Matt Casters
Roland Bouman
Jos van Dongen

Pentaho® Kettle Solutions

Building Open Source ETL Solutions with Pentaho Data Integration

www.allitebooks.com

www.allitebooks.com

Pentaho® Kettle Solutions

www.allitebooks.com

www.allitebooks.com

Pentaho® Kettle Solutions

Building Open Source ETL Solutions with Pentaho Data Integration

Matt Casters
Roland Bouman
Jos van Dongen

Wiley Publishing, Inc.

www.allitebooks.com

Pentaho® Kettle Solutions: Building Open Source ETL Solutions with Pentaho Data Integration

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-63517-9
ISBN: 9780470942420 (ebk)
ISBN: 9780470947524 (ebk)
ISBN: 9780470947531 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Control Number: 2010932421

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Pentaho is a registered trademark of Pentaho, Inc. All other trademarks are the property of their respective owners. Wiley Publishing, Inc. is not associated with any product or vendor mentioned in this book.

www.allitebooks.com

For my wife and kids, Kathleen, Sam and Hannelore.
Your love and joy keeps me sane in crazy times.

—Matt

For my wife, Annemarie, and my children, David, Roos,
Anne and Maarten. Thanks for bearing with me—I love you!

—Roland

For my children Thomas and Lisa, and for Yvonne, to whom
I owe more than words can express.

—Jos

www.allitebooks.com

www.allitebooks.com

About the Authors

Matt Casters has been an independent business intelligence consultant for many years and has implemented numerous data warehouses and BI solutions for large companies. For the last 8 years, Matt kept himself busy with the development of an ETL tool called Kettle. This tool was open sourced in December 2005 and acquired by Pentaho early in 2006. Since then, Matt took up the position of Chief Data Integration at Pentaho. His responsibility is to continue to be lead developer for Kettle. Matt tries to help the Kettle community in any way possible; he answers questions on the forum and speaks occasionally at conferences all around the world. He has a blog at `http://www.ibridge.be` and you can follow his `@mattcasters` account on Twitter.

Roland Bouman has been working in the IT industry since 1998 and is currently working as a web and business intelligence developer. Over the years he has focused on open source software, in particular database technology, business intelligence, and web development frameworks. He's an active member of the MySQL and Pentaho communities, and a regular speaker at international conferences, such as the MySQL User Conference, OSCON and at Pentaho community events. Roland co-authored the *MySQL 5.1. Cluster Certification Guide* and *Pentaho Solutions*, and was a technical reviewer for a number of MySQL and Pentaho related book titles. He maintains a technical blog at `http://rpbouman.blogspot.com` and tweets as `@rolandbouman` on Twitter.

Jos van Dongen is a seasoned business intelligence professional and well-known author and presenter. He has been involved in software development, business intelligence, and data warehousing since 1991. Before starting his own consulting practice, Tholis Consulting, in 1998, he worked for a top tier systems integrator and a leading management consulting firm. Over the past years, he has successfully implemented BI and data warehouse solutions for a variety of organizations, both commercial and non-profit. Jos covers new BI developments for the Dutch *Database Magazine* and speaks regularly at national and international conferences. He authored one book on open source BI and is co-author of the book *Pentaho Solutions*. You can find more information about Jos on `http://www.tholis.com` or follow `@josvandongen` on Twitter.

www.allitebooks.com

Credits

Executive Editor
Robert Elliott

Project Editor
Sara Shlaer

Technical Editors
Jens Bleuel
Sven Boden
Kasper de Graaf
Daniel Einspanjer
Nick Goodman
Mark Hall
Samatar Hassan
Benjamin Kallmann
Bryan Senseman
Johannes van den Bosch

Production Editor
Daniel Scribner

Copy Editor
Nancy Rapoport

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Marketing Manager
Ashley Zurcher

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Compositor
Maureen Forys,
Happenstance Type-O-Rama

Proofreader
Nancy Bell

Indexer
Robert Swanson

Cover Designer
Ryan Sneed

www.allitebooks.com

Acknowledgments

This book is the result of the efforts of many individuals. By convention, authors receive explicit credit, and get to have their names printed on the book cover. But creating this book would not have been possible without a lot of hard work behind the scenes. We, the authors, would like to express our gratitude to a number of people that provided substantial contributions, and thus help define and shape the final result that is *Pentaho Kettle Solutions*.

First, we'd like to thank those individuals that contributed directly to the material that appears in the book:

- Ingo Klose suggested an elegant solution to generate keys starting from a given offset within a single transformation (this solution is discussed in Chapter 8, "Handling Dimension Tables," subsection "Generating Surrogate Keys Based on a Counter," shown in Figure 8-2).
- Samatar Hassan provided text as well as working example transformations to demonstrate Kettle's RSS capabilities. Samatar's contribution is included almost completely and appears in the RSS section of Chapter 21, "Web Services."
- Thanks to Mike Hillyer and the MySQL documentation team for creating and maintaining the Sakila sample database, which is introduced in Chapter 4 and appears in many examples throughout this book.
- Although only three authors appear on the cover, there was actually a fourth one: We cannot thank Kasper de Graaf of DIKW-Academy enough for writing the Data Vault chapter, which has benefited greatly from his deep expertise on this subject. Special thanks also to Johannes van den Bosch who did a great job reviewing Kasper's work and gave another boost to the overall quality and clarity of the chapter.
- Thanks to Bernd Aschauer and Robert Wintner, both from Aschauer EDV (`http://www.aschauer-edv.at/en`), for providing the examples and screenshots used in the section dedicated to SAP of Chapter 6, "Data Extraction."
- Daniel Einspanjer of the Mozilla Foundation provided sample transformations for Chapter 7, "Cleansing and Conforming."

Thanks for your contributions. This book benefited substantially from your efforts.

Much gratitude goes out to all of our technical reviewers. Providing a good technical review is hard and time-consuming, and we have been very lucky to find a collection of such talented and seasoned Pentaho and Kettle experts willing to find some time in their busy schedules to provide us with the kind of quality review required to write a book of this size and scope.

We'd like to thank the Kettle and Pentaho communities. During and before the writing of this book, individuals from these communities provided valuable suggestions and ideas to all three authors for topics to cover in a book that focuses on ETL, data integration, and Kettle. We hope this book will be useful and practical for everybody who is using or planning to use Kettle. Whether we succeeded is up to the reader, but if we did, we have to thank individuals in the Kettle and Pentaho communities for helping us achieve it.

We owe many thanks to all contributors and developers of the Kettle software project. The authors are all enthusiastic users of Kettle: we love it, because it solves our daily data integration problems in a straightforward and efficient manner without getting in the way. Kettle is a joy to work with, and this is what provided much of the drive to write this book.

Finally, we'd like to thank our publisher, Wiley, for giving us the opportunity to write this book, and for the excellent support and management from their end. In particular, we'd like to thank our Project Editor, Sara Shlaer. Despite the often delayed deliveries from our end, Sara always kept her cool and somehow managed to make deadlines work out. Her advice, patience, encouragement, care, and sense of humor made all the difference and form an important contribution to this book. In addition, we'd like to thank our Executive Editor Robert Elliot. We appreciate the trust he put into our small team of authors to do our job, and his efforts to realize *Pentaho Kettle Solutions*.

—*The authors*

Writing a technical book like the one you are reading right now is very hard to do all by yourself. Because of the extremely busy agenda caused by the release process of Kettle 4, I probably should never have agreed to co-author. It's only thanks to the dedication and professionalism of Jos and Roland that we managed to write this book at all. I thank both friends very much for their invitation to co-author. Even though writing a book is a hard and painful process, working with Jos and Roland made it all worthwhile.

When Kettle was not yet released as open source code it often received a lukewarm reaction. The reason was that nobody was really waiting for yet another closed source ETL tool. Kettle came from that position to being the most widely deployed open source ETL tool in the world. This happened only thanks to the thousands of volunteers who offered to help out with various tasks. Ever since Kettle was open sourced it became a project with an every growing community. It's impossible to thank this community enough. Without the help of the developers, the translators, the testers, the bug reporters, the folks who participate in the forums, the people with the great ideas, and even the folks who like to complain, Kettle would not be where it is today. I would like to especially thank one important member of our community: Pentaho. Pentaho CEO Richard Daley and his team have done an excellent job in supporting the Kettle project ever

since they got involved with it. Without their support it would not have been possible for Kettle to be on the accelerated growth path that it is on today. It's been a pleasure and a privilege to work with the Pentaho crew.

A few select members of our community also picked up the tough job of reviewing the often technical content of this book. The reviewers of my chapters, Nicholas Goodman, Daniel Einspanjer, Bryan Senseman, Jens Bleuel, Samatar Hassan, and Mark Hall had the added disadvantage that this was the first time that I was going through the process of writing a book. It must not have been pretty at times. All the same they spent a lot of time coming up with insightful additions, spot-on advice, and to the point comments. I do enormously appreciate the vast amount of time and effort that they put into the reviewing. The book wouldn't have been the same without you guys!

—*Matt Casters*

I'd like to thank both my co-authors, Jos and Matt. It's an honor to be working with such knowledgeable and skilled professionals, and I hope we will collaborate again in the future. I feel our different backgrounds and expertise have truly complemented each other and helped us all to cover the many different subjects covered in this book.

I'd also like to thank the reviewers of my chapters: Benjamin Kallman, Bryan Senseman, Daniel Einspanjer, Sven Boden, and Samatar Hassan. Your comments and suggestions made all the difference and I thank you for your frank and constructive criticism.

Finally, I'd like to thank the readers of my blog at `http://rpbouman.blogspot.com/`. I got a lot of inspiration from the comments posted there, and I got a lot of good feedback in response to the blog posts announcing the writing of *Pentaho Kettle Solutions*.

—*Roland Bouman*

Back in October 2009, when *Pentaho Solutions* had only been on the shelves for two months and Roland and I agreed never to write another book, Bob Elliot approached us asking us to do just that. Yes, we had been discussing some ideas and already concluded that if there were to be another book, it would have to be about Kettle. And this was exactly what Bob asked us to do: write a book about data integration using Kettle. We quickly found out that Matt Casters was not only interested in reviewing, but in actually becoming a full author as well, an offer we gladly accepted. Looking back, I can hardly believe that we pulled it off, considering everything else that was going on in our lives. So many thanks to Roland and Matt for bearing with me, and thank you Bob and especially Sara for your relentless efforts of keeping us on track.

A special thank you is also warranted for Ralph Kimball, whose ideas you'll find throughout this book. Ralph gave us permission to use the Kimball Group's 34 ETL subsystems as the framework for much of the material presented in his book. Ralph also took the time to review Chapter 5, and thanks to his long list of excellent comments the chapter became a perfect foundation for Parts II, III, and IV of the book.

Finally I'd like to thank Daniel Einspanjer, Bryan Senseman, Jens Bleuel, Sven Boden, Samatar Hassan, and Benjamin Kallmann for being an absolute pain in the neck and thus doing a great job as technical reviewers for my chapters. Your comments, questions and suggestions definitely gave a big boost to the overall quality of this book.

—*Jos van Dongen*

Contents at a Glance

Introduction		xxxi
Part I	Getting Started	1
Chapter 1	ETL Primer	3
Chapter 2	Kettle Concepts	23
Chapter 3	Installation and Configuration	53
Chapter 4	An Example ETL Solution—Sakila	73
Part II	ETL	111
Chapter 5	ETL Subsystems	113
Chapter 6	Data Extraction	127
Chapter 7	Cleansing and Conforming	167
Chapter 8	Handling Dimension Tables	207
Chapter 9	Loading Fact Tables	245
Chapter 10	Working with OLAP Data	269
Part III	Management and Deployment	293
Chapter 11	ETL Development Lifecycle	295
Chapter 12	Scheduling and Monitoring	321

Chapter 13	Versioning and Migration	341
Chapter 14	Lineage and Auditing	357
Part IV	Performance and Scalability	375
Chapter 15	Performance Tuning	377
Chapter 16	Parallelization, Clustering, and Partitioning	403
Chapter 17	Dynamic Clustering in the Cloud	433
Chapter 18	Real-Time Data Integration	449
Part V	Advanced Topics	463
Chapter 19	Data Vault Management	465
Chapter 20	Handling Complex Data Formats	497
Chapter 21	Web Services	515
Chapter 22	Kettle Integration	569
Chapter 23	Extending Kettle	593
Appendix A	The Kettle Ecosystem	629
Appendix B	Kettle Enterprise Edition Features	635
Appendix C	Built-in Variables and Properties Reference	637
Index		643

Contents

Introduction **xxxi**

Part I **Getting Started** **1**

Chapter 1 **ETL Primer** **3**

OLTP versus Data Warehousing 3
What Is ETL? 5
The Evolution of ETL Solutions 5
ETL Building Blocks 7
ETL, ELT, and EII 8
ELT 9
EII: Virtual Data Integration 10
Data Integration Challenges 11
Methodology: Agile BI 12
ETL Design 14
Data Acquisition 14
Beware of Spreadsheets 15
Design for Failure 15
Change Data Capture 16
Data Quality 16
Data Profiling 16
Data Validation 17
ETL Tool Requirements 17
Connectivity 17
Platform Independence 18
Scalability 18
Design Flexibility 19
Reuse 19
Extensibility 19

Data Transformations 20
Testing and Debugging 21
Lineage and Impact Analysis 21
Logging and Auditing 22
Summary 22

Chapter 2 Kettle Concepts 23
Design Principles 23
The Building Blocks of Kettle Design 25
Transformations 25
Steps 26
Transformation Hops 26
Parallelism 27
Rows of Data 27
Data Conversion 29
Jobs 30
Job Entries 31
Job Hops 31
Multiple Paths and Backtracking 32
Parallel Execution 33
Job Entry Results 34
Transformation or Job Metadata 36
Database Connections 37
Special Options 38
The Power of the Relational Database 39
Connections and Transactions 39
Database Clustering 40
Tools and Utilities 41
Repositories 41
Virtual File Systems 42
Parameters and Variables 43
Defining Variables 43
Named Parameters 44
Using Variables 44
Visual Programming 45
Getting Started 46
Creating New Steps 47
Putting It All Together 49
Summary 51

Chapter 3 Installation and Configuration 53
Kettle Software Overview 53
Integrated Development Environment: Spoon 55
Command-Line Launchers: Kitchen and Pan 57
Job Server: Carte 57
Encr.bat and encr.sh 58
Installation 58

Java Environment 58
Installing Java Manually 58
Using Your Linux Package Management System 59
Installing Kettle 59
Versions and Releases 59
Archive Names and Formats 60
Downloading and Uncompressing 60
Running Kettle Programs 61
Creating a Shortcut Icon or Launcher for Spoon 62
Configuration 63
Configuration Files and the .kettle Directory 63
The Kettle Shell Scripts 69
General Structure of the Startup Scripts 70
Adding an Entry to the Classpath 70
Changing the Maximum Heap Size 71
Managing JDBC Drivers 72
Summary 72

Chapter 4 An Example ETL Solution—Sakila 73
Sakila 73
The Sakila Sample Database 74
DVD Rental Business Process 74
Sakila Database Schema Diagram 75
Sakila Database Subject Areas 75
General Design Considerations 77
Installing the Sakila Sample Database 77
The Rental Star Schema 78
Rental Star Schema Diagram 78
Rental Fact Table 79
Dimension Tables 79
Keys and Change Data Capture 80
Installing the Rental Star Schema 81
Prerequisites and Some Basic Spoon Skills 81
Setting Up the ETL Solution 82
Creating Database Accounts 82
Working with Spoon 82
Opening Transformation and Job Files 82
Opening the Step's Configuration Dialog 83
Examining Streams 83
Running Jobs and Transformations 83
The Sample ETL Solution 84
Static, Generated Dimensions 84
Loading the dim_date Dimension Table 84
Loading the dim_time Dimension Table 86
Recurring Load 87
The load_rentals Job 88

The load_dim_staff Transformation 91
Database Connections 91
The load_dim_customer Transformation 95
The load_dim_store Transformation 98
The fetch_address Subtransformation 99
The load_dim_actor Transformation 101
The load_dim_film Transformation 102
The load_fact_rental Transformation 107
Summary 109

Part II ETL 111

Chapter 5 ETL Subsystems 113
Introduction to the 34 Subsystems 114
Extraction 114
Subsystems 1–3: Data Profiling, Change Data Capture, and Extraction 115
Cleaning and Conforming Data 116
Subsystem 4: Data Cleaning and Quality Screen Handler System 116
Subsystem 5: Error Event Handler 117
Subsystem 6: Audit Dimension Assembler 117
Subsystem 7: Deduplication System 117
Subsystem 8: Data Conformer 118
Data Delivery 118
Subsystem 9: Slowly Changing Dimension Processor 118
Subsystem 10: Surrogate Key Creation System 119
Subsystem 11: Hierarchy Dimension Builder 119
Subsystem 12: Special Dimension Builder 120
Subsystem 13: Fact Table Loader 121
Subsystem 14: Surrogate Key Pipeline 121
Subsystem 15: Multi-Valued Dimension Bridge Table Builder 121
Subsystem 16: Late-Arriving Data Handler 122
Subsystem 17: Dimension Manager System 122
Subsystem 18: Fact Table Provider System 122
Subsystem 19: Aggregate Builder 123
Subsystem 20: Multidimensional (OLAP) Cube Builder 123
Subsystem 21: Data Integration Manager 123
Managing the ETL Environment 123
Summary 126

Chapter 6 Data Extraction 127
Kettle Data Extraction Overview 128
File-Based Extraction 128
Working with Text Files 128
Working with XML files 133
Special File Types 134

www.allitebooks.com

Database-Based Extraction 134
Web-Based Extraction 137
Text-Based Web Extraction 137
HTTP Client 137
Using SOAP 138
Stream-Based and Real-Time Extraction 138
Working with ERP and CRM Systems 138
ERP Challenges 139
Kettle ERP Plugins 140
Working with SAP Data 140
ERP and CDC Issues 146
Data Profiling 146
Using eobjects.org DataCleaner 147
Adding Profile Tasks 149
Adding Database Connections 149
Doing an Initial Profile 151
Working with Regular Expressions 151
Profiling and Exploring Results 152
Validating and Comparing Data 153
Using a Dictionary for Column Dependency Checks 153
Alternative Solutions 154
Text Profiling with Kettle 154
CDC: Change Data Capture 154
Source Data–Based CDC 155
Trigger-Based CDC 157
Snapshot-Based CDC 158
Log-Based CDC 162
Which CDC Alternative Should You Choose? 163
Delivering Data 164
Summary 164

Chapter 7 Cleansing and Conforming 167
Data Cleansing 168
Data-Cleansing Steps 169
Using Reference Tables 172
Conforming Data Using Lookup Tables 172
Conforming Data Using Reference Tables 175
Data Validation 179
Applying Validation Rules 180
Validating Dependency Constraints 183
Error Handling 183
Handling Process Errors 184
Transformation Errors 186
Handling Data (Validation) Errors 187
Auditing Data and Process Quality 191
Deduplicating Data 192

Handling Exact Duplicates 193
The Problem of Non-Exact Duplicates 194
Building Deduplication Transforms 195
Step 1: Fuzzy Match 197
Step 2: Select Suspects 198
Step 3: Lookup Validation Value 198
Step 4: Filter Duplicates 199
Scripting 200
Formula 201
JavaScript 202
User-Defined Java Expressions 202
Regular Expressions 203
Summary 205

Chapter 8 Handling Dimension Tables 207
Managing Keys 208
Managing Business Keys 209
Keys in the Source System 209
Keys in the Data Warehouse 209
Business Keys 209
Storing Business Keys 210
Looking Up Keys with Kettle 210
Generating Surrogate Keys 210
The "Add sequence" Step 211
Working with auto_increment or IDENTITY Columns 217
Keys for Slowly Changing Dimensions 217
Loading Dimension Tables 218
Snowflaked Dimension Tables 218
Top-Down Level-Wise Loading 219
Sakila Snowflake Example 219
Sample Transformation 221
Database Lookup Configuration 222
Sample Job 225
Star Schema Dimension Tables 226
Denormalization 226
Denormalizing to 1NF with the "Database lookup" Step 226
Change Data Capture 227
Slowly Changing Dimensions 228
Types of Slowly Changing Dimensions 228
Type 1 Slowly Changing Dimensions 229
The Insert / Update Step 229
Type 2 Slowly Changing Dimensions 232
The "Dimension lookup / update" Step 232
Other Types of Slowly Changing Dimensions 237
Type 3 Slowly Changing Dimensions 237
Hybrid Slowly Changing Dimensions 238

More Dimensions 239
Generated Dimensions 239
Date and Time Dimensions 239
Generated Mini-Dimensions 239
Junk Dimensions 241
Recursive Hierarchies 242
Summary 243

Chapter 9 Loading Fact Tables 245
Loading in Bulk 246
STDIN and FIFO 247
Kettle Bulk Loaders 248
MySQL Bulk Loading 249
LucidDB Bulk Loader 249
Oracle Bulk Loader 249
PostgreSQL Bulk Loader 250
Table Output Step 250
General Bulk Load Considerations 250
Dimension Lookups 251
Maintaining Referential Integrity 251
The Surrogate Key Pipeline 252
Using In-Memory Lookups 253
Stream Lookups 253
Late-Arriving Data 255
Late-Arriving Facts 256
Late-Arriving Dimensions 256
Fact Table Handling 260
Periodic and Accumulating Snapshots 260
Introducing State-Oriented Fact Tables 261
Loading Periodic Snapshots 263
Loading Accumulating Snapshots 264
Loading State-Oriented Fact Tables 265
Loading Aggregate Tables 266
Summary 267

Chapter 10 Working with OLAP Data 269
OLAP Benefits and Challenges 270
OLAP Storage Types 272
Positioning OLAP 272
Kettle OLAP Options 273
Working with Mondrian 274
Working with XML/A Servers 277
Working with Palo 282
Setting Up the Palo Connection 283
Palo Architecture 284
Reading Palo Data 285
Writing Palo Data 289
Summary 291

Part III Management and Deployment 293

Chapter 11 ETL Development Lifecycle 295

Solution Design 295
  Best and Bad Practices 296
    Data Mapping 297
    Naming and Commentary Conventions 298
    Common Pitfalls 299
  ETL Flow Design 300
  Reusability and Maintainability 300
Agile Development 301
Testing and Debugging 306
  Test Activities 307
  ETL Testing 308
    Test Data Requirements 308
    Testing for Completeness 309
    Testing Data Transformations 311
    Test Automation and Continuous Integration 311
    Upgrade Tests 312
  Debugging 312
Documenting the Solution 315
  Why Isn't There Any Documentation? 316
    Myth 1: My Software Is Self-Explanatory 316
    Myth 2: Documentation Is Always Outdated 316
    Myth 3: Who Reads Documentation Anyway? 317
  Kettle Documentation Features 317
  Generating Documentation 319
Summary 320

Chapter 12 Scheduling and Monitoring 321

Scheduling 321
  Operating System–Level Scheduling 322
    Executing Kettle Jobs and Transformations from the Command Line 322
    UNIX-Based Systems: cron 326
    Windows: The at utility and the Task Scheduler 327
  Using Pentaho's Built-in Scheduler 327
    Creating an Action Sequence to Run Kettle Jobs and Transformations 328
    Kettle Transformations in Action Sequences 329
    Creating and Maintaining Schedules with the Administration Console 330
    Attaching an Action Sequence to a Schedule 333
Monitoring 333
  Logging 333
    Inspecting the Log 333

Logging Levels 335
Writing Custom Messages to the Log 336
E-mail Notifications 336
Configuring the Mail Job Entry 337
Summary 340

Chapter 13 Versioning and Migration 341
Version Control Systems 341
File-Based Version Control Systems 342
Organization 342
Leading File-Based VCSs 343
Content Management Systems 344
Kettle Metadata 344
Kettle XML Metadata 345
Transformation XML 345
Job XML 346
Global Replace 347
Kettle Repository Metadata 348
The Kettle Database Repository Type 348
The Kettle File Repository Type 349
The Kettle Enterprise Repository Type 350
Managing Repositories 350
Exporting and Importing Repositories 350
Upgrading Your Repository 351
Version Migration System 352
Managing XML Files 352
Managing Repositories 352
Parameterizing Your Solution 353
Summary 356

Chapter 14 Lineage and Auditing 357
Batch-Level Lineage Extraction 358
Lineage 359
Lineage Information 359
Impact Analysis Information 361
Logging and Operational Metadata 363
Logging Basics 363
Logging Architecture 364
Setting a Maximum Buffer Size 365
Setting a Maximum Log Line Age 365
Log Channels 366
Log Text Capturing in a Job 366
Logging Tables 367
Transformation Logging Tables 367
Job Logging Tables 373
Summary 374

Part IV Performance and Scalability 375

Chapter 15 Performance Tuning 377
Transformation Performance: Finding the Weakest Link 377
Finding Bottlenecks by Simplifying 379
Finding Bottlenecks by Measuring 380
Copying Rows of Data 382
Improving Transformation Performance 384
Improving Performance in Reading Text Files 384
Using Lazy Conversion for Reading Text Files 385
Single-File Parallel Reading 385
Multi-File Parallel Reading 386
Configuring the NIO Block Size 386
Changing Disks and Reading Text Files 386
Improving Performance in Writing Text Files 387
Using Lazy Conversion for Writing Text Files 387
Parallel Files Writing 387
Changing Disks and Writing Text Files 387
Improving Database Performance 388
Avoiding Dynamic SQL 388
Handling Roundtrips 388
Handling Relational Databases 390
Sorting Data 392
Sorting on the Database 393
Sorting in Parallel 393
Reducing CPU Usage 394
Optimizing the Use of JavaScript 394
Launching Multiple Copies of a Step 396
Selecting and Removing Values 397
Managing Thread Priorities 397
Adding Static Data to Rows of Data 397
Limiting the Number of Step Copies 398
Avoiding Excessive Logging 398
Improving Job Performance 399
Loops in Jobs 399
Database Connection Pools 400
Summary 401

Chapter 16 Parallelization, Clustering, and Partitioning 403
Multi-Threading 403
Row Distribution 404
Row Merging 405
Row Redistribution 406
Data Pipelining 407
Consequences of Multi-Threading 408
Database Connections 408

Order of Execution 409
Parallel Execution in a Job 411
Using Carte as a Slave Server 411
The Configuration File 411
Defining Slave Servers 412
Remote Execution 413
Monitoring Slave Servers 413
Carte Security 414
Services 414
Clustering Transformations 417
Defining a Cluster Schema 417
Designing Clustered Transformations 418
Execution and Monitoring 420
Metadata Transformations 421
Rules 422
Data Pipelining 425
Partitioning 425
Defining a Partitioning Schema 425
Objectives of Partitioning 427
Implementing Partitioning 428
Internal Variables 428
Database Partitions 429
Partitioning in a Clustered Transformation 430
Summary 430

Chapter 17 Dynamic Clustering in the Cloud 433
Dynamic Clustering 433
Setting Up a Dynamic Cluster 434
Using the Dynamic Cluster 436
Cloud Computing 437
EC2 438
Getting Started with EC2 438
Costs 438
Customizing an AMI 439
Packaging a New AMI 442
Terminating an AMI 442
Running a Master 442
Running the Slaves 443
Using the EC2 Cluster 444
Monitoring 445
The Lightweight Principle and Persistence Options 446
Summary 447

Chapter 18 Real-Time Data Integration 449
Introduction to Real-Time ETL 449
Real-Time Challenges 450
Requirements 451

Transformation Streaming 452
A Practical Example of Transformation Streaming 454
Debugging 457
Third-Party Software and Real-Time Integration 458
Java Message Service 459
Creating a JMS Connection and Session 459
Consuming Messages 460
Producing Messages 460
Closing Shop 460
Summary 461

Part V Advanced Topics 463

Chapter 19 Data Vault Management 465
Introduction to Data Vault Modeling 466
Do You Need a Data Vault? 466
Data Vault Building Blocks 467
Hubs 467
Links 468
Satellites 469
Data Vault Characteristics 471
Building a Data Vault 471
Transforming Sakila to the Data Vault Model 472
Sakila Hubs 472
Sakila Links 473
Sakila Satellites 474
Loading the Data Vault: A Sample ETL Solution 477
Installing the Sakila Data Vault 477
Setting Up the ETL Solution 477
Creating a Database Account 477
The Sample ETL Data Vault Solution 478
Sample Hub: hub_actor 478
Sample Link: link_customer_store 480
Sample Satellite: sat_actor 483
Loading the Data Vault Tables 485
Updating a Data Mart from a Data Vault 486
The Sample ETL Solution 486
The dim_actor Transformation 486
The dim_customer Transformation 488
The dim_film Transformation 492
The dim_film_actor_bridge Transformation 492
The fact_rental Transformation 493
Loading the Star Schema Tables 495
Summary 495

Chapter 20 Handling Complex Data Formats 497
Non-Relational and Non-Tabular Data Formats 498
Non-Relational Tabular Formats 498
Handling Multi-Valued Attributes 498
Using the Split Field to Rows Step 499
Handling Repeating Groups 500
Using the Row Normaliser Step 500
Semi- and Unstructured Data 501
Kettle Regular Expression Example 503
Configuring the Regex Evaluation Step 504
Verifying the Match 507
Key/Value Pairs 508
Kettle Key/Value Pairs Example 509
Text File Input 509
Regex Evaluation 510
Grouping Lines into Records 511
Denormaliser: Turning Rows into Columns 512
Summary 513

Chapter 21 Web Services 515
Web Pages and Web Services 515
Kettle Web Features 516
General HTTP Steps 516
Simple Object Access Protocol 517
Really Simple Syndication 517
Apache Virtual File System Integration 517
Data Formats 517
XML 518
Kettle Steps for Working with XML 518
Kettle Job Entries for XML 519
HTML 520
JavaScript Object Notation 520
Syntax 521
JSON, Kettle, and ETL/DI 522
XML Examples 523
Example XML Document 523
XML Document Structure 523
Mapping to the Sakila Sample Database 524
Extracting Data from XML 525
Overall Design: The import_xml_into_db Transformation 526
Using the XSD Validator Step 528
Using the “Get Data from XML” Step 530
Generating XML Documents 537
Overall Design: The export_xml_from_db Transformation 537
Generating XML with the Add XML Step 538
Using the XML Join Step 541

SOAP Examples 544
Using the "Web services lookup" Step 544
Configuring the "Web services lookup" Step 544
Accessing SOAP Services Directly 546
JSON Example 549
The Freebase Project 549
Freebase Versus Wikipedia 549
Freebase Web Services 550
The Freebase Read Service 550
The Metaweb Query Language 551
Extracting Freebase Data with Kettle 553
Generate Rows 554
Issuing a Freebase Read Request 555
Processing the Freebase Result Envelope 556
Filtering Out the Original Row 557
Storing to File 558
RSS 558
RSS Structure 558
Channel 558
Item 559
RSS Support in Kettle 560
RSS Input 561
RSS Output 562
Summary 567

Chapter 22 Kettle Integration 569
The Kettle API 569
The LGPL License 569
The Kettle Java API 570
Source Code 570
Building Kettle 571
Building javadoc 571
Libraries and the Class Path 571
Executing Existing Transformations and Jobs 571
Executing a Transformation 572
Executing a Job 573
Embedding Kettle 574
Pentaho Reporting 574
Putting Data into a Transformation 576
Dynamic Transformations 580
Dynamic Template 583
Dynamic Jobs 584
Executing Dynamic ETL in Kettle 586
Result 587
Replacing Metadata 588
Direct Changes with the API 589
Using a Shared Objects File 589

www.allitebooks.com

OEM Versions and Forks 590
Creating an OEM Version of PDI 590
Forking Kettle 591
Summary 592

Chapter 23 Extending Kettle 593
Plugin Architecture Overview 593
Plugin Types 594
Architecture 595
Prerequisites 596
Kettle API Documentation 596
Libraries 596
Integrated Development Environment 596
Eclipse Project Setup 597
Examples 598
Transformation Step Plugins 599
StepMetaInterface 599
Value Metadata 605
Row Metadata 606
StepDataInterface 607
StepDialogInterface 607
Eclipse SWT 607
Form Layout 607
Kettle UI Elements 609
Hello World Example Dialog 609
StepInterface 614
Reading Rows from Specific Steps 616
Writing Rows to Specific Steps 616
Writing Rows to Error Handling 617
Identifying a Step Copy 617
Result Feedback 618
Variable Substitution 618
Apache VFS 619
Step Plugin Deployment 619
The User-Defined Java Class Step 620
Passing Metadata 620
Accessing Input and Fields 620
Snippets 620
Example 620
Job Entry Plugins 621
JobEntryInterface 622
JobEntryDialogInterface 624
Partitioning Method Plugins 624
Partitioner 625
Repository Type Plugins 626
Database Type Plugins 627
Summary 628

Appendix A The Kettle Ecosystem **629**
Kettle Development and Versions 629
The Pentaho Community Wiki 631
Using the Forums 631
Jira 632
##pentaho 633

Appendix B Kettle Enterprise Edition Features **635**

Appendix C Built-in Variables and Properties Reference **637**
Internal Variables 637
Kettle Variables 640
Variables for Configuring VFS 641
Noteworthy JRE Variables 642

Index **643**

Introduction

More than 50 years ago the first computers for general use emerged, and we saw a gradually increasing adoption of their use by the scientific and business world. In those early days, most organizations had just one computer with a single display and printer attached to it, so the need for integrating data stored in different systems simply didn't exist. This changed when in the late 1970s the relational database made inroads into the corporate world. The 1980s saw a further proliferation of both computers and databases, all holding different bits and pieces of an organization's total collection of information. Ultimately, this led to the start of a whole new industry, which was sparked by IBM researchers Dr. Barry Devlin and Paul Murphy in their seminal paper "An architecture for a business and information system" (first published in 1988 in *IBM Systems Journal*, Volume 27, Number 1). The concept of a business data warehouse was introduced for the first time as being "the single logical storehouse of all the information used to report on the business." Less than five years later, Bill Inmon published his landmark book, *Building the Data Warehouse*, which further popularized the concepts and technologies needed to build this "logical storehouse."

One of the core themes in all data warehouse–related literature is the concept of integrating data. The term *data integration* refers to the process of combining data from different sources to provide a single comprehensible view on all of the combined data. A typical example of data integration would be combining the data from a warehouse inventory system with that of the order entry system to allow order fulfillment to be directly related to changes in the inventory. Another example of data integration is merging customer and contact data from separate departmental customer relationship management (CRM) systems into a corporate customer relationship management system.

NOTE **Throughout this book, you'll find the terms "data integration" and "ETL" (short for extract, transform, and load) used interchangeably. Although technically not entirely correct (ETL is only one of the possible data integration scenarios, as you'll see in Chapter 1), most developers treat these terms as synonyms, a sin that we've adopted over the years as well.**

In an ideal world, there wouldn't be a need for data integration. All the data needed for running a business and reporting on its past and future performance would be stored and managed in a single system, all master data would be 100 percent correct, and every piece of external data needed for analysis and decision making would be automatically linked to our own data. This system wouldn't have any problems with storing all available historical data, nor with offering split-second response times when querying and analyzing this data.

Unfortunately, we don't live in an ideal world. In the real world, most organizations use different systems for different purposes. They have systems for CRM (Customer Relationship Management), for accounting, for sales and sales support, for supporting a help desk, for managing inventory, for supporting a logistics process, and the list goes on and on. To make things worse, the same data is often stored and maintained independently, and probably inconsistently, in different systems. Customer and product data might be available in all the aforementioned systems, and when a customer calls to pass on a new telephone number or a change of address, chances are that this information is only updated in the CRM system, causing inconsistency of the customer information within the organization.

To cope with all these challenges and create a single, integrated, conformed, and trustworthy data store for reporting and analysis, data integration tools are needed. One of the more popular and powerful solutions available is Kettle, also known as Pentaho Data Integration, which is the topic of this book.

The Origins of Kettle

Kettle originated ten years ago, at the turn of the century. Back then, ETL tools could be found in all sorts of shapes and forms. At least 50 known tools competed in this software segment. Beneath that collection of software, there was an even larger set of ETL frameworks. In general, you could split up the tools into different types based on their respective origin and level of sophistication, as shown in Figure 1.

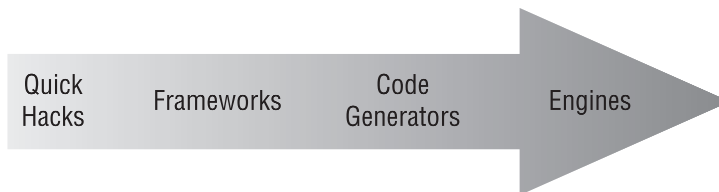


Figure 1: ETL tool generations

- **Quick hacks:** These tools typically were responsible for extraction of data or the load of text files. A lot of these solutions existed out there and still do. Words such as "hacker" and "hacking" have an undeservedly negative connotation. Business intelligence can get really complex and in most cases, the quick hacks make the difference between project disaster and success. As such, they pop up quite easily

because someone simply has a job to do with limited time and money. Typically, these ETL quick hack solutions are created by consultancy firms and are meant to be one-time solutions.

- **Frameworks:** Usually when a business intelligence consultant does a few similar projects, the idea begins to emerge that code needs to be written in such a way that it can be re-used on other projects with a few minor adjustments. At one point in time it seemed like every self-respecting consultancy company had an ETL framework out there. The reason for this is that these frameworks offer a great way to build up knowledge regarding the ETL processes. Typically, it is easy to change parameters for extraction, loading, logging, change data capture, database connections, and such.
- **Code generators:** When a development interface is added as an extra level of abstraction for the frameworks, it is possible to generate code for a certain platform (C, Java, SQL, and so on) based on sets of metadata. These code generators come in different types, varying from one-shot generators that require you to maintain the code afterward to full-fledge ETL tools that can generate everything you need. These kinds of ETL tools were also written by consultancy companies left and right, but mostly by known, established vendors.
- **Engines:** In the continuing quest by ETL vendors to make life easier for their users, ETL engines were created so that no code had to be generated. With these engines, the entire ETL process can be executed based on *parameterization* and *configuration*, i.e. the description of the ETL process itself as described throughout this book. This by itself does away with any code generation, compilation, and deployment difficulties.

Based on totally non-scientific samples of projects that were executed back then, it's safe to say that over half of the projects used quick hacks or frameworks. Code generators in all sorts of shapes and forms accounted for most of the rest, with ETL engines only being used in exceptional cases, usually for very large projects.

NOTE **Very few tools were available under an open source license ten years ago. The only known available tool was Enhydra Octopus, a Java-based code generator type of ETL tool (available at** `www.enhydra.org/tech/octopus/`**). To its credit and benefit to its users, it's still available as a shining example of the persistence of open source.**

It's in this software landscape that Matt Casters, the author of Kettle and one of the authors of this book, was busy with consultancy, writing quick hacks and frameworks, and deploying all sorts of code generators.

Back in the early 2000s he was working as a business intelligence consultant, usually in the position of a data warehouse architect or administrator. In such a position, you have to take care of bridging the well-known gap between information and communication technology and business needs. Usually this sort of work was done without a big-vendor ETL tool because those things were prohibitively costly back then. As such, these tools were too expensive for most, if not all, small-to-medium–sized projects. In

that situation, you don't have much of a choice: You face the problem time after time and you do the best you can with all sorts of frameworks and code generation. Poor examples of that sort of work include a program, written in C and embedded SQL (ESQL/C) to extract data from Informix; an extraction tool written in Microsoft Visual Basic to get data from an IBM AS/400 Mainframe system; and a complete data warehouse consisting of 35 fact tables and 90 slowly changing dimensions for a large bank, written manually in Oracle PL/SQL and shell scripts.

Thus, it would be fair to say that Matt knew what he was up to when he started thinking about writing his own ETL tool. Nevertheless, the idea to write it goes back as far as 2001:

> Matt: *"I'm going to write a new piece of software to do ETL. It's going to take up some time left and right in the evenings and weekends."*
>
> Kathleen (Matt's wife): *"Oh, that's great! How long is this going to take?"*
>
> Matt: *"If all goes well, I should have a first somewhat working version in three years and a complete version in five years."*

The Design of Kettle

After more than ten years of wrestling with ETL tools of dubious quality, one of the main design goals of Kettle was to be as open as possible. Back then that specifically meant:

- Open, readable metadata (XML) format
- Open, readable relational repository format
- Open API
- Easy to set up (less than 2 minutes)
- Open to all kinds of databases
- Easy-to-use graphical user interface
- Easy to pass data around
- Easy to convert data from/to any possible format

During the first two years, progress was slow while a lot of work was spent figuring out what the ultimate list of capabilities would be for the new ETL tool. The idea to create a parallel ETL engine comes from that time frame. Multiple negative experiences with quick hacks, frameworks, and code generators led to the conviction that the solution had to be engine-based.

Because Matt's background was primarily with the C programming language, he started dabbling with things like client/server code to test passing data between processes and servers. Testing different scenarios taught him a lot about performance bottlenecks in the encoding/decoding of data. As a consequence, one of the major

design principles became to leave rows of data untouched as much as possible. To date, this principle is still present in Kettle.

NOTE The name "Kettle" came originally from "KDE ETTL Environment" because the original plan was to write the software on top of the K Desktop Environment (`www.kde.org`). It was later renamed recursively "Kettle ETTL Environment" after that plan was dropped.

Ultimately, it was the lack of decent drivers for the multitude of relational databases that drove the development to the new and upcoming Java programming language. Work on that started in early 2003. The Standard Widget Toolkit (SWT) was chosen because Matt had prior negative experiences with the performance and look of the then available Java AWT (Abstract Window Toolkit). In contrast, SWT (Standard Widget Toolkit) used native operating system widgets to speed up things on the screen and comparatively looked good on all operating systems.

Combine a Java newbie with advanced ETL topics, and you will not be surprised to hear that for the first year of development, the Kettle codebase was a complete mess. The code didn't have packages; it was unstructured and had funky (C-style) naming conventions. Exception handling was unheard of and crashes were not exceptional. The only thing this embryonic version of Kettle had going for it really was the fact that it worked. It was capable of reading text files, reading from databases, and writing to databases, and it had a versatile JavaScript step that allowed you to get by most tough problems. Most of all, it was very flexible and easy to use. This was, after all, a business intelligence tool, not a Java project.

However, it was clear that at some point it needed to become a lot better. So help arrived in the early years in the form of a friend, Wim De Clercq, the co-owner of ixor (`www.ixor.be`) and a senior enterprise Java architect. He explained the basics of core Java concepts such as packages and exception handling. Time was spent reading up on design patterns such as singletons to simplify the code.

Listening to that advice meant performing massive amounts of code changes. As a consequence, it was not unusual back then for Matt to spend weekends doing nothing but re-factoring code in Eclipse, rewriting tens of thousands of lines of code. But, bit by bit, over the course of many weeks and months, things kept going in the right direction.

Kettle Gets Some Traction

These initial positive results were shared with peers, colleagues, and other senior BI consultants to hear what they thought of Kettle. That's how the news spread around slowly, and that's how in 2004, Kettle got deployed at the Flemish Traffic Center (`www.verkeerscentrum.be`) where billions of rows of data had to be integrated from thousands of data sources all over Belgium. There was no time to write new code and no money to buy a big name ETL tool—so Kettle entered the picture. The specific tasks that had to be performed at the traffic center led to many improvements that could be implemented in a full time capacity for the first time. Consequently, Kettle improved very fast in that period. For example, the database drivers improved dramatically because now

there were really diverse test cases. It was also around that time that messages got out to the world to let people know they could download a gratis (free of charge, not open source) copy of Kettle for their own use.

Reactions were few but mostly positive. The most interesting response came from a nice person named Jens Bleuel in Germany who asked if it was possible to integrate third-party software into Kettle, more specifically an SAP/R3 connector. Kettle version 1.2 was just deployed at the Traffic Center and it would certainly be possible to put code in there, but it didn't have a plugin architecture. Jens' request to integrate existing SAP/R3 code into Kettle was the main reason to develop the plugin system, and this became version 2.0 of Kettle. Ultimately, this effort took until the end of 2004. It was a fairly complete release with support for slowly changing dimensions, junk dimensions, 28 steps and 13 databases. It was then that the real potential of the tool started to show. This in turn led to the creation by Jens Bleuel of the very first Kettle plugin, ProSAPCON, used to read data from an SAP/R3 server.

Kettle Goes Open Source

There was a lot of excitement during that period, and Matt and Jens agreed to start promoting the sale of Kettle from the `kettle.be` website and via the newfound partner Proratio (`www.proratio.de`), the company where Jens was working at that time.

Improvements kept coming and evaluation requests and interest mounted. However, doing development and sales for a complete ETL tool is too big a task for any single person. Matt discovered also that working on Kettle was fun, but selling it was not. He had to find a way to concentrate on the fun part of Kettle development. So by late summer 2005, the decision was made to go open source. This would let Kettle sell itself as well as attract contributions from external developers.

When the code and free downloads of version 2.2 were first published in December 2005, the response was massive. The download package that was put up on JavaForge got downloaded around 35,000 times in the first week alone. The news spread all over the world pretty quickly.

Given the large number of open source projects that had turned into "abandonware," it was important to build a community around Kettle as fast as possible. That meant answering (literally) thousands of e-mails and forum posts in the next few years. Fortunately, help quickly arrived in the form of an open source business intelligence company called Pentaho (`www.pentaho.com`), which acquired the rights to the source code and employed Matt as lead developer of Kettle. Later, Kettle was re-branded as Pentaho Data Integration.

Help also arrived in the form of many developers, translators, reviewers, doc writers, and thousands of bug reports without whose help Kettle would not be where it is today.

About This Book

The beginnings of *Pentaho Kettle Solutions* go back to August 2009, around the time when the first book by Roland and Jos, *Pentaho Solutions*, was released. Just like anyone who has run his or her first marathon, they proclaimed to each other "never again." But, as

they saw the enthusiastic responses to the first book, the tone of the conversation gradually changed and soon they were saying that if there were to be another book, it had to be about Kettle and data integration. When Bob Elliot, the publisher of the first Pentaho book, approached them about the possibility of writing a Kettle book, the topics and table of contents were already well underway. To their relief (and surprise), both their spouses encouraged them to go ahead. The good news kept coming; after consulting Matt Casters about helping out with the review process, he offered to become one of the main authors, an offer that was gladly accepted, of course.

The same motivation that spurred the writing of *Pentaho Solutions* still holds today: There's an ongoing and increasing interest in open source and free software solutions, combined with a growing recognition that business intelligence (BI) solutions are essential in measuring and improving an organization's performance. These BI solutions require an integrated collection of data that is prepared in such a way that it is directly usable for analysis, reporting, and dashboarding. This is the key reason why most BI projects start with a data integration effort, and why this book is invaluable in assisting you with this.

Over the past decade, open source variants of more and more types of software have become commonly accepted and respected alternatives to their more costly and less flexible proprietary counterparts. The fact that software is open source is often mistaken for being free of cost, and although that might be true if you only look at the license costs, a BI solution cannot (and never will) be free of cost. There are costs associated with hardware, implementation, maintenance, training, and migration, and when this is all added up it turns out that licenses make up only a small portion of the total lifecycle cost of any software solution. Open source, however, is much more than a cheaper way of acquiring software. The fact that the source code is freely available to anyone ensures better code quality because it is more likely that bugs are found when more people have access to the source than just the core developers. The fact that open source software is built on open standards using standard programming languages (mostly Java) makes it extremely flexible and extensible. And the fact that most open source software is not tied to a particular operating system extends this flexibility and freedom even further.

What is usually lacking, however, is a good set of documentation and manuals. Most open source projects provide excellent quality software, but developers usually care more about getting great software out than delivering proper documentation. And although you can find many good sources of information about Kettle, we felt there was a need for a single source of information to help an ETL developer on his or her way in discovering the Kettle toolset and building robust data integration solutions. That is exactly what this book is for—to help you to build data integration solutions using Kettle.

Who Should Read This Book

This book is meant for anyone who wants to know how to deliver ETL solutions using Kettle. Maybe you're an IT manager looking for a cost-efficient ETL solution, an IT professional looking to broaden your skill set, or a BI or data warehouse consultant responsible for developing ETL solutions in your organization. Maybe you're a software developer with a lot of experience building open source solutions but still new to the world of data integration. And maybe you're already an experienced ETL developer

with deep knowledge of one or more of the existing proprietary tools. In any case, we assume you have a hands-on mentality because this is a hands-on book. We do expect some familiarity with using computers to deliver information, installing software, and working with databases, but most of the topics will be explained right from the start. Of course, the data integration concepts are explained as well, but the primary focus is on how to transform these concepts into a working solution. That is exactly why the book is called *Pentaho Kettle Solutions*.

What You Will Need to Use This Book

In order to use this book, you need only two things: a computer and an Internet connection. All the software we discuss and use in this book is freely available over the Internet for download and use. The system requirements for the computer you will need are fairly moderate; in fact, any computer that is less than four years old will do the job just fine, as long as you have at least 1 gigabyte of RAM installed and 2 gigabytes of free disk space available for downloading and installing software.

The various chapters contain URLs where you can find and download the software being used and the accompanying installation instructions. As for Pentaho, there are, apart from the actual source code of course, four versions of the software that you can use:

- **GA (General Availability) releases:** These are the stable builds of the software, usually not the most recent ones but surely the most reliable.
- **Release candidates:** The “almost ready” next versions of the software, possibly with a few minor bugs still in them.
- **Milestone releases:** These are created more frequently and allow you to work with recent versions introducing new features.
- **Nightly builds:** The most up-to-date versions of the software, but also the least stable ones.

When writing this book, we mostly worked with the nightly builds that generally precede the GA releases by three months or more. At the time of writing, the GA version of Kettle 4.0 has just been released. This means that when you read this book, the software used in this book will have already been put through its paces and most initial bugs will be fixed. This allows you to work through the material using a stable, bug-free product, and you can concentrate on building solutions, not fixing bugs.

The complete list with download options is available online at `http://wiki.pentaho.com/display/COM/Community+Edition+Downloads`.

What You Will Learn from This Book

This book will teach you:

- What data integration is, and why you need it
- The concepts that form the foundation of the Kettle solution
- How to install and configure Kettle, both on a single computer and a client/server environment

www.allitebooks.com

- How to build a complete end-to-end ETL solution for the MySQL Sakila demo database
- What the 34 subsystems of ETL are and how they translate to the Kettle toolkit
- How Kettle can be used for data extraction, cleansing and conforming, handling dimension tables, loading fact tables, and working with OLAP cubes
- What the Kettle development lifecycle looks like
- How to take advantage of Pentaho's Agile BI tools from within the Kettle development environment
- How to schedule and monitor jobs and transformations
- How to work with multiple developers and manage different versions of an ETL solution
- What data lineage, impact analysis, and auditing is, and how Kettle supports these concepts
- How to increase the performance and throughput of Kettle using partitioning, parallelization, and dynamic clustering
- How to use complex files, web services, and web APIs
- How to use Kettle to load an enterprise data warehouse designed according to the Data Vault principles
- How to integrate Kettle with other solutions and how to extend Kettle by developing you own plugins

How This Book Is Organized

This book explains ETL concepts, technologies, and solutions. Rather than using a single example, we use several scenarios to illustrate the various concepts, although the MySQL Sakila example database is heavily used throughout the book. When the example relies on a database, we have taken care to ensure the sample code is compatible with the popular and ubiquitous MySQL database (version 5.1).

These samples provide the technical details necessary to understand how you can build ETL solutions for real-world situations. The scope of these ETL solutions ranges from the level of the departmental data mart to the enterprise data warehouse.

Part I: Getting Started

Part I of this book focuses on gaining a quick and high-level understanding of the Kettle software, its architecture, and its capabilities. This part consists of the following chapters:

Chapter 1: ETL Primer—Introduces the main concepts and challenges found in data-integration projects. We explain what the difference between transaction and analytical systems is, where ETL fits in, and how the various components of an ETL solution work to solve data-integration problems.

Chapter 2: Kettle Concepts—Provides an overview of the design principles used as the foundations for Kettle and the underlying architecture of the software. We explain the basic building blocks, consisting of jobs, transformations, steps, and hops, and how they interact with each other. You'll also learn how Kettle interacts with databases and how this can be influenced by setting database-specific options. This chapter also contains a hands-on mini tutorial to quickly walk you through Kettle's user interface.

Chapter 3: Installation and Configuration—Explains how and where to obtain the Kettle software and how to install it. We explain which programs make up Kettle, and how these different programs relate to each other and to building ETL solutions. Finally, we explain various configuration options and files and where to find them.

Chapter 4: An Example ETL Solution—Sakila—Explains how to build a complete ETL solution based on the popular MySQL Sakila sample database. Based on a standard star schema designed for this chapter, you'll learn how to work with slowly changing dimensions and how to work with lookup steps for loading fact tables. An important topic is the use of mapping steps in a transformation to be able to re-use existing transformations.

Part II: ETL

The second part of this book is entirely devoted to the 34 ETL subsystems as laid out by Dr. Ralph Kimball and his colleagues from the Kimball Group in their latest book, *The Kimball Group Reader* (Wiley, 2010) and before that in the 2nd edition of *The Data Warehouse Lifecycle Toolkit* (Wiley, 2008), one of the best-selling data warehousing books in history. This part includes the following chapters:

Chapter 5: ETL Subsystems—Provides an introduction to the various subsystems and their categorization. The four categories used are Extracting, Cleansing and Conforming, Delivering, and Managing. In this chapter, we also explain how Kettle supports each of the subsystems. The chapter is not only the foundation to the other chapters in this part of the book, but is essential reading to understand the way ETL solutions in general should be architected.

Chapter 6: Data Extraction—Covers the first main category of subsystems consisting of data profiling, change data capture, and the extract system itself. We explain what data profiling is and why this should always be the first activity in any ETL project. Change data capture (CDC) is aimed at detecting changes in a source system for which we provide several solutions that can be used in conjunction with Kettle.

Chapter 7: Cleansing and Conforming—This second main ETL subsystem category is where the real action of the ETL process takes place. In most cases, it's not enough to read data from different sources and deliver it somewhere else. Data must be made uniform; redundant or duplicate data needs to be deleted; and multiple encoding schemes need to be conformed to a single uniform encoding

for the data warehouse. Many of the Kettle steps that can be used for transforming data are explained and used in example transformations, including the new Fuzzy Lookup step using advanced string matching algorithms that can be used to deduplicate data.

Chapter 8: Handling Dimension Tables—This is part of the third subsystem category, Delivering. We start by explaining what dimension tables are. We describe the various load and update types for these tables using the "Dimension lookup / update" step, and give special attention to subsystem 10, the surrogate key generator. Special dimension types such as time dimensions, junk, or heterogeneous dimensions and mini dimensions are covered as well, and we conclude by explaining recursive hierarchies.

Chapter 9: Loading Fact Tables—This chapter covers loading the different types of fact tables. The first half of the chapter is devoted to the various load strategies that can be used to update fact tables and explains how to accommodate for late- or early-arriving facts. We introduce and demonstrate the different bulk loaders present in Kettle. Apart from the most familiar fact table type, the transaction fact table, we also explain the periodic and accumulating fact tables. Finally, a new type of fact table is introduced, the *state oriented* fact table.

Chapter 10: Working with OLAP Data—This is an entire chapter devoted to only one of the subsystems (20, OLAP cube builder). In this chapter, we illustrate how to work with the three types of OLAP sources and targets: reading data from XML/A and Mondrian cubes, and reading from and loading Palo cubes.

Part III: Management and Deployment

Where the previous part of this book focused on how to build solutions, the chapters in this part focus on how to deploy and manage them.

Chapter 11: ETL Development Lifecycle—This chapter takes one step back and discusses how to design, develop, and test an ETL solution using the tools available in Kettle. We cover the new Agile BI tools and how they can help speed up the development process, and explain what types of testing are required before a solution can be delivered.

Chapter 12: Scheduling and Monitoring—Covers the different scheduling options. We describe standard operating system scheduling tools such as `cron` and the Windows Task Scheduler, and the built-in Pentaho BI scheduler. Monitoring running jobs and transformations can be done directly from the design environment, but we also show you how to use the logging tables to retrieve information about completed or failed jobs.

Chapter 13: Versioning and Migration—Explains how to keep different versions of Kettle jobs and transformations, enabling a roll back to a previous version if necessary. Another important topic covered in this chapter is the separation of development, test, acceptance, and production environments and how to migrate or promote Kettle objects from one stage to the next.

Chapter 14: Lineage and Auditing—In this chapter, you learn how to use the Kettle metadata to find out where data came from and where it is used. For auditing purposes, it's important to be able to keep track of when jobs ran, how long they took, and how many and what changes were made to the data. To accommodate for this, Kettle has extensive logging capabilities, which we describe in detail.

Part IV: Performance and Scalability

This part of the book is all about speeding up the ETL process. Several options are available for increasing extract, transform, and load speeds, and each chapter covers a specific class of solutions that can be used in a Kettle environment. The following chapters cover these topics:

Chapter 15: Performance Tuning—Explains the inner workings of the transformation engine and assists you in detecting performance bottlenecks. We present several solutions to improve performance and throughput. A large part of this chapter is devoted to speeding up the processing of text files and pushing data through the Kettle stream as fast as possible.

Chapter 16: Parallelization, Clustering, and Partitioning—Describes more techniques to increase Kettle's performance. Two classes of strategies exist: scale up and scale out. A scale up strategy aims at taking advantage of the processing power in a single machine by leveraging multi-core CPUs and large amounts of memory. Scale out means distributing a task to multiple different servers. We explain both strategies and the way they can be used from within Kettle.

Chapter 17: Dynamic Clustering in the Cloud—Shows you how to take advantage of the huge computing power that's available on demand nowadays. The chapter explains how to apply the clustering principles presented in Chapter 16 to a cloud environment, in this case the Amazon Elastic Computing Cloud (EC2). You'll learn how to dynamically expand and decline a computing cluster based on the expected workload in order to minimize cost and maximize peak performance.

Chapter 18: Real-Time Data Integration—Takes a look at how a perpetual stream of data can be handled by Kettle. As explained in this chapter, the Kettle engine is stream-based in its genes. All it takes to handle real-time, streaming data is to connect to a permanent data stream or message queue and fire up the transformation.

Part V: Advanced Topics

The Advanced Topics part of this book covers miscellaneous subjects to illustrate the power of Kettle, how to extend this power, and how to use it from third-party applications, for instance a custom Java application.

Chapter 19: Data Vault Management—Explains what the Data Vault (DV) modeling technique for enterprise data warehousing is and how Kettle can be used to load the three types of tables that make up a DV schema: hubs, links, and satellites.

Chapter 20: Handling Complex Data Formats—Shows you how to work with data of a non-relational nature. Different types of semi-structured and unstructured data are covered, including the way data in a key/value pair format can be transformed into regular tables, how to use regular expressions to structure seemingly unstructured data, and how to deal with repeating groups and multi-valued attributes. The ability to handle key/value pair data stores is becoming more relevant as many of the so-called *schema-less* or *NoSQL* databases store their data in this format.

Chapter 21: Web Services—Takes a deep dive into the world of data available on the World Wide Web. The chapter covers the main components of the Web, and describes the various methods for accessing data from the Web within Kettle, such as HTTP GET, POST, and SOAP. We also thoroughly explain data formats such as XML, JSON and RSS, and show how to process these formats with Kettle.

Chapter 22: Kettle Integration—Illustrates the various ways in which Kettle can be used from external applications. The Kettle API is described and several examples help you on your way to embedding Kettle jobs and transformations in a custom application. One of the easiest ways to do this is to use Pentaho Reports, which can use a Kettle transformation as a data source.

Chapter 23: Extending Kettle—Teaches you how to write your own plugins to augment the already extended capabilities of Kettle. This final chapter covers the prerequisites and tools you need, and describes how to develop the various types of plugins: steps, job entries, partitioning methods, repository types, and database types.

Appendixes

We conclude the book with a few quick references.

Appendix A: The Kettle Ecosystem—Draws a map of the Kettle world and explains who's involved, where to get (or give!) help, and how to interact with others using the forums. We also explain how to work with Jira, Pentaho's issue management system, to find out about and track bugs, monitor their status, and see what's on the roadmaps.

Appendix B: Kettle Enterprise Edition Features—Explains the differences between the two editions of Kettle and highlights the extra features available in the Enterprise Edition.

Appendix C: Built-in Variables and Properties Reference—Provides an overview of all default Kettle variables and properties you can call and use from within your Kettle solutions.

Prerequisites

This book is mainly about Kettle, and installing and using this software are the primary topics of this book. Chapter 3 describes the installation and configuration of Kettle, but there are other pieces of software you'll need in order to follow along with all the

examples and instructions in this book. This section points you to these tools and shows you how to get and install them.

Java

Kettle (and the rest of the Pentaho stack for that matter) runs on the Java platform. Although Java is ubiquitous and probably already installed on your system, we do provide installation instructions and considerations in Chapter 3.

MySQL

The MySQL database is the default database we use throughout the book. It can be obtained from the MySQL website at `http://dev.mysql.com/downloads/mysql`. The MySQL database is available for almost any current operating system and can be installed either by downloading and running the installer for your environment, or by using the standard repositories for a Linux system. If you're running Ubuntu and don't have a MySQL server installed yet, you can do so very quickly by opening a terminal screen and executing the command `sudo apt-get install mysql-server`. Because this will only install the server, you might also want to download and install the GUI tools to work with the database outside of the Kettle environment. Starting with version 5.2, the MySQL Workbench now contains the database modeling, management, and query tools in one integrated solution.

SQL Power Architect

To model your target environment, we strongly recommend using a data modeling tool because this capability is not available in the Kettle solution, nor anywhere else in the Pentaho toolset. One of the best open source solutions around is Power Architect, which is used in this book as well. You can find the tool on the download page of SQLPower, the Canadian company that develops and maintains Power Architect, which is located at `www.sqlpower.ca/page/architect`.

Eclipse

Kettle is programmed in Java, using the Eclipse IDE (Integrated Development Environment), and the final chapter of this book contains instructions for developing your own Kettle plugins using Eclipse. Eclipse is a versatile tool that can be used to program solutions in any programming language you can think of. Thanks to the architecture of the tool, it can also be used for data modeling, report creation, data mining, and so on by using plugins and switching to different perspectives. Eclipse can be obtained from the download page at `www.eclipse.org/downloads`. If you're running Ubuntu, it's in the standard repositories and can be installed either from the Software Center (Developer Tools ⇨ IDEs in 10.04 or Programming in 9.10), or by running `sudo apt-get install eclipse` from the command line.

On the Website

All the example material used in the book is available for download from the companion website at Wiley (`www.wiley.com/go/kettlesolutions`). The downloads are organized into folders for each chapter, in which you will find:

- Power*Architect data models for the sample databases in the book
- All PDI jobs and transformations
- SQL Scripts for examples and modifications

Further Resources

Numerous books are available on the specific topics covered in this book. Many chapters contain references for further reading and links to websites that contain additional information. If you are new to business intelligence and data warehousing in general (or want to keep up with the latest developments), here are some good places to start:

- `http://en.wikipedia.org/wiki/Business_intelligence`
- `http://www.kimballgroup.com`
- `http://b-eye-network.com`
- `http://www.tdwi.org`

We also encourage you to visit our websites, where you can find our contact information in case you want to get in touch with us directly:

- **Matt Casters:** `www.ibridge.be`
- **Roland Bouman:** `rpbouman.blogspot.com`
- **Jos van Dongen:** `www.tholis.com`

Part I

Getting Started

In This Part

Chapter 1: ETL Primer
Chapter 2: Kettle Concepts
Chapter 3: Installation and Configuration
Chapter 4: An Example ETL Solution—Sakila

www.allitebooks.com

CHAPTER 1

ETL Primer

The introduction of this book described the need for data integration. This chapter provides a starting point to the wonderful world of data integration and explains the differences and similarities among the three main forms of data integration: ETL, ELT, and EII. To fully understand the reasoning behind using a data warehouse and an ETL solution to load and update data, we start by explaining the differences between a transaction and an analysis database.

OLTP versus Data Warehousing

The first question one might ask is how source data systems differ from *business intelligence* (BI) systems (sometimes still called *decision support systems* or DSS). An individual transaction system, often denoted by the acronym *OLTP* (short for OnLine Transaction Processing), needs to be able to very quickly retrieve a single record of information. When multiple records are needed they are usually tied to a single key that has been retrieved before. Think of an order with the accompanying order lines in an order entry system or a personnel record with all salary and bonus information in an HR system. What's more: this data often needs to be updated as well, usually just one record at a time.

The biggest difference between an OLTP and a BI database (the *data warehouse*, or *DWH*) is the amount of data analyzed in a single transaction. Whereas an OLTP handles many concurrent users and queries touching only a single record or limited groups of records at a time, a data warehouse must have the capability to operate on millions of records to

answer a single query. Table 1-1 shows an overview of the major differences between an OLTP and a data warehouse.

Table 1-1: OLTP versus Data Warehouse

CHARACTERISTIC	OLTP	DATA WAREHOUSE
System scope/view	Single business process	Multiple business subjects
Data sources	One	Many
Data model	Static	Dynamic
Dominant query type	Insert/update	Read
Data volume per transaction	Small	Big
Data volume	Small/medium	Large
Data currency	Current timestamp	Seconds to days old
Bulk load/insert/update	No	Yes
Full history available	No	Yes
Response times	< 1 second	< 10 seconds
System availability	24/7	8/5
Typical user	Front office	Staff
Number of users	Large	Small/medium

Of course, it's not as black and white as this table might indicate. The distinctions listed are a rather classic way of looking at the two types of systems. More and more often, business intelligence systems are being used as part of the primary business process. A call center agent might have a screen in front of her with not only customer details such as name and address, but also information about order and payment history retrieved from an *operational data store* (ODS) or a data warehouse. Many CRM systems are already capable of showing a credit or customer score on-the-fly, items that have been pre-calculated in the data warehouse and are available on demand for front office workers. This means that the more the data warehouse is used for operational purposes, the more the same requirements apply as for OLTP systems, especially regarding system availability and data currency.

Probably the most discussed characteristic of the data warehouse is the required response time. Ten years ago, it wasn't a problem when a report query took one or two minutes to retrieve and display its data. Nowadays users expect response times similar to what they're accustomed to when using a search engine. More than ten seconds and users get impatient, start clicking refresh buttons (which will sometimes re-issue the query, making the problem even worse), and eventually avoid using the data warehouse because it's so slow. On the other hand, when the data warehouse is used for data mining purposes, analysts find a response time of several hours totally acceptable, as long as the result to their inquiry is valuable.

What Is ETL?

You know of course that ETL is short for extract, transform, and load; no secrets here. But what exactly do we mean by ETL? A simple definition could be "the set of processes for getting data from OLTP systems into a data warehouse." When we look at the roots of ETL it's probably a viable definition, but for modern ETL solutions it grossly oversimplifies the term. Data is not only coming from OLTP systems but from websites, flat files, e-mail databases, spreadsheets, and personal databases such as Access as well. ETL is not only used to load a single data warehouse but can have many other use cases, like loading data marts, generating spreadsheets, scoring customers using data mining models, or even loading forecasts back into OLTP systems. The main ETL steps, however, can still be grouped into three sections:

1. **Extract:** All processing required to connect to various data sources, extract the data from these data sources, and make the data available to the subsequent processing steps. This may sound trivial but can in fact be one of the main obstacles in getting an ETL solution off the ground.
2. **Transform:** Any function applied to the extracted data between the extraction from sources and loading into targets. These functions can contain (but are not limited to) the following operations:
   - Movement of data
   - Validation of data against data quality rules
   - Modification of the content or structure of the data
   - Integration of the data with data from other sources
   - Calculation of derived or aggregated values based on processed data
3. **Load:** All processing required to load the data in a target system. As we show in Chapter 5, this part of the process consists of a lot more than just bulk loading transformed data into a target table. Parts of the loading process include, for instance, surrogate key management and dimension table management.

The remainder of this section examines how ETL solutions evolved over time and what the main ETL building blocks look like.

The Evolution of ETL Solutions

Data integration needs have existed as long as data has been available in a digital format. In the early computing days, before ETL tools existed, the only way to get data from different sources and integrate it in one way or another was to hand-code scripts in languages such as COBOL, RPG, and later in Perl or PL/SQL. Although this is called the first generation of ETL solutions, it may surprise you that today, about 45 percent of all ETL work is still conducted by using hand-coded programs/scripts. This might have made sense in the days when ETL tools had a six-figure price tag attached to them, but currently there are many open source and other low-cost alternatives available

so there's really no point in hand-coding ETL jobs anymore. The main drawbacks of hand-coding are that it is:

- Error prone
- Slow in terms of development time
- Hard to maintain
- Lacking metadata
- Lacking consistent logging/error handling

The second generation of ETL tools (actually the first if we're talking about "tools" rather than the broader "solutions") tried to overcome these weaknesses by generating the required code based on the design of an ETL flow. In the early 1990s, products such as Prism, Carlton, and ETI emerged but most were acquired later by other ETL vendors. ETI is probably the only independent vendor left from those early days that still offers a code-generating solution. The fact that code generators are listed here as second-generation ETL solutions doesn't necessarily mean they are outdated. It's rather the contrary; code generators are alive and kicking, with Oracle's Warehouse Builder arguably being the most well-known product in this category. The popular open source tool Talend is another example of a code-generation solution.

Code generators have their pros and cons; the biggest disadvantage is that most code generators can work with only a limited set of databases for which they can generate code. Soon after the code generators came into use, a third generation of ETL tools emerged. These were based on an engine where all the data processing took place, and a set of metadata that stored all the connection and transformation rules. Because engines have a generic way of working and all the transformation logic is independent from both the source and the target data stores, engine-based ETL tools, in general, are more versatile than code-generating tools. Kettle is a typical example of an engine-based tool; other familiar names in this area are Informatica Powercenter and SQL Server Information Services.

Both code generators and engine-based tools offer some help in discovering the structure of the underlying data sources and the relationships between them, although some tools are more capable in doing this than others. They also require that a target data model is developed either before or during the design of the data transformation steps. After this design phase, the target schema has to be mapped against the source schema(s). This whole process is still very time consuming, and as a result, a new generation of data warehouse tools emerged that are model driven. MDA tools (for *Model Driven Architecture*) try to automate the data warehouse and data mart design process from the ground up by reading the source data model and generating both the target schema and all required data mappings to populate the target tables. There are only a few such tools on the market, with Kalido and BIReady being the most well known. They are no silver bullets, however; MDA tools still require a skilled data warehouse architect to reap the benefits from them. Although they cannot solve every data integration challenge they can be a huge time (and thus money) saver.

DATA WAREHOUSE VERSUS DATA MART

In this book, the terms *data warehouse* and *data mart* are often used as if they are interchangeable items. They're not, and they differ widely in scope, model, and applicability. A data warehouse is meant to be the single, integrated storehouse of (historical) data that can be used for supporting an organization's decision process. As such, it contains data covering a wide range of topics and business processes, for instance finance, logistics, marketing, and customer support. Often, a data warehouse cannot be accessed directly by end user tools. A data mart, in contrast, is meant for direct access by end users and end user tools, and has a limited specific analytical purpose, for instance Retail Sales or Customer Calls.

ETL Building Blocks

The best way to look at an ETL solution is to view it as a business process. A business process has input, output, and one or more units of work, the process steps. These steps in turn also have inputs and outputs, and perform an operation to transform the input into the output. Think, for example, of a claims department at an insurance company. There's a big sign on the door that says Claims Department, which tells the purpose and main process of the department: handling claims. Within the department, each desk or sub-department might have its own specialty: health insurance claims, car insurance claims, travel insurance claims, and so on. When a claim is received at the office, it is checked to find out to which desk it should be sent. The claims officer can then determine whether all required information to handle the claim is available and if not, send it back with further instructions to the submitter. Each day at 9 a.m. this process of handling claims starts, and it runs until 5 p.m.

This example is a lot like an ETL process: data arrives or is retrieved and a validation step determines what kind of data it is. The data is then sent to a specific transformation that is designed to handle that specific data. When the transformation can process the data, it's delivered to the next transformation or a destination table, and in the case of errors, it is transferred to an error handling routine. Each night at 3 a.m., the job is started by a scheduler and it ends when all data is processed.

You might now have a global feeling of how ETL processes are designed. From the preceding examples you can deduce that there must be some mechanism to control the overall process flow, and other more specific parts of the process that do the actual transformation. The first part is called a *job* in Kettle terminology, and the latter part consists of *transformations*. Jobs are the traffic agents of an ETL solution, and transformations are the basic building blocks. Individual transformations can be chained together in a logical order, just like a business process, to form a job that can be scheduled and executed. A transformation in turn can also consist of several steps. A *step* is the third basic building block of a Kettle solution, and the connection between steps and transformation is formed by *hops*. You'll read a lot more about jobs, transformations, steps,

and hops in the remainder of this book, but these four building blocks enable you to develop any imaginable ETL solution. Chapter 2 provides a more detailed introduction to these four concepts.

ETL, ELT, and EII

The term *data integration* encompasses more than just ETL. With ETL, data is extracted from one or more source systems and, possibly after one or more transformation steps, physically stored in a target environment, usually a data warehouse. To be able to distinguish between ETL and other forms of data integration, we need a way of classifying and describing these other mechanisms.

Figure 1-1 shows a classic example of a data warehouse architecture. In this figure there are multiple source systems, a staging area where data is extracted to, a central warehouse for storing all historical data, and finally data marts that enable end users to work with the data. Between each of these building blocks a data integration process is used, as shown by the ETL blocks.

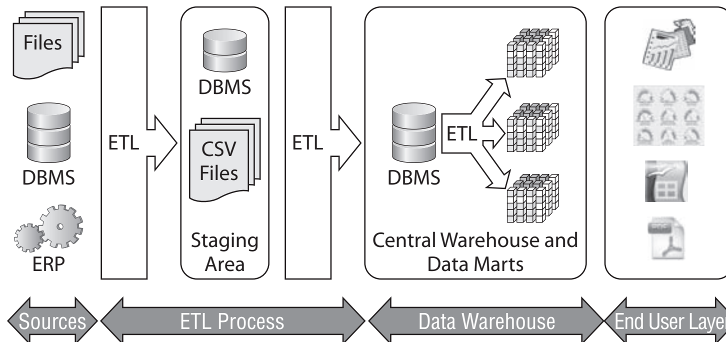


Figure 1-1: Classic data warehouse architecture

This is an architecture that has been used for the past 20 years and has served us well. In fact, many current data warehouse projects still use an architecture similar to the one shown in Figure 1-1. This picture clearly shows that ETL tools are used not only to extract data and load a data warehouse, but also to populate data marts and possibly other databases like an operational data store (not present in the diagram).

Figure 1-1 also shows that there is an intermediate step between the source systems and the data warehouse, called the *staging area*. This part of the overall architecture is merely a drop zone for data; it serves as an intermediate area to get data out of the source systems as quickly as possible. A staging area doesn't necessarily need to be a database system; in many cases, using plain ASCII files to stage data works just as well and is sometimes a faster solution than first inserting the data into a database table.

NON-ETL USE FOR ETL TOOLS

ETL tools are used for more than data warehouse purposes alone. Because they offer a wide range of connectivity and transformation options, another often seen use case is *data migration*. With the help of a tool like Kettle it is fairly easy to connect to database A and migrate all the data to database B. In fact, Kettle has two wizards (Copy Table and Copy Tables) that will handle this for you automatically, including generating the new target tables using the target database SQL syntax.

A third and more complex use case is *data synchronization*, meaning that two (or more) databases are being kept in sync using ETL tools. Although this can be achieved to a certain level, it is not the most common way of using ETL. Usually there are time constraints (changes made in database A need to be available in database B within the shortest achievable amount of time), which make the batch orientation of most ETL tools an unlikely choice for synchronization purposes.

ELT

ELT (short for extract, load, and transform) is a slightly different approach to data integration than ETL. In the case of ELT, the data is first extracted from the source(s), loaded into the target database, and then transformed and integrated into the desired format. All the heavy data processing takes place inside the target database. The advantage of this approach is that in general, a database system is better suited for handling large workloads where hundreds of millions of records need to be integrated. Database systems are also usually optimized for I/O (throughput), which helps to process data faster, too.

There is a big "but" here: In order to benefit from an ELT approach, the ELT tool needs to know how to use the target database platform and the specific SQL dialect being used. This is the reason there aren't a lot of ELT solutions on the market and why a general-purpose ETL tool such as Kettle lacks these capabilities. Nevertheless, most of the traditional closed source ETL vendors augmented their tools with pushdown SQL capabilities, basically resulting in supporting *ETLT* (extract, transform, load, transform) scenarios, where transformation can take place either within the engine (especially for operations not supported by the target database), or after loading inside the database. Leading database vendors such as Microsoft (SQL Server Integration Services) and Oracle (Oracle Warehouse Builder) have a headstart here and have had ETLT capabilities by design because their tools were already tightly integrated with the database. Oracle even bought Sunopsis some time ago, a company that created one of the few specialized ELT solutions on the market (now available as Oracle Data Integrator). Others, like Informatica and Business Objects, have added pushdown SQL capabilities to their products in later releases. An excellent overview of the pros and cons of ETL and ELT can be found in this blog by Dan Linstedt, the inventor of the Data Vault data warehouse modeling technique: `http://www.b-eye-network.com/blogs/linstedt/archives/2006/12/etl_elt_-_chall.php`.

A special product that should be mentioned here is LucidDB. This open source columnar BI database took the ETL and ELT concepts one step further and is capable of handling all the ETL functionality inside the database using extensions to standard ANSI SQL. To do this, LucidDB uses so called *wrappers* around different data sources. After a wrapper is defined for a source (which could be a database, a text file, or even a Web service), the source can be accessed using standard SQL to perform any operation that the SQL language supports. This architecture, of course, makes LucidDB also capable of acting as a lightweight EII solution (Enterprise Information Integration), which we cover in the next section.

EII: Virtual Data Integration

Both ETL and ELT move or copy data physically to another data store, from the OLTP to the data warehouse system. The reasons for using a separate data warehouse and hence, moving the data to that datastore, were explained in the earlier section "OLTP versus Data Warehousing." In more and more cases, however, there is no need to move or copy data. In fact, most users don't even care whether there is an ETL process and a data warehouse complemented with data marts: They just want access to their data! In a way, the data warehouse architecture displayed in Figure 1-1 is like the kitchen of a restaurant. As a customer, I don't really care how my food is prepared—I just want it served in a timely matter and it should taste great. Whatever happens behind those swinging doors is really none of my business. The same applies to a data warehouse: Users don't really care how their data is processed; they just want to access it quickly and easily.

So instead of physically integrating data, it is virtually integrated, making the data accessible in real time when it is needed. This is called *enterprise information integration*, or *EII*; other terms such as *data federation* and *data virtualization* are used as well and have the same meaning. The main advantage of this approach is, of course, the fact that data is always up-to-date. Another advantage is that there is no extra storage layer and no extra data duplication. Some data warehouse environments copy the same data three or four times: once in a staging area, then an operational data store (ODS), the data warehouse itself, and finally the data marts. By using virtual data integration techniques, the data is accessible for an end user as if it were a data mart, but in reality the EII tool takes care of all the translations and transformations in the background.

Although EII sounds like a winning strategy, it does have some drawbacks. Table 1-2 highlights the differences between using a physical and virtual data integration approach.

You can draw some conclusions from Table 1-2. One is that managing large volumes of cleansed, current data using a virtual approach will be challenging, if not impossible. Another conclusion might be that ETL is a tool that typically belongs in the physical integration category, but as you will see in Chapter 22, Pentaho Reporting can be used to invoke Kettle data integration jobs as a data source on an ad-hoc basis, offering some of the advantages of a virtual data warehouse solution combined with all the functionality of a full-fledged ETL tool.

Table 1-2: Virtual versus Physical Data Integration

CHARACTERISTIC	PHYSICAL	VIRTUAL
Data currency	○	●
Query performance/latency	●	⦿
Frequency of access	●	⦿
Diversity of data sources	⦿	●
Diversity of data types	⦿	●
Non-relational data sources	○	●
Transformation and cleansing	●	○
Performance predictability	●	⦿
Multiple interfaces to same data	○	●
Large query/data volume	●	○
Need for history/aggregation	●	○

Legend: ○=Weak, ⦿=Acceptable, ●=Strong

©Mark Madsen, Third Nature, Inc., 2009. All Rights Reserved. Used with Permission.

Data Integration Challenges

Data integration typically poses a number of challenges that need to be addressed and resolved before your solution is up and running. These challenges can be of a political, organizational, functional, or technical nature.

First and foremost, you'll need to find out which data is needed to answer the questions that your organization wants answered and build a solid business case and project plan for delivering that required information. Without a proper business case for starting a business intelligence project, you'll likely fail to get the necessary sponsorship. Technological barriers can be challenging but are in most cases removable; organizational barriers are much harder to take away. Although we won't cover these topics further in this book we just wanted to raise awareness about this important topic.

NOTE **For more information, see Ralph Kimball's *Data Warehouse Lifecycle Toolkit* (2nd edition). Chapter 3 addresses gathering business requirements.**

A good plan and a business case might get you the necessary support to start a project, but they are not enough to deliver successful solutions. For this, a solid methodology is needed as well, and of course a team of bright and experienced people won't hurt either. For many years IT projects were run using a waterfall approach where a project had its initiation phase, followed by design, development, testing, and moving

to production. For business intelligence projects, of which ETL is an important part, this never worked quite well. As you'll see in the following section, a more agile approach fits the typical steps in a BI project much better.

On a more detailed level, you need to face the ETL design challenges, and define how your jobs and transformations will be built, not in a pure technical sense, but in a more functional way. There are many ways in which an ETL tool can be used to solve a specific problem, and no matter which approach is taken, it's mandatory that the same conceptual design is used to tackle similar problems. For instance, if the team decides to stage data to files first, stick to that and don't mix in staging data to a database for some parts of the solution, unless absolutely necessary.

After solving the organizational, project, and design challenges, the first technical issue is finding out where to get the data from, in what format it is available, and what exactly makes up the data you're interested in. Not only might it be a challenge to get access to the data, but connecting to the systems that host the data can be a major issue, too. A lot of the data available in enterprise information systems resides on mainframe computers or other hard-to-access systems such as older proprietary UNIX editions.

Large data volumes are also a challenge. Extracting all the data from the source systems every time you run an ETL job is not feasible in most circumstances. Therefore you need to resolve the issue of identifying what has changed in your source systems to be able to retrieve only the data that has been inserted, updated, or deleted. In some cases, this issue cannot be gracefully resolved and a brute force approach needs to be taken that compares the full source data set to the existing data set in the data warehouse.

Other challenges have to do with the way the data needs to be integrated; suppose there are three different systems where customer data is stored, and the information in these systems is inconsistent or conflicting? Or how do you handle incomplete, inconsistent, or missing data?

Methodology: Agile BI

One of the first challenges in any project is to find a good way to build and deliver the solution, including proper documentation. This holds true for any software package, not only for ETL. Over the years, many project management and software development methodologies have seen the light of day. Maybe you remember the days of the structured analysis and design methodologies, as developed in the 70s by people like Ed Yourdon and Tom DeMarco. These approaches all had a so-called waterfall model in common, meaning that one step in the analysis or design phase needs to be complete before you can move on to the next one. You can find more background information about these methods at `http://en.wikipedia.org/wiki/Structured_Analysis`.

During the 80s and 90s, developers found that these structured, waterfall-based methods weren't always helpful, especially when requirements changed during the project. To cope with these changing requirements, different "agile" development methods emerged, with Scrum arguably being the best-known example. What's so special about agile development? To make that clear, the founders and proponents of agile methodologies came up with the Agile Manifesto, which declares the values of the agile methodology:

- Individuals and interactions over processes and tools

www.allitebooks.com

- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The Agile Manifesto (full text available on `http://agilemanifesto.org`) also contains 12 guiding principles that define what Agile is. In short, it's about:

- Early and frequent delivery of working software
- Welcoming changing requirements
- Business and IT working closely together
- Reliance on self-motivated developers and self-organizing teams
- Frequent, face-to-face conversations to discuss issues and progress
- Keeping it simple: maximizing the amount of work not done

NOTE **There is an abundant amount of information about agile development and the Scrum methodology available online. A good place to start is** `http://en.wikipedia.org/wiki/Scrum_(development)`**.**

Now what has all this to do with business intelligence, and more specifically, ETL? Well, in the case of Pentaho and Kettle: everything! Pentaho has always embraced agile development methods (especially Scrum) to incrementally develop and release new versions of their BI platform and components. Development phases are measured and communicated in Sprints and Milestones, which is also reflected in the download versions you can obtain from the CI (Continuous Integration) repositories. During 2009, Pentaho decided to translate the experience the company had with using agile development methods into an agile BI approach. The intention is not only to support BI developers with a solid methodology, but also to adapt the Pentaho BI suite and all constituent components in such a way that they enforce, enable, and support an agile way of working. The first part of the BI suite that was changed and extended is Kettle, which is the reason we introduced the agile concepts here. You'll see more about the agile capabilities of Kettle later in the book, but as you might already wonder how Kettle supports an agile way of working, here's a look at the basic capabilities Kettle has to offer.

Once you've installed Kettle, it will take only a couple of minutes before you have connected to a data source, read some data, added a transformation, and delivered the data to a destination table. Because Kettle is an engine-based solution, it automatically takes care of a lot of things for you, which helps speed up the process. Kettle also contains a vast (and perhaps at first glance overwhelming) number of standard components and transformation steps. These are prebuilt code blocks that also help in minimizing the development effort and maximizing the speed of solution delivery. Changes in data fields or data types are automatically propagated to subsequent steps in the process, and Kettle can also generate the change scripts needed to alter the final destination table. The integrated modeling and ad-hoc visualization tools enable you to directly show the results of your work to the end user and play with the data in an iterative

way. Developers and business users can therefore work closely together using the same toolset. Deviations from plan or from the user's expectations can be taken into account immediately and the jobs and transformations can be changed accordingly.

Because Kettle is tightly linked to the Agile BI initiative by Pentaho, it might be worthwhile to read up on the methodology and how Kettle supports it on the Agile BI wiki. You can find this info at `http://wiki.pentaho.com/display/AGILEBI/Welcome+to+Agile+Business+Intelligence`.

ETL Design

Even when (or perhaps, especially when) an agile approach is taken, your ETL process needs to be designed in one way or another. Because an ETL solution in many respects resembles a workflow or business process, it might make sense to use a flowchart drawing tool to create a high-level design before you start building. Most users will be familiar with flowchart diagrams and can comment and help make your design better. On a more detailed level, it is important to define which parts of the solution are to be reusable and which are not. For instance, creating a date dimension is usually a one-time effort within a data warehouse project. So for an individual project it makes sense to not spend too much time developing a flexible and database-independent solution and just develop a standard script for this. On the other hand, when you're a consultant working for many different customers, it absolutely makes sense to have a generic date dimension generator in your toolbox.

From this discussion, it's easy to see what the most important question is when you start building a data transformation: Should it be reusable in other parts of the solution or not? Depending on the answer, you might spend some extra time in making the transformation generic, for instance by adding extra parameters that enable you to choose a type of database, a date/time format, or other things that change between different solutions.

Data Acquisition

As explained earlier, getting access to and retrieving data from source systems is the first challenge you encounter when an ETL project is started. Don't automatically assume that this is only a technical problem; in many cases not being able to access data directly is caused by internal politics or guidelines. ERP vendors also try to make it difficult or even impossible to access the data in their systems directly. The widely used SAP/R3 system, for instance, has specific clauses in the software license that prohibit direct connections to the underlying database other than by the means provided by SAP. Most financial institutions that run their mission-critical systems on a mainframe also won't let you access these systems directly so you're dependent on data feeds delivered by FTP or via a web service. This isn't necessarily a disadvantage; the SAP system consists of more than 70,000 tables so finding the ones with the data you're interested in might be a very time-consuming task. For situations like this, you need tools that are able to interpret the ERP metadata, which displays a business view of the data. Fortunately, Kettle not only contains a standard data input step for acquiring data from SAP/R3, but also for getting data out of Salesforce.com, arguably the most used and advanced online CRM application. For other standard ERP

and CRM solutions such as SugarCRM, OpenERP, ADempiere, or Peoplesoft, you might have to revert to third-party solutions or build your own input step. The capability to read data from a mainframe directly is unfortunately not available so in those cases it's best to have this data delivered from the system in a readable format such as ASCII or UniCode (older mainframe systems still use EBCDIC). More information about accessing ancient Cobol systems and the specific file format challenges involved can be found at `http://jymengant.ifrance.com/jymengant/jurassicfaq.html`.

Beware of Spreadsheets

A major challenge in the field of data acquisition has to do with the way and the format in which the data is delivered. A notorious troublemaker in this area is Excel, so the best advice we can give is to just never accept a data delivery in Excel, unless you can be sure it's system-generated and that it's created on the same machine by a process that's owned by the same user. Excel data problems occur frequently when different internationalization settings are used, causing dates and numeric fields to change from one session to another. As a result, your carefully designed and tested transformation will fail, or at least will generate incorrect results (which is basically the same but harder to track). Most problems, however, are caused by users who will tell you they haven't changed anything but who did, perhaps even unknowingly.

Design for Failure

Even if access to the data isn't a problem and the solution you've built looks rock solid, you always need to make sure that the data source is available before you kick off a process. One basic design principle is that your ETL job needs to be able to fail gracefully when a data availability test fails. Kettle contains many features to do this. You can:

- Test a repository connection.
- Ping a host to check whether it's available.
- Wait for a SQL command to return success/failure based on a row count condition.
- Check for empty folders.
- Check for the existence of a file, table, or column.
- Compare files or folders.
- Set a timeout on FTP and SSH connections.
- Create failure/success outputs on every available job step.

It's also a good idea to add error handling at the job and transformation level. When loading a data warehouse, dependencies often exist between tables. A good example is that a fact table cannot be loaded before all dimension loads have completed. When one of the dimension loads fails, the complete job should fail, too. A good design then lets you correct the errors and enables you to restart the job where only the failed and not-yet-run parts will execute.

Change Data Capture

The first step in an ETL process is the extraction of data from various source systems and passing the data to the next step in the process. A best practice here is the intermediate storage of the extracted data in staging tables or files to make restarts possible without the need of retrieving all data again. This seems like a trivial task, and in the case of initially loading a data warehouse it usually is, apart from challenges incurred from data volumes and slow network connections. But after the initial load, you don't want to repeat the process of completely extracting all data again. This wouldn't be of much use anyway because you already have an almost complete set of data, and it only needs to be refreshed to reflect the current status. All you're interested in is what has changed since the last data load, so you need to identify which records have been inserted, modified, or even deleted. The process of identifying these changes and only retrieving records that are different from what you already loaded in the data warehouse is called *Change Data Capture* or *CDC*.

Basically, there are two main categories of CDC processes, *intrusive* and *non-intrusive*. By intrusive, we mean that a CDC operation has a possible performance impact on the system the data is retrieved from. It is fair to say that any operation that requires executing SQL statements in one form or another is an intrusive technique. The bad news is that most available methods to capture changed data are intrusive, leaving only one non-intrusive option. CDC is covered in depth in Chapter 6.

Data Quality

Much of what is said in the previous section applies here as well: you have to assume that there are quality problems in your data and therefore need to design your transformations to handle these problems. In fact, this isn't entirely true: data quality problems need to be resolved in the source systems, not in the ETL process. However, fixing data quality issues before starting a data warehouse project is a luxury that not many organizations can afford. There's always a pressing need to deliver a solution quickly, and even if serious data quality problems are discovered during the project, they are usually dealt with later or not at all. Hence knowing what's wrong with your data and knowing what to do about it are essential parts of the ETL developer's job description.

Two categories of tools are available to deal with data quality problems. First, you'll need to define a baseline by profiling the data and investigating how good the quality actually is, using a data profiling tool. The purpose of this exercise is twofold: to communicate the results of this exercise back to the data owner (hopefully a business manager with the authority to do something about it), and to serve as input for the data validation steps in the ETL jobs. Second, there are data quality tools that constantly monitor and augment the data based on business and quality rules. In Kettle, the Data Validation step serves as a built-in data quality tool.

Data Profiling

One of the first things to do when starting an ETL project is profile the source data. *Profiling* will tell you how much data there is and what it looks like, both technically

and statistically. The most common form of profiling is *column profiling*, where for each column in a table the appropriate statistics are created. Depending on the data type this will give you insight into things like the following:

- Number of NULL or empty values
- Number of distinct values
- Minimum, maximum, and average value (numeric fields)
- Minimum, maximum, and average length (string fields)
- Patterns (for example, ###-###-#### for phone numbers)
- Data distribution

Although most of these operations can be performed using Kettle transformations or just plain SQL, it's better to use a specialized tool such as Data Cleaner from eobjects. Chapter 6 covers data profiling in more detail.

WARNING **Data profiling will only get you so far; logical and/or cross-system quality issues cannot be detected by most data profiling tools, and in order to detect them, a global business glossary and metadata system needs to be in place first. These systems are still very rare.**

Data Validation

Profiling is meant for reporting and setting a baseline, while validation is part of the regular ETL jobs. A simple example is as follows: Some column in a source system can technically contain `NULL` values, but there is a business rule stating that this is a required field. Profiling revealed that there are several records with a `NULL` value in this column. To cope with this situation, a validation step is needed that contains the rule `NOT NULL` for this column, and when the column does contain a `NULL` value, an alternative action should be started. This could be omitting the record and writing it to an error table, replacing the `NULL` value with a default value such as `Unknown`, flagging the record as unreliable, or any other action that is deemed necessary.

ETL Tool Requirements

While this book is specifically about Kettle, it's useful to have an overview of the required features and functionality of an ETL tool in general. This will enable you to decide whether Kettle is the right tool for the job at hand. Each of the following sections first describes the requirement in general and then explains how Kettle provides the required functionality or feature.

Connectivity

Any ETL tool should provide connectivity to a wide range of source systems and data formats. For the most common relational database systems, a native connector (such

as OCI for Oracle) should be available. At a minimum, the ETL should be able to do the following:

- Connect to and get data from the most common relational database systems including Oracle, MS SQL Server, IBM DB/2, Ingres, MySQL, or PostgreSQL.
- Read data from ASCII files in a delimited or fixed format.
- Read data from XML files (XML is the lingua franca of data interchange).
- Read data from popular Office formats such as Access databases or Excel spreadsheets.
- Get files from external sites using FTP, SFTP, or SSH (preferably without scripting).

In addition to this, there might be the need to read data using a web service, or to read an RSS feed. In case you need to get data from an ERP system such as Oracle E-Business Suite, SAP/R3, PeopleSoft, or JD/Edwards, the ETL tool should provide connectivity options for these systems as well.

Out of the box, Kettle has input steps for Salesforce.com and SAP/R3. For other ERP or financial systems, an alternative or additional solution might be required. Of course it's always possible to have these systems export a data set to an ASCII file and use that as a source.

Platform Independence

An ETL tool should be able to run on any platform and even a combination of different platforms. Maybe a 32-bit operating system works for the initial development phase, but when data volumes increase and available batch windows decrease, a more powerful solution is required. In other cases, development takes place on a Windows or Mac development PC, but production jobs run on a Linux cluster. You shouldn't have to take special measures to accommodate for this in your ETL solution.

Scalability

Scalability is a big issue; data volumes increase year after year and your systems needs to be able to handle this. Three options should be available for processing large amounts of data:

- **Parallelism:** Enables a transformation to run many streams in parallel, thus utilizing modern multi-core hardware architectures
- **Partitioning:** Enables the ETL tool to take advantage of specific partitioning schemes to distribute the data over the parallel streams
- **Clustering:** Enables the ETL process to divide the workload over more than one machine

This last option especially can be cost prohibitive with proprietary ETL solutions that are licensed per server or per CPU.

Kettle, being a Java-based solution, runs on any computer that has a Java Virtual Machine installed. Any step in a transformation can be started multiple times in parallel to speed up processing. Kettle will then determine how the data is distributed over the different streams. For better control, a partitioning scheme can be used to make sure that each parallel stream contains data with the same characteristics. This resembles how database partitioning works, but Kettle has no specific facilities to work with database partitions. (The benefit of having such a capability is debatable because the database itself is probably better capable of distributing data over the partitions than an ETL tool would be.)

The most advanced scalability feature arguably is the clustering option, which lets Kettle spread the workload over as many machines as deemed necessary. Part IV of this book covers all these scalability features in depth, but a good source to whet your appetite is the white paper written by one of the technical reviewers of this book, Nicholas Goodman of Bayon Technologies. It can be found at `http://www.bayontechnologies.com/bt/ourwork/pdi_scale_out_whitepaper.php`.

Design Flexibility

An ETL tool should provide a developer the freedom to use any desirable flow design and should not limit people's creativity or design requirements by offering only a fixed way of working. ETL tools can be classified as either process- or map-based. A map-based tool offers a fixed set of steps between source and target data, thus severely limiting the freedom to design jobs. Map-based tools are often easy to learn and get you started very quickly, but for more complex tasks, a process-based tool is most likely the better choice. With a process-based tool like Kettle, you can always add additional steps or transformations if needed because of changes in the data or business requirements.

Reuse

Being able to reuse existing parts of your ETL solution is also an indispensable feature. An easy way of doing this is to copy and paste or duplicate existing transformation steps, but that's not really reuse. The term *reuse* refers to the capability to define a step or transformation once and call the same component from different places. Within Kettle this is achieved by the Mapping step, which lets you reuse existing transformations over and over as subcomponents in other transformations. Transformations themselves can be used multiple times in multiple jobs, and the same applies to jobs which can be reused as subjobs in other jobs as well.

Extensibility

There isn't a single ETL tool in the world that offers everything that's needed for every imaginable data transformation task, not even Kettle. This means that it must be possible to extend the basic functionality of the tool in some way or another. Almost all ETL tools offer some kind of scripting option to programmatically perform complex

tasks not available in the program itself. Only a few ETL tools, however, offer the option to add standard components yourself by offering an API or other means to extend the toolset. In between these options is a third way that lets you define functions that can be written using a script language and called from other transformations or scripts.

With Kettle, you get it all. Scripting is provided by the Java Script step, and by saving this as a transformation it can be reused in a mapping, resulting in a standard reusable function. In fact, any transformation can be reused in a mapping so creating standard components this way isn't limited to scripting alone. And Kettle is, of course, built with extensibility in mind, offering a plugin-enabled platform. The plugin architecture makes it possible for third parties to develop additional components for the Kettle platform. Several examples of these additional plugins are covered in this book, but it's important to note that all components you find in Kettle, even the ones that are available by default, are actually plugins. The only difference between built-in and third-party plugins could be the available support: If you buy a third-party plugin (for instance a SugarCRM connector), support is provided by the third party, not by Pentaho.

Data Transformations

A good deal of the work involved with an ETL project has something to do with transforming data. Between acquisition and delivery, the data needs to be validated, joined, split, combined, transposed, sorted, merged, cloned, de-duplicated, filtered, deleted, replaced, and whatnot. It's hard to tell what the minimum set of available transformations should be because data transformation requirements differ greatly between organizations, projects, and solutions. Nevertheless, there seems to be a common denominator of basic functions that most of the leading ETL tools (including Kettle) offer:

- Slowly Changing Dimension support
- Lookup values
- Pivot and unpivot
- Conditional split
- Sort, merge, and join
- Aggregate

The only difference between tools is the way these transformations need to be defined. Some tools, for instance, offer a standard SCD (Slowly Changing Dimension) transformation in a single step, while others generate the needed transformations with a wizard. Even Kettle doesn't cover all transformation requirements out of the box. A good example of a missing component is a hierarchy flattener. By *hierarchy* we mean a single table that refers to itself, for instance an employee table where each employee record has an employee ID and a manager ID. The manager ID in the employee record points to an employee ID of another employee who is the manager. Oracle has had a standard "connect by prior" function to cope with this for ages and some ETL tools have a similar feature; in Kettle you'd have to manually handle this issue.

Testing and Debugging

This requirement hardly needs further explanation. Even though an ETL solution is not (at least, we hope not) written in a program language such as Java or C++, it can be looked at as such. This means that what applies to application programming also applies to ETL development: Testing should be an integral part of the project. To be able to test, you need test cases that cover any possible (or at least, the most likely) scenario in a "what if" kind of way. The following are some examples of such scenarios:

- What if we don't get the data delivered on time?
- What if the process breaks halfway through the transformation?
- What if the data in column XYZ contains NULL values?
- What if the total number of rows transformed doesn't match the total number of rows extracted?
- What if the result of this calculation doesn't match the total value retrieved from another system?

Again, the message here is to design for failure. Don't expect that things will work; just assume that things will fail at some point. When designing tests, it's important to differentiate between *black box testing* (also known as *functional testing*) and *white box* testing. In case of the former, the ETL solution is considered a black box where the inner workings are not known to the tester. The only known variables are the inputs and the expected outputs. White box testing (also known as *structural testing*), on the other hand, specifically requires that the tester knows the inner workings of the solution and develops tests to check whether specific transformations behave as expected. Both methods have their advantages and disadvantages, which are covered in Chapter 11.

Debugging is an implicit part of white box testing and enables a developer or tester to run a program step by step to investigate what exactly goes wrong at what point. Not all ETL tools offer extensive debugging functionality where you can step through a transformation row by row, inspecting individual rows and variable allocations. Kettle offers extensive debugging features for both jobs and transformations, as covered in Chapter 11.

Lineage and Impact Analysis

A mandatory feature of any ETL tool is the ability to read the metadata to extract information about the flow of data through the different transformations. Data lineage and impact analysis are two related features that are based on the ability to read this metadata. Lineage is a backward-looking mechanism that will show for any data item where it came from and which transformations were applied to it. This would include calculations and new mappings, such as when price and quantity are used as input fields to calculate revenue. Even if the field's `price` and `quantity` are omitted from further processing, the data lineage function should reveal that the field `revenue` is actually based on `price` and `quantity`.

Impact analysis works the other way around: Based on a source field, the impact on the subsequent transformations and ultimately, destination tables is revealed. You can find in-depth coverage of these subjects in Chapter 14.

Logging and Auditing

The data in the data warehouse needs to be reliable and trustworthy because that's one of the purposes of a data warehouse: provide an organization with a reliable source of information. To guarantee this trustworthiness and have a system of record for all data transformations, the ETL tool should provide facilities for logging and auditing. Logging takes care of recording all the steps that are executed when an ETL job is run, including the exact start and end timestamps for every step. Auditing facilities create a complete trace of the actions performed on the data, including number of rows read, number of rows transformed, and number of rows written. This is a topic where Kettle actually leads the market, as you will see in Chapters 12 and 14.

Summary

This chapter introduced ETL and its history, and explained why data integration is needed. The basic building blocks of a Kettle solution were introduced briefly to give you a feeling for what will be covered in the rest of the book. We also explained the difference and similarities between ETL, ELT, and EII and showed the advantages of each method.

We presented the major challenges you might face when developing ETL solutions:

- Getting business sponsorship and creating a business case
- Choosing a good methodology to guide your work
- Designing ETL solutions
- Data acquisition and the problem with spreadsheets
- Handling data quality issues using profiling and validation

Finally, we highlighted the general requirements of an ETL tool and briefly described how Kettle meets the requirements for the following:

- Connectivity
- Platform independence and scalability
- Design flexibility and component reuse
- Extensibility
- Data transformations
- Testing and debugging
- Lineage and impact analysis
- Logging and auditing

All the topics introduced in this chapter are covered extensively in the rest of this book.

www.allitebooks.com

CHAPTER 2

Kettle Concepts

In this chapter we cover the various concepts behind Kettle. We take a look at the general design principles and describe the various data integration building blocks. First we show you how row level data integration is performed using transformations. Then we explain how you can handle basic workflow using jobs.

You will learn about the following Kettle concepts:

- Database connections
- Tools and utilities
- Repositories
- Virtual File Systems
- Parameters and variables
- Visual programming

Design Principles

Let's start with a look at some of the core design principles that have been put in place since the very beginning of Kettle's development. Previous negative experiences with other tools and frameworks obviously colored the decisions that have been taken. However, it's especially interesting to look at the positive things that were retained from these experiences.

- **Ease of development:** It's clear that as a data warehouse and ETL developer, you want to spend time on the creation of a business intelligence solution. Every hour you spend on the installation of software is wasted. The same principle applies to the configuration as well. For example, when Kettle came on the market, pretty much every Java-based tool that existed forced the user to explicitly specify the Java driver class name and JDBC URL just to create a database connection. This is not the sort of problem that can't be overcome with a few Internet searches but it is something that draws attention away from the real issues. Because of that, Kettle has always tried to steer clear of these types of problems.
- **Avoiding the need for custom program code:** In general, an ETL tool needs to make simple things simple and hard things possible. That means that such a tool needs to provide standard building blocks to perform those tasks and operations that are repeatedly required by the ETL developer. There is no barrier to programming something in Java or even JavaScript if it gets the job done. However, every line of code adds complexity and a maintenance cost to your project. It makes a lot of sense not to have to deal with it.
- **All functionality is available in the user interface:** There are very few exceptions to this golden principle. (The `kettle.properties` and `shared.xml` file in the Kettle home directory are the exceptions.) If you don't expose all functionality through a user interface, you are actually wasting the time of both the developer as well as the end user. Expert ETL users still need to learn the hidden features behind the engine.

  In Kettle, ETL metadata can be expressed in the form of XML, via a repository, or by using the Java API. One hundred percent of the features in these forms of ETL metadata can be edited through the graphical user interface.
- **No naming limitation:** ETL solutions are full of names: database connections, transformations and their steps, data fields, jobs, and so on all need to have a proper name. It's no fun creating ETL functionality if you have to worry about any naming restrictions, (such as length and choice of characters) imposed by your ETL tool. Rather, the ETL tool needs to be clever enough to deal with whatever identifier the ETL developer sees fit. This in turn allows the ETL solution to be as self-descriptive as possible, partly reducing the need for extra documentation, and allows for a further reduction in the ever-present maintenance cost of a project.
- **Transparency:** Any ETL tool that allows you to describe how a certain piece of work is done suffers from a lack of transparency. After all, if you would write the same functionality yourself, you would know exactly what is going on or you could at least figure it out. Allowing people to see what is going on in the various parts of a defined ETL workload is crucial. This, in turn, will speed up development and lower maintenance costs.

  The various parts of an ETL workload should not have a direct influence on one another. They should also pass data in the same order as defined. This principle of data isolation has a big impact on transparency that can only be appreciated by people who worked with ETL tools that didn't enforce this principle.

- **Flexible data paths:** For an ETL developer, creativity is extremely important. Creativity allows you to not only enjoy your work but also find the quickest path to a certain ETL solution. Kettle was designed from the ground up to be as flexible as possible with respect to the data paths that can be put in place. Distributing or copying data among various targets such as text files and relational databases should be possible. The inverse, merging data from various data sources, should also be very simple and part of the core engine.
- **Only map impacted fields:** While some find it visually pleasing to see hundreds of arrows map input and output fields in various ETL tools, it is also a maintenance nightmare in a lot of cases. Every building block in the ETL workload adds to the maintenance cost because fields are added and changed all the time during the development of any ETL solution.

  An important core principle of Kettle is that all fields that are not specified are automatically passed on the next building block in the ETL workload. This massively reduces the maintenance cost. It means, for example, that source fields can be added to the input and they will automatically show up in the output unless you take measures to prevent this.

The Building Blocks of Kettle Design

Every data integration tool uses different names to identify the various parts of the tool, its underlying concepts, and the things you can build with it, and Kettle is no exception. This section introduces and explains some of the Kettle specific terminology. After reading this section you will not only have an understanding of how data is transformed on a row level in a transformation and how workflow is handled with jobs, you will also learn about details like data types and data conversion options.

Transformations

A *transformation* is the workhorse of your ETL solution. It handles the manipulation of rows or data in the broadest possible meaning of the extraction, transformation, and loading acronym. It consists of one or more *steps* that perform core ETL work such as reading data from files, filtering out rows, data cleansing, or loading data into a database.

The steps in a transformation are connected by transformation *hops*. The hops define a one-way channel that allows data to flow between the steps that are connected by the hop. In Kettle, the unit of data is the row, and a data flow is the movement of rows from one step to another step. Another word for such a data flow is a *record stream*.

Figure 2-1 shows an example of a transformation in which data is read from a database table and written to a text file.

In addition to steps and hops, transformations can also contain *notes*. Notes are little boxes that can be placed anywhere in a transformation and can contain arbitrary text. Notes are intended to allow the transformation to be documented.

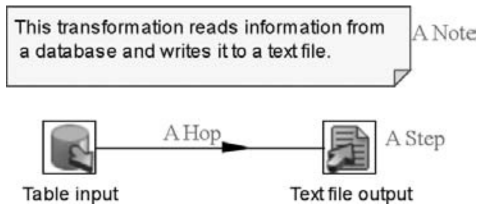


Figure 2-1: A simple transformation example

Steps

A *step* is a core building block in a transformation. It is graphically represented in the form of an icon; Figure 2-1 shows two steps, "Table input" and "Text file output." Here are some of the key characteristics of a step:

- A step needs to have a name that is unique in a single transformation.
- Virtually every step is capable of reading as well as writing rows of data (the single exception is the Generate Rows step, which only writes data).
- Steps write data to one or more *outgoing hops*. To the step that is connected at the other end of the hop, that hop is an *incoming hop*. Steps read data arriving through the incoming hops.
- Most steps can have multiple outgoing hops. A step can be configured to either *distribute* or *copy* data to its outgoing hops. When distributing data, the step alternates between all outgoing hops for each outbound row (this is known as a *round robin*). When copying data, each row is sent to all outgoing hops.
- When running a transformation, one or more copies of each step are started, each running in its own thread. During the run, all step copies run simultaneously, with rows of data constantly flowing through their connecting hops.

Beyond these standard capabilities, each step obviously has a distinct functionality that is represented by the step type. For example, in Figure 2-1, the "Table input" step executes an SQL query to read data from a relational database, and writes the data as rows to its outgoing hop(s), and the "Text file output" step reads rows from its incoming hop(s), and writes them to a text file.

Transformation Hops

A *hop*, represented by an arrow between two steps, defines the data path between the steps. The hop also represents a row buffer called a *row set* between two steps. (The size of the row sets can be defined in the transformation settings.) When a row set is full, the step that writes rows halts if there is room. When a row set is empty, the step doing the reading will wait a bit until rows are available again.

NOTE While creating new hops, please remember that loops are not allowed in transformations. That is because a transformation heavily depends on the previous steps to determine the field values that are passed from one step to another.

Parallelism

The simple rules enforced by the hops allow steps to be executed in a parallel nature in separate threads. That is because rows are forced through the transformation step network, causing a maximum amount of parallelism. The rules also allow data to be processed in a streaming fashion with minimal memory consumption. In data warehousing, you are often dealing with massive amounts of data so this is a core requirement for any serious ETL tool.

As far as Kettle is concerned, it is not possible to define an order of execution, and it is not possible or necessary to identify any start or end to a transformation. This is because all steps are executed in parallel: when a transformation is started, all its steps are started and keep reading rows from their incoming hops and pushing out rows to their outgoing hops until there are no more rows left, terminating the step's run. The transformation as a whole stops after the last step has terminated. That said, functionally a transformation almost always does have a definite start and an end. For example, the transformation shown in Figure 2-1 "starts" at the "Table input" step (because that step generates rows) and "ends" at the "Text file output" step (because that step writes the rows to file and does not lead them into another subsequent step for further processing).

The remarks made earlier about the (im)possibility of identifing a transformation's start and end may seem paradoxical or even contradictory. In reality, it isn't that complicated—it's just a matter of perspective. Although you may envision an individual row flowing through the transformation, visiting subsequent steps and thus following a definitive path from start to end, many rows are flowing through the transformation during the run. While the transformation is running, all steps are working simultaneously in such a way that it is impossible to pinpoint to which one step the transformation has progressed.

If you need to perform tasks in a specific order, refer to the "Jobs" section later in this chapter.

Rows of Data

The data that passes from step to step over a hop comes in the form of a *row* of data. A row is a collection of zero or more *fields* that can contain the data in any of the following data types:

- `String`: Any type of character data without any particular limit.
- `Number`: A double precision floating point number.
- `Integer`: A signed long integer (64-bit).

- BigNumber: A number with arbitrary (unlimited) precision.
- Date: A date-time value with millisecond precision.
- Boolean: A Boolean value can contain true or false.
- Binary*:* Binary fields can contain images, sounds, videos, and other types of binary data.

Each step is capable of describing the rows that are being put out. This row description is also called *row metadata*. It contains these pieces of information:

- **Name:** The name of the field should be unique in a row.
- **Data type:** The data type of the field.
- **Length:** The length of a String, or number of a BigNumber data type.
- **Precision:** The decimal precision of a number of a BigNumber data type.
- **Mask:** The representation format (or conversion mask). This will come into play if you convert numeric (Number, Integer, BigNumber) or Date data types to String. This happens, for example, during data preview in the user interface or during serialization to text or XML.
- **Decimal:** The decimal symbol in a number. This symbol is culturally defined and is typically either a dot (.) or a comma (,).
- **Group:** The grouping symbol. This symbol is also culturally defined and is typically either a comma (,), a dot (.), or a single quotation mark (').
- **Step origin:** Kettle keeps track of the origin of a field in the row meta-data. This allows you to quickly identify at which step in the transformation the field was last modified or.

Here are a few data type rules to remember when designing your transformations:

- All the rows in a row set always need to have the same layout or structure. This means that when you lead outgoing hops from several different steps to one receiving step, all layout of the rows of each of these hops needs to have the same fields with the same data types, and in the same order.
- Beyond the data type and name, field metadata is not enforced during the execution of a transformation. This means that a String is not automatically cut to the specified length and that floating point numbers are not rounded to the specified precision. This functionality is explicitly available in a few steps.
- By default, empty Strings ("") are considered to be the equivalent of NULL (empty).

NOTE The behavior that empty strings and NULL are considered equivalent can be changed by setting the KETTLE_EMPTY_STRING_DIFFERS_FROM_NULL variable. Further details can be found in Appendix C.

Data Conversion

Data conversion takes place either explicitly in a step like Select Values, where you can change the data type of a field, or implicitly, for example when you are storing numeric data in a `VARCHAR` column in a relational database. Both types of data conversion are handled in the exact same way by using a combination of the data and the description of the data.

Date to String Conversion

The internal `Date` representation contains all the information you need to represent any date/time with millisecond precision. To convert between the `String` and `Date` data types you only need to specify a conversion mask. For information on `Date` and `Time` formats, see from the table under "Date and Time Patterns" in the Sun Java API documentation, located at `http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html`. The mask can contain any of the Letter codes shown in that table for representation purposes. Other characters can be included if they are put between single quotes.

For example, Table 2-1 shows a few popular examples of date conversion masks for December 6, 2009 at 21 hours, 6 minutes and 54.321 seconds.

Table 2-1: Date Conversion Examples

CONVERSION MASK (FORMAT)	RESULT
yyyy/MM/dd'T'HH:mm:ss.SSS	2009/12/06T21:06:54.321
h:mm a	9:06 PM
HH:mm:ss	21:06:54
M-d-yy	12-6-09

Numeric to String Conversion

Numeric data (`Number`, `Integer`, and `BigNumber`) is converted to and from `String` using these field metadata components:

- Conversion mask
- Decimal symbol
- Grouping symbol
- Currency symbol

The numeric conversion mask determines how a numeric value is represented in a textual format. It has no bearing on the actual precision or rounding of the numeric data itself. You can find all the allowed symbols and formatting rules in the Java API documentation at `http://java.sun.com/j2se/1.4.2/docs/api/java/text/DecimalFormat.html`.

Table 2-2 shows a few popular examples of numeric conversion masks.

Table 2-2: A Few Numeric Conversion Mask Examples

VALUE	CONVERSION MASK	DECIMAL SYMBOL	GROUPING SYMBOL	RESULT
1234.5678	#,###.##	.	,	1,234.57
1234.5678	000,000.00000	,	.	001.234,56780
-1.9	#.00;-#.00	.	,	-1.9
1.9	#.00;-#.00	.	,	1.9
12	00000;-00000			00012

Other Conversions

Table 2-3 provides a short list of other conversions that occur between data types.

Table 2-3: Other Data Type Conversions

FROM	TO	DESCRIPTION
Boolean	String	This is converted to Y or N unless the length is 3 or higher. In that case, the result is true or false.
String	Boolean	A case-insensitive comparison is made and Y, True, Yes, and 1 all convert to true. Any other String is converted to false.
Integer Date	Date Integer	The Integer long value is considered to be the number of milliseconds that passed since January 1, 1970, 00:00:00 GMT. For example, September 12th 2010 at noon is converted to Integer 1284112800000 and vice-versa.

Jobs

In most ETL projects, you need to perform all sorts of maintenance tasks. For example, you want to define what needs to be done in case something goes wrong and how files need to be transferred; you want to verify if database tables exist, and so on. It's also important that these tasks be performed in a certain order. Because transformations execute all steps in parallel, you need a *job* to handle this.

A job consists of one or more *job entries* that are executed in a certain order. The order of execution is determined by the *job hops* between job entries as well as the result of the execution itself. Figure 2-2 shows a typical job that handles the loading of a data warehouse.

Like a transformation, a job can contain notes for documentation purposes.

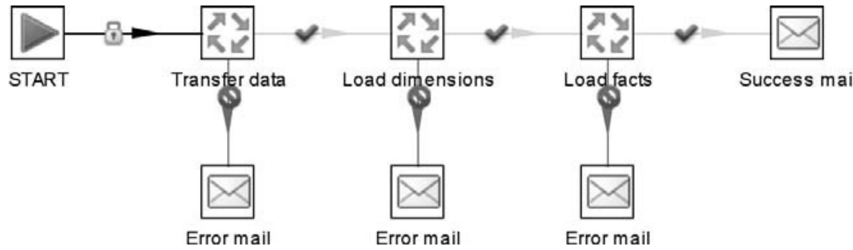


Figure 2-2: A typical job that updates a data warehouse

Job Entries

A *job entry* is a core building block of a job. Like a step, it is also graphically represented in the form of an icon. However, if you look a bit closer, you see that job entries differ in a number of ways:

- While the name of a new job entry needs to be unique, it is possible to create so-called *shadow copies* of a job entry. This will allow you to place the same job entry in a job on multiple locations. The shadow copy is backed by the same information, so that if you edit one copy, you edit them all.
- A job entry passes a result object between job entries. While this result object can contain rows of data, they are not passed in a streaming fashion. Rather, after the job entry finishes, the rows are transferred all at once to the subsequent job entry.
- All job entries are executed in a certain sequence by default. Only in special cases are they executed in parallel.

Because a job executes job entries sequentially, you must define a starting point. This starting point comes in the form of the special job entry called Start. As a consequence, you can only put one Start entry in any one job.

Job Hops

Job hops are used in a job to define an execution path between job entries. This is done in the form of the link between two job entries as well as a result evaluation type. This evaluation type can be any of these:

- **Unconditional:** This means that the next job entry is executed no matter what happened in the previous one. This evaluation type is indicated by a lock icon over a black hop arrow as shown in Figure 2-2.
- **Follow when result is true:** This job hop path is followed when the result of the previous job entry execution was true. This typically means that it ran without

a problem. This type is indicated with a green success icon drawn over a green hop arrow.

- **Follow when result is false:** This job hop path is followed when the result of the previous job entry execution was false, or unsuccessful. This is indicated by a red stop icon drawn over a red hop arrow.

The evaluation type can be set by using the hop's right-click menu or by cycling through the options by clicking on the small hop icons.

Multiple Paths and Backtracking

Job entries are executed using a so called *backtracking* algorithm. The path itself is determined by the actual outcome (success or failure) of the job entries. However, a back-tracking algorithm means that a path of job entries is always followed until the very end before a next possibility is considered.

Consider the example shown in Figure 2-3.

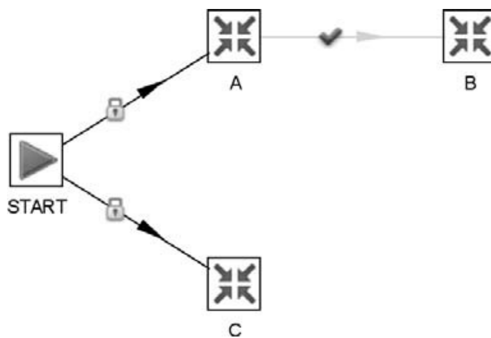


Figure 2-3: Serial execution of multi-paths using back-tracking

In the situation shown in Figure 2-3, job entries A, B, and C are executed in sequence:

- First the "Start" job entry searches all next job entries and finds "A" and "C".
- "A" is executed.
- "A" then searches all next job entries and finds C.
- "C" is executed.
- "C" searches all the next job entries but comes up blank.
- "A" has no further job entries to execute.

- "Start" has one more job entry to execute: "C".
- "C" is executed.
- "C" searches all next job entries but comes up blank.
- "Start" has no more job entries to execute.
- The job finishes.

However, because no order is defined, it could have been CAB as well.
The back-tracking nature of a job is important for two main reasons:

- The result of a job itself (for nesting purposes) is taken from the last job entry that was executed. Because the execution order of the job entries could be ABC or CAB we have no guarantee that the result of job entry C is taken. It could just as well be A.
- In cases where you create loops (which is allowed in a job) you will be putting a burden on the application stack as all the job entries and their results will be kept in memory for evaluation. See Chapter 15 for more information on this topic.

Parallel Execution

Sometimes it is necessary to execute job entries or entire jobs in parallel. This is possible, too. A job entry can be told to execute the next job entries in parallel, as shown in Figure 2-4.

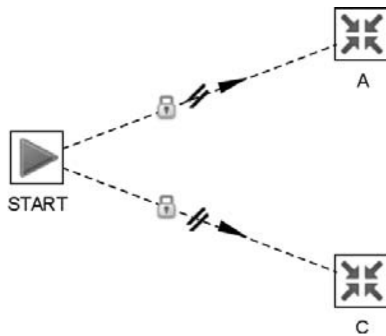


Figure 2-4: Parallel execution of job entries

In the example in Figure 2-4, job entries A and C are started at the same time. Please note that in instances where you have a number of sequential job entries, these are executed in parallel as well. For example, take the case illustrated in Figure 2-5.

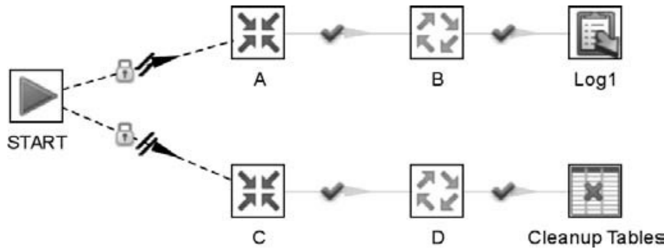


Figure 2-5: Two simultaneous sequences of job entries

In this situation, the job entries [A, B, Log1] and [C, D, Cleanup Tables] are executed in parallel in two threads. Usually this is a desired behavior. However, sometimes you want to have a few job entries executed in parallel and then continue with other work. In that situation, it's important to put all the job entries that need to be executed in parallel in a new job. This job can then be executed in another job, as shown in Figure 2-6.

Figure 2-6: Parallel load as part of a larger job

Job Entry Results

The result of the execution of a job entry not only determines what the next job entry is that will be executed. It also passes the result object itself to the next job entry. The result object contains the following pieces of information:

- **A list of result rows:** These result rows can be set in a transformation using the "Copy rows to result" step. They can be read back using the "Get rows from result" step. Certain job entries such as Shell, Transformation, or Job have an option to loop over the result rows, allowing for advanced parameterization of transformations or jobs.
- **A list of file names:** These file names are assembled during the execution of a job entry. The list contains the names of all the files a job entry comes in contact with. For example, if a transformation read and processed ten XML files, the names of the files read will be present in the result object. It is possible to access these file names in a transformation using the "Get files from result" step. This step will also expose the type of file. Type `GENERAL` is associated with all forms of input and output files whereas the `LOG` type is reserved for Kettle log files that are written.

- **The number of lines read, written, input, output, updated, deleted, rejected, and in error by a transformation:** For more information on how to configure these metrics to be passed, please see Chapter 14.
- **The exit status of a Shell job entry:** This will allow you to specifically evaluate the result of a shell script return value.

In addition to simple usage such as looping and file handling in certain job entries, it is also possible to perform more advanced evaluations using the JavaScript job entry. Table 2-4 describes the exposed objects and variables.

Table 2-4: Expressions You Can Use in the JavaScript Job Entry

EXPRESSION	DATA TYPE	MEANING
`previous_result.getResult()`	`Boolean`	`true` if the previous job entry was executed successfully, `false` if there was some error.
`previous_result.getExitStatus()` or `exit_status`	`Int`	Exit status of previous shell script job entry.
`previous_result.getEntryNr()` or `nr`	`int`	The entry number is increased every time a job entry is executed.
`previous_result.getNrErrors()` or `errors`	`long`	The number of errors, also available as variable `errors`.
`previous_result.getNrLinesInput()` or `lines_input`	`long`	The number of rows read from a file or database.
`previous_result.getNrLinesOutput()` or `lines_output`	`long`	The number of rows written to a file or database.
`previous_result.getNrLinesRead()` or `lines_read`	`long`	The number of rows read from previous steps.
`previous_result.getNrLinesUpdated()` or `lines_updated`	`long`	The number of rows updated in a file or database.
`previous_result.getNrLinesWritten()` or `lines_written`	`long`	The number of rows written to next step.

Continued

Table 2-4 *(continued)*

EXPRESSION	DATA TYPE	MEANING
`previous_result.getNrLinesDeleted()` or `lines_deleted`	`long`	The number of deleted rows.
`previous_result.getNrLinesRejected()` or `lines_rejected`	`long`	The number of rows rejected and passed to another step via error handling.
`previous_result.getRows()`	`List`	The result rows.
`previous_result.getResultFilesList()`	`List`	The list of all the files used in the previous job entry (or entries).
`previous_result.getNrFilesRetrieved()` or `files_retrieved`	`int`	The number of files retrieved from FTP, SFTP, and so on.

In the JavaScript job entry, it would as such be possible to specify alternative conditions to see whether or not a certain path should be followed.

For example, it is possible to count the number of rejected rows in a transformation. If that number is higher than 50, you could determine that the transformation failed, even though this is not the standard behavior. The script would be:

```
lines_rejected <= 50
```

Transformation or Job Metadata

Transformations and jobs are core building blocks in the Kettle ETL tool. As discussed, they can be represented in XML, stored in a repository, or in the form of the Java API. This makes it crucial to highlight a few core metadata properties:

- **Name:** The name of the transformation or job. Even though this is not an absolute requirement, we recommend that you use a name that is unique not only within a project, but even among various ETL projects and departments. This will help during remote execution or storage in a central repository.
- **Filename:** This is the file name or URL where the transformation or job is loaded from. This property is set only when the object is stored in the form of an XML file. When loaded from a repository, this property is not set.
- **Directory:** This is the directory (folder) in a Kettle repository where the transformation or job is loaded from. When it is loaded from an XML file, this property is not set.

- **Description:** You can use this optional field to give a short description of what the transformation or job does. If you use a Kettle repository, this description will be shown in the file listing of the Repository explorer dialog.
- **Extended description:** Another optional field that you can use to give an extended description to the transformation or job.

Database Connections

Kettle database connections are used by transformations and jobs alike to connect to relational databases. Kettle database connections are actually database connection *descriptors*: they are a recipe that can be used to open an actual connection with a database. By definition, the actual connection with the RDBMS is made available at runtime. The act of defining the Kettle database connection by itself does not open a connection to the database.

Unfortunately, very few databases behave in exactly the same way. As a result, we've seen a growing number of options appearing in the Database Connection dialog, shown in Figure 2-7, to cover all the possibilities.

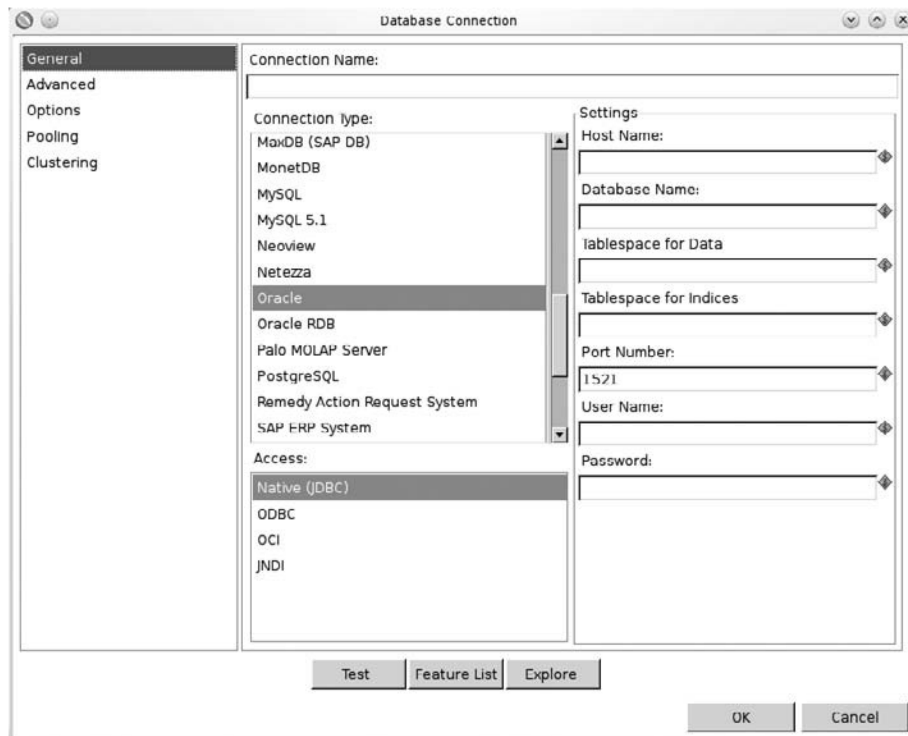


Figure 2-7: The Database Connection dialog

You need to specify three main options in the Database Connection dialog:

- **Connection Name:** The name identifies a connection. As such, it needs to be a unique name in any job or transformation.
- **Connection Type:** Select the type of relational database server that you want to access from the list. Your choice will determine the Settings and Access options that will be available. It will also influence what SQL dialect certain Kettle steps and job entries will use to generate SQL statements.
- **Access:** From this list, you select the type of access you want to use. Typically, we recommend that you use a Native (JDBC) connection. However, it is also possible to use a predefined ODBC data source (DSN), a JNDI data source, or an Oracle OCI (using Oracle naming services) connection.

Depending on the options you selected, you will be presented with a set of database-specific options to fill in on the right side of the dialog. For example, in Figure 2-7 you can see Oracle specific tablespace options that will only appear for that specific database type.

These typically include the following options:

- **Host Name:** This is the hostname or IP address of the database server.
- **Database Name**: For database servers that support multiple databases, you can specify the name of the database to access here.
- **Port Number**: The default port for the selected connection type will be filled in. Make sure to verify that this value is correct.
- **User Name and Password**: The user name and password you want to use to log in to the database server.

Special Options

For most users, the default settings of the database connections available from the General tab of the dialog are sufficient. However, once in a while you might need to use one of the options in the Advanced section of the dialog:

- **Supports boolean data type:** The Boolean (or bit) data type is treated differently in most types of database servers. Even between different versions of the same connection type there are important differences to be found. A lot of databases don't have any support for Boolean data types at all. For this reason, Kettle defaults to the population of a single character (`char(1)`) field with either `Y` or `N` for Boolean fields in a transformation. However, if you enable this option, Kettle will generate the correct SQL dialect for the chosen database type if the database supports a Boolean data type.
- **Quote all in database:** This forces all identifiers (table and field names) to be quoted. This is useful if you suspect that Kettle's internal list of reserved keywords is not up-to-date or if you want to retain the case of identifiers in the database.
- **Force all to lower case:** This option converts all identifiers (table and field names) to lowercase.

- **Force all to upper case:** This option converts all identifiers (table and field names) to uppercase.
- **The preferred schema name:** You can specify the preferred schema (also called *catalog* in certain databases) name for this connection. This schema is used when no other value is known.
- **SQL statements to execute after connecting:** These SQL statements can be used to modify certain connection parameters. For example, they are used to pass session license–related information to the database or to enable certain debugging features.

In addition to these advanced options, a number of database-specific features can be specified in the "Options" section of the dialog. There you can specify a list of driver-specific parameters. For a few database types (MySQL, for example) Kettle will set a few default values to help out. For a detailed list of options, consult the documentation of the database driver that you are using. For a number of database connection types Kettle will present this documentation in the form of a browser page that appears when you press the Help button on the Options tab of the dialog.

Finally, it is possible to enable the Apache Commons Database Connection pooling back end that ships with Kettle. Doing so will make sense only in those situations where you are running lots of very small transformations or jobs with short-lived database connections. It will not limit the number of simultaneous database connections in any possible way. See also Chapter 15 for more information.

The Power of the Relational Database

Relational databases are advanced pieces of software that usually specialized in areas such as joining, sorting, and merging data. Compared to a streaming data engine such as Kettle, they have one big advantage: the data they reference is already stored on disk. When a relational database performs join or sort operations, it can simply use references to the data and can avoid loading all the data in memory. This gives it a distinct performance advantage. The drawback is obviously that loading the data in a relational database is a separate performance bottleneck.

For ETL developers, this means that, when it is possible, it is usually beneficial to let a database perform a join or sort operation. If it is not possible to perform a join in a database because the data comes from different sources, have the database perform the sort operation to prepare for the join in the ETL tool.

Connections and Transactions

A Kettle database connection is only used during the execution of a job or transformation. In a job, every job entry opens and closes the actual connections to the database independently. The same happens in a transformation. However, because of the parallel nature of a transformation, each step that uses a Kettle database connection will open a separate actual database connection and start a separate transaction. While this generally provides great performance for most common situations, it can cause severe

locking and referential integrity problems when different steps update information in the same table.

To remedy the problems that may arise from opening multiple actual database connections, Kettle allows you to enable the "Make transformation database transactional" option. This option can be set for the entire transformation in the Transformation Settings dialog. When this option is enabled, the transformation open a single actual database connection for each different Kettle database connection. Furthermore, Kettle will perform a `COMMIT` when the transformation finishes successfully and a `ROLLBACK` if this is not the case..

Database Clustering

When one large database is no longer up to the task, you can think about using multiple smaller databases to handle the load. One way to spread the load is by partitioning the data with a technique that is called *database partitioning* or *database sharding*. With this method, you divide the entire dataset into a number of groups called partitions (or shards), each of which is stored in a separate database server instance. This architecture has the distinct advantage that it can greatly reduce the number of rows per table and per database instance. The combination of all the shards is called a *database cluster* in Kettle.

Typically, the partitioning method involves calculating the remainder of a division on an identifier to determine the target database server instance that stores the particular shard. This and other partitioning methods are available in Kettle (see Chapter 16 for more information).

The result of this partitioning calculation is a number between 0 and the number of partitions minus one. If you want to use this feature, this is the number of actual database connections that you need to specify in the Clustering section of the database connection dialog. For example, suppose you have defined five database connections to five different shards in a cluster. You could then execute a query in a Table Input step that you are executing in a partitioned fashion, as shown in Figure 2-8.

Figure 2-8: Reading information from a database cluster

As a result, the same query will be executed five times against the five shards. All steps in Kettle that open database connections have been modified to make use of this partitioning feature. For example, the Table Output step will make sure that the correct row arrives at the correct shard when executed in a partitioned fashion.

Tools and Utilities

Kettle contains a number of tools and utilities that help you in various ways and in various stages of your ETL project. The core tools of the Kettle software stack include:

- **Spoon:** A graphical user interface that will allow you to quickly design and manage complex ETL workloads.
- **Kitchen:** A command-line tool that allows you to run jobs
- **Pan:** A command-line tool that allows you to run transformations.
- **Carte:** A lightweight (around 1MB) web server that enables remote execution of transformations and jobs. A Carte instance also represents a slave server, a key part of Kettle clustering (MPP).

Chapter 3 provides more detailed information on these tools.

Repositories

When you are faced with larger ETL projects with many ETL developers working together, it's important to have facilities in place that enable cooperation. Kettle provides a way of defining repository types in a pluggable and flexible way. This has opened up the way for different types of repositories to be used. However, the basics will remain the same regardless of which repository type is used: They will all use the same graphical user interface elements and the stored metadata will also be the same. There are several types of repositories: database, Pentaho, and file repositories.

- **Database repository:** The database repository was created to centrally store ETL information in a relational database. A repository of this type is easy to create: Simply create a new database connection to an empty database schema. If you don't have anything in that schema yet, you can use the Database Repository dialog to create a new set of repository tables and indexes.

  Because of this ease of use, the database repository has remained popular despite the lack of important version management and relational integrity features.
- **Pentaho repository:** The Pentaho repository type is a plug-in that is installed as part of the Enterprise Edition of Pentaho Data Integration. It is backed by a content management system (CMS) that will make sure that all the important characteristics of the ideal repository are met, including version management and referential integrity checks.
- **File repository:** This simplified repository type allows you to define a repository in any kind of folder. Because Kettle uses a Virtual File System back end, this includes non-trivial locations such as zip files, web servers, and FTP servers.

Regardless of the type, the ideal repository should possess these features:

- **Central storage:** Store your transformations and jobs in a central location. This will give ETL users access to the latest view on the complete project.

- **File locking:** This feature is useful to prevent other users from changing a transformation or job you are working on.
- **Revision management:** An ideal repository stores all previous versions of transformations and jobs for future reference. It allows you to open previous versions as well as see the detailed change log. For more information see Chapter 13.
- **Referential integrity checking:** This feature will verify the referential integrity of a repository. This capability is needed to make sure there won't be any missing links, missing transformations, jobs, or database connections in a repository.
- **Security:** A secure repository prevents unauthorized users from changing or executing ETL logic.
- **Referencing:** It is very useful for a data integration developer to be able to reorganize transformations and jobs, or to be able to simply rename them. It should be possible to do this in such a way that references to these transformations and jobs are kept intact.

Virtual File Systems

Flexible and uniform file handling is very important to any ETL tool. That is why Kettle supports the specification of files in the broadest sense as URLs. The Apache Commons VFS back end that was put in place will then take care of the complexity for you. For example, with Apache VFS, it is possible to process a selection of files inside a `.zip` archive in exactly the same way as you would process a list of files in a local folder. For more information on how to specify VFS files, visit the Apache VFS website at `http://commons.apache.org/vfs/`.

Table 2-5 shows a few typical examples.

Table 2-5: Examples of VFS File Specifications

FILENAME	DESCRIPTION
Filename: `/data/input/customers.dat`	This file is defined using the classic (non-VFS) way and will be found and read as such.
Filename: `file:///data/input/customers.dat`	This same file will be read from the local file system using the Apache VFS driver.
Job: `http://www.kettle.be/GenerateRows.kjb`	This file can be loaded in Spoon, executed using Kitchen, and referenced in the Job job entry. Every time the XML file is transparently loaded from the web server.
Folder: `zip:file:///C:/input/salesdata.zip` Wildcard: `.*\.txt$`	This folder/wildcard combination can be entered in steps like "Text file input." The specification of the wildcard will search and read all the files in the specified zip file that end with `.txt`.

Parameters and Variables

It's very important when using a data integration tool that certain aspects of your work can be parameterized. This makes it easy to keep your work maintainable. For example, it's important that the location of a folder with input files be specified in one central location so that the various components of the data integration tool can make use of that value. In Kettle, you do this with variables.

Defining Variables

Each variable has a unique name and contains a string of any length. Variables are set either at system level or dynamically in a job. They have a specific scope that makes it possible to run the same job or transformation in parallel with different variables set on the same system, Java Virtual Machine, or J2EE container without a problem.

There are two main ways that a variable is instantiated: It can be set by the system or defined by the user. System variables include those defined by the Java Virtual Machine (such as `java.io.tmpdir`, the system's location for temporary files) and those defined by Kettle (such as `Internal.Kettle.Version`, containing the version of Kettle you're using).

NOTE **For more information on the various types of system variables and how they influence your environment, please see Appendix C.**

Variables can be set in multiple ways. The most common way of defining a variable is to place it in the `kettle.properties` file in the Kettle home directory (`${KETTLE_HOME}/.kettle`). The most convenient way of editing this file is by using the editor, located in version 4.0 under the "Edit ➪ Edit the kettle.properties file" menu.

You can also set variables dynamically during the execution of a job. There are various ways to do this, but the easiest way to do it is with a transformation. The Set Variables step allows you to set a variable with the appropriate scope for use in the current job. For more dynamic situations you can also use a JavaScript step or the User Defined Java Class step. Figure 2-9 shows a job that first sets a number of variables after which it uses them in a sub-job.

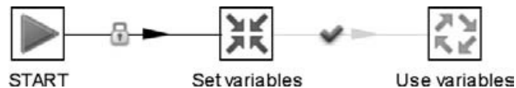


Figure 2-9: Defining and using variables in a job

Finally, variables can also be formally specified when you declare them as named parameters.

Named Parameters

If there are variables that you consider to be parameters for your transformation or job, you can declare them in the Parameters tab of the respective settings dialogs. Figure 2-10 shows the Parameters tab, where you can enter the parameter name, its default value, and a description.

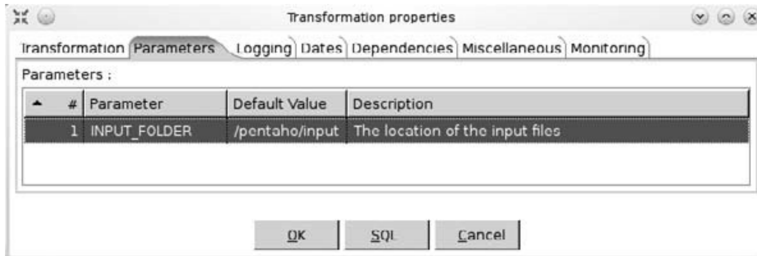


Figure 2-10: Defining named parameters

The main advantage of using named parameters is that they are explicitly listed, documented with a description, and carry an optional default value. This makes it obvious that you can set values for them in a job or a transformation.

Once you have defined parameters, you can specify values for them in all possible locations where you execute a transformation or job: in the execution dialogs, at the Pan or Kitchen command line, or in the Transformation or Job job entries.

For example, you could run a transformation with a non-default value for the input folder location:

```
user@host:$ sh pan.sh -file:/pentaho/read-input-file.ktr -param:
    INPUT_FOLDER=/tmp/input/
```

You can find more details on using Kitchen and Pan and specifying parameter and variable values on the command line in Chapters 3 and 12.

Using Variables

All input fields where variables can be used in Kettle are annotated with a diamond-shaped icon showing a red dollar sign on the upper-right of the input field. For example, the CSV Input step contains many fields that can be specified using variables, including the Filename, Delimiter, and Enclosure fields, as shown in Figure 2-11.

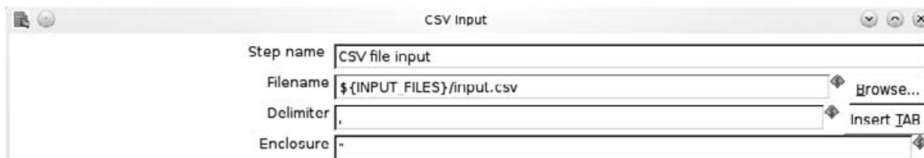


Figure 2-11: Using variables in Spoon

Entering variables is easy. Simply press Ctrl+Space to see the list of all variables that are known to Kettle at that time. If you hover over the input field, you will see the actual value of the complete expression.

As you can see in the example, variables are recognizable by the enclosing ${} symbols. The name of the variable is always enclosed within the curly brackets—for example, `${INPUT_FOLDER}`.

Even though it is less common, you can also surround the name of the variable with double percent signs (%%)—for example, `%%INPUT_FOLDER%%`.

One final special form of variable notation is the specification of hexadecimal values. You can specify binary values by listing a comma-separated list of hexadecimal values inside the `$[]` symbols (note that these are square brackets, not curly brackets). For example, the following value encodes the `123456` ASCII string `$[31,32,33,34,35,36]`. This method of entering data is useful on the rare occasion when you need to specify non-readable binary "character" symbols like `$[01]`.

NOTE **Variables are parsed at runtime in a recursive fashion. Because of this, it is possible to use variables in the value of another variable. This makes it easy to define variables in a very generic and re-usable fashion.**

Visual Programming

Kettle can be categorized under the group of *visual programming languages* (VPLs) because it lets its users create complex ETL programs and workflows simply by the graphical construction of diagrams. The diagrams in the case of Kettle are transformations and jobs. Visual programming is a core concept of Kettle because it allows you to quickly set up complex ETL jobs and lowers maintenance. It also brings the world of IT closer to the business requirements by hiding a lot of non-essential technical complexities.

NOTE **For more information on Visual programming languages, see** `http://en.wikipedia.org/wiki/Visual_programming_language`**.**

In this section, we show you how you can get started right away with Kettle. We do this with a simple example that reads data from a text file and stores it into a database. We do not intend this as a full primer or beginners' guide to Kettle. Rather, it is an introduction to get started for those who are in a hurry. For a more detailed practical example of how you can solve complex ETL problems, see Chapter 4.

NOTE **This chapter does not cover the installation or configuration of Kettle. If you have not yet installed Kettle, you'll need to refer to Chapter 3 and install Kettle to follow along with the example.**

Getting Started

Visual programming in Kettle is done with a graphical user interface called Spoon. Once you start Spoon on your system, you are presented with a welcome page leading you to information on how to get started, samples, documentation, and much more. In our case, we want to create a new transformation that will read data from a text file and store it into a database table. To do this, we click the "New file" icon (represented by a page of paper with a dog-eared corner) in the toolbar and select Transformation, as shown in Figure 2-12.

NOTE **Note that the principles described in this section for transformations are applicable to jobs as well.**

Figure 2-12: Creating a new transformation

When you create a new transformation, you are presented with an empty canvas on which you can design your ETL workload, as in Figure 2-13.

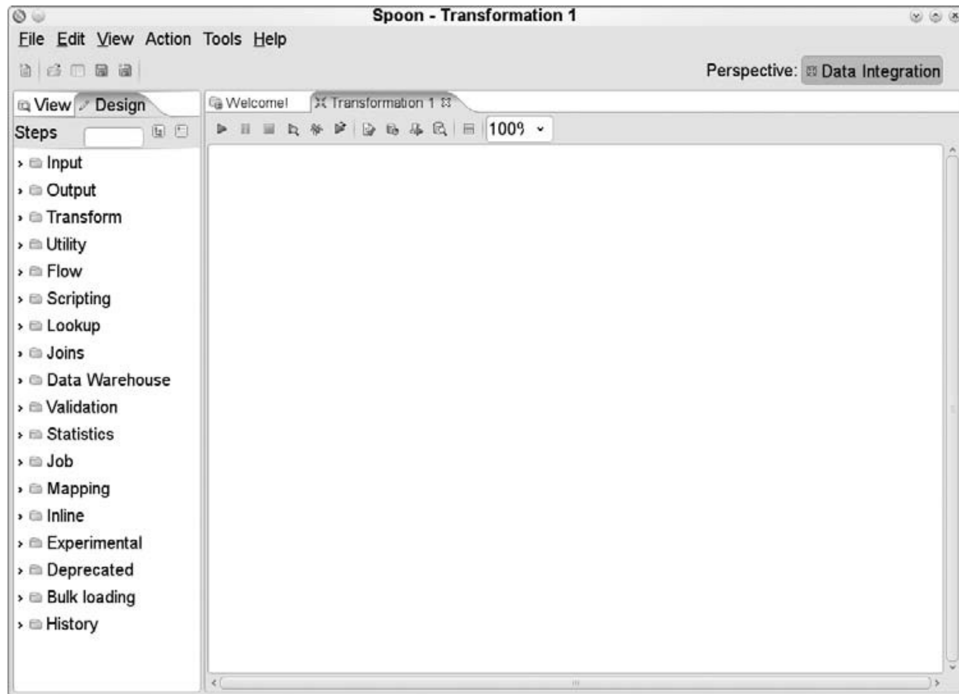


Figure 2-13: An empty canvas

Creating New Steps

On the left-hand side of the blank page, you can see all sorts of categories listed. Under these category headings, there are many steps that can be used to design the transformation. For this example, you are looking for a step that can read from a CSV file. Typing **CSV** in the quick search box next to the Steps label, as shown in Figure 2-14, will list all steps that might be useful.

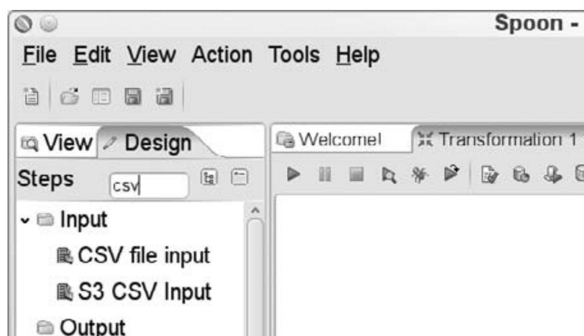


Figure 2-14: Finding CVS-related steps

In this case, you want to use the “CSV file input” step to read data. To place it in your transformation, simply click on the “CSV file input” step in the Input category and drag it onto the canvas on the right. A new icon will appear on the blank page with a “CSV file input” label below. This icon, shown in Figure 2-15, represents the functionality of reading a CSV file.

Figure 2-15: A brand new step

To configure this step, double-click on the icon or click the Edit (pencil) icon in the graphical slide-out menu and select “Edit step.” The slide-out menu appears when you hover over a step icon for a few seconds, as shown in Figure 2-16.

Figure 2-16: The graphical slide-out menu

You will be presented with a dialog that allows you to edit all aspects of the step, as shown in Figure 2-17.

CSV Input
Step name CSV file input
Filename Browse...
Delimiter , Insert TAB
Enclosure "
NIO buffer size 50000
Lazy conversion?
Header row present?
Add filename to result
The row number field name (optional)
Running in parallel?
File encoding

#	Name	Type	Format	Length	Precision	Currency	Decimal	Group	Trim type
1									

OK Cancel Preview Get Fields

Figure 2-17: Configuring the “CSV file input” step

NOTE **You can also access the dialog by selecting the step context menu (the downward arrow) from the graphical slide-out menu and selecting the "Edit step" option.**

For this example, you want to provide Kettle with the location of the file you're reading, the delimiter, the optional enclosure, and various other fields. Once that is done, you can test the step by clicking the Preview button that is found in most steps that read data. When you are satisfied with the output of the step, click the OK button and that part of the challenge will be completed.

Putting It All Together

The next step is to store the data from the CSV file in a database table. To accomplish this, you will use the quick search box again to look for a step that could accomplish this feat. Type in the word **table** and select the "Table output" step from the results. The mouse-over tooltip of that entry says: "Write information to a database table" and that is exactly what you want to do. Drag this step to the right of the "CSV file input" step that is already on the canvas. Your canvas should look like Figure 2-18.

Figure 2-18: Adding the "Table output" step

Before you configure a step, it's usually a good idea to connect it with a hop to its predecessor. This is useful because it allows Kettle to know what kind of information is being sent to the step. That in turn allows for the automatic configuration of a lot of step options.

The creation of a hop is easy: Simply click the Output icon (a page with a green arrow in the lower left corner) in the step's graphical slide-out menu. That will cause a gray arrow to be drawn on the canvas that will turn blue once you move it to the "Table output" step, as shown in Figure 2-19.

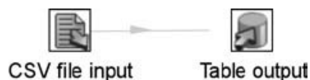


Figure 2-19: Creating a new hop

Clicking the left mouse button once when the arrow is blue will create a new hop. (Other ways of creating new hops include dragging from one step icon to another with the middle mouse button or with the left mouse button while keeping the Shift key pressed.) Because the "CSV file input" step is capable of error handling, you are asked which type of output you want to handle with this hop. If read errors in the step (conversion errors, missing fields, and so on) are not handled, the whole transformation

will fail. However, if you use error handling, the rows with an error in them will be sent to another step where they can be placed in another file or database table, as shown in Figure 2-20.

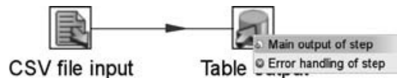


Figure 2-20: Selecting the type of output

In this simple example, you simply want to use the main output of the "CSV file input" step so select that option. Now both steps are connected with a hop.

All that's left to do now is edit the settings of the "Table output" step and you're done. Again, you can double-click on the "Table output" step, use the slide-out menu, or use the context menu to bring up the step's dialog. Figure 2-21 shows the dialog to configure the "Table output" step.

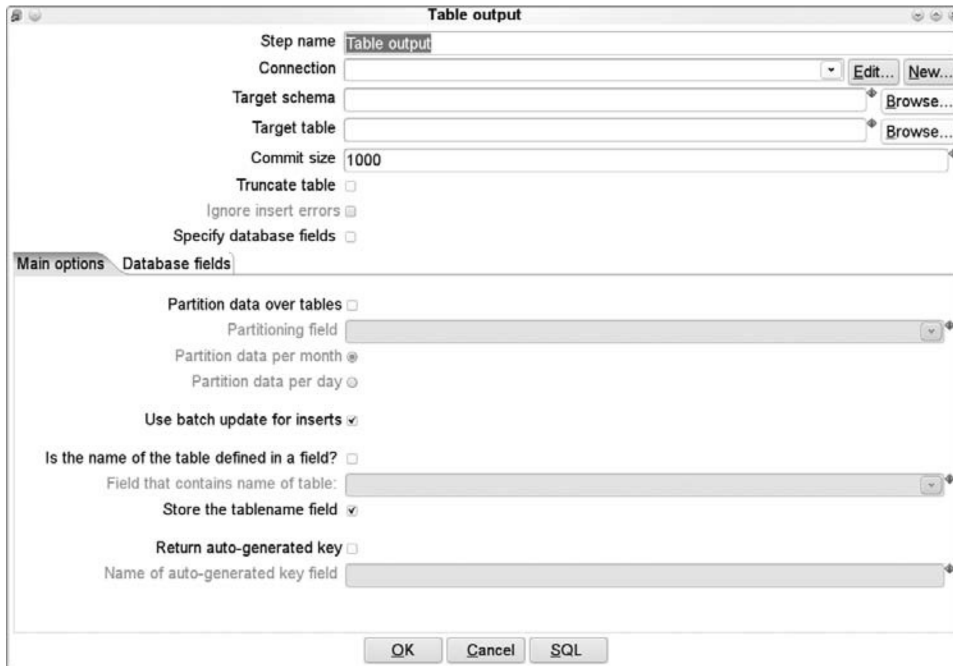


Figure 2-21: Configuring the "Table output" step

The first thing you notice when you open the dialog is that you need a database connection to make this step function properly. You can do this easily with the New button and the information in the "Database connections" section earlier in this chapter. You can then complete the required information in the dialog by specifying the schema and table name where the data is to go to. If the database table does not exist yet or needs

to be altered, you can use the SQL button at any time to generate the appropriate DDL for the specified database type.

When you are done, click OK and the transformation is finished. You can now click the Run icon (the green arrow pointing to the right) to execute the transformation. This will allow you to save the transformation and give it a name and description.

While this is a simple example, a lot of technology was used, including file reading, writing to a database, data conversion, error handling, parallel processing, and much more. However, to make it work, you didn't have to write a single line of programming code, nor was any code generated. The two main tasks, reading and writing, are clearly represented by the two icons on the transformation canvas, which makes it very easy to maintain. The transformation also addressed the complex issue of routing data simply with the creation of hops between steps. All this leads to a very short development time for your ETL solution.

Summary

In this chapter, we introduced some Kettle design considerations and explained the core Kettle building blocks. We finished with an introduction to visual programming in Kettle. Here are a few notable things you learned:

- Data in the form of rows of data is manipulated in a transformation by steps.
- Job entries are the basic components of a job. They are executed in sequence based on the result of a previous job entry.
- Database connections can be defined in transformations and jobs. You learned how to use the various parameters in the database dialog.
- There are different repository types and they each have specific benefits.
- You can use the Virtual File System to flexibly specify files in various locations.
- You can use variables and named parameters to make your transformations and jobs re-usable and easy to configure.
- How to get started with visual programming and how to easily create a transformation.

CHAPTER

3

Installation and Configuration

This chapter provides a high-level overview of the collection of tools included in a Kettle installation, and provides detailed instructions for their installation and configuration. Fortunately, installing Kettle is a fairly straightforward task, but it is helpful to refer to a single overview of all tasks involved, which is what this chapter aims to provide.

NOTE **If you've already successfully installed Kettle, it's likely you're already familiar with many of the topics discussed here. You may want to skim over some of this chapter.**

In addition to providing an overview, this chapter covers a few detailed configuration topics that apply to particular real-world Kettle scenarios. These sections may not make a lot of sense until you encounter those scenarios, so the subsequent chapters of the book will refer back to the relevant sections of this chapter.

Kettle Software Overview

Kettle is a single product, but consists of multiple programs that are used in different phases of the ETL development and deployment cycle. Each program serves a particular purpose and is more or less independent of the others. However, all of the programs depend on a common set of Java archives that make up the actual data integration engine. An overview of the main Kettle programs is shown in Figure 3-1.

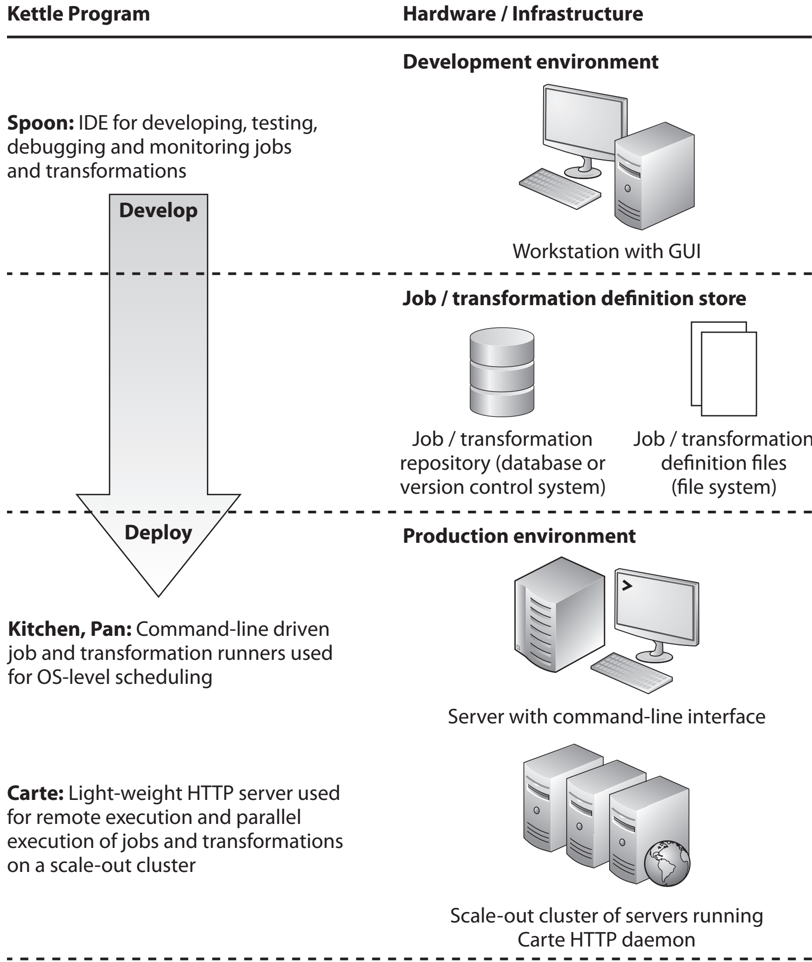


Figure 3-1: Overview of Kettle programs

The following list briefly explains the purpose of each of the different programs listed in Figure 3-1:

- **Spoon:** The integrated development environment. Offers a graphical user interface for creating and editing job and transformation definitions. Spoon can also be used to execute and debug jobs and transformations, and it also includes functionality for performance monitoring.
- **Kitchen:** A command line–driven job runner, which can be used to integrate Kettle with OS-level scripts. It is typically used to schedule jobs with a scheduler such as `cron`, `at`, or the Windows Task Scheduler.
- **Pan:** A command line–driven program just like Kitchen, but it is used for executing transformations instead of jobs.

- **Carte:** A light-weight server (based on the Jetty HTTP server) that runs in the background and listens for requests to run a job. Carte is used to distribute and coordinate job execution across a collection of computers forming a *Kettle cluster*.

These programs are described in more detail in the following sections.

Integrated Development Environment: Spoon

Spoon is Kettle's integrated development environment (IDE). It offers a graphical user interface based on SWT, the standard widget toolkit. As such, it is predominantly used as an ETL development tool.

You can start Spoon by executing the corresponding shell script found in the Kettle home directory. For Windows users, the script is called `Spoon.bat`, and for UNIX-like operating systems, the script is called `spoon.sh`. Windows users can also use the executable `Kettle.exe` to start Spoon.

NOTE In this book, Spoon is heavily used throughout Chapters 4 to 11, and you're guaranteed to know it inside out by the time you've finished this book.

A screenshot of Spoon is shown in Figure 3-2.

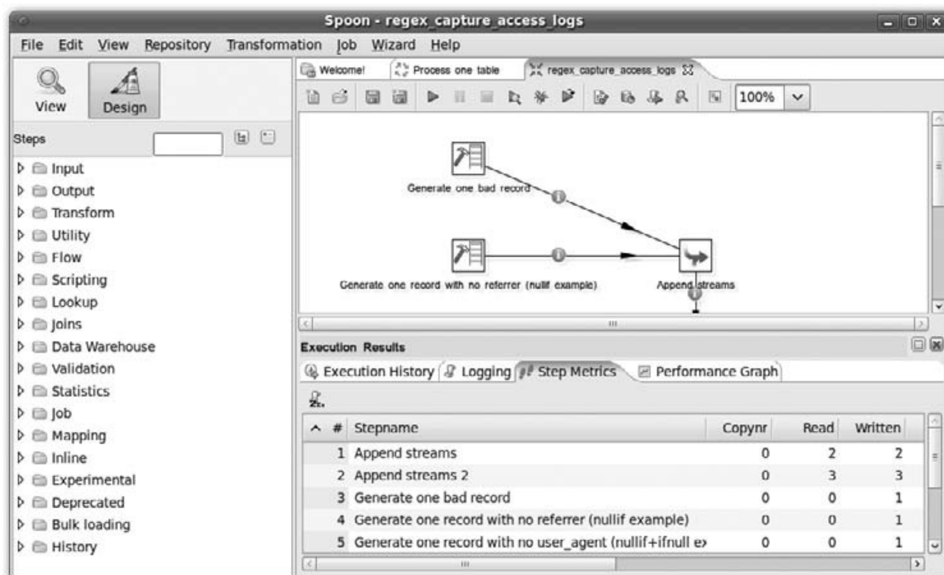


Figure 3-2: Spoon

In Figure 3-2, you can clearly see the main features of Spoon: a main menu bar across the top, and below that, an application window vertically divided in two main parts. The workspace on the right-hand side provides a tab interface to work with all jobs and

transformations that are currently open, and the left side of the application window features a tree view.

The workspace itself is horizontally divided: the upper part is called the *canvas*, where ETL developers can create jobs and transformations by simply drawing a diagram that models the effect of the job or transformation. In Figure 3-2, the activated tab shows a transformation diagram.

Diagramming a job or transformation is a matter of adding job entries and transformation steps onto the canvas. You can add elements to the canvas from the context menu or drag them from the tree view if it is in Design mode, as in Figure 3-2. These job entries and transformation steps can then be connected using *hops*. In the diagram, the hops are shown as lines drawn from the center of one job entry or transformation step to the other. The hops define the flow of control (in case of jobs) or flow of data (in case of transformations).

The mode of the tree view can be controlled using the two large toolbar buttons immediately above the tree view. In View mode, shown in Figure 3-3, the tree view offers an alternate view of the opened jobs and transformations, allowing easy access to the individual steps, hops and any other resources, such as database connections.

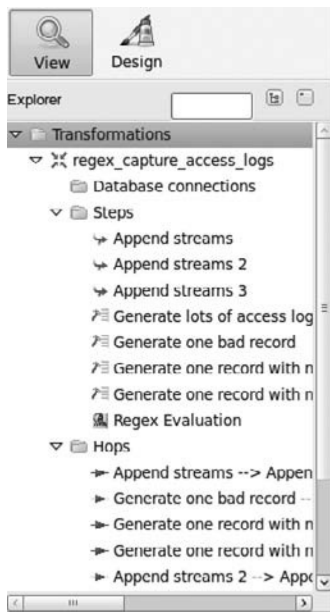


Figure 3-3: The tree view in View mode

Utilities for executing and debugging jobs and transformations are integrated into Spoon, allowing ETL developers to test their jobs and transformations without leaving the IDE. A number of these functionalities can be accessed through the buttons on the toolbar, which is located in the workspace right above the canvas.

NOTE **Testing and debugging are discussed in detail in Chapter 11.**

Spoon also features facilities for logging and monitoring the execution of jobs and transformations. Some of these functionalities are shown in the Execution Results panel, which is shown in the bottom half of the workspace in Figure 3-2.

NOTE **Monitoring is discussed further in Chapters 12 and 15.**

Command-Line Launchers: Kitchen and Pan

Jobs and transformations can be executed from within the graphical Spoon environment, but it is not very practical to do so unless you're developing, testing, or debugging. After the development phase, when you need to deploy your jobs and transformations, Spoon is not of much use.

For deployment, you typically need to be able to invoke your jobs and transformations from the command line so they can be integrated with shell scripts and the operating system's job scheduler. The Kitchen and Pan command-line tools were designed especially for this purpose. As such, Kitchen and Pan are typically used after the ETL development phase to execute jobs and transformations on production environments.

Pan and Kitchen are very similar in concept and usage, and the list of available command-line options is virtually identical for both tools. The only difference between these tools is that Kitchen is designed for running jobs, whereas Pan runs transformations.

Kitchen can be started by running its corresponding shell script from the Kettle home directory. The script is called `Kitchen.bat` for windows users, and `kitchen.sh` for UNIX-like systems. Similarly, the script for Pan is called `Pan.bat` on Windows, and `pan.sh` for UNIX-like operating systems.

NOTE **Kitchen and Pan are discussed in detail in Chapter 12.**

Job Server: Carte

The Carte program is a job runner, just like Kitchen. But unlike Kitchen, which runs immediately after invocation on the command line, Carte is started and then continues running in the background as a server (daemon).

While Carte is running, it waits and listens for requests on a predefined network port (TCP/IP). Clients on a remote computer can make a request to a machine running Carte, sending a job definition as part of the message that makes up the request. When a running Carte instance receives such a request, it authenticates the request, and then executes the job contained within it. Carte supports a few other types of requests, which can be used to communicate progress and monitor information.

Carte is a crucial building block in Kettle clustering. Clustering allows a single job or transformation to be divided and executed in parallel by multiple computers that are running the Carte server, thus distributing the workload.

NOTE **Carte and clustering are discussed in detail in Chapters 16 and 17.**

Encr.bat and encr.sh

The Kettle home directory contains a few more `.bat` and `.sh` scripts, which are better classified as utilities rather than standalone programs like Spoon, Pan, Kitchen, and Carte. One of these utilities deserves to be mentioned here: the `Encr.bat` and `encr.sh` scripts can be used to encrypt a plain-text password. While using Kettle, passwords pop up in a number of places:

- In database connections defined in jobs and transformations.
- In job entries and transformation steps that connect to a server, (such as an SMPT server, FTP server, or HTTP server).
- Carte requires authentication in order to process requests for job execution and progress information. With the default authentication method, the password is specified in the `kettle.pwd` file, which is discussed in more detail later in this chapter.
- In configuration files that are read by the various Kettle programs, such as the `kettle.properties` file discussed later in this chapter.

Although passwords can be entered in plaintext in all these instances, it is good practice to encrypt them with the `Encr.bat` or `encr.sh` script. This prevents an unauthorized person from seeing and abusing the plaintext password.

Installation

The following section describes the steps that are necessary to install and run Kettle programs.

Java Environment

Kettle is a Java program; it requires a Java runtime (that is, a Java Virtual Machine or JVM, and a set of standard classes) to run. You may already have Java installed, but for completeness, we discuss Java installation in the remainder of this section.

We recommend using the Sun Java Runtime Environment (JRE), version 1.6 if you just want to run Kettle programs. If you want to build Kettle from source and/or develop Kettle plugins, you should obtain the Sun Java Development Kit (JDK), also version 1.6.

NOTE You can try to use a JRE or JDK from other vendors if you like, but all examples that appear in this book were tested and developed using Sun JDK version 1.6

Installing Java Manually

The site `http://java.sun.com/javase/downloads/index.jsp` lets you download executable installation programs to install Java for a variety of operating systems, including Windows, Sun Solaris, and Linux. Simply choose the Java version of your

preference and the appropriate platform (for example, Linux x64 or Windows) to download an installation executable. After downloading, run the executable and follow the instructions provided by the installer.

If you are running an operating system for which the Sun website does not offer a suitable version, you should refer to the website of your operating system vendor. For example, Mac OSX users should go to `http://developer.apple.com/java/download/` and find a suitable installation executable there. As with the installation executables provided by Sun, executing such an installer should provide the appropriate instructions to complete the installation process.

Using Your Linux Package Management System

If you're using a popular Linux distribution such as Debian (or compatible distributions such as Ubuntu), Red Hat (or compatible distributions such as Fedora or CentOS), or SUSE/openSUSE, installing your Java environment can be done using the package management system of that particular distribution.

For example, on Ubuntu, Java can be installed conveniently using the synaptic package manager. Just search for a package called `sun-java6-jre` or `sun-java6-jdk` (for the runtime environment and the development kit respectively), select it, and click to install. Alternatively, you can install it from the command line using a utility such as `apt-get`:

```
shell> sudo apt-get install sun-java6-jdk
```

If your operating system does not offer package management, or the package management system does not provide the required Java version, you should go to the Oracle's Java downloads website and download an appropriate installation program yourself.

Installing Kettle

Kettle is distributed as a single compressed archive. Downloads are hosted on `sourceforge.net` as part of the Pentaho business intelligence project at `http://sourceforge.net/projects/pentaho`, and you can find all available versions beneath the Data Integration folder at `http://sourceforge.net/projects/pentaho/files`.

Versions and Releases

On the SourceForge site you'll find a separate folder for each available version, where the folder name indicates the version. For example, at the time of this writing, the current version is 4.0.0, which is contained in the `4.0.0-stable` folder, and the folder called `3.2.0-stable` contains archives for the stable release of the previous 3.2.0 version.

NOTE **In this book, all examples were tested on version 4.0, but it's possible and even likely that many things described in this book also apply to other Kettle versions.**

You may also find folders for more recent versions that have a `-RCxx` or `-Mxx` postfix instead of `-stable`. These folders correspond to a release candidate (RC) and a milestone (M) release respectively, where `xx` is a number indicating the specific version. For example, the folder `4.0.1-RC1` contains the release candidate for the upcoming 4.0.1 version.

NOTE SourceForge is the proper site for downloading the actual Kettle releases, but it does not provide the latest bugfix releases nor binaries for the latest development builds.

Binaries for the latest development builds for Kettle as well as other Pentaho products are continuously produced through the Hudson continuous integration platform. You can download the Hudson builds from `http://ci.pentaho.com/`**. For a specific product, simply click the respective tab and choose the appropriate download file there. For example, for Kettle, click the Data Integration tab.**

Bugfix releases are released periodically through SourceForge, but in some cases, you may not be able to wait that long. In such cases, you should consider checking out the Kettle source code from Subversion and building Kettle yourself from source.

You can check out the Kettle source from the Subversion URL `svn://source.pentaho.org/svnkettleroot/Kettle/trunk`**. This process is described in detail in Chapter 22. You can build using the Ant build tool, which is available from** `http://ant.apache.org/`**.**

Archive Names and Formats

The version-specific folders contain the actual archive file, which is available in `.zip` and `.tar.gz` formats. The archive files are named according to the pattern `pdi-ce-`*`version.extension`*, where *`pdi`* stands for Pentaho Data Integration and *`ce`* stands for Community Edition. For example, `pdi-ce-4.0.0-stable.zip` is the zip archive file for the current 4.0.0 stable release. Windows users would typically download the `.zip` archive, whereas users of UNIX-like systems should download a `.tar.gz` archive.

Downloading and Uncompressing

You can download the archive file of your choice directly from the SourceForge web page using your web browser. Alternatively, you can download it from the terminal using a command-line utility such as `wget`. For example, the following line would download the `.tar.gz` archive for the 4.0.0-stable release to the current working directory (the command should all be on one line, but is wrapped here to fit on the page):

```
shell> wget http://sourceforge.net/projects/pentaho/files/
Data%20Integration/4.0.0-M2/pdi-ce-javadoc-4.0.0-M2.tar.gz/download
```

Uncompressing the archive is done in the usual way for the respective format. For example, on UNIX-like systems the `.tar.gz` archive can be conveniently extracted using `tar` with a command line like this:

```
shell> tar -zxvf pdi-ce-4.0.0-stable.tar.gz
```

On Windows, you can use a utility such as Peazip, or the Windows integrated zip utility. You can typically invoke these by right-clicking on the `.zip` archive and choosing "Extract here" or "Extract to folder" from the context menu.

Kettle doesn't care to which location it is extracted on your system, so you should extract it to a location that is most suitable for your situation. For example, on a Windows development machine, it makes a lot of sense to create a `kettle` or a `pentaho` directory in the `Program Files` directory for this task. On UNIX-like systems, you could create such a directory in your home directory in case you're setting up your own development environment, while a location such as `/opt/pentaho` or `/opt/kettle` may be more suitable for a production environment.

Extracting the archive file yields a directory called `data-integration`, which contains the actual software and resources. It is advisable to rename the `data-integration` directory so that it reflects the version you originally obtained. A sensible choice is to rename it simply to the exact name of the archive file, minus the extension:

```
shell> mv data-integration pdi-ce-4.0.0-M2
```

In the remainder of this book, we will refer to *the Kettle home directory* to indicate any such directory containing the Kettle software.

Renaming the directory to something that clearly contains the Kettle version allows you and possibly others working in the same environment to see at a glance which version of Kettle is being used. It also allows you to maintain different versions of Kettle next to each other in a common parent directory, which can be convenient in case you need to test a newer version before upgrading.

Running Kettle Programs

All Kettle programs can be started using shell scripts which are located in the Kettle home directory. There are some minor differences between Windows and UNIX-like platforms because of the differences in the command shells in these respective platforms, but there are more similarities than differences.

Note that the script files assume that the Kettle home directory is also the current working directory. This means that you'll have to explicitly set the current working directory to the Kettle home directory when calling the Kettle scripts from your own scripts.

Windows

After extraction, Windows users can run Kettle programs simply by executing a batch file located within the Kettle home directory. For example, to design transformations and jobs,

you can double-click `Spoon.bat` to start up Spoon, or you can run `Kitchen.bat` directly from the command line, or call it from within your own `.bat` files to execute jobs.

UNIX-like systems

For UNIX-like systems, you can execute Kettle programs by running the corresponding `.sh` script. It is possible you still need to make the `.sh` files executable before you can run the Kettle programs. For example, if your current working directory is your Kettle home directory, you can run:

```
shell> chmod ug+x *.sh
```

You should now be able to run the Kettle programs by executing their respective scripts, provided Kettle home is your current working directory.

Creating a Shortcut Icon or Launcher for Spoon

Because you'll be running Spoon mostly from within a graphically enabled workstation, you might want to add a shortcut or launcher to your taskbar, menu, or desktop (or its equivalent on your system).

Adding a Windows Shortcut

Windows users can open the Windows Explorer and navigate to the Kettle home directory. From there, right-click on `Spoon.bat` and choose the "Create shortcut" option from the context menu. This will create a new shortcut (`.lnk` file) in the Kettle home directory that can be used to launch Spoon.

Right-click the newly created shortcut file and choose Properties from the context menu. This will open the properties dialog showing the shortcut tab page. In that tab page, the Target and Start in fields should be filled in correctly already, so do not edit those fields.

It's a good idea to add the Kettle icon to your shortcut to make it easier to recognize when it appears on your desktop. To do this, click the "Change icon" button, and use the "Browse…" button to navigate to the Kettle home directory. Select the `spoon.ico` file and confirm the changes by pressing OK. Then, confirm the shortcut's properties dialog, again by pressing OK.

You can now drag the shortcut to your desktop, quick launch toolbar, or Start menu. Typically, Windows will make a copy of the shortcut as soon as you drop it on any of these items, so you can place more shortcuts in other places without having to re-create the shortcut and associating the icon again.

Creating a Launcher for the GNOME Desktop

For the GNOME desktop, which is the default desktop environment on many popular Linux distributions, you can create a launcher on your desktop by right-clicking the desktop background and choosing Create Launcher from the context menu. This opens the Create Launcher dialog.

In the Create Launcher dialog, ensure that the Type field is set to Application. Enter Spoon as the name for the new launcher. Click the Browse button next to the Command

field. This will open a file browser dialog. Navigate to the Kettle home directory, and select the `spoon.sh` file. Click the Open button to close the file browser and assign the full command line to the command field of the launcher.

To specify an icon, click the button with the launchpad icon on the left of the Create Launcher dialog to open the "Browse icons" dialog. Use the Browse button to open a file browser, and navigate to the Kettle home directory. Click the open button to return to the "Browse icons" dialog, which should now display `spoon.ico` and `spoon.png` icons. Select the `spoon.png` icon and press OK to confirm.

Optionally, fill in a description and then confirm the Create Launcher dialog to place the launcher on your desktop. If you like, you can drag the launcher and drop it on the main menu to add it there. To create a launcher for one of the submenus, open the menu editor via System ➪ Preferences ➪ Main menu. There, navigate to the appropriate submenu (for example, Programming) and click the New Item button. This will open the Create Launcher dialog, which you can fill in as described above to add a launcher on the desktop.

Configuration

There are a number of factors in Kettle's environment that influence the way Kettle behaves. Some are genuine configuration files; others are pieces of external software that integrate with Kettle. Together, we refer to this as Kettle's configuration.

In the remainder of this section, you will learn which components make up Kettle's configuration, and how you should manage these when working with Kettle.

Configuration Files and the .kettle Directory

A number of files influence the behavior of Kettle programs. As such, these files are considered to be part of Kettle's configuration, and in many cases they should be managed when migrating and/or upgrading. The files are:

- `.spoonrc`
- `jdbc.properties`
- `kettle.properties`
- `kettle.pwd`
- `repositories.xml`
- `shared.xml`

Some of these files are read only by one Kettle program, and others are shared by more than one. For most (but not all) of these files, a separate instance is created in a `.kettle` directory, which by default resides beneath the home directory of each user, allowing each user to have its own settings. (On UNIX-like systems, the home directory is typically `/home/`*`<user>`*; on Windows, the home directory is by default located at `C:\Documents and Settings\`*`user`*, where *`user`* stands for the actual username.)

The location of the `.kettle` directory can be changed explicitly by setting it in a `KETTLE_HOME` environment variable. For example, on a production machine, you most likely want to ensure that all users use the same configuration for running jobs and transformations, and explicitly setting `KETTLE_HOME` to a single location ensures everybody will be using the same configuration files. Alternatively, it may be convenient to maintain one particular configuration per ETL project, in which case you would set the value of the `KETTLE_HOME` environment variable accordingly before running any of the Kettle program scripts.

The following subsections discuss each file in more detail.

.spoonrc

As the name implies, the `.spoonrc` file is used to store preferences and program state of Spoon. Other Kettle programs do not use this file. It is stored in the `.kettle` directory beneath the user's home directory, so there are multiple instances of this file, one for each Spoon user. Items that are typically stored in `.spoonrc` are:

- **General settings and defaults:** In Spoon, these settings can be viewed and edited in the General tab of the Kettle Options dialog. You can open the Options dialog via the main menu ➪ Edit ➪ Options.
- **Look and feel preferences, such as font sizes and colors:** In Spoon, these can be viewed and edited in the Look and Feel tab of the Kettle properties dialog.
- **Program state data:** Such as the most recently used files list.

Normally you should not edit `.spoonrc` manually. However, it does make sense to overwrite the `.spoonrc` file of a new Kettle installation with one that contains your preferences. For this reason it also makes sense to ensure `.spoonrc` is included in your file system backups.

jdbc.properties

For each Kettle installation there is a single `jdbc.properties` file, and it is stored in the `simple-jndi` directory beneath the Kettle home directory. This file is used for storing database connection data for database connection objects of the JNDI type. Kettle can utilize JNDI for referring to JDBC connection data such as the host IP address, and for user credentials, which can then be used for specifying database connection objects for transformations and jobs.

NOTE **JNDI stands for Java Naming and Directory Interface, which is a Java standard that allows names to be used to refer to a server or service. You can find out more about JNDI at the Sun website at** `http://java.sun.com/products/jndi/`**.**

Note that JNDI is just one way that Kettle can be used to specify database connection data; database connection data may also be stored inside the local database connection objects of a transformation or job or as part of the PDI repository. We include it here because JNDI database connections form part of the Kettle configuration.

In jdbc.properties, JNDI connection data is stored as a set of multiple lines, each of which specifies a key/value pair, separated by an equal sign. The value appears after the equal sign. The key appears before the equal sign, and is itself composed of the JNDI name and a property name, separated by a forward slash. The lines for one particular connection are defined by the JNDI name, tying together the set of properties that make up the data required to establish a connection. The following properties are defined:

- type: Value is always javax.sql.DataSource.
- driver: Fully qualified class name of the Java class that implements the JDBC Driver class.
- url: The JDBC URL that the driver should use to connect with the database.
- user: The username that should be used when establishing the connection.
- password: The user's password.

The following example illustrates how a JNDI connection is stored in jdbc.properties:

```
SampleData/type=javax.sql.DataSource
SampleData/driver=org.hsqldb.jdbcDriver
SampleData/url=jdbc:hsqldb:hsql://localhost/sampledata
SampleData/user=pentaho_user
SampleData/password=password
```

In this example, the JNDI name is SampleData, which can be used to establish a connection to an HSQL database as the user pentaho_user having the password password.

You can add your own connections in jdbc.properties, following the format of the SampleData JNDI connection snippet. Kettle currently does not offer any graphical user interface to do this, but it is quite easy to use a simple text editor for this task.

Because the connections defined in jdbc.properties can be used by transformations and jobs, you should take proper measures to manage this file. At the very least, you should include it in system backups.

Another thing to be aware of is how to handle deployment. After developing transformations and jobs that rely on JNDI connections, you should find some way to create JNDI connections of the same name in the jdbc.properties file located at the deployment target.

Depending on your particular scenario, you may want to assign exactly the same values to the properties as you specified in your local jdbc.properties file, ensuring that the connections you used in your development environment are exactly the same as the ones you use on your deployment platform. However, in many cases it is more likely that there are dedicated development and/or testing databases, in which case you want to set up your local jdbc.properties so that it uses your local development database. On the deployment target, your jdbc.properties file would define exactly the same JNDI names, but specify connection data corresponding to the production environment. Because the transformations and jobs only refer to the JNDI connection by name, you can transparently apply your transformations and jobs on any environment without having to change the job and transformations themselves.

kettle.properties

The kettle.properties file is a general store for global Kettle properties. Properties are to Kettle what environment variables are to an operating system shell: they are global string variables that you can use to parameterize your jobs and transformations. For example, you can create properties to hold database connection data, paths on the file system, or simply constants that play a role in your transformation or job.

You can edit the kettle.properties file using a plain-text editor. Each property is denoted on its own line as a key/value pair, separated by an equal sign. The key appears before the equal sign, and is used as property name; whatever comes after the equal sign is the property value. After defining such a property, you can refer to its value using the key name. Here's an example to illustrate what the contents of kettle.properties might look like:

```
#connection parameters for the db server
DB_HOST=dbhost.domain.org
DB_NAME=sakila
DB_USER=sakila_user
DB_PASSWORD=sakila_password

#path from where to read input files
INPUT_PATH=/home/sakila/import

#path to store the error reports
ERROR_PATH=/home/sakila/import_errors
```

With these properties in kettle.properties, transformations and jobs can refer to these values by using a notation like `${<propertyname>}` or `%%<propertyname>%%` in any configuration field of the transformation step or job entry that supports variables. For example, Figure 3-4 shows the configuration dialog of the CSV input step.

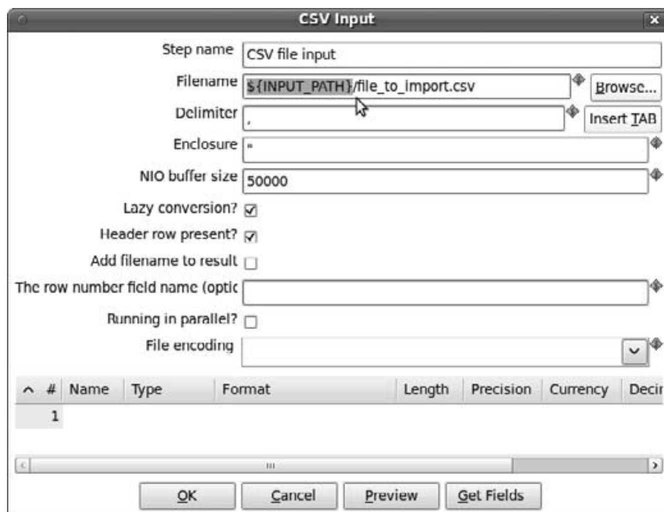


Figure 3-4: Referencing variables set in the kettle.properties file

As you can see, instead of "hard wiring" the directory in the transformation, the variable reference `${INPUT_PATH}` is used inside the value for the Filename field. You can use variables this way whenever there's a small dollar icon immediately next to the configuration dialog field. At runtime, the variable reference will evaluate to `/home/sakila/import` because this was specified in the `kettle.properties` file.

The way in which properties can be used, and even the syntax for creating them, is similar to what you saw for JNDI connection data in `jdbc.properties`. For example, you can use different `kettle.properties` files on your development and production environment, with the express purpose of managing the differences between them.

Despite the similarities between using `jdbc.properties` and `kettle.properties`, it is worthwhile to point out a few differences. First, currently JNDI can be used for database connections only, whereas properties can be used for any purpose. Also, keys for `kettle.properties` can have more or less arbitrary names, whereas the properties for JNDI connections are predefined and geared at configuring JDBC database connections.

A final point worth noting about the `kettle.properties` file is that there are a few predefined properties that are meant to configure a default repository. These settings have an effect only when you're using a repository to store transformations and jobs. The keys for these properties are:

- `KETTLE_REPOSITORY`: The name of the default repository
- `KETTLE_USER`: The name of the repository user
- `KETTLE_PASSWORD`: The password for the repository user

With these properties in place, Kettle programs will automatically use the repository identified by the value set for the `KETTLE_REPOSITORY` property and connect to it using the values set for `KETTLE_USER` and `KETTLE_PASSWORD` as credentials.

NOTE **Repositories are covered in more detail in the section about** `repositories.xml`**, later in this chapter, and in Chapter 13.**

kettle.pwd

Executing a job using the Carte service requires authentication. The exact method of authentication can be configured, but this is a specialized topic that is outside the scope of this chapter. By default, Carte is configured to support only basic authentication. For this type of authentication, the credentials are stored in the `kettle.pwd` file, which is located in the `pwd` directory beneath the Kettle home directory. By default, the contents of the `kettle.pwd` file are:

```
# Please note that the default password (cluster) is obfuscated
# using the Encr script provided in this release
# Passwords can also be entered in plaintext as before
#
cluster: OBF:1v8w1uh21z7k1ym71z7i1ugo1v9q
```

The last line is the only functional line. It defines a user called `cluster`, as well as the obfuscated password (which happens to be *cluster* as well). As the comment

indicates, the obfuscated password was generated using the `Encr.bat` or `encr.sh` script discussed previously.

If you are using the Carte service, (especially if that service is accessible from outside your private network), you should most certainly edit `kettle.pwd` and at least change the default credentials. There is no separate editor or tool available to do this: simply use any text editor and make the changes by directly editing the file.

repositories.xml

Kettle has the capability to manage transformations, jobs, and resources such as database connections using a centralized repository. If you do not use a repository, transformations and jobs will be stored in files, and each job or transformation will keep its own copy of configuration data for items such as database connections.

Kettle repositories can have a relational database as backend, or you can have plugins that utilize another data store as backend, such as a version control system like subversion. These features are discussed in Chapter 13.

To make it easier to work with repositories, Kettle maintains a list of known repositories in a `repositories.xml` file. There are two standard locations for a `repositories.xml` file:

- The `.kettle` directory beneath the user's home directory. This file is read by Spoon, Kitchen, and Pan.
- The Carte service will read the `repositories.xml` file in the directory from which Carte was started. If there is no `repositories.xml` file, it will use the default `repositories.xml` file stored in the `.kettle` directory beneath the user's home directory.

For the development platform, you are not required to edit this file manually; it is automatically maintained by Spoon whenever you connect to a new repository. But for deployment, things may be different. For each repository name referenced in the deployed transformations or jobs, an entry must be found with a matching name in whatever `repositories.xml` file is being used. And just as you saw for the `jdbc.properties` and `kettle.properties` files, it is possible or even likely that the actual repository that should be used on the deployment platform differs from that being used on the development platform.

In practice, it is most convenient to simply copy the `repositories.xml` file from the development environment and manually modify it for the deployment environment. This topic is covered in detail in Chapter 13.

shared.xml

Kettle supports a concept referred to as *shared objects*. Shared objects are things like transformation steps, database connection definitions, slave server definitions, and so on that are defined once and then reused (shared) among multiple transformations or jobs. Note that there is some overlap in functionality with repositories, which can also be used to share connections and slave server definitions. There are a few differences, however. A repository is typically a central store that can be accessed by multiple developers; as such it is an ideal way to maintain all objects pertaining to a particular project.

To share an object, activate the tree view View mode in Spoon, and locate the object that you wish to share. Right-click it, and then choose Share from the context menu. Be sure to save the file; otherwise the share is not recorded. Objects shared in this way will automatically be available in each new job or transformation you create, and can always be located in the tree view when in the View mode. However, shared steps and job entries are not automatically placed on the canvas; you'll have to drag them from the tree view unto the canvas in order to use them in your transformation or job.

Shared objects are stored in a file called `shared.xml`. By default, a `shared.xml` file is stored in the `.kettle` directory beneath the home directory of the current user. However, jobs and transformations can specify a custom location for the shared objects file. This allows you to conveniently manage and reuse all objects that are repeatedly needed in all transformations and jobs for a specific project.

For transformations as well as jobs, you can use Spoon to set the location of the shared objects file. For both jobs and transformations, this is done in the Properties dialog, which you can open via the Settings option of the main menu. For jobs, you can specify the file in the Shared objects file field on the Log tab page. For transformations, you'll find this field on the Miscellaneous tab page.

NOTE **You may use variables to specify the location of the shared properties file. For example, in a transformation you could use a location like the following:**

```
${Internal.Transformation.Filename.Directory}/shared.xml
```

This allows all transformations in the same directory to use the same shared objects file, regardless of the actual directory in which the transformation resides.

For deployment, you need to ensure that any shared object files that are directly or indirectly used in the developed transformations and jobs are also available on the deployment platform. Typically, the shared objects files should be identical for both environments, and any environment-specific settings should be taken care of using the `kettle.properties` file.

The Kettle Shell Scripts

In some cases, you may need to tweak the shell scripts that are used to launch the various Kettle programs. The most common reasons to do this are:

- To add additional entries to the Java classpath. This is necessary in case your job or transformation directly or indirectly (via plugins) refers to Java classes that are not by default accessible.
- To change the settings of the Java Virtual Machine, such as the amount of memory that it can utilize.

General Structure of the Startup Scripts

The structure of all the Kettle startup scripts is quite similar:

- An initial string is built to set the classpath later on. In this step, a number of core `.jar` files are added to the classpath.
- A string is built containing the file names of all `.jar` files residing in and below the `libext` directory.
- File names of other `.jar` files are added to the classpath. These file names are specific for the particular program that is invoked by the script. For example, the Spoon startup scripts add the names of a number of SWT `.jar` files in this step, which are used to build the Spoon graphical user interface.
- A string is built with a number of options for the Java Virtual Machine. The classpath built in the previous steps is included in that string. The option that sets the maximum heap size that can be utilized is added to the string in this step.
- The Java executable is executed using the string of options built in the previous steps. In this line the fully qualified name of the Java class that actually implements the specific Kettle program is passed to the Java executable.

Adding an Entry to the Classpath

Kettle allows you to use Java expressions in your transformations. For example, the JavaScript step allows you to instantiate Java objects and call their methods, and the User Defined Java Expression step lets you write Java expressions directly. These topics are discussed in detail in the "Scripting" section in Chapter 7.

You have to make sure that any classes that you want to use in your Java expressions are already included in the classpath. The easiest way to achieve this is to create a new directory beneath the `libext` directory located in the Kettle home directory and put any `.jar` files you want to add to the classpath in there. For the `.sh` scripts, this works perfectly because of the following lines:

```
# **************************************************
# ** JDBC & other libraries used by Kettle:       **
# **************************************************

for f in `find $BASEDIR/libext -type f -name "*.jar"`\
         `find $BASEDIR/libext -type f -name "*.zip"`
do
  CLASSPATH=$CLASSPATH:$f
done
```

These lines simply loop over all `.jar` and `.zip` files located in or below the `libext` directory. Unfortunately, the equivalent section for the `.bat` files looks like this:

```
REM Loop the libext directory and add the classpath.
REM The following command would only add the last jar:
REM FOR %%F IN (libext\*.jar) DO call set CLASSPATH=%CLASSPATH%;%%F
REM So the circumvention with a subroutine solves this ;-)

FOR %%F IN (libext\*.jar) DO call :addcp %%F
FOR %%F IN (libext\JDBC\*.jar) DO call :addcp %%F
FOR %%F IN (libext\webservices\*.jar) DO call :addcp %%F
FOR %%F IN (libext\commons\*.jar) DO call :addcp %%F
FOR %%F IN (libext\web\*.jar) DO call :addcp %%F
FOR %%F IN (libext\pentaho\*.jar) DO call :addcp %%F
FOR %%F IN (libext\spring\*.jar) DO call :addcp %%F
FOR %%F IN (libext\jfree\*.jar) DO call :addcp %%F
FOR %%F IN (libext\mondrian\*.jar) DO call :addcp %%F
FOR %%F IN (libext\salesforce\*.jar) DO call :addcp %%F
FOR %%F IN (libext\feeds\*.jar) DO call :addcp %%F
```

As you can see in the `.bat` scripts, the directories below the `libext` directory are hard-coded within the script. This means you'll have to add your own line here to ensure that the `.jar` files inside the subdirectory you added to `libext` are read and added to the classpath as well.

Changing the Maximum Heap Size

All Kettle startup scripts explicitly specify a maximum heap space. For example, in `pan.bat` you can find a line that reads:

```
REM ******************************************************************
REM ** Set java runtime options                                     **
REM ** Change 512m to higher values in case you run out of memory.  **
REM ******************************************************************

set OPT=-Xmx512M -cp %CLASSPATH% ...more options go here...
```

If you are experiencing Out of Memory errors when running certain jobs or transformations, or if the computer that you're running Java on has substantially more physical memory available than 512 Megabytes, you can try to increase the value. When changing the amount of memory, just be careful to change the number 512 only to some other integer. Other modifications, such as accidentally removing the trailing M, will lead to unexpected results that may be hard to debug.

NOTE **For the original documentation on the command-line parameters, see** `http://java.sun.com/javase/6/docs/technotes/tools/solaris/java.html`**.**

Managing JDBC Drivers

Kettle ships with a large number of JDBC drivers. A given JDBC driver typically resides in a single Java archive (`.jar`) file. Kettle keeps all of its JDBC drivers in the `JDBC` directory beneath the `libext` directory, which resides in the Kettle home directory.

To add a new driver, simply drop the `.jar` file containing the driver into the `libext/JDBC` directory. The scripts for starting Spoon, Kettle, and Pan automatically loop through the contents of this directory, and add all these `.jars` to the classpath.

NOTE **The new database driver does not become automatically available to any Kettle program that happens to be running at the time of adding the new driver. Only those `.jar` files that are present in the `libext/JDBC` directory before running the startup script will be available to Kettle.**

When you upgrade or replace a driver, be sure to also remove the old `.jar` file. If you want to keep the old `.jar` file, be sure to move it to a directory that is completely outside of the Kettle home directory, or one of its descendent directories. This ensures you can't accidentally load the old driver again.

Summary

This chapter provided an overview of the Kettle programs. We discussed how to install Kettle, and how to manage several aspects of its configuration. In particular, you learned:

- Spoon is the integrated development environment you use to create and design transformations and jobs.
- Kitchen and Pan are command-line lauchers for running Kettle jobs and transformations respectively.
- Carte is a server to run Kettle jobs remotely.
- How to install Java (which is a pre-requisite for running Kettle programs).
- How to obtain Kettle from sourceforge, and how to install it.
- How to use the Kettle scripts to run the individual Kettle programs.
- The different files that make up a Kettle's configuration.

CHAPTER

4

An Example ETL Solution—Sakila

After the gentle introduction to ETL and Kettle provided by the first two chapters and the installation and configuration guide provided in Chapter 3, it's finally time to get a bit of hands-on experience with Kettle. Using a fairly uncomplicated yet sufficiently realistic ETL solution, this chapter will give you a quick impression of Kettle's features and capabilities. In addition, you'll get just enough experience in using the Spoon program to follow through with the more complicated examples in the remainder of this book.

This chapter does not provide a detailed step-by-step instruction on how to build this solution yourself. Instead, we provide all the necessary instructions to set up this solution on your own system using transformations and jobs that are available on this book's website at `www.wiley.com/go/kettlesolutions`. As we discuss the design and constructs used in the example, we will refer to specific chapters and sections in the remainder of this book that contain more detailed descriptions of a particular feature or technique used in the example.

Sakila

Our example ETL solution is based on a fairly simple star schema that can be used to analyze rentals for a fictitious DVD rental store chain called Sakila. This star schema is based on the sakila database schema, which is a freely obtainable sample database for MySQL.

NOTE **The sakila sample database was originally developed by Mike Hillyer who was at the time a technical writer for MySQL AB. Since its release in March 2006, the sakila sample database has been maintained and distributed by the MySQL documentation team. More information about the sample database can be found in the relevant MySQL documentation pages at** `http://dev.mysql.com/doc/sakila/en/sakila.html`**. You can find download and installation instructions in that document. For your convenience, the SQL scripts for setting up the sakila database are also available on the book's website in the download area for this chapter, and this chapter provides detailed installation instructions.**

There is a port for Sakila available for PostgreSQL called pagila. This can be obtained via the dbsamples project hosted on pgFoundry at `http://pgfoundry.org/projects/dbsamples`**. You can find instructions to set up pagila at** `http://www.postgresonline.com/journal/index.php?/archives/36-REST-in-PostgreSQL-Part-1-The-DB- components.htm`**.**

The sample ETL solution described in this chapter is designed to periodically extract new or changed data from the original sakila schema, which thus acts as source database. The data is then transformed to fit the heavily denormalized rental star schema. Finally, the data is loaded into the rental star schema, which thus acts as the target database.

The following sections briefly discuss the schemas of both source and target databases. This should help you later on to understand how the sample ETL solution works and why it was built this way.

The Sakila Sample Database

For a full description of all objects in the sakila database schema, you should refer to the official documentation at `http://dev.mysql.com/doc/sakila/en/sakila.html`. However, the schema is quite simple and easy to grasp, and its essence can be easily understood when looking at the business process.

DVD Rental Business Process

The sakila database is a sample database to support the primary business process for a chain of brick-and-mortar DVD rental stores. The following list defines a few key activities of its business process to help you understand how the sakila database could support this business process:

- Each store maintains its own inventory of films for rental, which is updated by one of the store's staff members whenever customers pick up or return DVDs.
- Some descriptive data concerning the films are maintained, such as its categories (such as action, adventure, comedy, and so on), cast, rating, and whether the DVD

has some special features (such as deleted scenes and theatrical trailers). This data may be used to print out labels to physically mark DVD boxes.

- In order to become a customer, people have to register with one of the stores belonging to the chain.
- Customers can enter any of the stores and pick up one or more DVDs for rental. Customers are also expected to return previously rented DVDs within a fixed rental duration that is specified per DVD.
- Rentals must be paid for. Any customer can make a payment at any time for any rented item.

Sakila Database Schema Diagram

Figure 4-1 shows the database schema for the sakila sample database . The diagram reveals a typical normalized schema (having a respectable number of heavily inter-related tables), which is optimized for online transactional processing (OLTP).

NOTE For your convenience, the diagram is available as a Power*Architect file on this book's website. You can find the file, `sakila.architect`, in the download area beneath the folder for this chapter. The diagram was prepared with version 0.9.16 of the software.

Sakila Database Subject Areas

Figure 4-1 suggests an organization into four subject areas. This categorization allows the reader to quickly identify the key tables of the database schema along with their most important related tables. While you could think of other useful ways to categorize the tables in the database schema, the categories we suggest are:

- **Films:** Comprises the `film` table and a number of tables containing additional information pertaining to films, such as `category`, `actor`, and `language`.
- **Stores:** Has the `store` table and the related `staff` and `inventory` tables.
- **Customers:** Holds the `customer` table along with the customer-related `rental` and `payment` tables.
- **Location:** Holds the `country`, `city`, and `address` tables, which participate in the normalized storage of customer, store, and staff addresses.

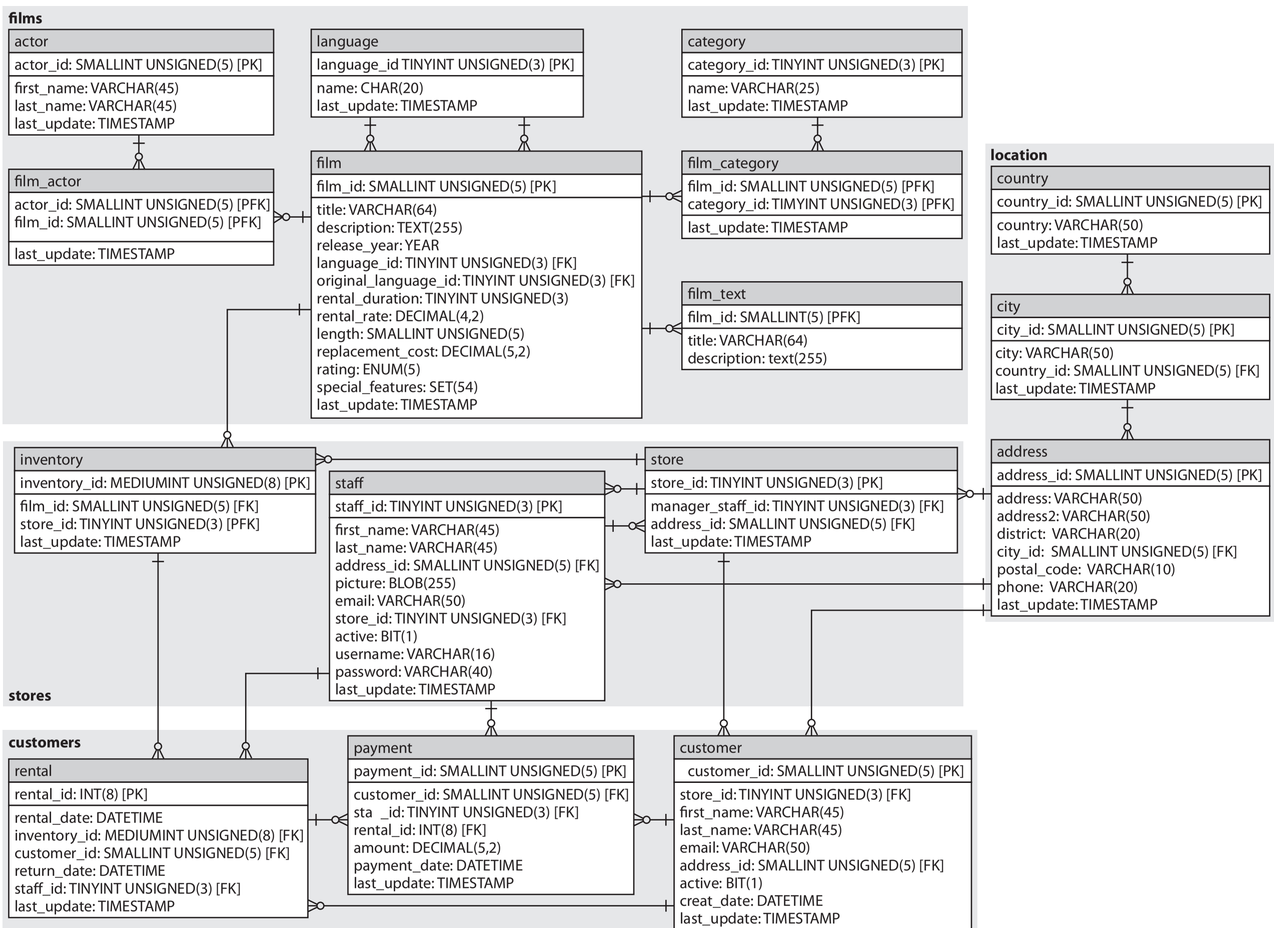


Figure 4-1: The Sakila sample database

General Design Considerations

To further understand the sakila database schema, it helps to be aware of a few general design considerations:

- The sakila schema uses singular object names as table names.
- Every table has an automatically incrementing surrogate primary key. The column that stores the key value is easily recognized as the column having a name like *<table-name>*_id. For example, the key of the film table is the film_id column.
- Foreign key constraints consistently refer to the primary key and the foreign key column keeps the name of the original primary key column. For example, the address_id column of the store table refers to the address_id column of the address table.
- Every table has a last_update column of the TIMESTAMP type, which is automatically updated to the current date and time whenever a row is added or modified.

Installing the Sakila Sample Database

You can download the sakila sample database from the MySQL documentation website. It is currently distributed as a compressed archive containing MySQL-compatible SQL script files. The archive is available in the .tar.gz format as well as the .zip format. You can find the links to the actual archives on this page: http://dev.mysql.com/doc/index-other.html. Alternatively, you can download these archives from this book's website in the download folder for Chapter 4.

Before you can install the sakila sample database, you should first install a recent version of the MySQL RDBMS software. You'll need at least version 5.0, but we recommend the latest stable version. For instructions on obtaining, installing, and configuring MySQL, please refer to the official MySQL documentation at http://dev.mysql.com/doc/refman/5.1/en/installing.html.

After you have installed the MySQL RDBMS software and downloaded the sakila sample database archive, refer to the sakila setup guide at http://dev.mysql.com/doc/sakila/en/sakila.html for detailed installation instructions. Alternatively, use the following instructions:

1. Uncompress the .tar.gz or .zip archive containing the SQL script files. You should now have two files, sakila-schema.sql and sakila-data.sql.
2. Connect to MySQL using the mysql command-line client. Be sure to log on as a user that has the appropriate permissions to create a new database.
3. Run the sakila-schema.sql and sakila-data.sql scripts by using the SOURCE command. For example, if you unpacked the script in /home/me/sakila, you could type:

```
mysql> SOURCE /home/me/sakila/sakila-schema.sql
```

and then:

```
mysql> SOURCE /home/me/sakila/sakila-data.sql
```

The Rental Star Schema

The rental star schema is directly derived from the original sakila sample database. It is just one example of a number of possible dimensional models that focuses on the rental business process.

Rental Star Schema Diagram

A diagram of the rental star schema is shown in Figure 4-2.

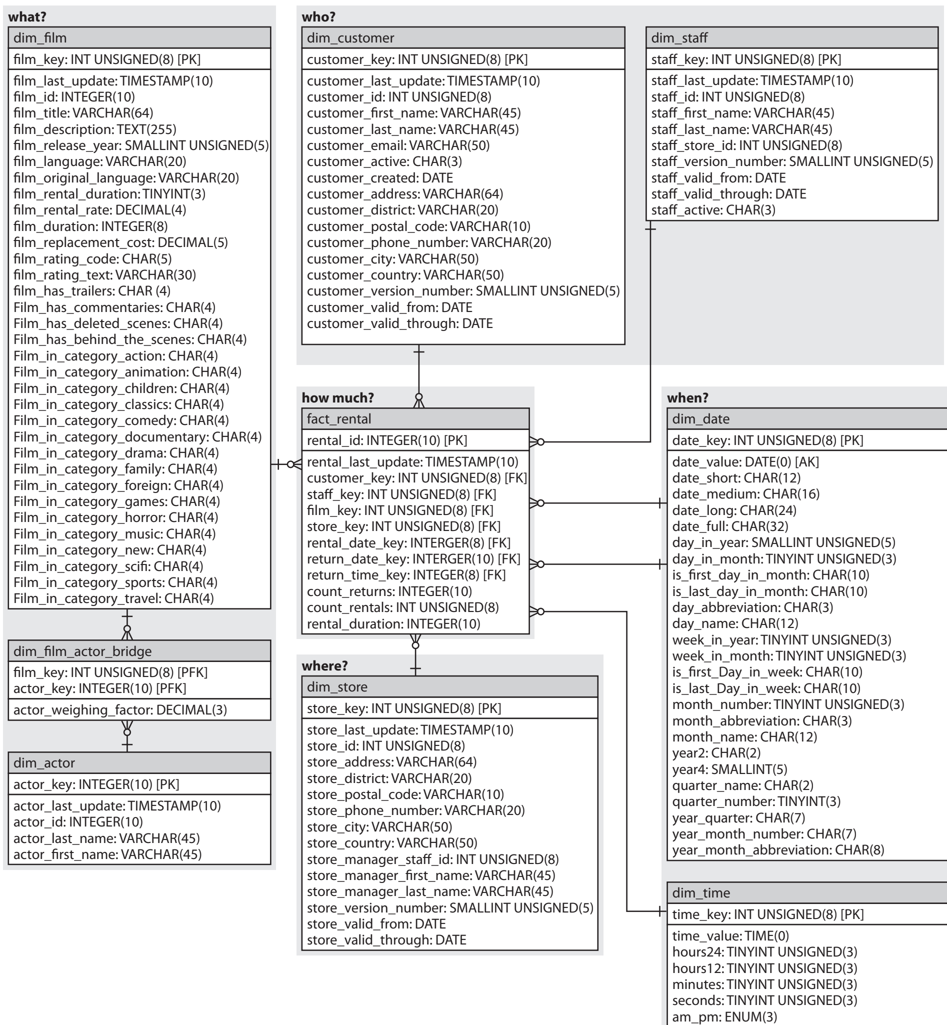


Figure 4-2: The rental star schema

NOTE **If you want to examine the star schema more closely yourself, you can find a copy of the Power*Architect file on this book's website, located in this chapter's folder. It's called** `sakila-rental-star.architect`**. The diagram was prepared with SQL Power Architect version 0.9.16. The version is important, as older versions of the software may not be able to load the file.**

Figure 4-2 reveals a typical dimensionally modeled database schema, having one central fact table called `fact_rental`, which is related to multiple dimension tables. A dimensional model like this is optimal for online analytical processing (OLAP) requirements. It is also a typical star schema because almost all dimensions are modeled as a single, denormalized dimension table that is only related to the fact table and not to the other dimension tables.

NOTE **There is one notable exception to the dimension tables not being related to each other: the** `dim_actor` **and** `dim_film` **tables are related to each other with a so-called** ***bridge table*** **called** `dim_film_actor_bridge`**. The purpose of this construct is described in the subsection "Dimension Tables" later in this chapter.**

The following sections describe the elements found in this star schema and their purpose.

Rental Fact Table

The fact table is called `fact_rental` and contains a few columns to store the quantitative metrics pertaining to the performance of the rental business process (`count_returns`, `count_rentals`, and `rental_duration`). In addition, it contains a set of columns to refer to the keys of the dimension tables. The dimension table rows identified by these key columns provide the context for the state of the business at the time when particular values for the metrics were obtained.

The `fact_rental` table corresponds directly to the original `rental` table in the sakila schema: One row in the `rental` table generates one row in the `fact_rental` table.

Dimension Tables

We previously explained that the rental star schema models each dimension as a single dimension table. The dimension tables all follow the naming convention `dim_`*`<dimension-name>`*, where *`<dimension-name>`* is a descriptive name for the subject of the dimension.

Analogous to how we categorized the sakila database diagram (Figure 4-1) in a number of subject areas, Figure 4-2 suggests that the dimension tables are organized into four groups containing conceptually related dimensions (plus a fifth group, "how much?," that is in the fact table):

- **who?:** In this group, we find the dim_customer and dim_staff dimension tables, which represent the customer and the staff member who participate in a rental. Both dimensions are modeled as type 2 slowly changing dimensions: the special %_version_number, %_valid_from, and %_valid_through columns serve to distinguish between multiple records pertaining to the same physical customer or staff member. This allows one to track the history.
- **when?:** This group contains dimension tables that mark the point in time when a rental or a return occurred. The dim_date dimension represents the calendar day. This is a so-called role-playing dimension because it is used to mark both the rental as well as the return date. The dim_time dimension is used to identify in which moment during the day a rental occurred.
- **where?:** The dim_store dimension is used to identify from which store the DVD was rented. Like the dim_staff and dim_customer dimension tables, this is also a type 2 slowly changing dimension, having a set of columns to keep track of the different store record versions through time.
- **what?:** This group contains the dim_actor and dim_film dimension tables, which have the role of being the subjects of a rental. Only the dim_film table is directly related to the fact_rental table because the film is the actual object that is rented or returned. But because a film has a cast, consisting of multiple actors, the actors are in a sense also a subject of the rental. This is where the so-called bridge table dim_film_actor_bridge comes into play, which relates actors to films. In addition, it stores a weighting factor that can be used to attenuate the values of the metrics in the fact table proportional to how much a particular actor contributed to a film. By multiplying the raw metrics with the weighting factor, one can analyze rentals in terms of actors and still treat the attenuated metrics as if they were additive. For example, one can sensibly answer the question: In the last month, how many rentals did we have for films with Robert De Niro or Al Pacino (or both)?

In the case of the rental star schema, the derivation of dimension tables from the original sakila schema is so straightforward that each dimension table (save for the dim_date and dim_time tables) corresponds directly with one table in the original sakila schema. For example, the dim_store dimension table corresponds with the original store table, and the dim_actor table corresponds with the original actor table.

Keys and Change Data Capture

With the exception of dim_date and dim_time, the dimension tables each have their own automatically incrementing column that serves as the surrogate primary key. The dim_date and dim_time tables also have a key but as you will see later, this is a so-called *smart key*. These smart keys are directly derived from parts of the date and time value respectively, which has some practical advantages for the ETL process, as well as for partitioning the fact table.

The key values of the dimension tables are used by the `fact_rental` table to refer to the dimension tables. For any given dimension table, the key column is called `<dimension-name>_key`, where `<dimension-name>` is the dimension name without the `dim_` prefix that is used in the table name.

When we discussed the general design considerations of the sakila database schema, we mentioned that every table in the source schema has a `last_update` column, which is used to store a `TIMESTAMP` value of the last change (or addition) on a per-row basis. As you will see in the following section, the existence of these columns is extremely convenient for capturing data changes, which is necessary for incrementally loading the rental star schema. Although there are several ways to exploit this feature, the rental star schema takes a very straightforward approach: Each dimension table keeps its own copy of the `last_update` column to store the values from the `last_update` column of its corresponding table in the original sakila schema. This allows you to do a single query on each dimension table to obtain the date/time of the last loaded row, which can then be used to identify and extract all added and/or changed rows in its corresponding table in the source database.

In addition to the surrogate primary key, each dimension table also contains a column that stores the value of the primary key column from the original sakila schema. For example, the `dim_film` table in the rental star schema corresponds to the `film` table in the original sakila schema and thus has a `film_id` column to store the values from the corresponding `film.film_id` column. As you will see in the next section, these columns are extremely important to determine whether to add or update rows in the dimension tables to reflect the changes that occurred in the source database since the last load.

Installing the Rental Star Schema

You can download SQL script files for the rental star schema from this book's website. Just like the sakila sample database, the script files are archived and available as a `.zip` and a `.tar.gz` archive.

So, the installation procedure for the rental star schema is the same as for the original sakila sample database: Simply unpack the archive and use the `SOURCE` command in the MySQL command line utility to execute the scripts. The only difference with the sakila sample database are the actual file names. For the rental star schema, the files for the schema and the data are called `sakila_dwh_schema.sql` and `sakila_dwh_data.sql`, respectively.

Note that the ETL solution described later in this chapter will actually load the tables in the rental star schema based on the contents of an existing sakila schema, so you might want to consider installing only the schema and not the data. If you do load the data from the SQL script file, then the ETL solution will still run fine: It just won't find any new changes to load.

Prerequisites and Some Basic Spoon Skills

We are about to dive into the details of our sample ETL solution, and you are encouraged to follow along and examine its nuts and bolts directly—"live" so to speak on your

own computer, running your own copy of Spoon. So before you actually look at the individual transformations and jobs of the sample ETL solution, it is a good idea to first obtain the files that make up the sample solution and verify that you can open them.

In addition, for those readers who are not yet familiar with Spoon, it might be a good idea to obtain a few basic skills in working with Spoon. Working through the "Visual Programming" section in Chapter 2 is an excellent way to get started. The following section lists a few basic operations just to get you started. As the Spoon user interface is quite intuitive, we're confident that you'll be able to figure out many things about working with Spoon yourself as you go.

Setting Up the ETL Solution

You can obtain all these files from this book's website in the folder for Chapter 4. All files are available in `.zip` and `.tar.gz` archives called `ch4_ktr_and_kjb_files`. Simply download the archive, and extract it to some location on your hard disk.

Creating Database Accounts

The transformations for the ETL solution use two particular database accounts to access the sakila and rental star schema: a `sakila` account, which is used to read from the `sakila` sample database, and a `sakila_dwh` account, which is used to read from and write to the rental star schema. To create these accounts, use the `mysql` command line client to log into MySQL as a user with `SUPER` privileges, such as the built-in `root` account. Then, enter these commands:

```
CREATE USER sakila IDENTIFIED BY 'sakila';
GRANT ALL PRIVILEGES ON sakila.* TO sakila;

CREATE USER sakila_dwh IDENTIFIED BY 'sakila_dwh';
GRANT ALL PRIVILEGES ON sakila_dwh.* TO sakila_dwh;
```

For your convenience, these commands are also available as a SQL script file (`create_sakila_accounts.sql`), which is included in the `ch4_ktr_and_kjb_files` archive along with the kettle transformation and job files.

Working with Spoon

We assume you already installed Kettle and can successfully run the Spoon program prior to setting up the sample ETL solution. For instructions on how to install Kettle and run the Spoon program, see Chapter 3.

Opening Transformation and Job Files

You can open individual files in Spoon by choosing the main menu and then File ➪ Open. This will open a file browser, which you can use to navigate to the directory where you extracted the archive.

Opening the Step's Configuration Dialog

We pointed out previously that this chapter does not provide step-by-step instructions to build the ETL solution yourself. Instead, we are confident that you'll get an idea how to do this yourself by simply going back and forth between the book and the transformation in Spoon. So, if you're curious about some detail of the inner workings of the transformations, good! Simply double-click on any step of your transformation to open a dialog that reveals the specific configuration of that step.

Don't worry if you don't immediately understand everything that you might find inside these configuration dialogs. In the subsequent chapters, a great number of different steps are covered in detail, with specific examples on how to configure and use them in a practical situation.

Examining Streams

Kettle transformations are all about manipulating streams of records. Therefore, it is quite natural to want to try to understand the effect of a particular transformation step by examining the layout of its incoming and outgoing streams. To do this, right-click on the step, and choose either the "Show input fields" or the "Show output fields" option from the context menu. This will open a dialog showing the name, data type, format, originating step, and much more for all fields that together make up the stream. For an example of what this looks like, see Figure 4-3.

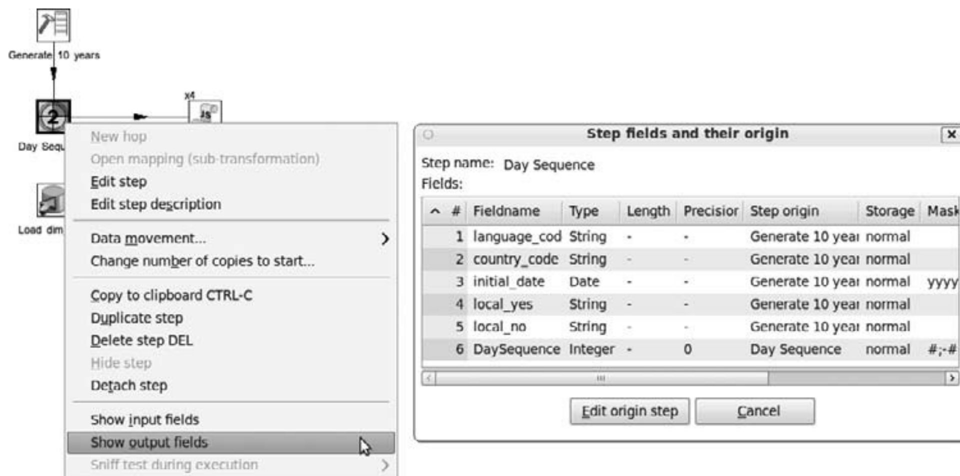


Figure 4-3: Examining the layout of incoming and outgoing streams

Running Jobs and Transformations

If you need to run a job or transformation, find the "Run this transformation or job" button (hereafter: Run button) on the toolbar. It's the button with the green arrowhead pointing to the right, which appears as the first button on the toolbar right above the

workspace. You will notice a dialog popping up, which you can use to configure all kinds of run-time settings. For now, however, you can simply confirm the dialog by pressing OK.

As soon as your transformation is running, you should see the Execution Results pane appear at the bottom of Spoon's workspace. You can see what it looks like in the bottom right side of Figure 3-2 in the preceding chapter. In this pane, use the "Step Metrics" tab page to get an overview of which step is doing what, and how fast it is doing it. This is most useful for troubleshooting performance problems. Use the Logging tab for a text-based log output. This is most useful for debugging and troubleshooting errors. These and other tabs are discussed in depth in Chapters 12 and 15.

While the transformation is running, the Run button will appear to be disabled. Instead, the Pause and Stop buttons located immediately to right of the Run button will be enabled, allowing you to suspend or abort the running transformation. For now, ignore these buttons; simply use them as an indicator that the transformation is still running and hasn't finished yet.

The Sample ETL Solution

Now that we have described the respective database schemas, we can examine how the ETL solution manages to load the data from the sakila sample database into the rental star schema. In the remainder of this section, we will describe each job and transformation that we used to load the rental star schema.

Static, Generated Dimensions

The `dim_date` and `dim_time` dimension tables are static: They are initially loaded with a generated dataset and do not need to be periodically reloaded from the sakila sample database (although one might need to generate more data for the `dim_date` table at some point to account for dates in the far future).

In Kimball's 34 ETL subsystems framework, covered in Chapter 5, dimensions such as time and date belong to the domain of the *Special Dimensions Manager*. Although that term may suggest a specialized device that "manages the special dimensions," it is better to think of it as a conceptual bucket to classify all dimension types that cannot be neatly derived from the source system. In the case of Kettle, there is nothing special about loading date and time dimensions: You simply create a transformation that generates whatever data you feel will fit the requirements and load it into its respective dimension table(s).

Loading the dim_date Dimension Table

The transformation file for loading the `dim_date` dimension table is called `load_dim_date.ktr`. Figure 4-4 shows what it looks like in Spoon.

A brief description of the steps in Figure 4-4 appears in the next few paragraphs.

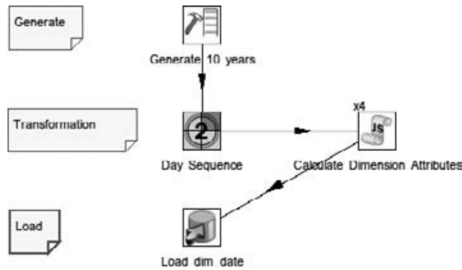


Figure 4-4: The load_dim_date transformation

Generate 10 years

The transformation works by first generating about 10 years worth of rows ($10 \times 366 = 3660$) using a step of the Generate Rows type, which is labeled "Generate 10 years" in Figure 4-4. The generated rows have a few fields with constant values, and one of these constants is the initial date, which was set at `2000-01-01`. Other constants include the language and country code.

Day Sequence

The generated rows are fed into the Day Sequence step in Figure 4-4. This step is of the Sequence type, and its purpose is to generate an incrementing number for each row passed by the incoming stream. In the subsequent step, this sequence number will be added to the initial date in order to generate a series of consecutive calendar dates.

Calculate Dimension Attributes

The Calculate Dimension Attributes step is the heart of the transformation shown in Figure 4-4. This is a step of the Modified Java Script Value type. In this step, the field values from the stream incoming from the Day Sequence step are bound to JavaScript variables, which are then used to compute various representations of the date, in different formats.

One of the first computations performed in the Calculate Dimension Attributes step is the addition of the sequence number and the initial date, yielding the calendar day. Then, subsequent JavaScript expressions are applied to the calendar day result to yield different formats that represent the date or a part thereof. In addition to generating all kinds of human-friendly date formats, a JavaScript expression is also used to generate the smart key used to identify the rows in the `dim_date` table.

Some of these JavaScript expressions use the language and country code (originally specified in the initial "Generate 20 years" step) to achieve locale-specific formats. For example, this allows the date `2009-03-09` to be formatted as `Monday, March 9, 2009` if the country code is `gb` and the language code is `en`, whereas that same date would be formatted as `lundi 9 mars 2009` if the country code is `ca`, with `fr` as language code.

This Calculate Dimension Attributes step illustrates two more points of interest. You may have noticed that this step is adorned with a little "x4" label, which appears at the left top of the square icon. This label is not part of the step itself; it is a result from spawning multiple copies of this step. You can specify the number of copies by right-clicking

on a step and choosing "Change number of copies to start" from the context menu. The benefits of doing this are explained in detail in Chapters 15 and 16.

Load dim_date

The final step of the transformation is a "Table output step" labeled "Load dim_date." This step accepts the rows from the stream incoming from the "Calculate Dimension Attributes" step, and generates and executes the appropriate SQL command(s) to insert rows into the `dim_table` dimension table.

Running the Transformation

If your database is running and you set up the `sakila_dwh` schema and corresponding user account, you should be able to simply run the transformation and load the `dim_date` table. The transformation should complete in a matter of seconds on a reasonably idle modern laptop or desktop computer.

Loading the dim_time Dimension Table

The transformation file for loading the `dim_time` dimension table is called `load_dim_time.ktr`. The transformation is shown in Figure 4-5.

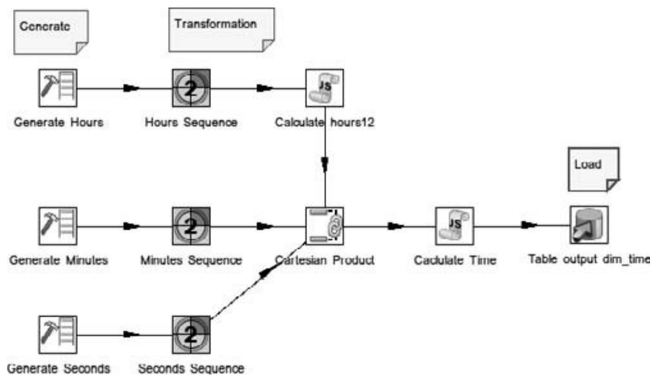


Figure 4-5: The load_dim_time Transformation

The `load_dim_time` transformation introduces one new type of step: In addition to steps of the types Generate Rows, Sequence, Modified Java Script Value, and "Table output," it also uses a step of the type "Join rows (cartesian product)." Again, we briefly discuss the key components of the transformation in the paragraphs that follow.

Generating Hours, Minutes, and Seconds

Just like the `load_dim_date` transformation shown in Figure 4-4, the `load_dim_time` transformation initially uses the Generate Rows step type to generate the data. However, in this case, three instances of the Generate Rows step type run in parallel:

- The step labeled "Generate hours" generates 24 rows to represent the number of hours per day.

- The step labeled "Generate minutes" generates 60 rows, representing the number of minutes per hour.
- The step labeled "Generate seconds" also generates 60 rows, but these represent the number of seconds per minute.

Each of the Generate Rows steps is immediately followed by its own step of the Sequence type to actually obtain a series of consecutive integer values representing 0–23 hours, 0–59 minutes, and 0–59 seconds respectively. In the hours branch, the sequence step is followed by the Modified Java Script Value step labeled "Calculate hours12" to obtain a 2×12 hour representation of the hour. In addition, this step also calculates the AM/PM indicator to complement the 12 hour notation.

Cartesian Product

The following point of interest in the `load_dim_time` transformation is the "Cartesian product" step. This step is an instance of the "Join rows (cartesian product)" step type, which joins all incoming streams and creates one outgoing stream that consists of the Cartesian product of the incoming streams. That is, it will generate all possible combinations of all rows found in the incoming streams, and for each combination, emit one record containing all the fields found in the incoming streams.

It is possible to restrain steps of the "Join rows (cartesian product)" step type to generate only specific combinations of rows, but in this case generating all combinations is intentional, leading to a total of 24 * 60 * 60 = 86400 rows, each representing one second in a 24-hour day. (This is actually not entirely correct because it does not take any leap seconds into account. For now we will ignore this minor issue and move on.)

Calculate Time and Table out dim_time

The "Cartesian product" step is followed by another Modified Java Script Value type step and a Table Output type step. Functionally, these steps are analogous to the last two steps in the `load_dim_date` transformation: The values in the incoming stream are modified using JavaScript expressions to generate all desired representations of the time of day, including the smart key of the `dim_time` dimension table, and then this data is inserted into the dimension table, finalizing this transformation.

Running the Transformation

Here, the same instruction applies as to the `load_dim_date` transformation. Assuming your database is running, and you properly set up the `sakila_dwh` schema and database account, you should be able to run this transformation and load the `dim_time` dimension table within a couple of seconds.

Recurring Load

In the previous section, we discussed the construction and load of the special `dim_time` and `dim_date` dimension tables. Because loading these tables is virtually a one-shot operation, you shouldn't consider them to be truly part of the ETL process proper.

In this section, we cover the process of extracting any data changes that occurred in the sakila schema since the last load of the rental star schema (if ever), and then transforming that data to load the dimension tables and fact table of the rental star schema.

The load_rentals Job

The entire ETL procedure for the rental star schema is consolidated into one single Kettle job called `load_rentals.kjb`. This does all the work of updating and maintaining the dimension tables and loading the fact table. You can open jobs in Spoon via the main menu, just as you can with transformations. Figure 4-6 shows what the job looks like in Spoon.

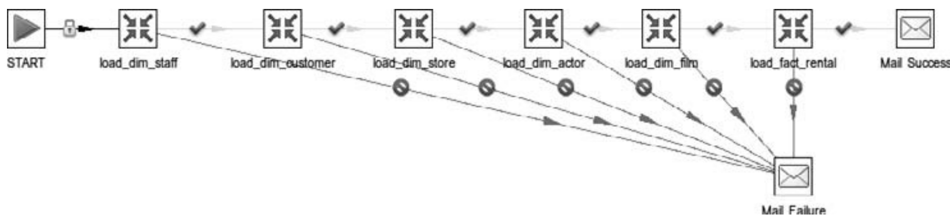


Figure 4-6: The load_rentals job

We discuss the key elements of this job in the remainder of this section.

Start

The first element in the load_rentals job shown in Figure 4-6 is the job entry labeled START. Every valid job has exactly one START job entry, which forms the entry point of the entire job: This is where job execution begins.

Various Transformation Job Entries

The START job entry is connected to a sequence of transformation job entries. The hop between the START job entry and the first transformation job entry is adorned with a little lock: This symbol indicates an unconditional hop, which means that the execution path of the job as a whole will always follow this path, even if the preceding job entry did not execute successfully. Later in this section, you learn about other job hop types.

The transformation job entries that follow the START job entry all refer to a particular transformation that performs a distinct part of the process. For example, the `load_dim_staff` job entry executes the `load_dim_staff.ktr` transformation (which loads the `dim_staff` dimension table), and the subsequent `load_dim_customer` job entry executes the `load_dim_customer.ktr` transformation (which loads the `dim_customer` dimension table), and so on and so forth.

To see exactly which transformation file is associated with a particular transformation job entry, double-click the job entry. A dialog titled "Job entry details for this transformation" will pop up and allow you to configure the job entry. For example, double-click the `load_dim_store` job entry (the third transformation shown in Figure 4-6), and you'll get a configuration dialog like the one shown in Figure 4-7.

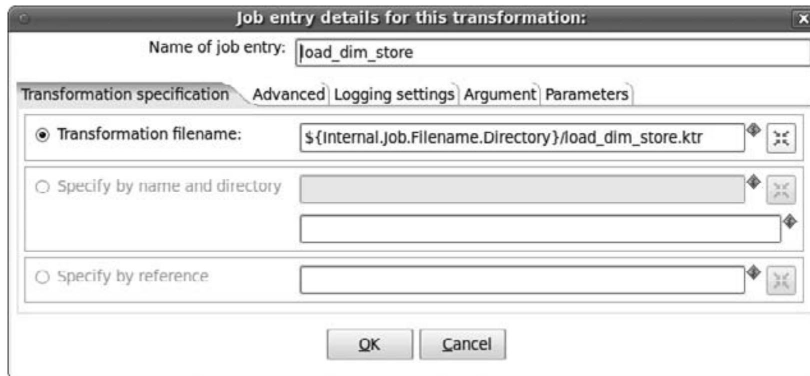


Figure 4-7: The configuration dialog for the load_dim_store transformation job entry

Note that in Figure 4-7, the field "Name of job entry" contains the value `load_dim_store`. The value of this field directly corresponds to the label shown for that job entry on the canvas shown in Figure 4-6. Also note that in Figure 4-7, the "Transformation filename" field has the value:

```
${Internal.Job.Filename.Directory}/load_dim_store.ktr
```

The first part, `${Internal.Job.Filename.Directory}`, indicates a built-in Kettle variable that evaluates at runtime to whatever directory the current job resides in. Together with the remainder of the value, `/load_dim_store.ktr`, this will form the file name of an existing transformation file.

NOTE **Variables are an important tool to make your transformations and jobs more independent of particular resources such as files and servers. The example shown in Figure 4-7 illustrates this well: Instead of hard-wiring the exact path on the file system in the job entry, the location can be specified relative to the location of the current job itself using the** `${Internal.Job.Filename.Directory}` **built-in variable.**

Variables were briefly discussed in Chapter 3 and will be discussed in more detail throughout this book.

You should realize that the occurrence of `load_dim_store` in the "Transformation filename" fields is completely unrelated to the value used for the "Name of job entry" field. That is to say, you are free to choose whatever name you like to label the job entry. For the load_rentals job example, the choice was made to let the job entry names correspond directly and unambiguously with the name of the associated transformation file.

To open the transformation itself, right-click the transformation job entry and choose the "Open transformation" option from the context menu. We recommend that you keep the load_rentals job open while working through this chapter, as this is an easy and convenient way to browse through all the relevant transformations.

Different Types of Hops and Flow of Execution

In the load_rentals job, each transformation step entry has a success hop: This is the hop that is adorned with a green checkmark and points straight ahead to the following job entry. Each transformation job entry also has an failure hop, which is adorned with a red Do Not Proceed icon and points down to a Mail Failure job entry.

When you run the job, the job entries are executed in a serial fashion: First, the `load_dim_staff` job entry will be started and execute its transformation. If that finishes successfully, the job will follow the "success" hop and proceed to the `load_dim_customer` job entry, which will in turn execute its associated transformation, and so on, until the last transformation is executed successfully and the job proceeds with the final Mail Success job entry. Because there are no job entries beyond the Mail Success job entry, the job will finish here.

Of course, it is also possible that a particular job entry does not execute successfully. In this case, the job will follow the error hop and proceed to the Mail Failure job, thereby prematurely finishing the job.

Serial Execution

Note that the serial execution of the job entries is quite different from what happens when executing a transformation: In a transformation, all steps are started at once and execute simultaneously, processing rows as they are queued at the input side of the step, and emitting them on the output side.

The serial execution of job entries is an excellent way of synchronizing work that must be done according to a particular flow. For example, in this particular job, the START job entry is first followed by a sequence of job entries (`load_dim_staff` through `load_dim_film`) that load a dimension table. The very last transformation job entry (`load_fact_rental`) is responsible for loading the `fact_rental` fact table. This ensures that the dimension tables will always be loaded with whatever new dimension rows are required before the fact table is loaded with any rows that attempt to reference these new dimension rows.

Mail Job Entries

The Mail job entries were briefly mentioned when we discussed the different types of hops. The significance of the Mail job entries in this transformation is to always provide some sort of notification to a system administrator about the final status of a job: Either all transformation job entries succeed, and a success message is sent, or one of them fails, ending the job prematurely and sending a failure notice. (Sending notifications only in the event of failure would be a bad idea because it would then be impossible to distinguish between successful execution and a broken mailserver.)

To witness the effect of the Mail job entries, you should configure them so the entries know which mail server to use and how to authenticate against the mail server (if required). To configure the mail entries, simply double-click them, and fill out the appropriate fields in the Addresses and Server tab pages. If you can't figure out exactly what to fill in, then don't worry at this point—you should still be able to run this job and load the rental star schema without properly configuring these job entries. You just won't receive a notification e-mail to inform you about the status of the job.

Running the Job

Running jobs is done in exactly the same way as running transformations. Running the entire job may take a little while, but will typically finish in less than a minute. You do not need to worry about restarting the job, or running the individual transformations called by this job afterward, as it should simply detect that there aren't any changes to process if the rental star schema is up-to-date.

The load_dim_staff Transformation

The first transformation to be executed by the load_rentals job is called `load_dim_staff`. You can either open the transformation from the context menu of the `load_dim_staff` job entry within the load_rentals job, or open the `load_dim_staff.ktr` file directly using the Spoon main menu. Figure 4-8 shows the design of the `load_dim_staff` transformation.

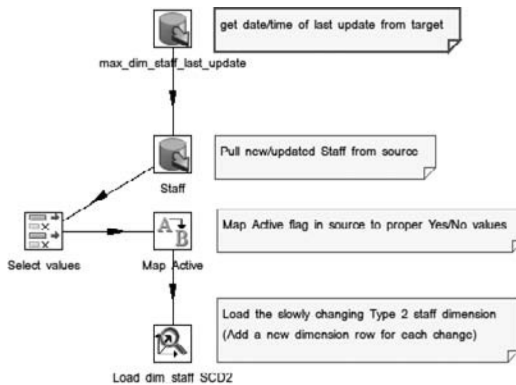


Figure 4-8: The load_dim_staff transformation

The purpose of this transformation is to load the `dim_staff` dimension table. We will discuss exactly how this is done in the remainder of this section.

Database Connections

All transformations included in the load_rentals job define two distinct database connections: one is called sakila, which points to the sakila sample database (the source); the other is called sakila_dwh and points to the rental star schema database (the target). Note that these connections correspond directly to the two database accounts that were set up in the "Prerequisites and Some Basic Spoon Skills" section earlier in this chapter. If you haven't set up these two database accounts yet, now is a good time to revisit the subsection on "Creating Database Accounts" before you proceed.

You can see these database connections in the left pane tree view if you switch to View mode and then expand the "Database connections" folder. Figure 4-9 shows an example of what this may look like.

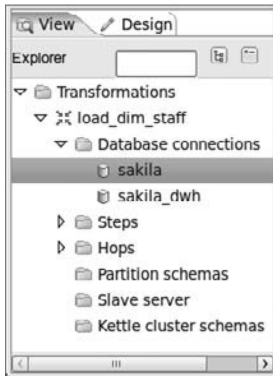


Figure 4-9: The database connections in the sidebar

To examine or modify the definition of a database connection, double-click it in the tree view. The Database Connection dialog will pop up. For example, double-clicking the sakila connection should pop up a dialog like the one shown in Figure 4-10.

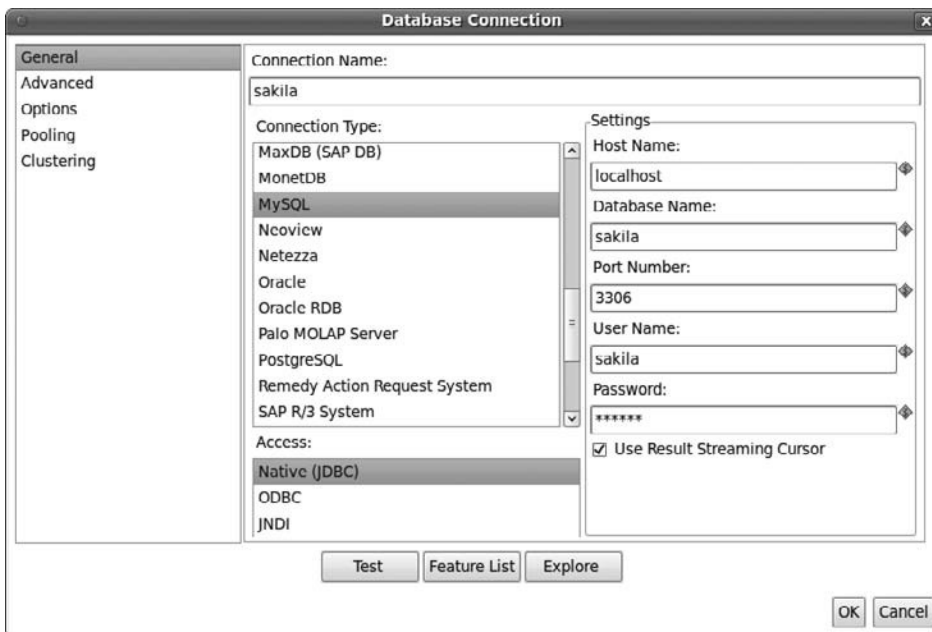


Figure 4-10: The Database Connection dialog for the sakila connection

As you can see in Figure 4-10, quite a lot is going on in this dialog. Because this is one of the most frequently used dialogs in Kettle, we will briefly discuss some of the elements in this dialog.

In the far left side of the dialog, you can select from among the categories of configuration tasks. Typically, you can do almost everything you need from the General category (which is the selected category in the dialog shown in Figure 4-10), but in some cases, you need to visit the others, too. For example, in Chapter 17 you learn how to use the Clustering page to define a cluster of database servers.

On the right side of the dialog, you first see a "Connection name" field. This is used to provide a unique name for this connection. This name will then be used throughout the job or transformation to refer to this connection.

Immediately beneath the "Connection name" field are two columns. The right column contains a Connection Type list and an Access list. Use the Connection Type list to select the RDBMS product that you would like to connect to. In the unlikely event that you find that none of the 30-something list entries match your database server, you can use the "Generic database" option to specify an arbitrary JDBC connection.

From the Access list you can select which interface should be used to connect to the database. In the vast majority of cases, you should be able to use Native (JDBC) access. In the rare case that you cannot obtain a JDBC driver for your database, you can try to establish an ODBC connection. ODBC connections are implemented through the Sun JDBC/ODBC bridge.

NOTE **In addition to JDBC and ODBC, you can also use JNDI connections. JNDI connections are technically also JDBC connections, but defined using the Java Naming and Directory Interface. Configuring JNDI connections is explained in more detail in Chapter 3.**

The left side of the dialog contains a Settings pane. The Settings pane displays a number of fields that must be used to specify the actual connection details. It is hard to define exactly which fields will be listed here, as it is entirely dependent upon the selections made in the Connection Type list as well as the Access list.

Changed Data Capture and Extraction

The first two steps of the load_dim_staff transformation are of the "Table input" type. Steps of this type can execute a SQL statement against a preconfigured database connection and turn the results retrieved from a RDBMS into an outgoing record stream.

The `max_dim_staff_last_update` step uses the connection to `sakila_dwh` to execute a query like this:

```
SELECT  COALESCE(
            MAX(staff_last_update),
            '1970-01-01'
        )                   AS max_dim_staff_last_update
FROM    dim_staff
```

The intention of this query is to obtain a single row from the `dim_staff` dimension table that returns the most recent date/time when either an update or an insert was performed on the `staff` table in the sakila source schema. This works because the `dim_staff.staff_last_update` column is filled with values coming from the

`staff.last_update` column, which in turn are automatically generated whenever an insert or an update occurs on the respective row.

The "Table output" step labeled Staff is used to extract staff member data. It executes a query like this against the sakila schema:

```
SELECT  *
FROM    staff
WHERE   last_update > ?
```

The question mark (`?`) in this query denotes a placeholder for a value. It is used in the `WHERE` clause to select only those rows from the `staff` table for which the value of the `last_update` column is more recent than the value of the placeholder.

The value for the placeholder is supplied by the previous `max_dim_staff_last_update` step, which we know yields the most recent date/time of the staff rows that were already loaded into the `dim_staff` dimension table. Effectively, this setup ensures that the outgoing stream of the Staff step will only contain rows that are newer than the already loaded rows, thus capturing only the changed data.

Capturing changed data is a major topic in any ETL solution, and recognized as ETL subsystem 2. It is discussed in more detail in Chapters 5 and 6.

Converting and Recoding the Active Flag

The next two steps in the transformation are "Select values" and "Map active." Together, these serve to convert a flag of the Boolean data type that indicates whether the staff member is currently working for the Sakila rental store chain into a textual Yes/No value, which is more suitable for presentation in reports and analyses.

Steps of the "Select values" type can be used to perform general manipulations on the incoming stream, such as removing fields, renaming fields, converting field values to another data type, and optionally applying a particular output format. The "Map active" step is of the "Value mapper" type. As the name implies, steps of this type can be used to look up a particular output value depending on the value of a particular field in the incoming stream, thus mapping a value.

The process of converting and recoding data is one example of conforming data. This is recognized as ETL subsystem 8 and is described in more detail in Chapters 5 and 9. Both the "Select values" step and the "Value mapper" step offer functionality that is very useful for cleaning and conforming activities.

Loading the dim_staff Type 2 Slowly Changing Dimension Table

The final step in the `load_dim_staff` transformation is loading the `dim_staff` table. This table is a type 2 slowly changing dimension: Any changes in the source database of the original row result in the addition of another version of the dimension row in the target database.

All rows that represent different versions of the same staff member can be grouped through the original `staff_id` key value. In addition, the dimension table has a pair of `staff_valid_from` and `staff_valid_through` columns, which are used to indicate to which period in time that particular version of the `staff` row applies. As a bonus,

there's also a staff_version_number column, which maintains the version number of the row.

Managing slowly changing dimensions is ETL subsystem 9 (Slowly Changing Dimension Processor), which is further explained in Chapter 5. Kettle offers a specialized "Dimension lookup / update" step, which makes maintaining a type 2 slowly changing dimension a relatively easy task. The exact usage of this particular step is beyond the scope of this chapter, but will be fully explained in Chapter 8.

The load_dim_customer Transformation

The load_dim_customer transformation is associated with the second transformation job entry in the load_rentals job. Its purpose is to load the dim_customer dimension table. The transformation is shown in Figure 4-11.

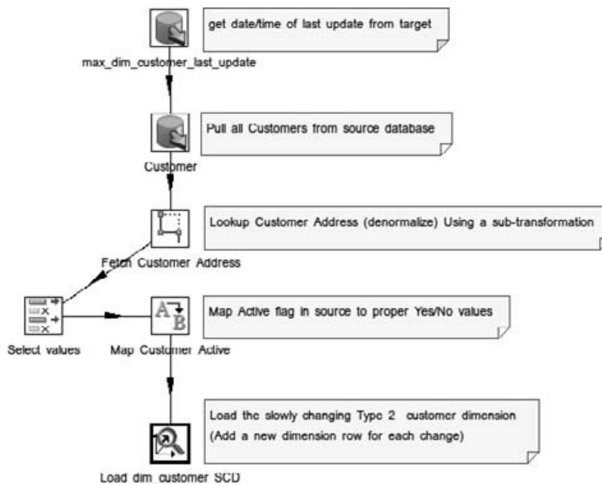


Figure 4-11: The load_dim_customer transformation

The load_dim_customer transformation has virtually the same structure as the load_dim_staff transformation shown previously in Figure 4-8:

- The transformation starts with two steps of the "Table output" type to capture the changed data and extract it.
- The transformation ends with a "Dimension lookup/update," which serves to maintain the type 2 slowly changing customer dimension, and load the date into the dim_customer dimension table.
- Before loading the data into the dim_customer dimension table, the values from the active column in the customer table are recoded into an explicit Yes/No field.

The Fetch Customer Address SubTransformation

The `load_dim_customers` transformation contains one extra kind of step in addition to the steps it has in common with the `load_dim_staff` transformation shown in Figure 4-8. The extra step is labeled Fetch Customer Address and is of the "Mapping (subtransformation)" type. Steps of the "Mapping (subtransformation)" type allow the reuse of an existing transformation.

To examine to which reusable transformation a step of the "Mapping (subtransformation)" type refers, simply double-click it to open its configuration dialog. Figure 4-12 shows an example of the configuration of the Fetch Customer Address step in the `load_dim_customers` transformation. To load the transformation itself, simply right-click the step and choose the "Open mapping (subtransformation)" option from the context menu.

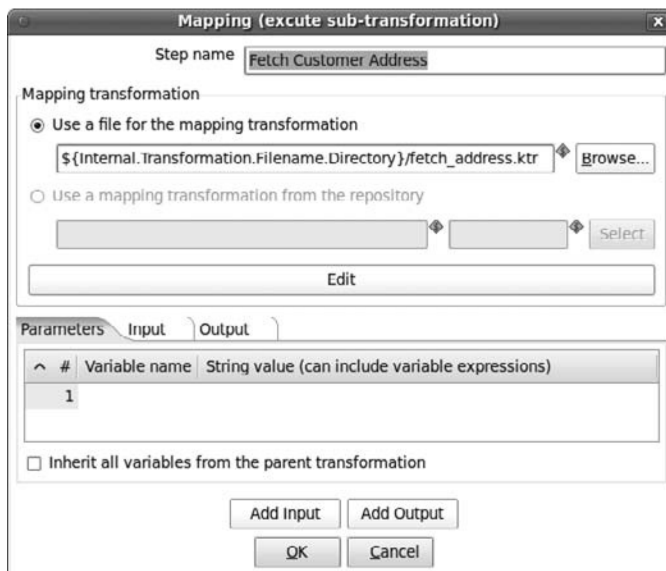


Figure 4-12: The configuration of the Fetch Customer Address step

As you can see in Figure 4-12, the Fetch Customer Address step refers to the `fetch_address.ktr` file. This transformation is described later in this chapter. Steps of the "Mapping (subtransformation)" type allow variables to be used to refer to a particular transformation file, just like transformation job entries (refer to Figure 4-7).

NOTE **Note that in this case, the variable is slightly different from the one used in transformation job entries. Instead of** `${Internal.`**`Job`**`.Filename.Directory}`**, the Fetch Customer Address step uses** `${Internal.`**`Transformation`**`.Filename.Directory}`**. At runtime, this evaluates to the directory of the current transformation—that is to say, the directory wherein the** `load_dim_customer.ktr` **file resides.**

Possible confusion may arise from the fact that the variable that was used earlier as shown in Figure 4-7, `${Internal.Job.Filename.Directory}`, could also have been used for the Fetch Customer Address step. However, this would have evaluated to the location of the `load_rentals.kjb` job file because this is the context job that is calling the `load_dim_customers` transformation.

In our setup of the sample ETL solution, it would not have made a difference as long as the `load_dim_customers` transformation was called from the `load_rentals` job because the job file resides in the same directory as the transformation files. However, it is good to be aware of the possibility that the variables `Internal.Job.Filename.Directory` and `Internal.Transformation.Filename.Directory` do not necessarily refer to the same location. You can experience this first hand when running the `load_dim_customers` transformation directly (that is, not calling it from a job): In this case, the variable `Internal.Job.Filename.Directory` would not have been initialized because it is not applicable, whereas the `Internal.Transformation.Filename.Directory` variable would still refer to the directory of the current transformation file.

Functionally, the purpose of the Fetch Customer Address step is to look up the customer's address. This is necessary because of the denormalized design of the `dim_customer` dimension table, which contains the customer's address data coming from the `sakila.address` table as well as its related `sakila.city` and `sakila.country` tables. The `dim_store` dimension table follows the same denormalized design, and because the same tables are involved, this is an excellent opportunity for reuse.

NOTE **The effect of a step of the "Mapping (subtransformation)" type is similar to the way some programming languages allow another source file to be included in the main source code file, with similar benefits—a piece of logic that is required multiple times needs to be built and tested only once, and any maintenance due to changing requirements or bug fixes can be applied in just one place.**

Often, reuse barely needs justification: It can mean enormous savings in development and maintenance time as compared to the most likely alternative, which is duplication of transformation logic. But to be fair, reuse also has its price, as you have a greater responsibility to design a stable interface for the reusable component, and modifying the reusable component could negatively impact the total ETL solution in many places instead of just one.

In the particular case of the rental star schema, you could argue that needing to look up addresses twice was wrong in the first place: If the star schema design had been a little more relaxed, and you allowed for a central conformed location dimension table, the `dim_customer` and `dim_store` dimension tables could have been snowflaked and could use what Kimball refers to as a *location outrigger*, essentially normalizing those dimensions that need location data by referring to a single conformed location dimension table. Basically, this is

another form of reuse, but at the level of the data warehouse instead of in the ETL procedure.

The question of whether to normalize or to denormalize is excellent tinder to spark a religious war between various BI and data warehousing experts, and we do not want to side with either approach at this point. Rather, we just want to use this opportunity to mention that Kettle has the option of reusing transformation logic, and that this can be powerful in fighting duplication of logic. Whether using this tool to solve this particular problem in the rental star schema is appropriate is a question we leave for the reader to answer.

The load_dim_store Transformation

The load_dim_store transformation has a structure that is very similar to that of the load_dim_customer transformations. The purpose of the load_dim_store transformation is to load the dim_store type 2 slowly changing dimension table. The load_dim_store transformation is shown in Figure 4-13.

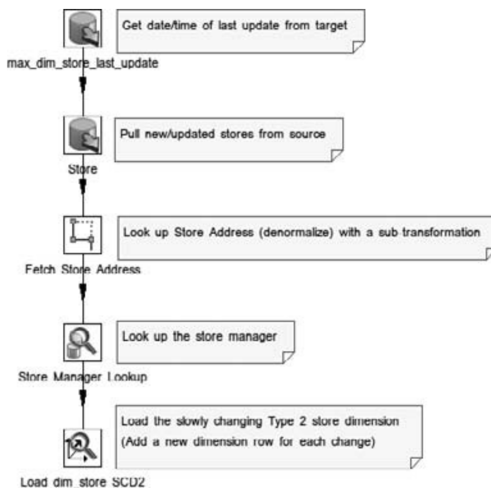


Figure 4-13: The load_dim_store transformation

The mechanism for capturing changed data is exactly the same as the one used in the load_dim_staff and load_dim_customer transformations, as is the usage of the "Dimension lookup / update" step type for the task of actually loading the table and maintaining the dimension history. The obvious difference is that in this case, the source table is store and the target table is dim_store. And just like the load_dim_customer transformation, the load_dim_store transformation uses a "Mapping (subtransformation)" step to call upon the fetch_address transformation to fetch the store's address data.

The load_dim_store transformation also introduces one step type we haven't encountered before: the Store Manager Lookup step, which is of the "Database lookup"

type. This step type is used to look up values in a database, based on a SQL query. The SQL query is parameterized using values from the incoming stream, and typically these parameters map to a primary key or unique constraint defined at the database level. The outgoing stream contains all the fields present in the incoming stream, plus the fields that are added by the lookup. This can be used for a number of purposes:

- **Cleaning and conforming data:** Similar to a Value Mapper step, a "Database lookup" step can be used to clean and conform data, the difference being that the lookup values for the Value Mapper step have to be entered as literals, whereas the lookup values for the "Database lookup" step are sourced from a relational database.
- **Denormalization:** Similar to how the join operator in a SQL statement pulls together columns from multiple tables, the "Database lookup" step combines the fields from the incoming record stream with the columns fetched by the database query.
- **Dimension key lookups:** When loading fact tables, the "Database lookup" step can be quite useful to find keys in a dimension table based on the natural keys present in the main stream.

NOTE Kettle offers other lookup steps, which offer functionality beyond the simple "Database lookup" step, such as the "Dimension lookup / update" step and the "Combination lookup / update" step, which can be used for lookups of a more advanced nature such as for a type 2 slowly changing dimension or maintaining a junk dimension. This subject is covered in detail in Chapter 8.

Thus, the "Database lookup" step is a useful general-purpose tool, which can be used in several of the ETL subsystems described in Chapter 5: subsystem 8 (Data Conformer), subsystem 14 (Surrogate Key Pipeline) and subsystem 17 (Dimension Manager System).

Functionally, the Store Manager Lookup in the `load_dim_store` transformation serves to denormalize the data sourced from the `store` table in the source database, which is necessary before the data can be loaded into the `dim_store` dimension. So in this case, the "Database lookup" step is used as a dimension handler (subsystem 17).

The fetch_address Subtransformation

We just mentioned how the `load_dim_customer` and `load_dim_store` transformations use a "Mapping (subtransformation)" step to call the `fetch_address` transformation file as part of the main transformation. You can load the transformation either directly from the main menu by opening the `fetch_address.ktr` file, or you can use choose the "Open mapping (subtransformation)" option in the context menu of either the "Fetch Customer address" or "Fetch Store address" steps in the `load_dim_customer` or `load_dim_store` transformation, respectively. The `fetch_address` transformation is shown in Figure 4-14.

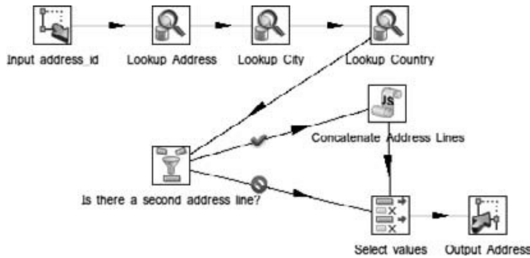


Figure 4-14: The fetch_address transformation

The remainder of this section describes the key elements in this transformation.

Address Data Lookup Cascade

The fetch_address transformation consists mainly of a series of steps of the "Database lookup" type. In Figure 4-14, these are the steps labeled as Lookup Address, Lookup City, and Lookup Country. In a sense, these steps form a *lookup cascade*: The Lookup Address step fetches a row from the address table based on the value of an address_id field in the incoming stream, and as a result of the lookup, a city_id field (among others) is added to the outgoing stream. In turn, the Lookup City step uses the value of the city_id field to fetch the corresponding row from the city table, and the city fields, including the country_id, are added to the outgoing stream. Finally, this country_id is used to fetch a row from the country table. By this time, the outgoing stream has gained all address data corresponding to the value of the address_id field in the incoming stream of the Lookup Address step.

More Denormalization: Concatenating Address Lines

In the sakila database, the address table contains two columns for storing multiple-line addresses: address and address2. The dim_customer and dim_store dimension tables only support a single address column, so something must be done when dealing with a multi-line address.

The fetch_address transformation contains a step of the Filter type labeled "Is there a second address line?" that splits the incoming stream of addresses into two outgoing streams: one stream having multi-line addresses, and one stream having only single-line addresses.

The multi-line addresses follow the "true" outgoing stream, which is the top stream in Figure 4-14. You can recognize it because the corresponding hop is adorned with a checkmark. These addresses are then further processed using JavaScript, concatenating both address lines into a single one.

Both the processed multi-line addresses as well as the single-line addresses are then led into the "Select values" step. In this transformation, the "Select values" step has the task of selecting only those fields that should be exposed to the calling transformation. For example, this step discards all the intermediate key fields such as city_id and country_id.

While it is possible to simply expose all the fields that have accumulated in the course of the fetch_address transformation, it would be a bad idea. It would be harder for

anybody trying to call this as a subtransformation because you would have to figure out which fields are really useful and which aren't. In addition, a calling transformation may start to accidentally or intentionally rely on these fields, and when maintaining the `fetch_address` transformation you would have to be committed to ensuring that these fields remain supported, too.

Subtransformation Interface

The initial and final steps of the `fetch_address` transformation are of the "Mapping input specification" (labeled "Input address_id") and "Mapping output specification" (labeled "Output Address") types respectively. These step types are typical for any transformation that is used as a subtransformation, and they define the interface of the subtransformation.

Steps of the "Mapping input specification" type can be seen as special input steps that define the point(s) where the incoming stream from the calling transformation is to be injected into the subtransformation, and determine which fields to draw from the incoming stream. For example, the "Input address_id" step specifies it expects to receive an incoming stream containing an `input_id` field from the calling transformation. Steps of the "Mapping output specification" type can be seen as special output steps that define at which point the stream of the local transformation is to be exposed to the calling transformation as an outgoing stream.

The load_dim_actor Transformation

The `load_dim_actor` transformation is responsible for loading the `dim_actor` dimension table. Figure 4-15 shows what it looks like.

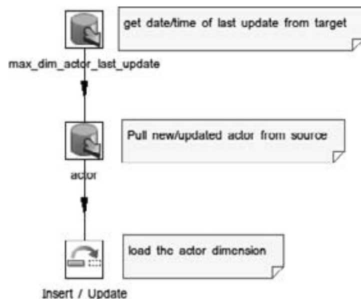


Figure 4-15: The fetch_address transformation

This is by far the simplest transformation yet. It uses the same changed data capture device as all prior transformations. It introduces one new step type: Insert / Update.

The Insert / Update Step Type

The `load_dim_actor` transformation uses an Insert / Update step to load the date into the `dim_actor` table. Steps of this type work by checking a number of fields in the

incoming stream against the key of a database table. If the database already contains a row matching the key, it performs an update, assigning the values from a number of specified fields to corresponding database columns. If the row does not already exist, it inserts those values and creates a new row instead.

For the dim_actor table, this is exactly what we want: The dim_actor dimension table is not a slowly changing dimension (at least, not type 2). We do require an update, for example, when a typo in an actor's name is corrected in the sakila source database.

The load_dim_film Transformation

The load_dim_film transformation has the responsibility of loading both the dim_film dimension table and the dim_film_actor_bridge table. The transformation is shown in Figure 4-16.

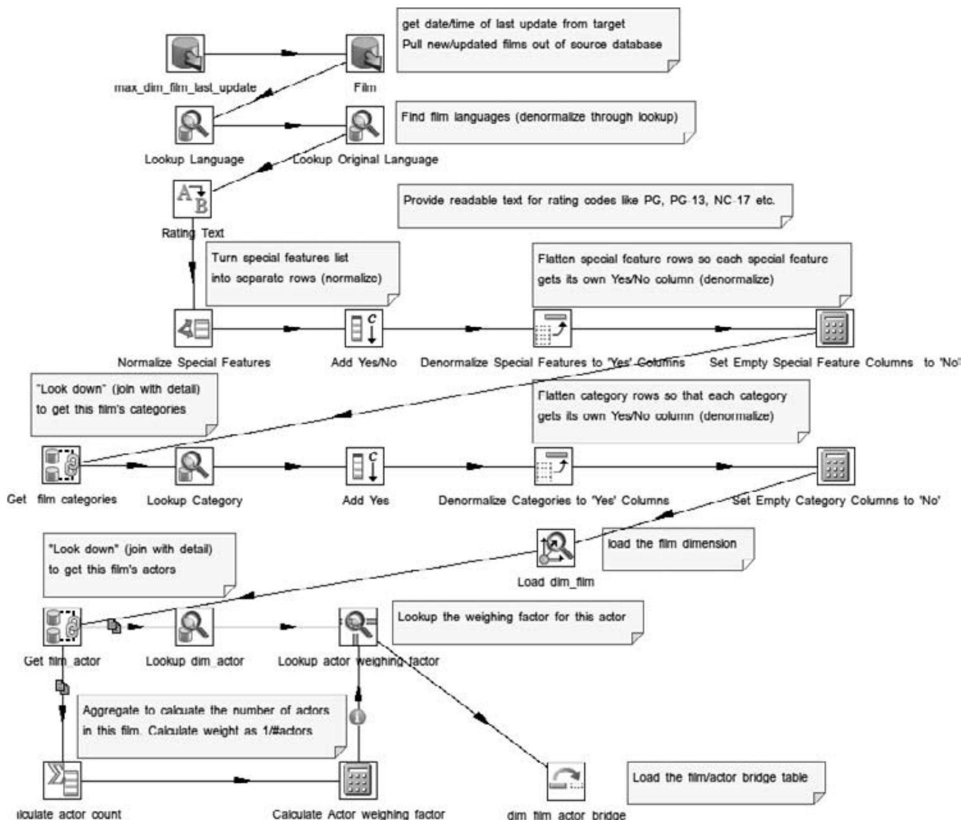


Figure 4-16: The load_dim_film transformation

The load_dim_film transformation is the most complex transformation we've shown so far. In part, this is because it loads two tables. If you look at Figure 4-16, you can see the top half, which runs from the initial max_fim_film_last_update step way

down to the load_dim_film step. This part is responsible for loading the dim_film dimension table. The remainder, starting from the "Get film_actor" step up to and including the final dim_film_actor_bridge step, is responsible for loading the dim_film_actor_bridge table. Another reason for the size and complexity is that the rental fact star schema solves several multi-valued dimension problems, which all happen to surround the dim_film table. Finally, the dim_film table itself has its share of regular denormalization issues as well, namely the references to both the film's language and the film's original language.

First, let's take a look at the elements that the load_dim_film transformation has in common with the transformations you've seen so far:

- The usual changed data capture feature is found in the initial two "Table input" steps, max_film_last_update and Film.
- Two "Database lookup" steps, "Lookup Language" and "Lookup Original Language" are used to denormalize the film's language data.
- The codes that appear in the rating column of the original film table are recoded using a Value Mapper step labeled Rating Text.

We won't discuss this part of the transformation any further as it should be rather clear how this is achieved with the techniques described prior to this section. As for the yet undescribed techniques:

- The special_features field in the original film table is a non-atomic list of values (based on the MySQL SET data type), which is transformed into a set of Yes/No flags (the film_has_% columns) in the dim_film table.
- The possible multiple categories stored in the film_category and category tables in the original sakila schema are flattened and denormalized to yet another collection of Yes/No flags (the film_is_% columns) in the dim_film table.

This concludes the summary of transformations required to load the dim_film table. As for the dim_film_actor_bridge table, there are two related but distinct things going on:

- For each row added to the dim_film dimension table, the list of actors has to be retrieved, and for each actor, the value of the actor_key must be looked up in the previously loaded dim_actor table in order to add rows to the dim_film_actor_bridge table.
- For each row that is added to the dim_film_actor_bridge table, a weighting factor has to be calculated and assigned to achieve the desired allocation of the dim_actor dimension to the metrics in the fact_rental fact table.

The following sections discuss some of these transformation techniques in more detail.

Splitting the special_features List

In order to create the flags for the special features, the comma-separated list of values stored in the original film table is first normalized to a set of rows. This allows for much more opportunity to process the values for individual special features.

The Normalize Special Features step is responsible for parsing and splitting the list of special features. This step is of the “Split field to rows” type. Steps of this type have the potential of seemingly duplicating a single row in the input string into just as many rows in the output stream as there are distinct items in the list.

For example, if one row from the incoming stream has a `special_features` list such as `'Deleted Scenes,Trailers'`, then the Normalize Special Features step will emit two rows to the output stream: one having only `'Deleted Scenes'` and one having only `'Trailers'`. Apart from the `special_feature` field, the remaining fields will be exactly identical across these two rows (thus seemingly duplicated).

Flattening Individual Special Features to Yes/No Flags

Although normalizing the list of special features to multiple rows of individual special feature values increases your opportunities for processing the individual special feature value, it also introduces a de-duplication problem. Somehow we would like to re-flatten (and thus, deduplicate) the multiple rows for each film, but now storing each individual special feature value in its own column instead of a list in a single column. This is achieved with the Denormalize Special Features to ‘Yes’ Columns step.

The Denormalize Special Features to ‘Yes’ Columns step is of the “Row denormaliser” type. It is called this because it pivots a field collection of rows into a repeating group of columns. This is best explained by examining the configuration of the Denormalize Special Features to ‘Yes’ Columns step (see Figure 4-17).

Figure 4-17: Configuration of the Denormalize Special Features to ‘Yes’ Columns step

Three items in the configuration dialog shown in Figure 4-17 are of major importance:

- **The key field:** This is the field that you wish to be denormalized into a repeating group.
- **The fields that make up the grouping:** In this grid you identify all the fields that define how the multiple rows are to be flattened. Basically, these are all the fields from the incoming stream that you want to retain.

- **Target fields:** This grid defines which fields are to be added to the outgoing stream. This is used to specify how the occurrence of unique values in the key field should be mapped to a repeating group. For example, if the first row in this grid specifies that whenever the key field happens to have the value `Trailers`, the value `Yes` should be stored in the newly created field called `tmp_has_trailers`.

NOTE The explanation of the "Target fields" in the "Row denormaliser" configuration dialog is slightly incorrect. The value that is to be stored in the new field is not simply `Yes`; it is actually whatever happens to be the value of the field called Yes. Figure 4-16 includes a step labeled Add Yes/No (immediately preceding the Denormalize Special Features to 'Yes' Columns step), which actually defines two constant fields, called Yes and No, having the respective values `Yes` and `No`.

Creating Flags for the Film Categories

In the sakila database, a film can belong to multiple categories. This information is stored in the `film_category` table, which has foreign keys to both the `film` table as well as the `category` table. In the rental star schema, the category data is stored in the `dim_film` dimension table by creating a Yes/No flag for each category. The approach to loading this data is in part akin to the solution used for the special features.

In Figure 4-16, the "Get film_categories" step is responsible for obtaining the list of categories per film. This is a step of the "Database join" type. The "Database join" step type is a close relative of the "Database lookup" step type, which you first encountered in our discussion of the `load_dim_store` transformation earlier in this chapter.

Like the "Database lookup," the "Database join" step type executes a SQL query that is parameterized using values from the incoming stream against a database. But whereas the parameters for the "Database lookup" step type are typically mapped to the columns of a primary key or unique constraint in order to look up a single row from the database, the parameters for the "Database join" step type typically map to the columns of a foreign key in order to retrieve a collection of related detail rows. Just like the "Database lookup," the "Database join" step type adds a number of specified fields from the database to the outgoing stream, but whereas the "Database lookup" step can at most emit one row to the outgoing stream for each row in the incoming stream, the "Database join" step type typically emits one row to the outgoing stream for each row returned from the database. Another way of putting it is to say that the "Database join" step mirrors the behavior of a "Database lookup" step, and performs a "Database look-down."

The effect is quite similar to the situation that you saw when denormalizing the special features in the previous subsection: The "Get film_categories" step has the potential of seemingly duplicating the incoming rows while adding the category. And just as we described for the special features flags, a "Row denormaliser" step type labeled Denormalize Categories to 'Yes' Columns is used to re-flatten the rows corresponding to one film and create a repeating group of Yes/No fields.

Loading the dim_film Table

The actual loading of the `dim_film` dimension table is achieved by the load_dim_film step. This step is of the "Combination lookup" type, which bears some similarity to the "Dimension lookup / update" step type, which you first encountered in our discussion of the `load_dim_staff` transformation much earlier in this chapter.

We discuss the "Combination lookup" step type in more detail in Chapter 8, "Handling Dimension Tables." For now, just think of this step as similar to the Insert / Update step that was used in the `load_dim_actor` transformation: It will insert a row into the `dim_film dimension` if it doesn't exist already, and update the dimension row if it does.

One difference with the Insert / Update step type is that the "Combination lookup" step type will also return the key of the dimension row. This is exactly why we chose this particular step type here: We need the key of `dim_film` rows so we can use them later in the transformation to load the `dim_film_actor_bridge` table.

Loading the dim_film_actor_bridge Table

The first step in loading the `dim_film_actor_bridge` table is to fetch the film's actors from the `film_actor` table in the sakila database. This is achieved by another step of the "Database join" type, labeled "Get film_actor." You need to obtain two more pieces of data to actually load the bridge table, and this is why the "Get film_actor" step has two outgoing hops.

First, you need to look up the value of the `actor_key` from the `dim_actor` table. This is a straightforward task for the "Database lookup" step, and in the `load_dim_film` transformation this is achieved by the "Lookup dim_actor" step.

Second, you need to calculate a weighting factor and store that together with each `film_key/actor_key` combination in the `dim_film_actor_bridge` table. Because the sakila database doesn't store any data that could help you determine the actual contribution of each actor to a film, you simply assume that each actor has an equal weight. So, to calculate the weighting for any actor of a given film, you can simply do the following:

```
Weighting factor = 1 / <#actors in this film>
```

This calculation is achieved by two steps in the `load_dim_film` transformation:

- The "Calculate actor count" step is of the "Group by" type, and does two things: Like the `GROUP BY` clause in the SQL language, it groups rows from the incoming stream based on a specified field (or collection of fields). In addition, it can also generate one or more aggregate values for the entire group. The grouped rows are then emitted to the outgoing steam, with extra fields for any aggregates calculated by the "Group by" step (where applicable).

  In this case, the rows were grouped using the `film_key`, and in addition, we seized the opportunity to calculate the number of actors (per `film_key`), which is returned to the outgoing stream in the `count_actors` field.

- The "Calculate Actor weighting factor" step is of the Calculator type and performs the division `1/count_actors` and assigns the result to the `actor_weighting_factor` field.

We just obtained the actor_key and the actor_weighting_factor, but these data are still confined to their own stream. The "Lookup actor weighting factor" step does the job of rejoining them. This step is of the "Stream lookup" type, and uses the film_key in the stream coming out of the "Lookup dim_actor" step to find a matching row in the stream coming out of the "Calculate Actor weighting factor" step, thereby conveying the actor_weighting_factor field from the "Calculate Actor weighting factor" to the outgoing stream of the "Lookup actor weighting factor" step.

After performing the "Lookup actor weighting factor" step, we gained a single stream having the film_key, actor_key, and actor_weighting_factor. This stream can be conveniently loaded into the dim_film_actor_bridge table using a simple step of the Insert / Update step.

The load_fact_rental Transformation

The load_fact_rental transformation is the final transformation that is called from the load_rentals job. All prior transformations in the load_rentals job have the purpose of loading a dimension table, and this transformation concludes the ETL solution by loading the fact_rental fact table. Figure 4-18 shows what the transformation looks like.

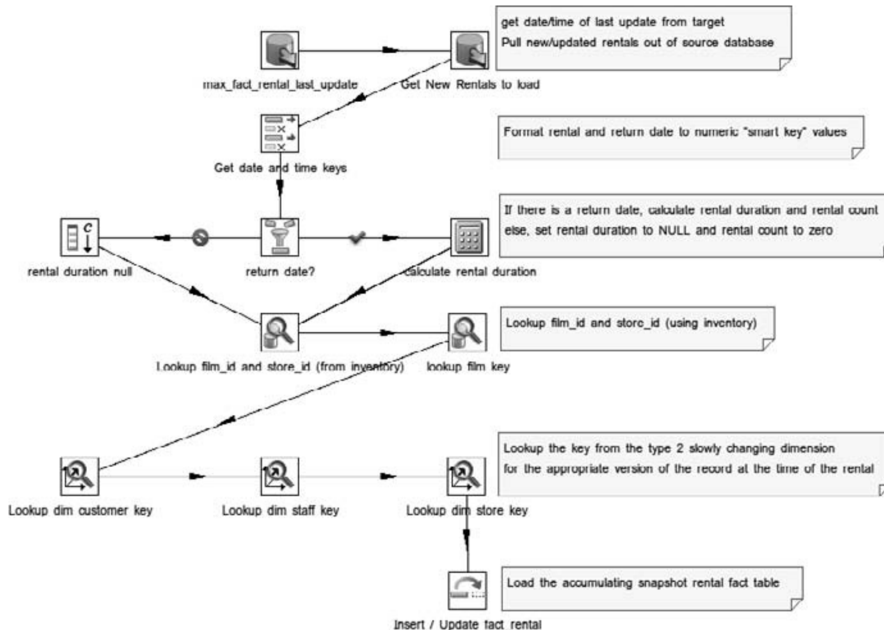


Figure 4-18: The load_fact_rental transformation

The structure of the load_fact_rental transformation is quite unlike what we encountered in any of the other transformations that are called from the load_rentals job. The other transformations are mostly occupied with various operations to denormalize data in order to load a dimension table, whereas the essence of this transformation lies

in calculating metrics and looking up corresponding dimension keys. So although the `load_fact_rental` transformation hardly introduces any new step types, it's still a good idea to describe its key elements.

Changed Data Capture

The transformation starts with the familiar pattern of two "Table output" steps to capture changed data. This is achieved by the max_fact_rental_last_update and "Get New Rentals to load" steps.

A minor difference in comparison to the other transformations is that instead of the `last_update` column, the `rental_id` column is used to detect the changes. This works just as well as the `last_update` column because the values for the `rental_id` are generated and automatically incrementing. The only drawback to using the `rental_id` is that we won't detect any updates to the `rental` table. For now, we will assume that this is not a problem.

Obtaining Date and Time Smart Keys

In our discussion of the `load_dim_date` and `load_dim_time` transformations, we mentioned that the `dim_time` and `dim_date` dimension tables use a smart key. The value of these smart keys is obtained simply by converting the various date and time values to a particular string format. This is achieved by the "Get date and time keys" step. This allows us to retrieve the keys without actually querying the dimension tables.

Calculating Metrics

The `fact_rental` table has three metrics:

- `count_rentals`: This is a so-called *factless fact*. By definition, the existence of a row in the original `rental` table, and thus in the `fact_rental` table, automatically counts as one rental. This is implemented at the database level by giving the column a default value of 1.
- `count_returns`: Depending on whether the rental is known to have been followed up by a return, this column is either `NULL` (which means that so far, the rental has not been followed up by a return) or `1` (which means that the rental is known to have been followed up by a return).
- `rental_duration`: Depending on whether the rental is known to have been followed up by a return, this column is either `NULL` (which means that so far, the rental has not been followed up by a return) or the total duration between the rental and return date, measured in seconds.

Because the `fact_rental` table keeps track of both rentals as well as returns, the status of the metrics for one particular fact row may change over time. This is an extremely simple but complete example of an accumulating snapshot fact table.

The Filter step labeled "return date?" is responsible for checking if the return date is available. The "true" path leads to a Calculator step labeled "calculate rental duration," which does the job of setting the `count_returns` metric to 1 and subtracting the rental date from the return date to calculate the value of the `rental_duration`

metric. The "false" path leads to the "rental duration null" step, which assigns `NULL` to the `count_returns` and `rental_duration` metrics.

Looking Up Keys of Regular Dimension Tables

The rental table in the sakila database has an `inventory_id` column to reference a particular inventory, rather than referring directly to a row in the `store` or `film` table. So to look up the dimension key values, we first need to look up the inventory. This is achieved by the "Lookup film_id and store_id (from inventory)" step. The `store_id` and `film_id`, which are thus obtained can now be used to look up the actual dimension keys.

Looking Up Keys of Type 2 Slowly Changing Dimension Tables

For the type 2 slowly changing dimensions, you cannot simply use the original ID column to look up a dimension row because the type 2 slowly changing dimensions can have multiple rows for one such particular key. Whenever you have to look up a row from a type 2 slowly changing dimension table, you need to know the ID as well as the context period. Only then can you try to find the correct version of the dimension row.

In Figure 4-18, this is achieved by the "Lookup dim customer key," "Lookup dim store key," and "Lookup dim staff key" steps, which are all of the "Dimension lookup / update" type. We already encountered this step type when loading the respective dimension tables in the load_dim_customer, load_dim_store and load_dim_staff transformations. But this time, this type of step is used to look up dimension keys in the dimension table, instead of loading new rows into the dimension tables. This is explained in further detail in Chapter 8.

Loading the Fact Table

The final step in the transformation is an Insert / Update step. Because this is an accumulating snapshot fact table, we need to be able to update the fact table in case the status of the rental changes (that is, when the rental is followed up by a return).

Summary

In this chapter, we covered a lot of ground. Using a fairly simple example source database we explained key features in Kettle required to load a target star schema. This chapter covered:

- The business process of the fictitious Sakila DVD rental business and the structure and purpose of the sakila sample database
- The Sakila rental star schema, and how it relates to the original sakila sample database
- Basic Spoon skills, including how to open and run jobs and transformations, several ways for setting up database connections, how to edit and review the

configuration of job entries and transformation steps, and how to examine the hops between them

- Using jobs to organize the execution flow of transformations and send notifications of success and failure via e-mail
- Obtaining database data using steps of the "Table input" type and parameterizing this step to set up a simple changed data capture mechanism
- Recoding values with a Value Mapper type step
- Loading a type 2 slowly changing dimension and looking up dimension keys
- How to reuse a transformation by calling it as subtransformation
- Loading a type 1 slowly changing dimension using a Insert / Update step
- How to normalize a single valued list into rows, and then denormalize rows into a set of columns
- Calculating aggregates and metrics

While all these things were discussed in this chapter, the focus was mainly on "the bigger picture" and subsequent chapters will discuss the underlying concepts of the example and provide detailed practical information about the many Kettle features.

Part II

ETL

In This Part

Chapter 5: ETL Subsystems
Chapter 6: Data Extraction
Chapter 7: Cleansing and Conforming
Chapter 8: Handling Dimension Tables
Chapter 9: Loading Fact Tables
Chapter 10: Working with OLAP Data

CHAPTER

5

ETL Subsystems

As surprising as it may sound, until a few years ago there was no book available that was solely dedicated to the challenges involved with ETL. Sure, ETL was covered as part of delivering a BI solution, but many people needed more in-depth guidance to help them successfully implement an ETL solution, independent of the tools used. The book *The Data Warehouse ETL Toolkit* by Ralph Kimball and Joe Caserta (Wiley Publishing, 2004) filled that gap. A bit later, the ideas of that book found their way into an article, "The 38 Subsystems of ETL," which added more structure to the various tasks that are part of an ETL project.

NOTE **The original article can still be found online at** `http://intelligent-enterprise.informationweek.com/showArticle.jhtml?articleID=54200319`**. The most recent version can be found in *The Kimball Group Reader,* article 11.2, "The 34 Subsystems of ETL," pp. 430–434 (Wiley 2010). The names of the subsystems in this book are taken from the latter reference since the names have been altered slightly compared to earlier publications.**

In 2008, Wiley published the second edition of one of the best-selling BI books ever: *The Data Warehouse Lifecycle Toolkit*, also by Ralph Kimball and his colleagues in the Kimball Group. In that book, the subsystems were restructured a second time, resulting in a slightly condensed list consisting of 34 ETL subsystems. We were fortunate to get Ralph's permission to use this list as the foundation for Part II of this book, in which we show you many practical ways that Kettle can help to implement those 34 subsystems. This chapter serves as an introduction and points to the chapters in the book where the

specific subsystems are covered. In this way, this chapter serves as a cross-reference that will help you translate the theoretical foundation found in the books written by Kimball et al. to the practical implementations using Kettle.

This chapter assumes you're familiar with the concepts behind dimensional modeling and data warehousing. If you're not, Part II of our first book, *Pentaho Solutions* (Wiley, 2009), offers an excellent introduction. The previously mentioned books written by Ralph Kimball and company are also highly recommended, as is his classic introductory work, *The Data Warehouse Toolkit, Second Edition* (Wiley, 2002).

Introduction to the 34 Subsystems

As explained in *The Data Warehouse Lifecycle Toolkit*, the 34 subsystems provide a framework that helps us understand and categorize the implementation and management of an ETL solution. Before you can start such an implementation, you need a clear understanding of the requirements, existing systems, and available skills and technology to know what is expected and what the enablers and constraints are when delivering the solution. Many of the 34 subsystems are about managing the solution, primarily because the lifetime of a system only begins when the project result is delivered. Management is one of the four main components that comprise the subsystems. These components are:

- **Extraction:** In Chapter 1, the ETL introduction, we introduced data extraction as a major challenge in any ETL effort. Subsystems 1 through 3 cover this component.
- **Cleaning and conforming:** No matter what data warehouse architecture is used, at some point the data needs to be cleansed and conformed to business requirements. In a Kimball-style data warehouse, these steps are executed before the data enters the data warehouse ("single version of the truth"). When applying the Data Vault architecture, the data is added to the data warehouse "as is" ("single version of the facts") and cleansing/conforming takes place in the subsequent steps. Subsystems 4 through 8 cover this part of the process.
- **Delivering:** Thirteen of the 34 subsystems are devoted to delivering the data to the target database. *Delivering* means more than just writing the data to a certain destination, but also covers all transformations needed to get the data in the required dimension and fact tables.
- **Managing:** As mentioned earlier, a system must be managed and monitored when it is part of the basic information infrastructure of an organization. An ETL solution is no exception to this rule, so subsystems 22 to 34 cover these management activities.

Extraction

The first step in any ETL solution is invariably the extraction of data from various sources, as we explained in Chapter 1. Of the many challenges involved with getting

access to source data, the political issues are often the hardest ones to tackle. The "owner" of a system is usually charged with preventing unauthorized users from accessing the system because of the potential for decreased performance when another process is accessing the system. Another concern is the restrictions enforced in the systems license: some ERP vendors (such as SAP or Oracle) prohibit direct access to the underlying databases on the penalty of refusing further support.

Subsystems 1–3: Data Profiling, Change Data Capture, and Extraction

As Chapter 6 is entirely devoted to the subjects of profiling, change data capture (CDC), and extraction, we won't go into much detail on these three subsystems here. In short, they provide the following:

- **Subsystem 1: Data Profiling System**—This subsystem is aimed at gaining insight into the content and structure of the various data sources. A profile provides simple statistics such as row counts and number of NULL values in a column, but also more sophisticated analysis such as word patterns. At the time of this writing, the latest development version of Kettle has only limited data profiling capabilities. Until more advanced built-in tools become available, you can use Kettle in conjunction with other open source tools to gain insight into the data you're accessing in various source systems, as you'll see in the next chapter.
- **Subsystem 2: Change Data Capture System**—The goal of this subsystem is to detect changes in source systems that were made since the last time the data warehouse was loaded. Currently, no special tools are available within Kettle to help you get a change data capture solution up and running, but a couple of techniques such as time-stamped CDC or taking snapshots can be applied without the use of any additional tools.
- **Subsystem 3: Extraction System**—The third subsystem, extraction, is meant for getting data from different sources into the ETL process. Kimball makes a clear distinction between file- and stream-based extraction, the latter not necessarily meaning a real-time stream. From a Kettle perspective, this distinction is a bit artificial because the moment the data is accessed, whether from a database, real-time feed, file, Web service or any other source, the data flows through the transformations as a stream. The only distinction that really matters is whether the source data can change during the time that the Kettle job runs. So it's not a matter of file versus stream, but more static versus dynamic. This distinction becomes very important in case of failure. As we explained in Chapter 1, any ETL solution should be designed for failure. When your source is static (as most files will be), it's easy to restart the job from the start. In case of a dynamic source such as a transaction database, it could be that the data has already changed since you started the failed job. Think, for instance, about a job that loads sales data. All dimensions have loaded correctly, but then halfway through processing the sales order lines, there's a power failure. It could well be that since the load started, a new customer with new sales transactions has been added to the source system.

Unless you fully reprocess the dimension information first, this would cause loading transaction records with unknown customer references. Depending on the kind of CDC solution implemented, recovery from these kinds of failures can be very tricky.

Cleaning and Conforming Data

The rationale behind adding steps to clean the data and conform it to business requirements before adding the content to the data warehouse is the simple fact that there isn't a single organization in the world that doesn't have a data quality problem. Organizations that have all their critical information stored in a single system are pretty rare, too; usually there are one or more systems in place to support the primary business process, with possibly separate solutions for business processes such as finance, HR, purchasing, or customer relationship management. Each of these systems stores the data in a way that is unique to that particular system. This means that in system A, the customer gender could be stored as F (for female), M (for male), and U (for unknown), whereas system B would have the same information coded as 0, 1, and NULL. All these sources need to be aligned to whatever rules apply for the data warehouse.

Subsystem 4: Data Cleaning and Quality Screen Handler System

Cleansing or cleaning data refers to fixing or tidying up dirty data that comes into the ETL process. We cannot stress enough that the most sensible location to clean this data is in the source systems where the data originates. However, there's usually a sharp contrast between the available time to develop the data warehouse on the one hand, and the time needed to execute the required data quality projects on the other. So for better or worse, you'll have to find a way to get a cleaned set of information to the end user. That doesn't mean that all efforts to improve data quality at the source are doomed to fail; on the contrary, an ETL project is preeminently suited to support data quality improvement projects. First, the profiling phase shows clearly what's wrong with the data. Second, the business rules that apply to the cleansing transformations in the ETL processes are not much different from what is needed in the source systems. And last but not least, involvement of the business users is mandatory to define those business rules because only the business users can tell you what the correct values should be.

Ideally, business users/data owners, source system developers/managers, and ETL developers can start an ongoing data quality process. In many cases, however, the biggest source for incorrect data is the users that enter data into the system.

For one of our clients, we recently developed a solution that consisted of an ETL process to read and transform source data from an operational system. The target is an inspection system where all required data are visually inspected and flagged if incorrect. The ETL transformation automatically flags common errors such as empty required fields and incorrectly formatted or missing phone numbers. On a weekly basis, the results of these quality checks are reported back to the people responsible for data entry in the source system. Although the business requirement is a 100 percent error-free data entry process, the average rate of correct data entries was below 50 percent before

the improvement project started. By making the errors visible to everyone involved, the percentage of correctly entered data went up to almost 90 percent in the first year. This example shows how a simple ETL process, combined with a few standard reports, can help to educate users and thus help in improving overall data quality.

In Chapter 7, you can read how Kettle can be used to achieve the results described in the case study along with many other scenarios for cleansing and conforming data.

Subsystem 5: Error Event Handler

The error event handler is also covered in Chapter 7. The purpose of such a handler is to record every error event that occurs during an ETL load. This enables an organization to constantly monitor and analyze errors and their causes, whether the errors are due to data quality issues, system errors, or other causes. Kimball mentions the need for a separate error event schema to capture those errors, but in the case of Kettle a separate schema isn't necessary. As you'll see in Chapter 7, Kettle already contains many features to handle error event logging out of the box.

Subsystem 6: Audit Dimension Assembler

While an error event schema is separated from all other data in the data warehouse, an audit dimension is an intrinsic part of the data warehouse tables. The audit dimension is a special table that is linked to each fact table in the data warehouse and contains metadata about the changes made to the fact table, including information about the actual event such as the exact load date and time and the quality indication for the loaded record(s). In fact, the addition of an audit dimension augments the dimensional model in such a way that many of the advantages of using the Data Vault architecture (see Chapter 19) apply to the dimensional data warehouse as well. Details of the audit dimension are also covered in Chapter 7.

Subsystem 7: Deduplication System

Deduplication is probably the hardest problem to tackle in an ETL project, even more so because most ETL tools don't offer a facility to automate this process in an easy way. In almost all cases, deduplication is about eliminating double entries of customer data, or trying to unify conflicting customer data from different source systems. Although customer data represents the biggest source for data to be deduplicated, other kinds of data can have the same problems. Any entity that can be classified as "reference data" and can contain a fairly large number of entries is prone to duplicates. Examples of these are product or supplier data.

If you're familiar with Microsoft Access, you might think that deduplication is a trivial problem that is easily tackled with the Find Duplicate Query Wizard. Well, for exact matches in key information such as a phone number or street address, the problem is indeed trivial. Unfortunately, life usually doesn't get that simple. Names and addresses are misspelled or abbreviated in different ways, phone numbers are entered incorrectly, new addresses are entered in one system but not in others, and the list goes

on and on. In many cases, fuzzy logic, pattern matching, soundex functions, and other data mining techniques must be applied to get a grip on this problem. Determining that *JCJM VanDongen* is the same person as *Dongen, van Jos* or that *Bouman RP* equals *Roland Bouman* can be a daunting task. Several advanced (and expensive) tools are available that do an excellent job at this but unfortunately Kettle isn't one of them. Nevertheless, Chapter 7 includes some pointers on how to handle deduplication from within Kettle, and of course you can always revert to a solution like DQ Guru by SQLPower, a specialized and powerful tool that can help you deduplicate reference data. DQ Guru is open source; you can find more info at `http://www.sqlpower.ca/page/dqguru`.

Subsystem 8: Data Conformer

A conforming system builds on the functionality delivered by the deduplication subsystem and the previous data quality steps. Its purpose is to conform all incoming fact records from various source systems to the same conformed dimension records. Think, for example, about an organization that has a complaints management system in place, which is likely to have its own customer database. In order to link the complaints records to sales information, the customer data from the sales system and the complaints management system need to be integrated into a single customer dimension. When loading the facts from both the sales and the complaints management system, the data needs to be linked to these unique customer records in the customer dimension table. A common approach to this problem is to keep all the natural keys from the different source systems so that during fact loads, lookups can be performed against the correct key column in the dimension table. Chapter 9 covers these fact loads and lookup functions in Kettle in more detail.

Data Delivery

Delivery of new data includes a lot more than just appending new records to a target database. First, there are many ways to update dimension tables, as reflected in the different *slowly changing dimensions* techniques. You need to generate surrogate keys, look up the correct dimension key values, make sure the dimension records are loaded before the fact records arrive, and prepare the fact records for loading. Loading facts can also be challenging because of the large data volumes, a requirement to update fact records, or both. There are also special tables and data storage options such as OLAP databases that require special attention. This is why a large number of the 34 subsystems are part of the data delivery category.

Subsystem 9: Slowly Changing Dimension Processor

Slowly Changing Dimensions (SCDs) are the cornerstone of the Dimensional Data Warehouse or Bus Architecture. Remember that a dimension is a table which contains the information needed to analyze or group facts. A customer dimension might, for instance, contain the field `city`, making it possible to summarize and compare customer sales by city. Whenever a customer moves to a different city, a change to the customer table in the

source system is made. The slowly changing dimension processor takes care of handling those changes according to the defined rules for each column in the dimension table. Basically there are three SCD types:

- **Overwrite:** Replaces the old value with the new one
- **Create new rows:** Marks the current row as "old" and sets an *end* time stamp, while creating a new record at the same time which is flagged as "current" with a new *start* timestamp
- **Add new column:** Adds a new column to the table to store the updated value while keeping the current value in its original column.

In Chapter 7 of *Pentaho Solutions*, we extended this list with the following three SCD types:

- **Add a mini-dimension:** Table attributes that change more frequently are separated from the main dimension table and stored in their own table.
- **Separate history table:** Each change is stored as an historical record in a separate table, together with the change type and change timestamp. A table like that would be useful to answer questions such as "how many customers moved from Florida to California last year?"
- **Hybrid:** A combination of types 1, 2, and 3 (1 + 2 + 3 = 6).

Chapter 8 covers the first three SCD types and explains how Kettle can be used to support those different approaches.

Subsystem 10: Surrogate Key Creation System

The ETL system needs to be able to generate surrogate keys. Within Kettle this is relatively easy because it contains a special Add Sequence step that can generate artificial keys within Kettle or call a database sequence generator. This is great for an initial load or when creating a time dimension, but not for updating dimension tables. That's where the "Dimension lookup / update" and "Combination lookup / update" steps come in. Both these steps have three ways of generating a new surrogate key value:

- Use table (actually: column) maximum + 1.
- Use a database sequence.
- Use an auto increment field.

The latter is also supported by the Table Output step.

Subsystem 11: Hierarchy Dimension Builder

Of special consideration is building and managing the hierarchies in the data warehouse. In fact, the complete name for this subsystem is *hierarchy dimension builder for fixed, variable, and ragged hierarchies*. Hierarchies are the means by which users analyze data on different levels of aggregation. A simple example of a hierarchy can be found in

the time dimension. In reality, most time dimensions contain more than one hierarchy: one for Year-Quarter-Month-Day, and one for Year-Week-Day. The time dimension is also a perfect example of a *balanced hierarchy,* where all levels have an equal depth and the same number of members. More complex hierarchies can be found in organizational structures, which often are *unbalanced* or *variable* (subtrees of variable depth) or *ragged* (equal depth but some levels have no data). For an example of the latter, think of geographic information where a hierarchy might look like Country-Region-State-City. Some countries do not have regions, states, or both, causing a ragged hierarchy. Both unbalanced and ragged hierarchies are often implemented as a *recursive* relation in a source system. Kettle offers some (but not all!) capabilities to flatten these hierarchies, as you will see in Chapter 8.

Subsystem 12: Special Dimension Builder

In addition to the slowly changing category of dimensions, a dimensionally modeled data warehouse usually contains at least one special dimension, the time dimension. Also considered special are the following types of dimensions:

- **Junk dimensions (also called *garbage dimensions*):** Contain "leftover" attributes that need to be available for analysis but don't fit in other dimension tables. Items such as status flags, yes/no, and other low cardinality fields are good candidates for putting in a junk dimension.
- **Mini-dimensions:** Used to split off fast-changing attributes from a big or monster dimension, but also useful in other cases. This is why we list mini-dimensions as SCD type 4.
- **Shrunken or rolled dimensions:** Subsets of regular dimension tables that are created and updated from their base dimension to avoid inconsistencies. Shrunken dimensions are needed when data is aggregated to accommodate for a lower level of granularity, such as when details are stored on day level but an aggregate is rolled up to a monthly level.
- **Static dimensions:** Usually small translation or lookup tables that have no origin in a source system, such as descriptions for status codes and gender.
- **User maintained dimensions:** Custom descriptions, groupings, and hierarchies not available in source systems but required for reporting. These dimensions can be of any kind; what distinguishes them is the fact that the content is maintained by users, not the data warehouse team (although this is often initially the case) or an ETL process.

Most of the dimension-loading techniques are covered in Chapter 8. User-maintained dimensions are a bit different; they also require an application to maintain the user data, a topic that is beyond the scope of this book. We do have a tip, however: Take a look at Wavemaker, an open source rapid application development tool that lets you build a maintenance screen in minutes. You can find more information at `http://dev.wavemaker.com/`.

Subsystem 13: Fact Table Loader

Before fact records can enter the data warehouse, the data must be prepared for loading. Fact table building is not a process per se; it's just listed as a separate subsystem to raise the awareness of the three different kinds of fact tables:

- **Transaction grain fact table:** Each transaction or event, such as a point of sale transaction or a call made, is recorded as a separate fact row and, as such, loaded into the data warehouse.
- **Periodic snapshot fact table:** Not every recorded action is stored in the data warehouse; rather, a "picture" of the data taken at regular intervals is stored, such as daily or monthly inventory levels or monthly account balances.
- **Accumulating snapshot fact table:** The content of the record stored in the fact table is constantly updated when new information becomes available; the data warehouse record always reflects the latest available data. Think of an order process: this is a process with several separate dates such as an order date, expected ship date, actual ship date, expected delivery date, actual delivery date, and payment date. As the process progresses, the order record is updated with new expected or actual dates. Each time the data warehouse is loaded, this new information updates the existing fact record as well.

(Please note that in the last version of the 34 subsystems documentation, the full name of this subsystem is "Fact table loader for transaction, periodic snapshot, and accumulating snapshots grains.") Fact loads usually consist of many, sometimes millions, of rows. In order to handle these loads quickly, most database systems have some kind of bulk loader, which circumvents the regular transaction engine and loads the data into the destination table directly. Sometimes, in order to further speed up the process, all the indexes on the fact table are dropped before the load starts and re-created again after the load finishes. Chapter 9 covers the different fact table load techniques in depth.

Subsystem 14: Surrogate Key Pipeline

This subsystem takes care of retrieving the correct surrogate key to use with the fact record that's being loaded. The term "pipeline" is used because the fact load looks like a process where each subsequent step performs a lookup using the column's natural key to find the corresponding surrogate key. To make this process as efficient as possible, the current distinct set of natural and surrogate keys from the dimension table can be preloaded into memory. Chapter 9 explains how this is achieved in Kettle using the "Database lookup" and "Stream lookup" steps.

Subsystem 15: Multi-Valued Dimension Bridge Table Builder

Bridge tables are needed to allow for variable depth hierarchies, such as a customer with subsidiaries and sub-subsidiaries where the organizations at each level can purchase goods. If you want to be able to roll up the data to the mother company level, you need a way to accomplish this. A bridge table can do that task. You can also use a

bridge table when there are multiple dimension entries that have a relation to a single fact or other dimension table. Think of movie ticket sales and movie actors; if you want to summarize the revenue by actor, you need a bridge table with weighing factors to distribute the revenue over all the individual actors that performed in the movie (or those that are listed on the movie poster). Kettle doesn't provide specific functionality for building and maintaining bridge tables, but in Chapter 4 we showed a small example in the Sakila transformations.

Subsystem 16: Late-Arriving Data Handler

Until now our discussion assumed that all data arrived or was extracted at the same time. Unfortunately this isn't always the case: both fact and dimension records can arrive late. For fact records this doesn't have to be a big problem; the only extra measure to be taken is looking up the surrogate key that was valid at the time of the transaction. It suffices to add extra conditions for `valid_from` and `valid_to` dates. Those fields are available in the "Dimension lookup / update" step by default. Late-arriving dimension data is a more serious problem: in this case the facts have already been loaded but the dimension information wasn't up-to-date at the time of loading. When the dimension updates finally arrive and result in a new record in case of a type 2 SCD, the newly created surrogate key should be used to update an existing fact row that contains the previous version. A variation to this is when a new fact arrives that contains a natural customer key that's not yet known in the dimension table. In that case, you need to create a new dimension record first, with all fields set to a default or dummy value and use the surrogate key from this record. Later, when the correct customer data arrives from the source system, the dummy values can be updated. Late-arriving dimension data is covered in Chapter 8, late-arriving facts in Chapter 9.

Subsystem 17: Dimension Manager System

The 34 subsystems describe the dimension manager as "the centralized authority who prepares and publishes conformed dimensions to the data warehouse community." This centralized authority is responsible for all tasks related to dimension management, and as such is more a way of organizing things. In Chapter 8, we cover how to best organize the management of dimension tables using Kettle.

Subsystem 18: Fact Table Provider System

This subsystem is another organizational approach and handles the activities involved with creating, managing, and using fact tables within the data marts. Note that subsystem 17 and 18 work together as a pair: the fact table provider subscribes to the dimensions managed by the dimension manager and attaches them to their fact tables. Chapter 9 provides more detail about this subsystem.

Subsystem 19: Aggregate Builder

For as long as databases have been used for analytical purposes, there has been an unquenchable thirst for performance. This "need for speed" led to several solutions; among these, the use of aggregate tables is the solution with the most dramatic impact. Decreasing the average response time from 30 minutes to a couple of milliseconds has always been a great way to make customers happy, and that's what aggregate tables can do. Unfortunately, having aggregate tables in place is not enough; they need to be maintained (still something Kettle can do for you), and your database needs to be aware of the available aggregates to take advantage of them. This is where there is still a sharp distinction between closed source products such as Oracle, SQL Server, and DB/2 (which all have automatic aggregate navigation capabilities), and open source databases such as MySQL, PostgreSQL, and Ingres. The only open source product that is aggregate table–aware is Mondrian, but these aggregate tables are better created and maintained using Mondrian's own Aggregation Designer. Alternatively, you can use one of the special analytical databases around such as LucidDB, InfoBright, MonetDB, InfiniDB, or Ingres/Vectorwise, or, in case of LucidDB, a combination of a fast database and the Pentaho Aggregation Designer. Generation and population of aggregation tables are one-time–only activities, however; neither LucidDB nor PAD maintains the aggregates after the data warehouse is refreshed. In Chapter 9, we discuss how you can use Kettle to do this for you.

Subsystem 20: Multidimensional (OLAP) Cube Builder

OLAP databases have a special (storage) structure that enables these cubes to pre-aggregate data when it is loaded. Some OLAP databases can only be written and not updated, so in those cases the data needs to be flushed before an updated set can be loaded again. Other OLAP databases (for example, Microsoft Analysis Services) allow fact updates but have their own loading mechanism that's not available within Kettle. Chapter 10 is entirely devoted to handling OLAP data and shows how the Palo plugin can be used to populate a Palo cube.

Subsystem 21: Data Integration Manager

This subsystem is used to get data out of the data warehouse and send it to other environments, usually for offline data analysis or other special purposes such as sending an order overview to a specific customer. In Chapter 22, we show you how Kettle can be used to make this data available on a regular basis.

Managing the ETL Environment

The final section of this overview contains the 14 ETL subsystems required for managing the environment. Because Part III of this book covers all these subsystems in depth, we provide just a brief overview here with the pointers to the respective chapters.

- **Subsystem 22: Job Scheduler**—The Community Edition of Kettle doesn't have its own scheduler but relies on external schedulers such as the Pentaho BI Scheduler or `cron`. Chapter 12 contains everything related to scheduling and logging of jobs.
- **Subsystem 23: Backup System**— Backing up the intermediate data obtained and created during ETL processing should be part of your ETL solution. Ralph Kimball recommends staging (backing up) the data in three places in the ETL pipeline: 1) immediately after extracting, before any modifications are made to the data; 2) after cleaning, deduplicating, and conforming while possibly still in flat file or normalized data formats; and 3) after final preparation of the BI-accessible data sets. Backing up the data warehouse itself is usually not the responsibility of the ETL team but you should work closely together with the DBAs to create a failure-proof solution.
- **Subsystem 24: Recovery and Restart System—**An important part of ETL design is being able to restart a job when it fails somewhere during the process. Missing or duplicate entries need to be avoided at all cost so this subsystem is quite important. Following the strategy described in the previous subsystem makes restarting a failed job a lot easier. Chapter 11 covers this aspect of the ETL design as part of the broader testing and debugging topic.
- **Subsystem 25: Version Control System, and Subsystem 26: Version Migration System from development to test to production—**There are several ways to implement a version control system; Chapter 13 covers these topics in depth. Kettle has built-in versioning capabilities, but only in the Enterprise Edition. This doesn't mean that you can't use a separate version control system such as SVN or CVS. It also doesn't mean that version management should be considered an afterthought; we'd like to repeat the note written by Ralph Kimball on this subject.

  You do have a master version number for each part of your ETL system as well as one for the system as a whole, don't you? And, you can restore yesterday's complete ETL metadata context if it turns out that there is a big mistake in the current release? Thank you for reassuring us.

  The Data Warehouse Lifecycle Toolkit, 2nd Edition,
  by Ralph Kimball et al., Wiley, 2008

- **Subsystem 27: Workflow Monitor**— Ever tried to bake a cake without using a timer or an oven thermostat? Pretty hard, isn't it? It's similar to running an ETL job without having the means to monitor the process in detail to show you exactly what's going on. How many rows have been processed and how fast were they processed? How much memory is consumed? Which records are being rejected and why? All these questions are answered by the workflow monitor, or, in Kettle terminology, the logging architecture. You can learn more about logging and the Kettle logging architecture in Chapters 12 and 14.
- **Subsystem 28: Sort System**—For some operations (such as the Kettle "Group by" and Sorted Merge steps), the data needs to be sorted first. Of course Kettle has a "Sort rows" step for this that operates in memory and pages to disk if the

dataset becomes too large, but for extremely large files a separate sort tool might be necessary. We don't cover these specialized tools but simply rely on the "Sort rows" step to do our sorting.

- **Subsystem 29: Lineage and Dependency Analyzer—**ETL systems should provide both lineage and impact analysis capabilities. Lineage works backward from a target data element and shows where it originated and what operations are performed on the data. Dependency or impact analysis works the other way around: From the viewpoint of a source column all following steps and transformations are displayed, showing the impact a change on that particular field or table will have on the rest of the system. At the time of this writing, Kettle has some capabilities to show the impact of a transformation step on a database, but full lineage and dependency analysis is still planned for a future version of Kettle Enterprise Edition. Chapter 14 covers what can be done using the current version by reading the Kettle metadata.
- **Subsystem 30: Problem Escalation System—**In case something goes wrong (and believe us, it will!), you need to be notified as soon as possible. Chapter 7 covers error handling and notification.
- **Subsystem 31: Parallelizing/Pipelining System—**To be able to process large amounts of data in a short timeframe, tasks should be able to run in parallel, maybe even with the workload spread over multiple machines. Chapters 15 and 16 cover these topics. Kettle's easy-to-use clustering capabilities are especially worth mentioning; this enables an organization to dynamically add capacity (for example, during a nightly batch load) and turn it off again when it's no longer needed (when the job has completed). Running these operations in a cloud environment such as Amazon's Elastic Computing Cloud (EC2) avoids large hardware investments while at the same time offering a large on-demand capacity at very low operational costs.
- **Subsystem 32: Security System—**Security and compliance are hot topics in modern IT environments. The data warehouse, where all of an organization's information is available in an integrated fashion, is an especially strong target for data theft. And because in a lot of cases the ETL process has direct access to the source systems, the ETL solution itself is an attractive target as well.
- **Subsystem 33: Compliance Reporter—**Most of the measures that are needed for full compliance are already covered by other subsystems. Compliance means that there should be a complete audit trail of where the data came from and what operations have been performed on it (lineage), what the data looked like when it was received into the data warehouse (timestamped backups), what the value was at each particular point in time (audit table; SCD type 2), and who had access to the data (logging). A good data modeling technique that has compliance written all over it is the Data Vault, covered in Chapter 19.
- **Subsystem 34: Metadata Repository Manager—**The goal of this final subsystem is to capture all business, process, and technical metadata related to the ETL system. An important part of this metadata is to document the system, which is covered in Chapter 11, and of course the entire Kettle architecture is metadata-driven, as discussed in Chapter 2.

Summary

This chapter introduced and explained the 34 ETL subsystems, as defined by Ralph Kimball, and linked all these subsystems to available Kettle components and relevant chapters in this book. The list of subsystems can also be viewed as the definition of ETL architecture in general: it prescribes what each subsystem should cover, not exactly how it should be implemented or exactly what the tool should do. As such, it is a magnificent list of requirements to validate any ETL solution available, not just Kettle. The four main areas that make up the 34 subsystems are:

- **Extraction:** Getting the data from the various source systems
- **Cleaning and conforming data:** Transforming and integrating the data to prepare it for the data warehouse
- **Delivering data:** Loading and updating the data in the data warehouse
- **Managing the environment:** Controlling and monitoring the correct processing of all components of the ETL solution

As this chapter also showed, there are a few tasks that are not yet fully covered by Kettle (most notably data lineage and impact analysis) but in general, Kettle is an excellent tool that can handle even the most challenging data integration task.

CHAPTER

6

Data Extraction

The first step in an ETL process is getting data from one or more data sources. As we discussed in Chapters 1 and 5, this is a demanding task because of the complexity and variety of these different data sources. In a traditional data warehouse environment, data is usually extracted from an organization's transaction systems, such as financial applications or ERP systems. Most of these systems store their data in a relational database such as MySQL, Oracle, or SQL Server. As challenging as this may be from a functional point of view (we'll take a closer look at ERP systems later in this chapter), technically it's pretty straightforward to connect to a MySQL database using a JDBC driver and extract data from it. It gets more interesting when the database isn't relational and there's also no driver available to connect to it. In those situations you often end up having the data delivered in a flat file format such as a comma-separated ASCII file. An even trickier variation to this topic is data that is owned by someone else and is stored outside the corporate firewall, perhaps by a client or vendor company. In that case, a direct connection is usually not feasible so getting flat files might be the only option. In the case of data stored on the Internet, even flat files are not an option. (Imagine yourself calling Google, asking the company to FTP you some data set on a regular basis.) As you'll see later in this chapter, Kettle provides several ways to read data from the Internet, ranging from a simple RSS reader to a Salesforce.com connector or a web services reader.

The first section of this chapter is an overview of the various components available in Kettle for extracting data. The next section is about data profiling, an important but often undervalued task in an ETL project. Data profiling shows you the structure of the data and, by calculating a set of useful statistics, gives you insight into the content and

quality of the extracted data. Kettle already contains some profiling functionality, but we'll cover a more complete tool for this task as well—eObjects.org DataCleaner.

The third section in the chapter covers Change Data Capture (CDC) and the way in which Kettle can be used to support the various CDC techniques. The last section covers the techniques that can be used to hand off the data to the next step in the ETL process.

Kettle Data Extraction Overview

The first thing a novice Kettle user will discover when starting Spoon for the first time is the fact that extraction steps are actually called *input* steps. This makes perfect sense, as these steps input data into the Kettle data stream. The second thing that strikes most people is the large number of available input steps. Although these inputs cover most of Kettle's functionality to extract data, it's not the complete set of available data handling steps. Generally the steps needed to prepare the data for reading (especially with files) are available at the job level, while the steps for the actual reading of the data are available at the transformation level. To clarify this, the following sections use a categorization of the options for handling data and data extraction that's based on the type of data extracted, not necessarily the distinction Kettle uses in the Spoon interface. Note that this is just a general overview of what's available; for detailed job and transformation step documentation and samples use the Kettle documentation available online. Just select Help ➪ Show the welcome screen from the menu bar to open the documentation home page.

File-Based Extraction

When using files in an ETL process, the distinction made in Kettle is pretty straightforward: Basic read and write operations are all available as transformation steps, whereas anything that has to do with file management (moving, copying, creating, deleting, comparing, compressing, and uncompressing) can be found in the "File management" job steps.

NOTE **It is not necessary to create a file before you can use a "Text file output" step. This step will create the file automatically if it's not found.**

Working with Text Files

Text files are probably the easiest ones to handle with an ETL tool. There's little magic involved in reading from or writing to them—they are easily transportable, can be compressed efficiently, and any plain editor can be used to view them (unless, of course, the file is several gigabytes big and you only have Windows Notepad at your disposal). Basically, there are two flavors of text files:

- **Delimited:** Each field or column is separated by a character or tab. Common terms used for these files are CSV (for Comma Separated Values) and tab-delimited files.

- **Fixed width:** Each field or column has a designated width and length. Although fixed-width file formats are very reliable, they require more work to set up the file definition. Kettle offers some visual aids with the Get Fields option in the "Fixed file input" step, but if you can choose between a delimited and fixed width format, the delimited version is the preferred solution

For both types of files, the file encoding (the character set used for the file) can be selected. UTF-8 is more or less the standard these days, but other encodings such as US-ASCII or Windows-1252 are still widely used. In order to get the correct translation of the data in your files, it is important to set the right file encoding. Unfortunately, in many cases you cannot tell what encoding is right by just looking at the file or even at the content, so it's always a good idea to have both sender and receiver of the file conform to the same standard.

TIP **There's a simple trick to reveal the file encoding: Just open the file in your favorite browser and select View ➪ Character Encoding (Firefox) or View ➪ Encoding (Internet Explorer).**

The most basic text file input available is the "CSV file input" step, which lets you use a single delimited file as input. Before you can process a file using this step (or even show the content of the file), the delimiter and fields need to be specified. If you're not sure what delimiter or enclosure is being used, you'll have to revert to a text editor to first visually inspect the contents of the file. Processing multiple files in a single load is also quite cumbersome with the "CSV file input" step and its sibling, the "Fixed file input" step.

Both these input steps are basically simplified versions of the "Text file input" step, which we think is the more powerful and preferred solution for handling text data. This step is capable of a lot more than the previously mentioned input steps, including:

- Reading file names from a previous step.
- Reading multiple files in a single run.
- Reading files from compressed `.zip` or `.gzip` archives.
- Showing the content of the data file without specifying the structure. Note that you must specify the Format (DOS, Unix, Mixed) before you can view a file because Kettle needs to know what line delimiter is used.
- Specifying an escape character. This can be used to read fields containing commas in a comma-separated file. A common escape character is a backslash, enabling the value "wait\, then continue" to be read correctly as "wait, then continue" without reading the comma as a field separator.
- Error handling.
- Filtering.
- Date format locale specification.

All this power comes at a price, however; the "Text file input" step will take up more memory and processing power than both the "CSV file input" step and the "Fixed file input" step.

EXAMPLE: PROCESSING MULTIPLE TEXT FILES

In this case study, we'll show you how a common scenario can be translated into a Kettle solution. The scenario is as follows:

- **Get the directory name from a parameter table.**
- **Specify a subset of files to be read based on a search string.**
- **Read the files.**

The example files are `custfile1.txt` **and** `custfile2.txt`**, each containing 25,000 rows of customer data. You can obtain these files from the book's companion website at** `www.wiley.com/go/kettlesolutions` **or have them generated by the Fake Name Generator on** `http://www.fakenamegenerator.com`**. To follow along with this example, download these files and put them in a separate directory. You can name this directory and the files anything you like, but to follow along make sure the file names start with** `cust`**.**

The first task is to create a new transformation and add a "Text file input" step. Before you can accept file names from a previous step, you must define the file layout, so open the step, browse to one of the files and add it to the Selected files list. Before you can show the content of the file, make sure that the correct Format (DOS, Unix, Mixed) is set in the `Content` **tab. Now you can open the file using the "Show file content" button, which will reveal the fact that there is a single header line, and the fields are delimited with the pipe (|) symbol. Use this information to set the correct values for Header, Number of header lines, Separator, and Enclosure in the Content tab.**

After defining the file layout, select the Fields tab and click Get Fields. Kettle will now try to determine what type of data is stored in the respective fields. Although Kettle does a pretty good job of guessing the field info, it's not without its flaws. Fields that start with a number (such as an address field containing values in the form `'2342 Wilwood Street'`**, or a field with values like** `'23;44;33'`**) will be identified as integer data, which is incorrect. In this example, the field** `StreetAddress` **is read as an Integer field with length 4, which needs to be changed into String with length 35. Later in this chapter we look at using data profiling as another means to determine what type of data each field contains.**

To verify that your data looks correct, click Preview rows. If Kettle appears to have captured your file layout and data types correctly, you can close this step. We continue by adding the steps needed to dynamically provide the file name information to the "Text file input" step you just created. These additional steps will precede the one you just created.

First, add a "Get file names" step. Note that you can add multiple files or directories in the "Selected files" list, just like in the "Text file input" step. For now just enter the directory name and the wildcard for matching only the customer files. This last option requires a regular expression in the form `^cust.+`**, which will search for all files starting with the string** `cust`**. Figure 6-1 shows what the step looks like after entering this information.**

EXAMPLE: PROCESSING MULTIPLE TEXT FILES

Figure 6-1: Get customer files

You can now preview the result and see whether the correct files will be found by clicking Preview rows; this will display a result similar to the one shown in Figure 6-2.

Figure 6-2: Preview of Get File Names

Next, connect the step Get File Names to the "Text file input" step. Open the latter step, mark the "Accept file names from previous step" checkbox and select Get File Names as the step to read filenames from. The field to use as a file name is, of course, `filename`. Note that you cannot use the "Preview rows" option anymore, but you can run a Preview on the step to check that it works. Also note that you could have used the same path and regular expression directly in the Text file input step, but there's a good reason for using an extra step. The Get File Names step enables you to read the directory name and file mask from a previous step, and when this information is stored in an external parameter table, you can create a very flexible solution. We don't have such a table available but you can mimic an external table by using a Data Grid step. Add a Data Grid step to the canvas and create two string fields (`source_dir` and `source_pattern`) on the Meta tab and enter the same directory and regex information used before. The Data tab should now look similar to the one in Figure 6-3.

Continued

EXAMPLE: PROCESSING MULTIPLE TEXT FILES *(continued)*

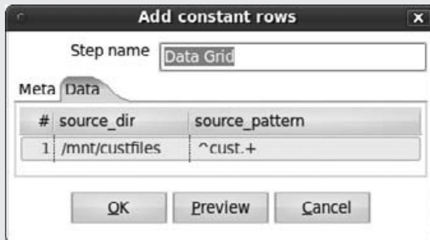


Figure 6-3: Data Grid with directory name

The "Get file names" step can now be adjusted by selecting the "Filename is defined in a field" checkbox and selecting the `source_dir` and `source_pattern` fields, as shown in Figure 6-4.

Figure 6-4: Get file names

In production environments, the Data Grid step should be replaced by a "Table input" step. Another approach could be to split the transformation in two and use a separate step to assign a variable based on a value retrieved from a parameter table. The assigned variable can then be used in a subsequent step that only needs to contain the "Text file input" step where the file name is replaced by the variable. The regex needs to be entered manually in that case.

As you can see from these examples, there are many approaches to achieve the same result. It all depends on how flexible your solution needs to be; a more flexible solution could take longer to design and build, but can be parameterized in such a way that it can run unaltered on development, test, and production environments. More on solutions design can be found in Chapter 11.

Working with XML files

XML is short for *eXtensible Markup Language* and is an open standard for defining and describing both structure and content of data in a plain text format. Although an XML extension might reveal the fact that you are dealing with an XML file, that's only part of the story. XML is a meta language that forms the basis for more specific implementations such as the OpenOffice OpenDocument format or formats like RSS or Atom. Many other systems are capable of exchanging information in an XML format, which makes XML a kind of lingua franca for data exchange. An XML file is basically just text and can be opened with any text editor such as Notepad or vi. This means that all available file management and file transfer operations can be used for XML files as well.

XML files are not only basic text files, but need to adhere to strict specifications as well. Kettle has four validation options to check whether and how an XML file can be processed:

- **Check if XML file is well formed:** Basic check of whether all opening/closing tags are complete and the nesting structure is well balanced.
- **DTD Validator:** Checks the content of the XML file based on a Data Type Definition file, which can be either internal (contained inside the XML file) or external (separate DTD file).
- **XSD Validator (Job):** Checks the content of the XML file based on an XML Schema Definition file.
- **XSD Validator (Transformation):** Same as previous, but can also check valid XML inside a specific input field such as a database column that contains XML data.

After checking the validity of the structure, the XML can be read using the "Get data from XML" input step. The main challenge when working with XML is to decipher the nesting structure of the file. The result of the step is a flattened and un-nested data structure that can be used to store the data in a relational database. Conversely, the "Add XML column" step is used to convert flat data to XML data.

If you need to transform the XML file into anything else—another XML file with a different structure, plain text, or an HTML file—you'll need the "XSL transformation" job step. XSL is short for *eXtensible Stylesheet Language*, and *XSLT* is the abbreviation for XSL Transformations, an XML language for transforming XML documents. As with the XSD validator, there is a step with the same name at both the job and the transformation level. The latter lets you apply an XSL transformation to a stream field containing XML data; the former reads in an entire XML file. An excellent resource with an in-depth explanation of XSLT including examples is the Wikipedia page `http://en.wikipedia.org/wiki/XSL_Transformations`. In order to try the examples on the web page, it suffices to save the example files to your local machine and use these in your own Kettle XSL transformation step.

Chapter 21 covers everything you need to know about XML formats and the way to read data from and write data to XML files.

Special File Types

Somewhere in between files and real databases are file formats that look like a database but really aren't. This doesn't need to cause any problems: The problems arise when people start using these file systems as if they were real databases. Kettle contains a collection of steps to deal with these file-based storage types, which we'll list here for completeness:

- **Access Input:** Many organizations have Access databases that are essential to running their operations. They typically started out as lightweight prototypes or single-user solutions, and then got "discovered" by other users and put on a network drive, where they suddenly became mission-critical solutions. Kettle contains an Access Input step that lets you retrieve data from these files, but note that the database cannot be secured (there's no way to pass username and password). Older versions such as Access 2000 are also not supported. If you need to access older or secured databases, use an ODBC connection from the "Table input" step.
- **XBase Input:** Back in the 1980s and early 1990s, dBase III, IV, and V were popular PC-type "databases." Kettle offers the XBase Input step to read data from these files, which can still be useful if you need to obtain data from, for example, scientific research. Many of these data sets are still available as DBF files.
- **Excel Input:** We cannot stress enough that you should do everything in your power to avoid having to use Excel as input. If you must, however, the Kettle step for reading those files works great and looks and operates a lot like the "Text file input" step discussed earlier.

For other, special, file types such as LDAP, LDIF, ESRI Shapefiles, and Property files, please refer to the online steps documentation.

Database-Based Extraction

With the term *database*, people usually refer to an RDBMS (Relational Database Management System) such as Oracle, SQL Server, or MySQL. In the context of data extraction, this is still a valid way to look at the database world, but things are shifting toward other approaches as well. Most notable are the "no-sql" or "not only SQL" databases such as Hadoop, Hypertable, CouchDB, or Amazon SimpleDB. An extensive overview of this category of data stores is available at `http://nosql-database.org/`. For our discussion, we'll initially stick to the more classic view of the database world, starting with the data extraction workhorse, the "Table input" step. Although this step is well documented in the online step guide, a working example showing how to work with parameterization and variable substitution is the subject of the next case study.

EXAMPLE: CREATING PARAMETERIZED QUERIES

For these examples, we'll use the Sakila database that was used in Chapter 4. In fact, one example of using parameterized queries is already described in "The load_dim_staff Transformation" section of that chapter. Basically, there are two ways to parameterize a query: using variable substitution and using parameters. The latter technique is used in Chapter 4, and boils down to the following: The step prior to the "Table input" step retrieves one or more values that are passed on to the "Table input" step, where these values are placed inside the query where question marks are located. A small but completely working solution illustrating this principle is shown in Figure 6-5.

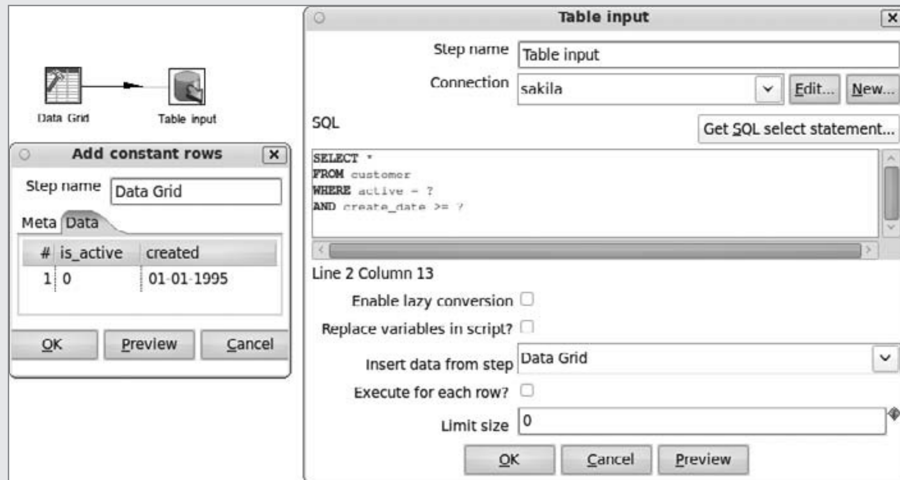


Figure 6-5: Parameterized query

The Data Grid step is used here for illustrative purposes and can be replaced by any other step that can get these parameters to the "Table input" step. You'll see another example in the CDC discussion later in this chapter.

The second technique is based on using variables. These variables have to be set in a separate transformation that must be executed prior to the execution of the transformation containing the "Data input" step; otherwise, it cannot be guaranteed that the variables get their value assigned prior to the start of the "Data input" step. Figure 6-6 shows this Set Vars transformation, which can be used as the first transformation in a job.

Continued

EXAMPLE: CREATING PARAMETERIZED QUERIES *(continued)*

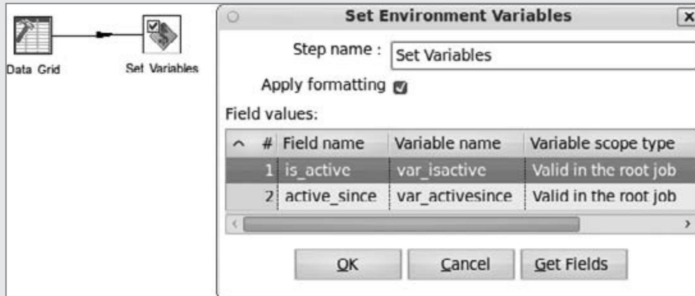


Figure 6-6: Set Variables transformation

The transformation using these set variables now contains a single "Table input" step where the variables are being replaced by the assigned values, as shown in Figure 6-7.

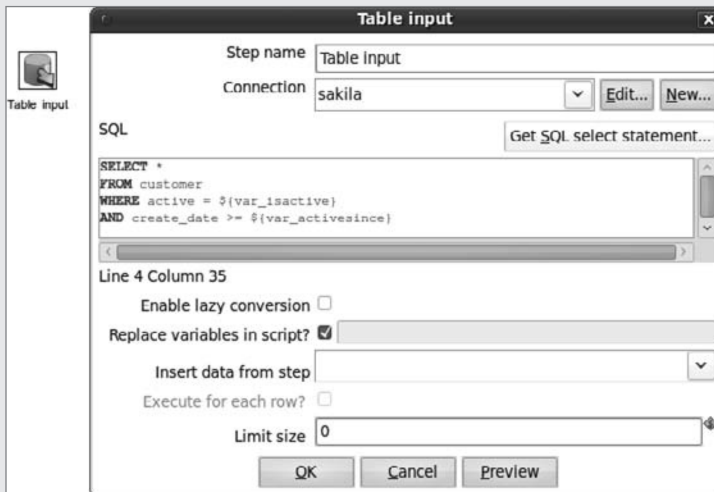


Figure 6-7: Table input with variable replacement

For completeness, the screenshot in Figure 6-8 shows the job with the respective transformations to be executed to set the variables and read the data from the database based on the parameterized query.

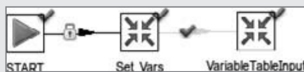


Figure 6-8: Variable-based "Table input" job

Web-Based Extraction

Getting data from the Web poses its own challenges but can sometimes be remarkably simple, too. Kettle offers various options to get data from the Web, but none of these options is a so-called "HTML scraper." This means that the data needs to be exposed in a structured manner in order to be readable by a Kettle transformation. Because Chapter 21 covers using web services and web APIs in depth, we only touch on the various options in this section.

Text-Based Web Extraction

A file can simply be put on a website and be made accessible via HTTP. If this is the case, getting the data into Kettle is a piece of cake because you simply use the URL as the file name in a "Text file input" step that uses the Apache VFS and thus handles HTTP as a file location automatically. The following URL with ISO 3166 country names and codes provides an example of such a file: `http://www.ip2location.com/download/iso3166.txt`.

HTTP Client

A second way to retrieve structured data from the Internet is to use the "HTTP client" step. This is a simple way of retrieving data because it can only make a call to a URL and the result is returned as a string. This result can be regular text that can be saved as a delimited file, or in an XML format that needs to be processed using the "Get data from XML" step. If you want to experiment with the different options for retrieving data using the HTTP client, you can use the GeoNames web services on `http://ws.geonames.org`. As an example, let's retrieve country info for the Netherlands using an HTTP client step.

First, create a new transformation and add a Generate Rows step. You need this because the HTTP client is a lookup step and needs input to be triggered; otherwise it doesn't do anything. Set the Limit to 1 and add a single field called `countryurl` of type `String`. The value of the field will be the URL that will be called, and can be used as follows:

- `http://ws.geonames.org/countryInfo` retrieves all country info in XML format.
- `http://ws.geonames.org/countryInfoCSV` retrieves all country info in CSV format.
- Adding a specific country code by using the country parameter limits the result to one specified country. For instance the URL `http://ws.geonames.org/countryInfoCSV?country=NL` will return only the information for the Netherlands in CSV format.

Next, add an "HTTP client" step and connect the two steps. For the "HTTP client" step, select the "Accept URL from field?" checkbox and select `countryurl` as the URL field name. Now you can do a preview, which will output the data similar to what you can see in Figure 6-9.

Figure 6-9: HTTP client preview

Using SOAP

SOAP, an acronym for *Simple Object Access Protocol*, is not as simple as the name implies. Rather than reviewing the ongoing discussion between advocates and opponents of the SOAP protocol, we'll limit ourselves to covering the basics needed to use a SOAP call for retrieving Internet data. You can find background information about SOAP and web services at `http://en.wikipedia.org/wiki/Web_service`.

Chapter 21 takes a close look at web data extraction, including the use of SOAP, so if you want to dig right in just move on to that chapter.

Stream-Based and Real-Time Extraction

The data integration world is becoming more and more real-time, meaning that there should be minimal delay between the occurrence of an event and the data showing up in a report or analysis. Although most people will look at Kettle as a batch-oriented ETL tool, it is in fact process-type agnostic. In Chapter 18, we show you how to retrieve streaming data from Twitter using Kettle, and in Chapter 22 you see that Kettle can also be used to deliver an extraction result in real-time data to a Pentaho Report.

Working with ERP and CRM Systems

Enterprise Resource Planning, or ERP, systems are meant to support all of an organization's business processes, from HR to procurement to production, shipping, and invoicing. The history of ERP systems started some 30 years ago when the first Manufacturing Requirements Planning (MRP) systems were designed to support manufacturing processes. Later, these first-generation MRP systems with a narrow focus were succeeded by Manufacturing Resources Planning (MRP II) solutions that covered a broader range of business processes but were still somewhat limited in scope and still mainly targeted at supporting manufacturing processes. As these systems matured and started to cover the full range of business processes, including HR, finance, and customer management, the *M* in MRP seemed a bit outdated so it was replaced by the *E* for Enterprise. This highlighted the fact that any organization could now benefit from this class of software solutions, not only manufacturing companies.

Over the past decades, many MRP and ERP vendors emerged (and some also went out of business again). The most well-known commercial solutions today with the biggest market shares and revenues are SAP/R3, Oracle E-Business Suite, and Microsoft Dynamics AX, with the former two targeting large multinational corporations while the latter is more focused on the Small and Medium Business (SMB) market. For many companies, however, ERP might have offered the breadth of solutions, but not the depth needed for specific purposes. Organizations such as Siebel and PeopleSoft (now both Oracle), Salesforce.com, and Amdocs thrived on the increased interest in specialized solutions for HR, CRM, and Sales support.

As a result of high costs and complexity of the proprietary solutions, open source alternatives soon followed; currently ERP applications such as OpenBravo, OpenERP, Adempiere, and TinyERP can in many cases compete against their high-priced commercial counterparts. One of the biggest open source success stories in the sales force automation and CRM space is SugarCRM, which offers both a community and professional/enterprise edition.

This section first explains what's special about ERP data and then shows you how the new SAP Input step can be used to obtain data from the market-leading SAP/R3 ERP system.

ERP Challenges

ERP systems such as SAP, J/D Edwards, and Lawson have two things in common: a huge collection of tables in the database and an extreme form of normalization and data encoding. This makes it hard, if not impossible, to access the underlying database directly. More often than not, the tables and columns have cryptic names such as f4211 instead of Sales Order. SAP in particular is a notoriously complex system, and the license terms of the software even prohibit direct access through a database connection. To date, a standard SAP installation consists of over 70,000 tables and most of the logic and metadata is programmed in the application, not in the database. The only way to get meaningful information out of the database is to use the interfaces that SAP offers. Almost all ERP vendors offer a solution like the one depicted in Figure 6-10, where an abstraction layer is used to access the underlying data. These abstraction layers have several advantages:

- No need to know the underlying database structure.
- Easy navigation of available components.
- The vendor can change the implementation of the database without "breaking" the extraction logic.

There could be a cost involved as well: performance. The process of calling an API that needs to do translations by looking up metadata and then accessing the database, translating the data based on the metadata, and sending it back to the calling process takes time. Nevertheless, it is usually the only way to get to the data so it's a good idea to first check how much time the extraction process takes and whether it fits in available batch windows.

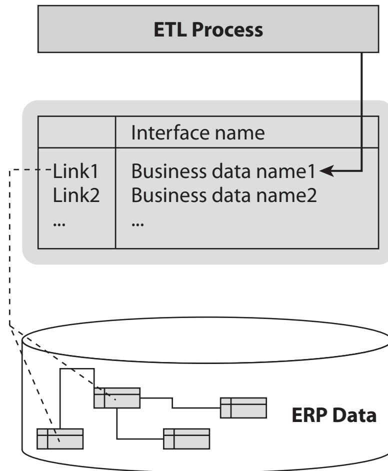


Figure 6-10: Business data layer

Kettle ERP Plugins

Because of the plugin architecture of Kettle, several commercial third-party plugins have been available for some time to get data from systems such as SAP or SugarCRM. The Pentaho community developed several new open source plugins for Kettle 4, so for some of these systems, data access support is now available in the standard installation as well. The following options are available at the time of this writing:

- **SAP Input step:** Allows Kettle to retrieve data from an SAP installation using SAP interface calls.
- **SalesForce.com input step:** Allows Kettle to retrieve data from SalesForce using the standard web service calls.
- **SalesForce.com output steps:** Using the same published set of web services, it is possible to insert, update, delete, and upsert data in a SalesForce database.

Working with SAP Data

There are many ways to get data from an SAP system, most notably by invoking Remote Function Calls (RFCs) or calling a Business Application Programming Interface (BAPI). Austrian Pentaho Partner Aschauer EDV developed the open source Kettle plugin that can work with both RFCs and BAPIs. The following workflow demonstrates how the SAP plugin can be used to retrieve Billing Document information from the Financial module of SAP/R3.

NOTE **The authors wish to thank Bernd Aschauer and Robert Wintner, both from Aschauer EDV (**`http://www.aschauer-edv.at/en`**), for providing the example screenshots used in this section.**

In order to be able to use the SAP Input step, the SAP Java Connector library (`sapjco3.jar` or similar) needs to be downloaded from the SAP support site and copied to the Kettle `libext` directory. After this, you can drag an SAP Input step to the canvas and connect to an SAP system. An SAP connection requires more than a server address, username and a password; Figure 6-11 shows an example connection with the correct values entered.

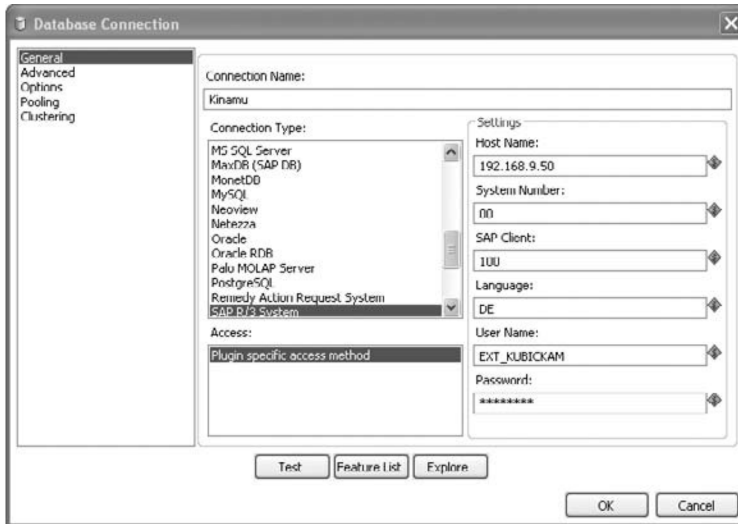


Figure 6-11: SAP/R3 connection

As shown in Figure 6-11, each SAP instance has a system number, and for connecting you'll need to provide the client ID and language used as well. This is exactly the same information as required to connect to SAP using the standard SAP client. After you enter the required information, the Test button on the connection screen will show whether the connection can be made successfully.

After making the connection, you can use the function browser by clicking the "Find it" button in the SAP Input step screen. This will open a dialog box where you can search for functions with a specific name using wildcards, as shown in Figure 6-12.

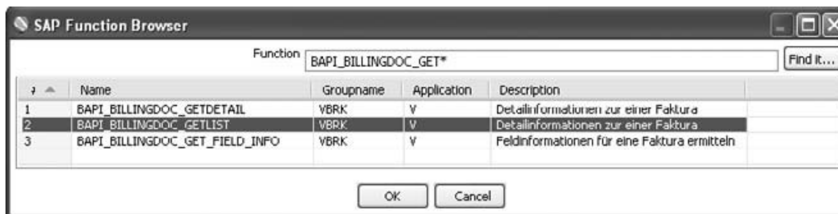


Figure 6-12: SAP Function Browser

Each selected function can have multiple input and output fields, which will be displayed by the SAP Input step after you select the required function, as shown in Figure 6-13.

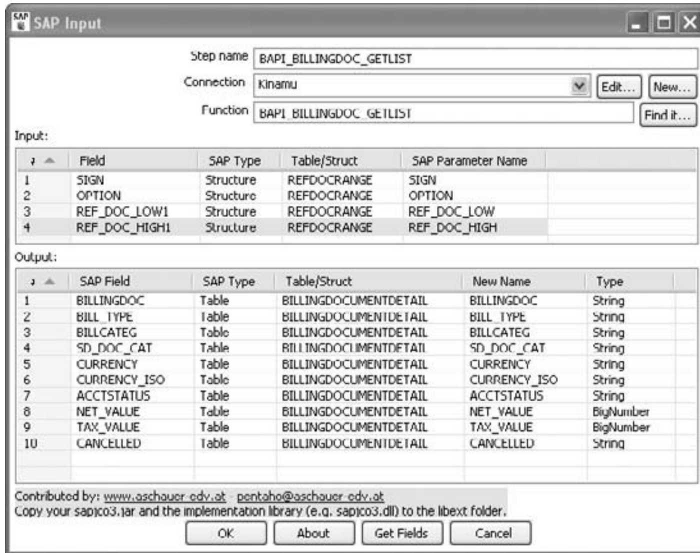


Figure 6-13: SAP Input step

The values for the input fields can be set in Kettle prior to using the SAP Input step by using a Data Grid or Generate Rows step, or by retrieving the data from another source such as an external parameter table. In this case, it suffices to define four string columns in a Generate Rows step and limit the row generation to a single row, as displayed in Figure 6-14.

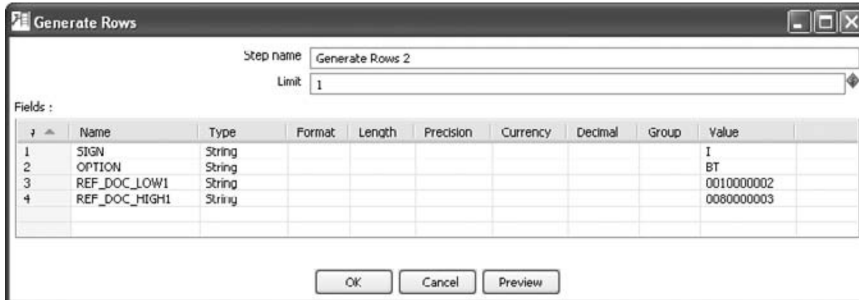


Figure 6-14: Generate Rows for SAP Input step parameters

A complete transformation to test whether this will give you the required data looks remarkably simple then. Figure 6-15 shows the required steps to set input values, retrieve data from SAP, and send the stream to a Dummy plugin.

Figure 6-15: Kettle transformation reading SAP data

The preview issued on the Dummy step will show that the input step extracts the required data, as displayed in Figure 6-16.

Examine preview data

Rows of step: Dummy (do nothing) 2 (66 rows)

#	BILLINGDOC	BILL_TYPE	BILLCATEG	SD_DOC_CAT	CURRENCY	CURRENCY_ISO	ACCTSTATUS	NET_VALUE	TAX_VALUE	CANCELLED
1	0090000000	F2	L	M	EUR	EUR	C	45	9	
2	0090000001	F2	L	M	EUR	EUR	E	11350.5	0	X
3	0090000002	S1	L	N	EUR	EUR	E	11350.5	0	
4	0090000025	ZTE	L	M	EUR	EUR	E	226.2	45.24	X
5	0090000026	S1	L	N	EUR	EUR	E	226.2	45.24	
6	0090000027	ZTE	L	M	EUR	EUR		226.2	45.24	
7	0090000025	ZTE	L	M	EUR	EUR	E	226.2	45.24	X
8	0090000026	S1	L	N	EUR	EUR	E	226.2	45.24	
9	0090000027	ZTE	L	M	EUR	EUR		226.2	45.24	
10	0090000030	ZTE	L	M	EUR	EUR		30.16	7.63	
11	0090000000	F2	L	M	EUR	EUR	C	45	9	
12	0084000000	F8	L	U	EUR	EUR	D	0	0	
13	0084000001	F8	L	U	EUR	EUR	D	13	0	
14	0090000004	F2	L	M	EUR	EUR	G	600	0	
15	0090000005	F2	L	M	EUR	EUR	C	30	6	X
16	0090000006	S1	L	N	EUR	EUR	C	30	6	
17	0090000007	F2	L	M	EUR	EUR	C	30	6	
18	0090000036	F2	L	M	EUR	EUR	C	0	0	
19	0090000037	F2	L	M	EUR	EUR	C	105	0	
20	0090000038	F2	L	M	EUR	EUR	C	105	0	
21	0090000013	FAZ	P	M	EUR	EUR	C	4400	880	
22	0090000014	F2	D	M	EUR	EUR	C	19800	3960	
23	0090000015	F2	D	M	EUR	EUR	C	2200	440	
24	0090000016	FAZ	P	M	EUR	EUR	E	4460	892	X
25	0090000017	FAS	P	N	EUR	EUR	E	4460	892	
26	0090000018	FAZ	P	M	EUR	EUR	C	4460	892	
27	0090000019	F2	D	M	EUR	EUR	C	20070	4014	
28	0090000020	F2	D	M	EUR	EUR	C	2230	446	
29	0090000016	FAZ	P	M	EUR	EUR	E	4460	892	X
30	0090000017	FAS	P	N	EUR	EUR	E	4460	892	
31	0090000018	FAZ	P	M	EUR	EUR	C	4460	892	

Close

Figure 6-16: SAP Billingdoc data

This first example showed you how to use a standard function that returned the data in a directly usable format. One of the more generic RFCs is the `RFC_READ_TABLE` function, which can be used to read data from any table. The drawback of this step is twofold: You need to know something of the structure of the data that needs to be retrieved (at least the field names), and the return value is a single column that you need to split manually. Nevertheless, it's a flexible way of getting any kind of data from an SAP system. Figure 6-17 shows what the "SAP Input" step looks like when using the `RFC_READ_TABLE` function to retrieve data from the G/L Account Master table called SKAT, which contains the account descriptions. Other important master tables in an SAP/R3 FI (finance) installation are KNB1 (Customer Master) and LFA1 (Vendor Master), names that still show the German heritage of the product: KN is short for *Kunde* (Customer), and LF for *Lieferant* (Vendor).

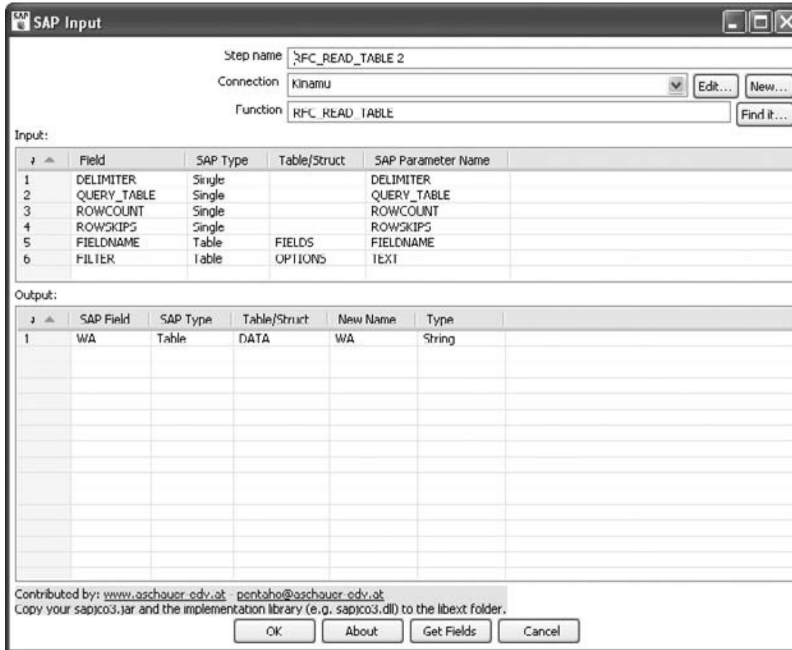


Figure 6-17: SAP Input step using RFC_READ_TABLE

Figure 6-17 also shows that the structure of the input and output is entirely different from the first example transformation. Now you need to specify the delimiter, the table name, the number of rows to read and skip, and the fieldnames and filter to use. As you can see in the screenshot, the fieldname and filter are of type Table, meaning that multiple values can be provided. The Generate Rows step used to create this input data is displayed in Figure 6-18 and shows how this step can be used to pass the correct values to the SAP Input step.

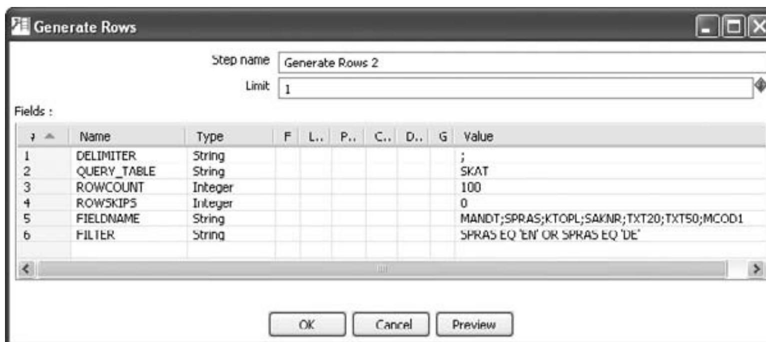


Figure 6-18: SAP RFC_READ_TABLE input specification

Before the data can be used, it has to be split, now using the same delimiter specified in the Generate Rows step. The Kettle "Field splitter" step type makes this a very straightforward process, as shown in Figure 6-19.

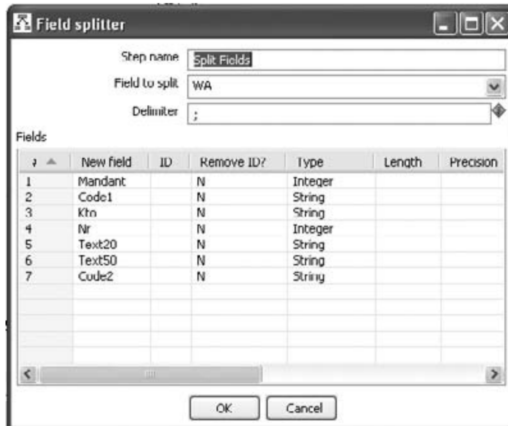


Figure 6-19: Using the "Field splitter" to divide output fields into columns

If you take a careful look at Figures 6-18 and 6-19, you'll notice that different field names are being used. That's because in order to access the fields in an SAP system, you'll need to exactly specify the field names used. The Field splitter lets you define your own names for the fields; the only thing that's important is the ordinal position and data type of the field, not the name. Figure 6-20 then shows the final output of the `RFC_READ_TABLE` function call that retrieved 100 rows as specified.

Examine preview data

Rows of step: Split Fields (100 rows)

	Mand...	Code1	Kto	Nr	Text20	Text50	Code2
1	100	D	CACH	10000	Kasse CHF	Kasse CHF ...	KASSE CHF
2	100	D	CACH	10030	Kasse EUR	Kasse EUR ...	KASSE EUR
3	100	D	CACH	10090	WB Fremdwährungen	WB Fremdwährungen ...	WB FREMDWÄHRUNGEN
4	100	D	CACH	10100	Post	Post ...	POST
5	100	D	CACH	10102	Post Ausgangszahl.	Post Ausgangszahlung ...	POST AUSGANGSZAHLUNG
6	100	D	CACH	10105	Post Zwischenbuch.	Post Zwischenbuchungen ...	POST ZWISCHENBUCHUNGEN
7	100	D	CACH	10109	Post Deb.-Geldeing.	Post Debitoren-Geldeingang ...	POST DEBITOREN-GELDEINGAN
8	100	D	CACH	10200	Bank A Kontokorrent	Bank A Kontokorrent ...	BANK A KONTOKORRENT
9	100	D	CACH	10201	Bank A Ausg.-Schecks	Bank A Ausgangsschecks ...	BANK A AUSGANGSSCHECKS
10	100	D	CACH	10202	Bank A Ausg.Üw Inl.	Bank A Ausgangsüberweisung...	BANK A AUSGANGSÜBERWEISUN
11	100	D	CACH	10203	Bank A Ausg.Üw Ausl.	Bank A Ausgangsüberweisung...	BANK A AUSGANGSÜBERWEISUN
12	100	D	CACH	10204	Bank A Bankeinzug	Bank A Bankeinzug ...	BANK A BANKEINZUG
13	100	D	CACH	10205	Bank A Sonst. Zw.b.	Bank A Sonstige Zwischenbuc...	BANK A SONSTIGE ZWISCHENB
14	100	D	CACH	10208	Bank A Scheckeing.	Bank A Scheckeingang ...	BANK A SCHECKEINGANG
15	100	D	CACH	10209	Bank A Deb-Geldeing.	Bank A Debitoren-Geldeingan...	BANK A DEBITOREN-GELDEING
16	100	D	CACH	10210	Bank B Kontokorrent	Bank B Kontokorrent ...	BANK B KONTOKORRENT
17	100	D	CACH	10220	Bank C Kontokorrent	Bank C Kontokorrent ...	BANK C KONTOKORRENT
18	100	D	CACH	10230	Bank D Kontokorrent	Bank D Kontokorrent ...	BANK D KONTOKORRENT
19	100	D	CACH	10240	Bank A Konto in EUR	Bank A Konto in EUR ...	BANK A KONTO IN EUR
20	100	D	CACH	10245	Bank A Verr.kto. EUR	Bank A Verrechnungskonto in ...	BANK A VERRECHNUNGSKONTO
21	100	D	CACH	10250	Bank A Konto in USD	Bank A Konto in USD ...	BANK A KONTO IN USD
22	100	D	CACH	10255	Bank A Verr.Kto. USD	Bank A Verrechnungskonto in ...	BANK A VERRECHNUNGSKONTO
23	100	D	CACH	10300	Bank D Sparkonto	Bank D Sparkonto ...	BANK D SPARKONTO
24	100	D	CACH	10310	Bank E Anlagekonto	Bank E Anlagekonto ...	BANK E ANLAGEKONTO
25	100	D	CACH	10390	WB Fremdw.-Konten	WB Fremdwährungskonten ...	WB FREMDWÄHRUNGSKONTEN
26	100	D	CACH	10400	Checks	Checks ...	CHECKS

Close

Figure 6-20: RFC_READ_TABLE output showing G/L Accounts

ERP and CDC Issues

The SAP/R3 examples in the previous section showed an easy way to obtain data from this ERP system, and based on user demand there will undoubtedly appear more ERP input steps as time will pass. There is one major issue with this way of working, however: there is no way of retrieving only changed, inserted, or deleted data. To work around this, you can take one of the following three approaches:

- **File delivery**: The SAP team can be asked to develop a custom program that delivers only changed data in a text file. In many projects this is the default way of operating since access to the SAP system is prohibited for the ETL process
- **Work with snapshots:** In the Snapshot-Based CDC section later in this chapter we'll show you how to use snapshots and the Kettle Merge Join step to detect changes in data sets.
- **Augment the RFCs:** For CDC purposes, it could make sense to add additional parameters to an RFC to enable filtering changed data, but that needs to be done on the SAP side.

Data Profiling

Data profiling is the process of collecting statistics and other information about the data available in different source systems. The obtained information is invaluable for the further design of your data warehouse and ETL processes. Data profiling is also an important part of any data quality initiative; before quality can be improved, a baseline must be established indicating what the current state of the data is. Profiling can be performed at three different levels:

- **Column profile:** Collects statistics about the data in a single column.
- **Dependency profile:** Checks for dependencies within a table between different columns.
- **Join profile:** Checks for dependencies between different tables.

The starting point for profiling is always the column-level profile, which generates useful information about the data in a column, including but not limited to:

- **Number of distinct values:** How many unique entries does the column contain?
- **Number of NULL and empty values:** How many records have no value or an empty value?
- **Highest and lowest values**: Not only for numeric but also for textual data.
- **Numeric sum, median, average, and standard deviation:** Various calculations on the numeric values and value distribution.

- **String patterns and length:** Are the values correctly stored? (For example, German postal codes should contain five digits.)
- **Number of words, number of upper and lowercase characters:** What's the total number of words in the column and are the words all upper, lower or mixed case?
- **Frequency counts:** What are the top and bottom *N* items in a column?

Most data-profiling tools can deliver this information and sometimes even more. It gets trickier when you look at the profiling within a single table to identify correlations and interdependencies. Examples of this are combinations of postal code-to-city, city-to-region, and region-to-country. Obviously, a city name is dependent on a postal code, the region name on the city, and the country on the region. These dependencies violate the third normal form so when finding these relationships in a third normal form source system, you should take extra care, especially regarding the address information. Sometimes the relations are not very clear or are even confusing, which makes it hard to distinguish correct from incorrect entries. This is exactly the reason that so many expensive address matching and cleansing solutions exist. Take, for instance, the city-region combination: There are more than ten states in the United States with a city named Hillsboro. Without additional knowledge of the country or ZIP codes, it's hard to tell whether a record contains erroneous information or not. For these cases, you'll need external information to validate the data against.

Inter-table relationships are easier to profile; it is simply a matter of evaluating whether a relationship is correctly enforced. In an order entry system, it shouldn't be possible to find a customer number in the order table that does not exist in the customer table. The same relationship test can be used to find out how many customers are in the customer table but not (yet) in the order table. The same applies to products and order details, inventory and suppliers, and so on.

Using eobjects.org DataCleaner

Currently, the community edition of the Pentaho BI suite does not contain data profiling capabilities, so we will use a tool named DataCleaner developed by the open source community eobjects.org. (Kettle 4.1 will contain a Data Profile feature as one of the Database Explorer functions.) The software can be obtained from `http://datacleaner.eobjects.org/` and is very easy to install. On Windows, you simply unzip the package and start `datacleaner.exe`. On a Linux machine, after unpacking the `tar.gz` file you first need to make the `datacleaner.sh` shell script executable to start the program. If you are using the GNOME desktop environment, this is very easy: Just right-click the file and open the properties. Then go to the Permissions tab and select the Execute option "Allow executing file as program." Now you can double-click on the `datacleaner.sh` file and the program will start. If you want a more convenient way to start the program next time, you can create a shortcut (in Windows) or a launcher (in GNOME).

DataCleaner performs three main tasks:

- **Profile:** All the column profiling tasks described earlier. The idea here is to gain insight into the state of your data. You can thus use the profiling task whenever you want to explore and take the temperature of your database.

- **Validate:** To create and test validation rules against the data. These validation rules can then later be translated (by hand) into Pentaho Data Integration validation steps. The validator is useful for enforcing rules onto your data and monitoring the data that does not conform to these rules.
- **Compare:** To compare data from different tables and schemas and check for consistency between them.

From these descriptions, it's immediately clear that DataCleaner does not provide intra-table profiling capabilities as a direct option, but there are other ways to accomplish this with the tool, as we'll show later.

The first thing you need, of course, is a connection to the database you want to profile. For each type of database, DataCleaner needs the corresponding driver, which enables the communication between the tool and the database. Before we explain how to add drivers and connections, let's take a first look at the available functions. DataCleaner starts with the New task panel open, which allows you to choose one of the three main options: Profile, Validate, and Compare. Click on Profile to start a new profiling task. You'll see an almost empty two-pane screen with some options and the "No data selected" indication in the left pane (see Figure 6-21).

Figure 6-21: Profiling task

Now select "Open database," select the DataCleaner `sampledata` entry, and click Connect to Database. All other fields have been set already. When you open the PUBLIC tree node on the left by clicking on the + sign, the list with tables appears. Each table can be opened individually, which displays the available columns. To add a column to the data selection, just double-click it. You'll notice that the table name is added to the Table(s) field, and the column to the Column(s) field. To remove a column from the selection, double-click it again or use Clear Selection to completely remove the selected tables and columns. The Preview option shows a sample of the selected data; the number of rows to be retrieved can be adjusted after clicking the button. The default value often suffices to get a first impression of the content of the data. Each table gets its selected columns displayed in a separate window.

Next to the "Data selection" tab is the Metadata tab. When you click this tab, the technical metadata of the selected columns is displayed. The field type, field length, and especially the Nullable indication give you a first impression of the kind of data to be expected.

Adding Profile Tasks

After selecting some columns to profile, you can add different profiles. DataCleaner contains the following standard profile options:

- **Standard measures:** Row count, number of NULL values, empty values, highest and lowest value.
- **String analysis:** Percentage of upper and lowercase characters, percentage of non-letter characters, minimum and maximum number of words, and the total number of words and characters in the column.
- **Time analysis:** Lowest and highest date value, plus number of records per year.
- **Number analysis:** Highest, lowest, sum, mean, geometric mean, standard deviation, and variance.
- **Pattern finder:** Finds and counts all patterns in a character column. Mostly used for phone numbers, postal codes, or other fields that should conform to a specific alpha-numeric pattern. Pattern examples are `9999 aa` (4 digits, space, 2 characters), `aaa-999` (3 characters, hyphen, 3 digits).
- **Dictionary matcher:** Matches the selected columns against the content of an external file or another database column (a "dictionary").
- **Regex matcher:** Matches columns against a regular expression.
- **Date mask matcher:** Matches text columns against date patterns; this cannot be used with date fields, only with text fields containing date and/or time information.
- **Value distribution:** Calculates the top and bottom N values in a column based on their frequency, or ranks the number of occurrences and calculates the frequency percentage for each value. The value for *N* can be any number between 0 and 50; the default is 5.

The collection of profiles in a task is very flexible; it's possible to add profiles of the same type to a single task. Each task can be saved as well, but this will only save the connection and task profiles, not the profiler results. This last option is a separate function and saves the results in an XML file, which is unfortunately for the moment a one-way street; DataCleaner cannot read these files back. Persisting profile results is part of the roadmap for future releases.

Adding Database Connections

One of the first things to do when setting up the data profiling environment is to add the correct database drivers and store the connections to your own databases for easy

selection. The first task is pretty straightforward; in the main DataCleaner screen, select File ➪ Register database driver. There are two ways to add a new driver. The first is to automatically download and install the driver. This option is available for MySQL, PostgreSQL, SQL Server/Sybase, Derby, and SQLite. The second way of doing this is to manually register a `.jar` file with the drivers. To help you find the drivers, DataCleaner contains the option to visit the driver website for the most common database drivers, such as those for Oracle or IBM DB2. After downloading a driver, you'll need to reference it by selecting the file and the correct driver class. For MySQL, we will use the Automatic download and install option.

NOTE **If you already installed the MySQL JDBC driver, there's no need to download it again; just register your existing `.jar` file.**

Adding the connection so you can select it from the drop-down list in the Open Database dialog box is a bit more complicated. For that you need to alter the DataCleaner configuration file, which can be found in the DataCleaner folder and is called `datacleaner-config.xml`. To edit XML files, it's best to use a plain-text editor that understands the XML syntax. For the Windows platform, the open source Notepad++ can be used; on a Linux machine, just right-click the file and open with Text editor. Look for the part in the file that says:

```
<!-- Named connections. Add your own connections here. -->.
```

Below this line there's an empty entry for the drop-down list; just leave that where it is. The second entry is the connection to the sample data. Copy the sample data part that starts with `<bean` and ends with `</bean>`, including the start and end `bean` tags. Paste it right below the closing tag of the sample data entry and adjust the information to reflect your own settings. The following code shows the entry as it should look for the connection to the sakila database on your local machine:

```
<bean class="dk.eobjects.datacleaner.gui.model.NamedConnection">
   <property name="name" value="sakila database" />
   <property name="connectionString"
value="jdbc:mysql://localhost:3306" />
   <property name="username" value="sakila" />
   <property name="password" value="sakila" />
   <property name="tableTypes">
     <list>
         <value>TABLE</value>
     </list>
   </property>
</bean>
```

To have DataCleaner also connect to the correct catalog, for instance the Sakila catalog, an extra line should be added below the password property line, like this:

```
<property name="catalog" value = "sakila" />
```

We don't recommend storing passwords in plain-text files; in fact, we strongly oppose doing so, and in this case you can leave the password field empty as well. In that case, you'll need to provide the password each time you create a new profiling task.

To use DataCleaner with sources other than the sakila database, you can find examples of the XML bean-element for other popular databases in the online DataCleaner documentation.

Doing an Initial Profile

The DataCleaner profiler has been optimized to allow you to do a rather quick and at the same time insightful profile with little effort. To get started with profiling, you can add the Standard Measures, String Analysis, Number Analysis, and Time Analysis profiles by repeatedly clicking the Add Profile button in the top-right corner of the Profile task window. You can apply these profiles to all the columns of your database to get the initial insight.

Working with Regular Expressions

Regular expressions, or *regexes*, are a way of masking and describing data, mainly for validation purposes but also to find certain patterns in a text. Several books have been written about working with regular expressions so we refer to existing information here. DataCleaner contains both a regex matcher as one of the profiles as well as a regex validation as part of the validator. Before you can use regular expressions, you'll need to add them to the Regex catalog in the main DataCleaner screen. Initially this catalog is empty, but it's easy to add regexes. When you click "New regex," three options appear. The first one is to create a new regex manually and the last one is to get a regex from the `.properties` file. The second option is the most interesting: When you select Import from the RegexSwap, an online library is opened with a large collection of existing regexes to pick from. It is also possible to contribute your own regexes to the RegexSwap at `http://datacleaner.eobjects.org/regexswap` for others to (re)use. After importing a regex from the RegexSwap, you can open it to change its name and the expression itself, and there's an option to test the expression by inputting strings you want to validate. If the RegexSwap doesn't fulfill your needs, a vast number of regular expressions are available on other Internet websites as well. The site `http://regexlib.com`, for example, contains regexes for U.S. phone numbers and ZIP codes. Another great site, especially if you want to learn the regular expression syntax, is `www.regular-expressions.info`.

For a very simple example of using a regex validation, create a new Profiler task, connect to the Sakila database, and select the `Address` table in the `Data Selection` tab. Now add a `Regex matcher` with the `Add profile` option and keep only the `Integer` regex to validate the `phone` field in the address table, as shown in Figure 6-22.

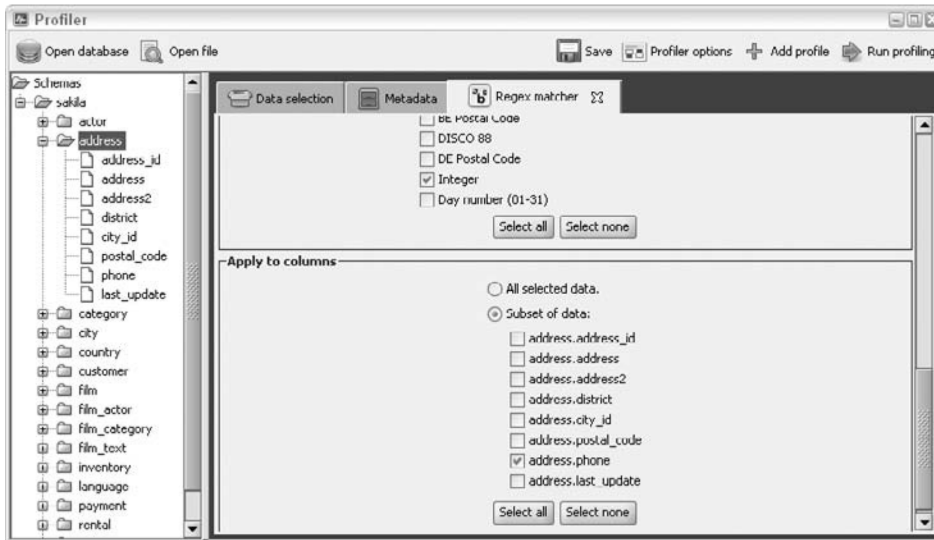


Figure 6-22: Regex match of phone number

Regular expressions are also a powerful component in the Kettle toolkit for handling all kinds of things. You already saw one example when we used a regular expression to find all `.txt` files in a directory. Chapters 7 and 20 also contain several examples of how regular expressions can be used in your transformations.

Profiling and Exploring Results

When the definition of a profile is complete, the option `Run profiling` starts the process. DataCleaner will display a status screen where you can also monitor the progress of the profile process. When the profiler is finished, a results tab is added to the screen, one for each table that contained profiled columns. Figure 6-23 shows the output of the profile task from the preceding section.

Figure 6-23: Profiler results

As is clearly visible, there are two phone numbers that do not match the regular expression, which enables us to show another nice DataCleaner feature: drilling down to the details. Clicking on the green arrow next to the two found exceptions opens the screen shown in Figure 6-24.

Detail data for [No Matches, phone]

address_id	address	address2	district	city_id	postal_code	phone	last_update
1	47 MySakila D...	<null>	Alberta	300	""	""	2006-02-15 0...
2	28 MySQL Bou...	<null>	QLD	576	""	""	2006-02-15 0...

Figure 6-24: Profiler results

This isn't the end, however. When you right-click, you'll get two export options: one for selected cells and one for the entire table. The latter option will also add the column headers to the clipboard; the first one just copies the selected data. Selected cells don't have to be adjacent. By using the Ctrl key, you can, for instance, select only the address ID and phone number and copy those columns to the clipboard. After that, you can easily paste the data into a spreadsheet or other file for further exploration.

Validating and Comparing Data

Validation works in a similar fashion as the profiling task but adds some capabilities. You can check for null values or do a value range check to find out whether entries in a column fall between a lower and upper value bound. The most advanced feature is the JavaScript evaluation, which lets you use any JavaScript expression for evaluating data. The difference is the output: The validation task will display only the entries that do not pass the tests with a count of the records. The DataCleaner roadmap includes future plans to integrate the profile and validation tasks and offer a single integrated interface for both tasks.

Data comparison enables you to compare data from different databases or schemas, or compare data against a file. This task therefore can be used to check whether all customers in the `rental` table also exist in the `customer` table and to perform similar comparison tasks.

Using a Dictionary for Column Dependency Checks

DataCleaner does not provide an out-of-the-box solution to verify combinations of columns or whether one dependent column contains an invalid entry based on the information in another column. There is, however, a way to do these analyses by using a dictionary combined with database views. A DataCleaner *dictionary* is a text file containing values that can be used to validate data in a database table. For example, you can download the ISO country table, store the values in a text file, and use this text file as a catalog to verify the entries in a country column. If you take this one step further, it's also possible to store multiple concatenated fields per row and create a view in the database, which concatenates the columns to be validated in the same way. Now the view

can be profiled using the dictionary with the concatenated entries; each row that does not correspond to the correct values in the text file will be recognized by DataCleaner. As an alternative to using text files, it's also possible to use a "real" database dictionary. This database dictionary needs to be added to the DataCleaner configuration file as explained in the section "Adding Database Connections."

Alternative Solutions

Very few open source alternatives exist for standalone or embedded data profiling. The data modeling tool from SQLPower, which we introduce shortly, has some basic profiling capabilities, and Talend offers a Data Profiler as well. If any of these tools work for you, just use them. Another frequently used alternative for data profiling is creating custom SQL scripts for data-profiling purposes. We would recommend this only if you have very specialized requirements that are not provided out-of-the-box by DataCleaner. Although it's outside the scope of this book, it is possible to extend DataCleaner's functionality with your own customer profiling tasks, which gives you a faster, more reliable, and more flexible solution than completely starting from scratch.

Text Profiling with Kettle

As stated before, Kettle doesn't offer profiling capabilities similar to DataCleaner, with one notable exception: text file profiling. If you profile a text file with DataCleaner by using "Open file," you'll notice that all fields are of type `Varchar`. Another limitation is that DataCleaner recognizes only commas, semicolons, and tabs as field delimiters, which limits the value of the tool when working with text files. In this case, Kettle might be a better alternative. When using the "Text file input" step, as shown earlier, the Get Fields step does a pretty good job of recognizing the field types and lengths. It's not a perfect solution, however; when a single field contains multiple separated integer values (for instance `21;33;56`) or a time value in the form `21:34`, Kettle sees this as an integer of length 2, not a string of length 8 or 5. This is a known issue which should be resolved in Kettle 4.1.

CDC: Change Data Capture

The first step in an ETL process is the extraction of data from various source systems and storing this data in staging tables. This seems like a trivial task and in the case of initially loading a data warehouse it usually is, apart from challenges incurred from data volumes and slow network connections. But after the initial load, you don't want to repeat the process of completely extracting all data again (which wouldn't be of much use anyway because you already have an almost complete set of data that only needs to be refreshed to reflect the current status). All you're interested in is what has changed since the last data load, so you need to identify which records have been inserted, modified, or even deleted. The process of identifying these changes and only retrieving records that are different from what you already loaded in the data warehouse is called *Change Data Capture*, or *CDC*.

NOTE **Parts of this section were published earlier in Chapter 6 of *Pentaho Solutions*.**

Basically there are two main categories of CDC processes, *intrusive* and *non-intrusive*. By intrusive, we mean that a CDC operation has a possible performance impact on the system the data is retrieved from. It is fair to say that any operation that requires executing SQL statements in one form or another is an intrusive technique. The bad news is that three of the four ways to capture changed data are intrusive, leaving only one non-intrusive option. The following sections offer descriptions of each solution and identify their pros and cons.

Source Data–Based CDC

Source data-based CDC relies on the fact that there are attributes available in the source system that enable the ETL process to make a selection of changed records. There are two alternatives:

- **Direct read based on timestamps (date-time values):** At least one update timestamp is needed for this alternative but preferably two are created: an insert timestamp (when was the record created) and an update timestamp (when was the record last changed).
- **Using database sequences:** Most databases have some sort of auto-increment option for numeric values in a table. When such a sequence number is used, it's easy to identify which records have been inserted since the last time you looked at the table.

Both of these options require extra tables in the data warehouse to store the information regarding the last time the data was loaded or the last retrieved sequence number. A common practice is to create these parameter tables either in a separate schema or in the staging area, but never in the central data warehouse and most certainly not in one of the data marts. A timestamp or sequence-based solution is arguably the most simple to implement and for this reason also one of the more common methods for capturing change data. The penalty for this simplicity is the absence of a few essential capabilities that can be found in more advanced options:

- **Distinction between inserts and updates:** Only when the source system contains both an insert and an update timestamp can this difference be detected.
- **Deleted record detection:** This is not possible, unless the source system only logically deletes a record, i.e., has an end or deleted date but is not physically deleted from the table.
- **Multiple update detection:** When a record is updated multiple times during the period between the previous and the current load date, these intermediate updates get lost in the process.
- **Real-time capabilities:** Timestamp or sequence-based data extraction is always a batch operation and therefore unsuitable for real-time data loads.

USING TIMESTAMPED CDC IN KETTLE: AN EXAMPLE

The sakila sample database used in Chapter 4 offers a good source for demonstrating the use of timestamps to capture changed data because all tables contain an *update* timestamp, and the `Customer` table even contains an *insert* timestamp as well. For our purposes, the `Customer` table is best because it can distinguish between inserts and updates. To use the timestamps, you need to store the last load date somewhere in a Kettle property or a parameter table. For this simple example, you'll just create a `cdc_time` table in the `sakila_dwh` catalog that consists of two fields, a `last_load` timestamp and a `current_load` timestamp. Initially, both the `last_load` and `current_load` timestamps are set to a very early date-time value (the latter will be set to the current time when we start loading).

First, create the table to hold the timestamps:

```
CREATE TABLE `cdc_time` (
   `last_load` datetime,
  `current_load` datetime)
```

Then set the default values:

```
insert into cdc_time values ('1971-01-01 00:00:01','1971-01-01
00:00:01')
```

The logic is as follows:

1. **When the load job starts, the value of the field `current_load` is set to the actual time of the job start. To do this, use a Get System Info step and create a field `sysdate` of type `system date (fixed)`. Then, add an Insert / Update step, create a hop between the Get System Info step and the Insert / Update step. In the "Update fields" section, map the table field `current load` to the stream field `sysdate`. Set Update to `Y` and make sure the there's a condition for the lookup section as well. Setting `current_load` as the `Table` field and `IS NOT NULL` as the `Comparator` will do the job.**
2. **The query that retrieves data from the customer table needs to be restricted by using the begin and end dates you just created. Logically this would look like the following:**

   ```
   (create_date >= last_load AND create_date < current_load)
   OR
   (last_update >= last_load AND last_update < current_load).
   ```

   In order for this code to use the values from the `cdc_time` table, you need two table input steps: one to read the values from the `cdc_time` table and one to select the customers. If you look at the restriction you'll notice that both `last_load` and `current_load` appear twice in the condition. This means you need to retrieve them twice as well by using the following query in the first input step:

   ```
   SELECT
     last_load last1
   ```

USING TIMESTAMPED CDC IN KETTLE: AN EXAMPLE

```
, current_load cur1
, last_load last2
, current_load cur2
FROM cdc_time
```

In the next step, where the customers are selected, check the "Replace `variables` **in script?" box and select the previous input step in the "Insert data from step" dropdown list (you need to create a hop first). The** `SELECT` **statement can now be extended with the following** `WHERE` **condition:**

```
WHERE
(create_date >= ? AND create_date < ?)
OR
(last_update >= ? AND last_update < ?)
```

The question marks will be replaced sequentially by the values returned from the first table input step, so the first question mark will get the value of last1, the second one the value of cur1, and so on.

3. **A distinction between inserts and updates is easily made by checking whether the** `create_date` **and** `last_update` **values are equal:**

```
CASE
     WHEN create_date = last_update THEN 'new'
     ELSE 'changed'
END AS flagfield
```

4. **After the load is completed without errors, the value of the field** `current_load` **is copied to the** `field last_load`**. If an error occurs during the load, the timestamps remain unchanged, enabling a reload with the same timestamps for last and current load. This can easily be accomplished by using an "Execute SQL script" step with the following query:**

```
update cdc_time set last_load = current_load
```

The reason for using two fields (especially the `current_load` **field) is that during a load new records can be inserted or altered. In order to avoid dirty reads or deadlock situations, it's best to set an upper limit for the** `create` **and** `update` **timestamps.**

Trigger-Based CDC

Database triggers can be used to fire actions when you use any data manipulation statement such as `INSERT`, `UPDATE`, or `DELETE`. This means that triggers can also be used to capture those changes and place these changed records in intermediate change tables in the source systems to extract data from later, or to put the data directly into the staging tables of the data warehouse environment. Because adding triggers to a database will be prohibited in most cases (it requires modifications to the source database, which is often not covered by service agreements or not permitted by database administrators)

and can severely slow down a transaction system, this solution, although functionally appealing at first, is not implemented very often.

An alternative to using the triggers directly in the source system would be to set up a replication solution where all changes to selected tables will be replicated to the receiving tables at the data warehouse side. These replicated tables can then be extended with the required triggers to support the CDC process. Although this solution seems to involve a lot of overhead processing and requires extra storage space, it's actually quite efficient and non-intrusive since replication is based on reading changes from the database log files. Replication is also a standard functionality of most database management systems, including MySQL, PostgreSQL, and Ingres.

Trigger-based CDC is probably the most intrusive alternative described here but has the advantage of detecting all data changes and enables near real time data loading. The drawbacks are the need for a DBA (the source system is modified) and the database-specific nature of the trigger statements.

Snapshot-Based CDC

When no timestamps are available and triggers or replication are not an option, the last resort is to use snapshot tables, which can be compared for changes. A snapshot is simply a full extract of a source table that is placed in the data warehouse staging area. The next time data needs to be loaded, a second version (snapshot) of the same table is placed next to the original one and the two versions compared for changes. Take, for instance, a simple example of a table with two columns, `ID` and `Color`. Figure 6-25 shows two versions of this table, snapshot 1 and snapshot 2.

Snapshot_1

ID	COLOR
1	Black
2	Green
3	Red
4	Blue

Snapshot_2

ID	COLOR
1	Grey
2	Green
3	Blue
4	Yellow

Figure 6-25: Snapshot versions

There are several ways to extract the differences between those two versions. The first is to use a full outer join on the key column ID and tag the result rows according to their status (I for Insert, U for Update, D for Delete, and N for None) where the unchanged rows are filtered in the outer query:

```
SELECT * FROM
(SELECT  CASE
                WHEN t2.id IS NULL THEN 'D'
                WHEN t1.id IS NULL THEN 'I'
```

```
                    WHEN t1.color <> t2.color THEN 'U'
                    ELSE 'N'
                END AS flag
,               CASE
                    WHEN t2.id IS NULL THEN t1.id
                    ELSE t2.id
                END AS id
,               t2.color
FROM                 snapshot_1 t1
FULL OUTER JOIN snapshot_2 t2
ON                  t1.id = t2.id
) a
WHERE flag <> 'N'
```

That is, of course, when the database supports full outer joins, which is not the case with MySQL. If you need to build a similar construction with MySQL there are a few options, such as the following:

```
SELECT 'U' AS flag, t2.id AS id, t2.color AS color
FROM  snapshot_1 t1 INNER JOIN snapshot_2 t2 ON t1.id = t2.id
WHERE t1.color != t2.color
UNION ALL
SELECT 'D' AS flag, t1.id AS id, t1.color AS color
FROM snapshot_1 t1 LEFT JOIN snapshot_2 t2 ON t1.id = t2.id
WHERE t2.id is NULL
UNION ALL
SELECT 'I' AS flag, t2.id AS id, t2.color AS color
FROM snapshot_2 t2 LEFT JOIN snapshot_1 t1 ON t2.id = t1.id
WHERE t1.id IS NULL
```

In both cases the result set is the same, as shown in Figure 6-26.

Flag	ID	COLOR
U	1	Grey
D	3	NULL
I	5	Yellow

Figure 6-26: Snapshot compare result

Most ETL tools nowadays contain standard functionality to compare two tables and flag the rows as I, U, and D accordingly, so you will most likely use these standard functions instead of writing SQL. Kettle is no exception to this rule and contains the "Merge rows" step. This step takes two *sorted* input sets and compares them on the specified keys. The columns to be compared can be selected and an output flag field name needs to be specified. To demonstrate this, let's first extract the Customer table from the Sakila database you created in Chapter 4, and then make a couple of changes

to the source table and create a snapshot-based CDC transformation using Kettle to find the differences.

1. The first step is to store a second version of the customer table, either in a separate database (the `sakila_dwh` catalog from Chapter 4 might do just fine), or in the Sakila database itself. In a real-life scenario, it's highly unlikely that you'll be allowed to create additional tables in a source system and the data warehouse is probably out as well, so a separate staging area like a `sakila_stg` database would be best.

NOTE **Don't forget to enable Boolean support for the database connections in this example; otherwise Kettle will create the field `valid` as a character string with length 1. Boolean support is enabled by selecting the first option in the Advanced tab of the Database Connection dialog box.**

   For this example, we used the `sakila_dwh` database you already created in Chapter 4, where a `customer_2` table is created and loaded with the original customer data from Sakila.

2. The next step is to make some alterations to the data, for instance change a last_name, make a customer inactive, and insert an extra row. These alterations need to be made in the source system, in this case the sakila database. To show that Kettle is capable of detecting deletions as well, you can insert an extra row in the staging table. This row won't exist in the source system and thus a row deletion is mimicked.
3. Next you need to create the CDC transformation itself. Create two "Table input" steps, one for `sakila` and one for `sakila_dwh`. Select all fields and make sure to sort the data on the key field(s) in ascending order because the Merge Rows step requires sorted input. After the two input steps are created, add a "Merge rows (diff)" step and connect both inputs to this step. Open the Merge step and select the reference and comparison origins, the name for the flag field (that will contain the values `unchanged`, `changed`, `new`, and `deleted`) and the comparison and match keys. Figure 6-27 shows what the step values should look like in this case.
4. You can now issue a Preview on the Merge Rows step to check whether the solution is working. Because all rows will be passed by this step, you'll probably want to filter out the unchanged rows by using a "Filter rows" step. With the condition `flagfield = identical`, you can send all unchanged rows to a dummy output and only keep the new, changed, or deleted rows for further processing. One way of doing this is to pass the data to a subsequent transformation in the job by using a "Copy rows to result" step. Another way is to add a "Synchronize after merge" step that automatically handles the inserts and updates based on the flag field returned from the Merge Rows step. Figure 6-28 shows what the Synchronize step looks like when used in this example.

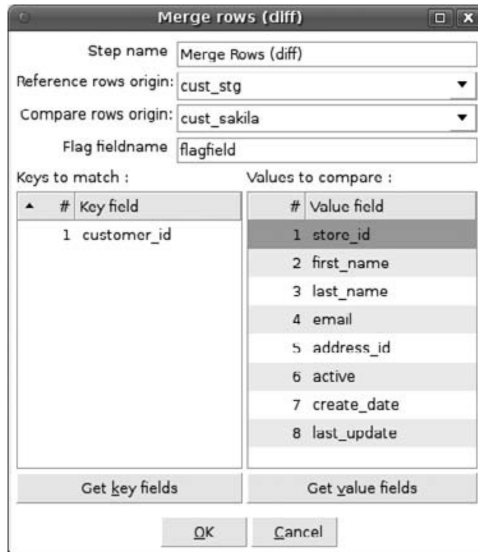


Figure 6-27: Merge Rows settings

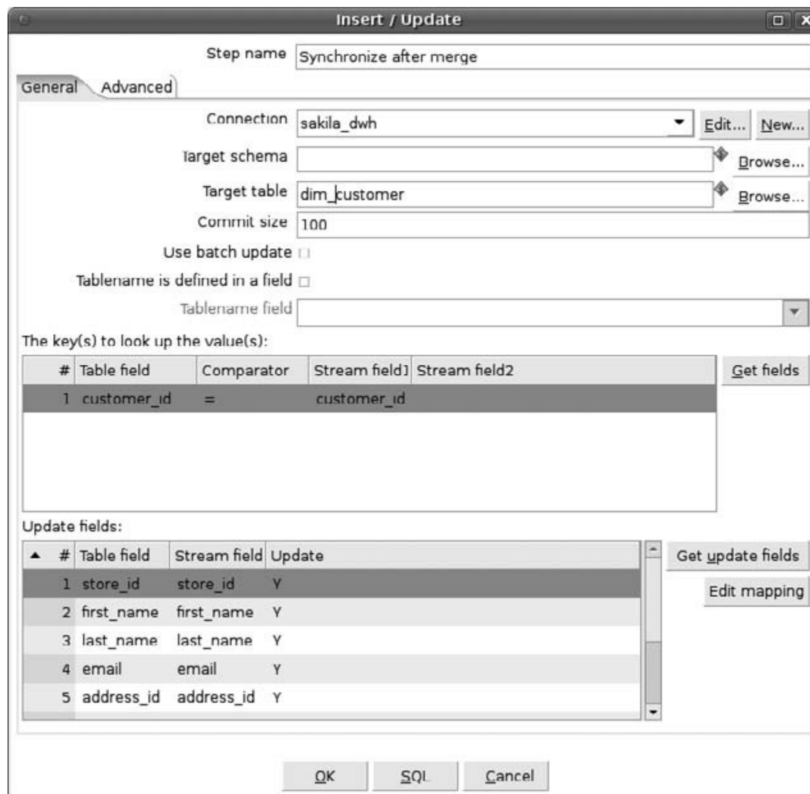


Figure 6-28: Kettle "Synchronize after merge" step

The completed transformation is displayed in Figure 6-29 and is also available from the book's companion site (`sakila_custdiff.ktr`).

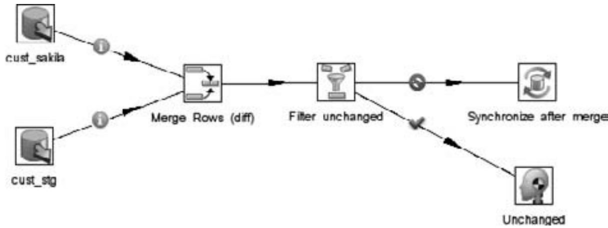


Figure 6-29: Kettle snapshot CDC solution

The final proof that the solution works is shown in Figure 6-30, where a Preview is executed for a CDC_Rows step (the name given to the "Copy rows to result" step).

Examine preview data

Rows of step: CDC_Rows (4 rows)

#	customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update	flagfield
1	1	1	MARY	SMITH	MARY.SMITH@sakilacu	5	Y	2006/02/14 22:04:36.000	2010/03/28 21:07:48.000	changed
2	2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@s	6	N	2006/02/14 22:04:36.000	2010/03/28 22:07:48.000	changed
3	600	2	MATT	CASTERS	MATT.CASTERS@pent	605	Y	2010/03/28 21:12:10.000	2010/03/28 21:02:22.000	new
4	601	1	ROLAND	BOUMAN	ROLAND.BOUMAN@p	5	Y	2010/03/28 21:07:48.000	2010/03/28 21:07:48.000	deleted

Close

Figure 6-30: CDC results Preview

As is shown in the preceding example, snapshot-based CDC can detect inserts, updates, and deletes, which is an advantage over using timestamps, at the cost of extra storage for the different snapshots. There can also be a severe performance issue when the tables to be compared are extremely large. For this reason we added the SQL illustration because for this kind of heavy lifting, the database engine is often better suited than an engine-based ETL tool such as Kettle. Chapter 19 includes an example of CDC detection for a Data Vault data model using Kettle.

Log-Based CDC

The most advanced and least intrusive form of change data capture is to use a log-based solution. Every insert, update, and delete operation run in a database can be logged. In cases using a MySQL database, the binary log has to be enabled explicitly in the Administrator tool (Startup variables ➪ Logfiles). From that moment on, all changes can be read in near-real time from the database log and used for updating the data in the data warehouse. The catch here is that this sounds simpler than it actually is. A binary log file needs to be transformed first into an understandable form before the entries can be read into a subsequent process.

The MySQL installation contains a special tool for this purpose, *mysqlbinlog*. This tool can read the binary format and translate it into a somewhat human-readable format, and can output the read results to a text file or directly into a database client (in case of a restore operation). Mysqlbinlog has several other options, with the most important one for our purposes being the fact that it can accept a start and/or end timestamp to read only part of the log file. Each entry also has a sequence number that can be used as an offset, so there are two ways to prevent duplicates or missing values when reading from these files.

After the mysqlbinlog output is written to a text file, this file can be parsed and read, for instance by a Kettle input step that reads the data and executes the statements on the corresponding staging tables. For other databases there are similar solutions, and some offer a complete CDC framework as part of their data warehouse solution.

The drawback of using a database-specific set of tools is obvious: it only works with a single database. For cases where you need to use a log-based solution in a heterogeneous environment, several commercial offerings such as Oracle GoldenGate and Attunity Stream are available. A possible disadvantage of these offerings might be the price tag attached to them. An interesting alternative, which is relatively new on the market, is the open source Tungsten Replicator that offers advanced master-slave replication options. Tungsten Replicator supports both statement- and row-based replication and is thus a more advanced solution than can be accomplished with the standard binlog reader. Future releases will also support Oracle and PostgreSQL databases, making it a very compelling product. More information is available on the website at `http://www.continuent.com/community/tungsten-replicator`.

Which CDC Alternative Should You Choose?

As you've seen in the previous sections, each of the described options for identifying and selecting changed data has strengths and weaknesses. Some alternatives require adaptations to the source database by a database administrator (DBA), some can support real-time loading of data, and others support only a partial discovery of changes. Table 6-1 summarizes these points to help you decide which option is most applicable in your situation.

Table 6-1: CDC Option Comparison

	TIMESTAMP	SNAPSHOT	TRIGGERS	LOG
Insert/update distinction?	N	Y	Y	Y
Multiple updates detected?	N	N	Y	Y
Deletes identified?	N	Y	Y	Y
Non intrusive?	N	N	N	Y
Real-time support?	N	N	Y	Y
DBA required?	N	N	Y	Y
DBMS independent?	Y	Y	N	N

Delivering Data

After getting changed data using one of the scenarios described, the data needs to be stored somewhere for further processing. Kettle offers many ways to accomplish this; one way was already used in the snapshot-based CDC example. The following list is a selection of the most common options:

- **Use a table output step:** The data can be stored in a separate staging table. The advantage of this is that regular SQL operations can be used in subsequent steps but at the cost of extra overhead and latency. Using a database table is not always the fastest or most efficient solution.
- **Text file output:** In many situations, this will be a preferred solution; data can be written as fast as the disks can store it without the overhead incurred when using a database, and the resulting file is portable to any other system as well.
- **XML output:** Only a viable option if the data needs to be processed by an external system that requires XML input; otherwise use one of the other solutions because of the large overhead incurred when using an XML format.
- **Serialize to file:** This type of output uses a special file format and must be used in conjunction with the accompanying "De-serialize from file" step. It can offer some speed advantages over regular flat files because Kettle doesn't need to parse the file again when it's read, at the cost of being a purely proprietary format.
- **Copy rows to result:** This will keep the data in memory to be pushed to a subsequent transformation that can read the data using a "Get rows from result" step. Probably the fastest option available to push data from one transformation to the next but limited by memory requirements for your data.

Summary

This chapter covered the extraction process from various source systems using Kettle. The first section explained the four main options for retrieving source data:

- File-based extraction using flat files and XML formatted files
- Database-based data extraction using parameterized "Table input" steps
- Web-based data extraction using Web service calls
- Streaming data extraction

Special attention was given to extracting data from ERP and CRM systems, which usually boils down to using a business data layer or API since accessing the database directly is either prohibited or technically cumbersome. We showed how the new SAP Input step can be used to retrieve data from the widely used SAP/R3 ERP system.

The next section covered data profiling, the important process of gaining an understanding of the structure and quality of the data that needs to be extracted. The section

introduced eObjects.org DataCleaner, an open source data-profiling tool that can be used independently because the community edition of Kettle does not provide data-profiling capabilities.

The last part of the chapter explained the different scenarios available for Change Data Capture and showed how Kettle can be used for the following types of CDC:

- Timestamp-based CDC
- Snapshot-based CDC
- Trigger-based CDC
- Log-based CDC

CHAPTER 7

Cleansing and Conforming

For many people, the core value of ETL is hidden behind the *T*, which denotes the *Transform* capabilities. Many people are still not very comfortable with a single *T* for all the work that takes place in this phase. Ralph Kimball has referred to ETL as ECCD, short for *Extract, Cleanse, Conform, and Deliver*, and Matt Casters used the recursive acronym Kettle as a name for the ETL tool where the double *T* stands for *Transportation and Transformation*. In any case, this chapter addresses the four of the 34 subsystems that cover cleansing and conforming data, or more specifically:

- **Subsystem 4:** Data-Cleansing
- **Subsystem 5:** Error Event Schema
- **Subsystem 6:** Audit Dimension Assembler
- **Subsystem 7:** Deduplication

The examples in this chapter are all based on the Sakila customer and address tables, but here we have messed up the data a bit in order to make the cleansing steps actually do something. We created duplicate records, misspelled some names, and added a few extra rows with our own information. The script needed to make these modifications, `sakilamods.sql`, can be downloaded from the book's companion site at `www.wiley.com/go/kettlesolutions`, but you can, of course, make your own modifications as well. The requirements for having the example transformations perform the designated task are described for each example to make it easy to follow along.

A large part of this chapter is devoted to data validation because before you can cleanse data, it must be clear which data must be cleansed and what conditions the data must meet to be considered valid. This area is closely related to data profiling,

already covered in Chapter 6. Profiling is a first step to gain an understanding of the quality and composition of the data. The results of the data-profiling initiative can then be used to build data-cleansing and validation steps.

The importance of the topics covered in this chapter cannot be overstated; in 2003, The Data Warehouse Institute (TDWI) estimated that data quality problems cost U.S. businesses $600 billion each year. Things have probably gotten even worse since then.

Data Cleansing

Cleansing data is one of the core reasons for using an ETL tool in the first place. On the other hand, data cleansing is also a much debated function because it is questionable whether the ETL process is the right place to cleanse your data, or at least whether the data warehouse would be the right place to have cleansed data in. Before we move on to what cleansing is all about, let's briefly look at the main issue with data cleansing.

Cleansing data is a part of a much broader topic—*data quality*—which in turn is a part of the encompassing subject of *data governance*. Arguably, data quality issues should be resolved at the root, which are the transactional and reference data systems. If this data is not clean in the source systems, cleaning it up before loading it into the data warehouse would cause a disconnected state between the information in the source systems and the data in the data warehouse. Data Vault advocates (see Chapter 19) take a different stand in this discussion: Data should be loaded in the data warehouse "as is" and only be cleansed (on request) when moved into a data mart. In this way, there's always a factual representation of how the data was at the time of loading and thus you have an historical logbook of an organization's data. The amount of work for an ETL developer is not reduced by moving the cleansing process more downstream; there's even a risk that there will be more work because data marts are becoming disposable entities. Designing reusable data-cleansing transformations is therefore an important part of the ETL design process.

Kettle offers a myriad of steps to help you with cleansing incoming data, whether it is extracted from the data warehouse or directly from the source systems.

NOTE According to Data Quality guru Arkady Maydanchik there are five categories of data quality rules:

- **Attribute domain constraints: Basic rules that restrict allowed values of individual data attributes.**
- **Relational integrity rules: Enforce identity and referential integrity of the data and can usually be derived from relational data models.**
- **Rules for historical data: Include timeline constraints and value patterns for time-dependent value stacks and event histories.**
- **Rules for state-dependent objects: Place constraint on the lifecycle of objects described by so-called state-transition models.**
- **General dependency rules: Describe complex attribute relationships, including constraints on redundant, derived, partially dependent, and correlated attributes.**

See Arkady Maydanchik, *Data Quality Assessment*, Technics Publications, LLC, 2007.

If we translate the five categories as defined by Maydanchik to the world of Kettle, the first one is obviously the easiest to handle and will be covered in depth in the following sections. Referential integrity rules can also easily be checked using the available lookup steps, but identity integrity is a bit more complex than that. It means that each record in the database points to a single, unique, real-world entity, and that no two records point to the same entity such as a person or a product. We'll look at how this rule can be checked in the "Deduplicating Data" section of this chapter. Beyond the first two basic categories it becomes more complex because there are no standard steps available to accomplish these tasks. Still, it is entirely possible to handle those constraints as well using Kettle. A state-dependent constraint for the sakila database, for instance, is that there cannot be a return without a rental, and that each rental should (at some point in time) have a corresponding return, or a payment for lost property. The allowed period between rentals and returns is a related constraint that can easily be checked. When an accumulating snapshot fact table is in place, this check will be pretty straightforward because each date will usually be equal to or greater than the previous date. Think about a shipment that cannot take place before an order is placed, and a delivery that cannot be done prior to a shipment.

In general, enforcing or checking dependency rules in Kettle will require multiple steps, as you'll see when we cover address validation. An address consists of multiple columns, where multiple validations are needed to check whether a specific address exists in a city, whether the postal code matches the available options for city names, and whether the city name exists in the list of allowable names for a state or country.

NOTE For more in-depth information about the different data quality rules and how to assess data quality, we recommend the book *Data Quality Assessment,* Technics Publications, LLC, 2007, by the already mentioned Arkady Maydanchik. Another good resource for anything data quality–related is the website `www.dataqualitypro.com`**, which also contains a series of articles that explain the data quality categories we introduced in this section.**

Data-Cleansing Steps

There is not a single Data Cleanse step, but rather, there are many steps and other places within a Kettle transformation where data can be cleansed. The data cleansing process starts when extracting data: Many of the input steps contain basic facilities to read the data in a specified format, especially when working with dates and numbers.

WARNING The "Table input" step lends itself perfectly to modified SQL to clean up the data as much as possible before entering the rest of the process. Be careful here, however, because this makes for very hard-to-maintain solutions. It's usually better to read the data "as is" and use Kettle steps to perform the modifications.

There is another reason to avoid SQL modifications to the extracted data: It's impossible to audit data that has been pre-cleansed before it enters the Kettle transformation. Most of the Kettle steps allow you to define an error stream or conditional processing to redirect data that doesn't satisfy certain requirements, as you will see in the "Error Handling" section of this chapter.

The steps in the Transform folder offer many different options for cleansing data. A powerful step and feature-rich example is the Calculator step, which you used in some of the previous chapters. It's nearly impossible to list all the available options, simply because the list of calculations keeps growing at a steady pace. Some of the more useful calculations for cleansing data are the following:

- **ISO8601 week and year numbers:** Most countries outside the United States use a different week numbering based on the ISO8601 standard. Not all databases are capable of deriving the correct week and year number from a date so this is a very helpful calculation.
- **Casing calculations:** The First letter, UpperCase, and LowerCase calculations allow for uniformity in string casing.
- **Return/remove digits:** These can be used to split strings containing both characters and digits in their respective text and numeric parts. A nice example is the fine-grained Dutch postal code, which consists of four digits and two characters (for example, `1234 AB`).

Another step in the list that deserves a special mention is the "Replace in string" step. This seems like a very simple step to replace parts of a string with some other value, but the option to use regular expressions here makes it an extremely powerful solution for many cases. The standard Kettle examples contain a `Replace in string` transformation that illustrates all the available options in this step. Other useful Transform steps are the ones for splitting fields or fields to rows based on a separator value, a "String cut (substring)" step, and the Value Mapper step. This last one is a simple step to replace certain values of the input data with other, conformed values. For instance, the `load_dim_staff` transformation in Chapter 4 uses a Value Mapper step to replace the source values `Y` and `N` with `Yes` and `No` respectively.

With the steps described, you can probably tackle a fair amount of the data-cleansing tasks that need to be performed. There will, however, always be the tough cases that require more programming power. We cover these scripting steps later in this chapter. There are also several related steps that can be used for data-cleansing purposes, such as the special validation steps to verify a credit card number or e-mail address. We look at those as well in the following sections.

One lookup step needs a special mention here, however, because it doesn't return a True/False decision based on some rule, but is able to use "fuzzy" matching algorithms. It is the "Fuzzy match" step that is being covered in-depth in the "Deduplicating Data" section. We mention it here because it contains a collection of string-matching algorithms. Those algorithms are also available in the Calculator step;, the related sidebar explains the purpose of each of the algorithms and the differences among them.

STRING MATCHING ALGORITHMS IN KETTLE

Kettle 4 contains two places where you can find similar algorithms: the Calculator step and the "Fuzzy match" step. The list of available algorithms is almost similar, but the way they work is different; the Calculator step can compare two fields in the same row, but the "Fuzzy match" step can use a lookup table to look at all the records. Before starting to use the steps and algorithms, it's probably a good idea to understand what they actually do. When you open the "Fuzzy match" step and click on the Algorithm drop-down list, you'll see a long list of possible choices with exotic names such as Damerau-Levenshtein, Jaro Winkler, and Double Metaphone. All these algorithms have one thing in common: They are aimed at matching strings. The way they do that, however, varies, which makes some algorithms more suited for, for example, deduplication efforts. The following list briefly explains the various options.

- **Levenshtein and Damerau-Levenshtein: Calculates the distance between two strings by looking at how many edit steps are needed to get from one string to another. The former only looks at inserts, deletes, and replacements, while the latter adds transposition as well. The score indicates the minimum number of changes needed; for instance, the difference between** `CASTERS` **and** `CASTRO` **is only 2. (Step 1: Delete the E; Step 2: Replace S with O.)**
- **Needleman-Wunsch: Also an algorithm used to calculate the similarity of two sequences, and mainly used in bioinformatics. The algorithm calculates a *gap penalty*; hence it will give a score of –2 for the** `CASTERS` **to** `CASTRO` **example above.**
- **Jaro and Jaro-Winkler: Calculates a similarity index between two strings. The result is a fraction between 0 (no similarity) and 1 (identical match). When calculating the Levenshtein distance between** `CASTERS` **and** `POOH` **you'll get 7, while the Jaro and Jaro-Winkler distance will be 0 because there is no similarity between the two strings. Levenshtein finds a distance because you can always get from one to the other by replacing, inserting, and deleting characters.**
- **Pair letters similarity: Only available in the "Fuzzy match" step, this algorithm chops both strings in pairs and compares the sets of pairs. In this case,** `CASTERS` **and** `CASTRO` **will be transformed into the following two arrays with the following values:** `{CA, AS, ST, TE, ER, RS}` **and** `{CA, AS, ST, TR, RO}`**. Now the similarity is calculated by dividing the number of common pairs (multiplied by two) by the sum of the pairs from both strings. In this example there are three common pairs (**`CA`**,** `AS`**,** `ST`**) and eleven pairs in total, resulting in a similarity score of (2*3)/11 = 0.545 (which is, in fact, pretty good).**
- **Metaphone, Double Metaphone, Soundex, and RefinedSoundEx: These algorithms all try to match strings based on how they would "sound" and are also called *phonetic algorithms*. The weakness of all these phonetic algorithms is that they are based on the English language and won't be of much use in a French, Spanish, or Dutch setting.**

Continued

STRING MATCHING ALGORITHMS IN KETTLE *(continued)*

The $64,000 question is, of course, which one to pick, and the answer must be that it all depends on what problem you need to solve. For information retrieval (for instance, retrieving all book titles on Amazon that include the word *Pentaho* with a relevance score attached to them), a pair letter similarity is very useful, but for the purpose of finding matching duplicates possibly containing misspellings, Jaro-Winkler is an excellent choice. Caution is needed, however; `Jos van Dongen` **and** `Davidson` **have a higher Jaro-Winkler similarity score than** `Jos van Dongen` **and** `Dongen, J van`**, but no human being would pick the former over the latter as a possible duplicate candidate.**

Using Reference Tables

In many cases, especially when correcting address information, you will need to access external master or reference data. Data entry applications might already use this kind of data, for instance to display a conformed country or state list. In other cases, it's not so straightforward. Suppose you have a customer complaints system that only has a free form text field for entering the product information. In order to clean this up, some intelligent text preprocessing is needed to match the entered product names to the master tables, which contain the complete product reference data. The most sought after external reference data is correct address information. If this data is available for free, you're lucky; typically a costly subscription is needed to get a monthly or quarterly update from the various companies that maintain and sell this information.

TIP **The site** `www.geonames.org` **contains a broad collection of web services for country, city, and postal code lookup and verification. For the United States, it even contains address-level information.**

Conforming Data Using Lookup Tables

There are a couple of ways to work with reference data. In its most simple form, it's a matter of doing a lookup based on an incoming field and flagging the field as erroneous when no direct match is found. This has limited value as it will indicate only that the data doesn't conform to a reference value. In order to increase the success rate of the lookup, it is better to first try to conform the input stream value as much as possible before the lookup is executed by using any of the data-cleansing steps described earlier.

As an example, let's look at city and postal code lookup. Almost every country uses a postal code, which links to a region, city or, in case of the Netherlands, a specific part of a street. Dutch postal codes are the most fine-grained in the world, which makes it possible to uniquely identify an address by knowing the postal code and the house number. Many times, however, people make mistakes when entering a postal code so in order to avoid getting incorrect data only the digits from a postal code can be extracted

and used to look up the correct city. The following example shows how a combination of calculation and lookup steps can help you determine whether a combination of postal code and city is correct.

First, you need some input data to cleanse; in this example, you use a Data Grid step and add some (almost) dummy data, as depicted in Figure 7-1.

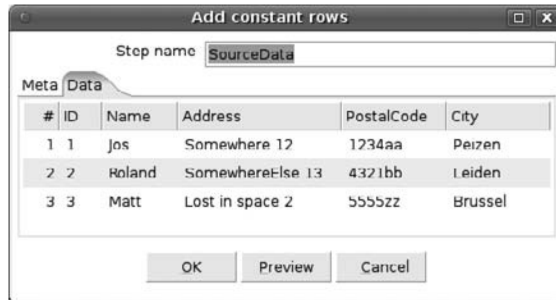


Figure 7-1: Source data to cleanse

The first cleansing step is to get only the digits from the field PostalCode using a Calculator step. The calculation used is, of course, “Return only digits from string A.” The new field should get a meaningful name like PC4. Then you’ll need a reference table. Because this is still a small example using dummy data, you can again use a Data Grid step to mimic a real reference table. The data is displayed in Figure 7-2.

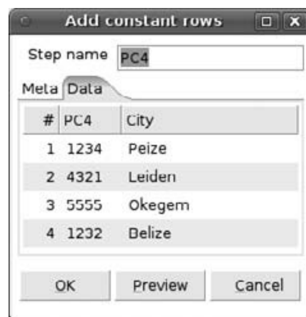


Figure 7-2: Reference data

To retrieve the city name from the lookup table using the four-digit PC4 field, you can use a “Stream lookup” step. Note that in this case, each lookup will succeed, but in real-life scenarios, it is very well possible that some of the codes cannot be found in the reference table. In order to be able to handle those records later in the process, it is advisable to use an easily recognizable default value that is filled in when the lookup fails. Figure 7-3 shows the completed Stream lookup step with the value `***unknown***` for the default field.

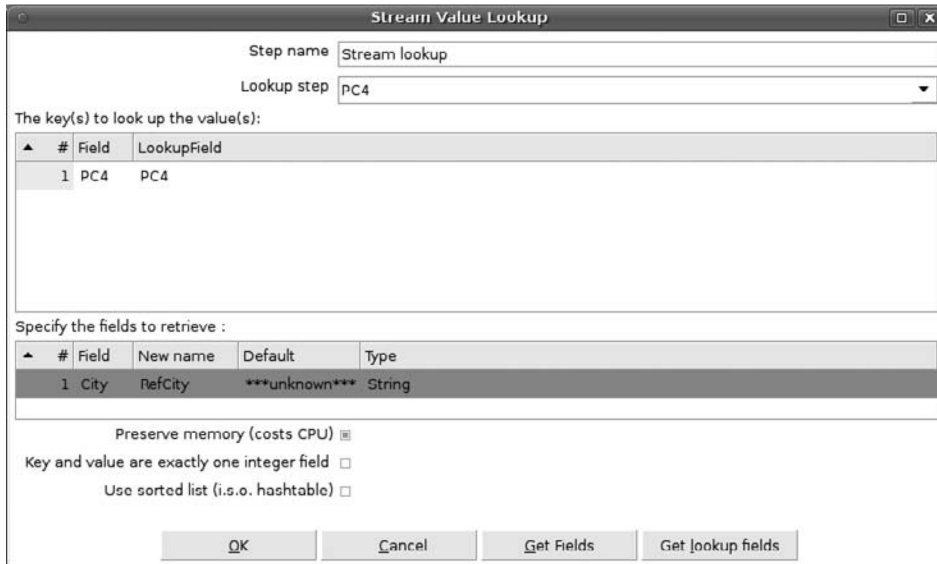


Figure 7-3: Stream lookup

The reason we're using something with a prefix *** and suffix *** is twofold. First of all, it makes a visual inspection of the data easier because the exceptions clearly stand out. Second, none of the string-matching algorithms will find any similarity between the source city value and the lookup value. This second reason allows you to further process the data and look for typing errors or complete mistakes. To illustrate what we mean by that, add a second Calculator step and use City and RefCity as Field A and Field B, respectively. The new field can be named something like cityscore because you'll be using a Jaro-Winkler similarity match algorithm. When you now run a Preview on the transformation you just created, it should output the results displayed in Figure 7-4.

Examine preview data

Rows of step: Preview (3 rows)

#	ID	Name	Address	PostalCode	City	pc4	RefCity	cityscore
1	1	Jos	Somewhere 12	1234aa	Peizen	1234	Peize	0.966666667
2	2	Roland	SomewhereElse 13	4321bb	Leiden	4321	Leiden	1.0
3	3	Matt	Lost in space 2	5555zz	Brussel	5555	Okegem	0.436507937

Close

Figure 7-4: Preview source and reference data

In the first record, you'll notice a very high similarity score between the source and reference city name, so someone probably made a data entry error. The second row is a perfect match (score 1) and can be safely processed further as well. The third row, however, causes a problem: The city names are very different, as shown by the low similarity score (anything below 0.8 could be considered at least suspicious). The problem is that you cannot

tell from this data where the mistake was made: Is the postal code correctly entered but the city name is wrong? Or did someone enter the correct city name but make a mistake when entering the postal code? In order to verify this, a second, reversed validation is necessary. By "reversed" we mean that the lookup to the reference table is now based on city, and the returned postal code can be compared to the reference table. If it then turns out that the city names match and the postal code is very similar (for example, 5556 in this example), it is safe to conclude that someone made an error when typing in the postal code.

Of course many modern applications will automatically check whether a combination of address, postal code, and city is correct when the data is entered. A surprisingly large number of organizations, however, use these validations only for their internal applications, while on the other hand, a customer-facing website allows any kind of data inconsistencies, which makes it very hard to consolidate the data from these different sources.

Conforming Data Using Reference Tables

A special class of reference tables is the *data conformation* master. A well-known example is the coding used for *gender*. Some systems will use the letters `M`, `F`, and `U` for Male, Female, and Unknown; other systems will allow a NULL value for unknown gender; another system might have the full values for `Male` and `Female`; and things like `0` and `1`, or `0` (unknown), `1` (male), and `2` (female) occur as well. To complicate things further, there are also applications that have a separate value for child (`C`) and even might use `F` for Father and `M` for Mother. Variations and combinations are also possible, so this list can go on and on. When the data from all these different systems must be consolidated, a translation needs to be made from all these various encodings to one single coding scheme. A single master table to support this is preferred over different lookup tables per system because there will probably be other conversions needed as well, and a single lookup table is easier to maintain.

Two basic requirements must be fulfilled:

- Every possible value from the source system needs a translation.
- The translation must lead to a single set of values.

Based on the gender example explained earlier, a master table could look like the one displayed in Table 7-1. The fields ref_code and ref_name are the conformed return values you want to obtain, src_sytem is the source system where the data is read from, and src_code contains the possible values that can occur in the input stream.

Table 7-1: Gender Code Reference Table

ID	REF_CODE	REF_NAME	SRC_SYSTEM	SRC_CODE
1	M	Male	Sales	1
2	F	Female	Sales	2
3	M	Male	Web	male

Continued

Table 7-1 *(continued)*

ID	REF_CODE	REF_NAME	SRC_SYSTEM	SRC_CODE
4	F	Female	Web	female
5	M	Male	CRM	F
6	F	Female	CRM	M
7	U	Unknown	CRM	C

Note that this is only an example and the data is displayed in a denormalized fashion for easy reference. It does, however, adhere to the requirements because it leads to only three values in the final destination system: `M`, `F`, and `U`. Now it is fairly easy to translate incoming values from different source systems to a single value for the data warehouse.

To show how this works, we created the preceding table in the same `sakila_dwh` catalog we already used in previous chapters.

NOTE Because the sakila database doesn't contain a code for gender, we used one of the customer data files obtained from `www.fakenamegenerator.com` for Chapter 6 as the source for the following examples.

Conforming data like this can be set up as a separate reusable transformation; the only thing needed to make it generic is to have the `source_system` value passed as a variable to the mapping that takes care of the lookup. There are a couple of ways to go about this; you can first build the complete transformation including the gender lookup step, take the lookup part out, and add the required mappings, or you can directly start by building the reusable step first before building the `outer` transformation that will use the mapping. Either way, there is one thing you'll find out soon enough, and that is that you can't just use a regular database lookup step here if you chose to pass a variable. The "Database lookup" step only allows filtering based on input row values, and because the source system name is not in one of the columns, you first need to add this to the input stream as well. With an "Add constant" step this isn't much of a problem, but adding this value as a constant to the transformation that calls the mapping somehow this doesn't feel right. Let's have a look at what will happen then (see Figure 7-5).

We've added an extra value for every row that's being transformed, just to be able to set a filter condition. Besides being an inefficient solution to the problem, the lookup table is also cached completely while only a small part of the table is needed. This is not a big problem with the few rows in this example, but it could be an issue with more voluminous lookup tables.

A better solution is to use a variable that can be used to filter the data, in combination with a table input step and a stream lookup. Figure 7-6 shows what the main transformation looks like.

on	Domain	system_constant
›rmation officer	SeriousBlog.com	Web
›r	FuelPlants.com	Web
›pairer	URLRankings.com	Web
t	AwesomePoetry.com	Web
otographer	AbacusMath.com	Web
machine tool setter	ChestPimples.com	Web
: controller	Airzilla.com	Web
	AutomobileRetailer.com	Web
lirector	TacomaHandyman.com	Web
›ods operator	BuyCherries.com	Web

Figure 7-5: System name as constant

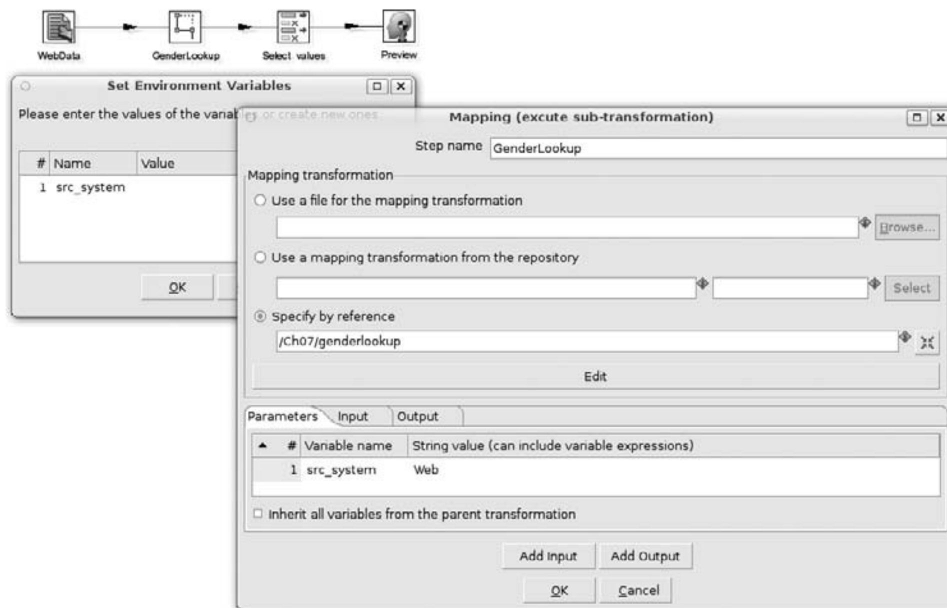


Figure 7-6: Passing a variable

First, a variable is defined, which you can give a value to in the Parameters tab of the mapping step. In this example, the value `Web` is passed to the `genderlookup` transformation. Figure 7-7 in turn shows this reusable transformation that picks up the variable in the "Table input" step.

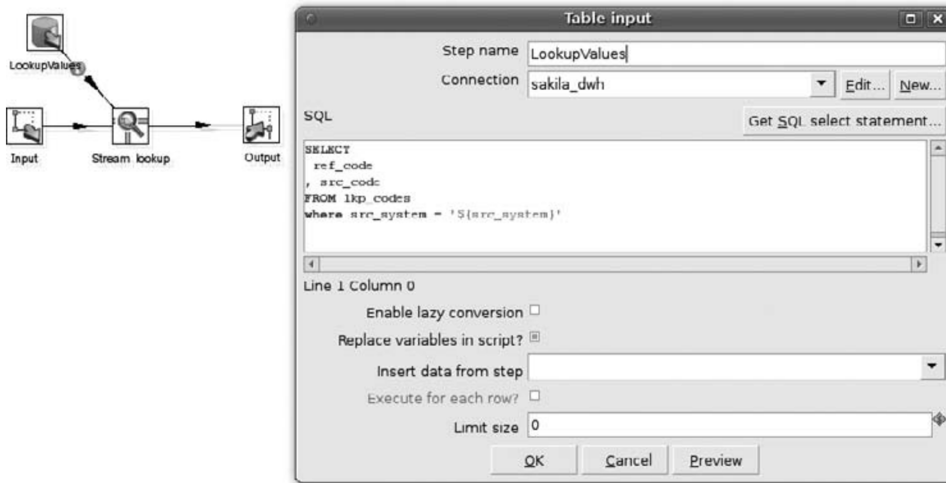


Figure 7-7: Using the passed variable

The stream lookup in this transformation is pretty straightforward; just add the `src_code (input) = src_code (lookup table)` condition and specify the return field. Please note that a default value is mandatory here to be able to handle NULL and unknown values. The completed "Stream lookup" step is displayed in Figure 7-8.

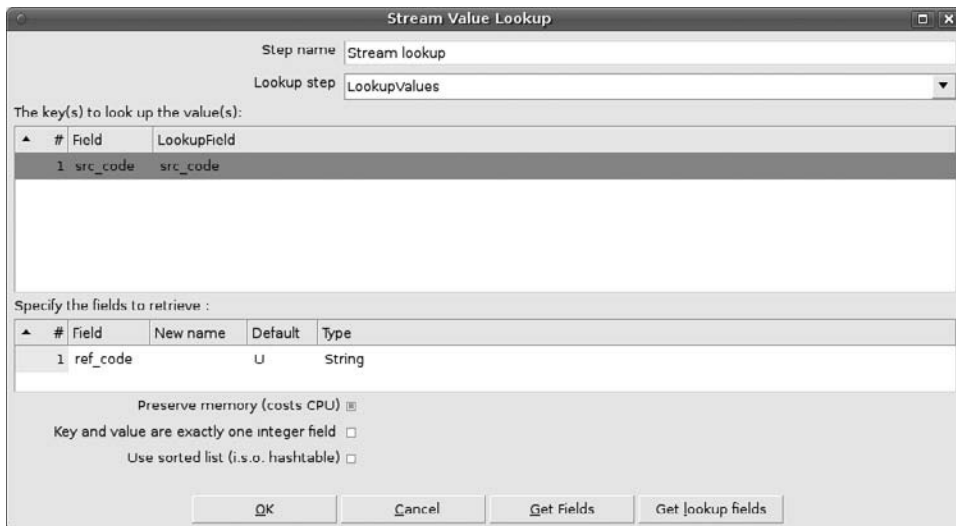


Figure 7-8: Generic "Stream lookup"

NOTE The source data can contain NULL values, but NULL isn't a real value in a database. A comparison like `NULL = NULL` will therefore always fail. This is the reason why NULL is not listed as a separate value in the lookup table and why a default value in the lookup steps is mandatory.

Data Validation

In many situations, data must adhere to certain rules. We've already covered working with reference tables as an example of this, where the rule is, of course: "value must be present in reference table." This is an easy form of data validation because the possible values that can occur are known beforehand. Most data validation rules are, however, a bit more complex than that. Consider the following examples:

- E-mail addresses must be in a valid format.
- Input values must be in upper/lowercase.
- Dates must be in the format `dd-mm-yyyy`.
- Phone numbers must comply with the format `xxxx-xxxx-xxxx`.
- Amounts cannot exceed the value `X`.
- Subscribers must be at least 18 years old.
- IBAN (International Bank Account Number) must be valid.

This list can easily be extended with hundreds of other examples but the idea behind data validation is probably clear by now: Check whether the data conforms to predefined (business) rules and flag or reject any record that doesn't meet these requirements. Note, however, that these examples are all part of the *domain attribute constraints* category. The workhorse in Kettle that's capable of handling all these validations is the Data Validator step. When this step is added to the canvas and opened for the first time, it is completely empty, but as soon as the first validation rule is added by clicking the "New validation" button, you might be overwhelmed by all the different options that are available. It's not that complicated, however; the different options are a result of being able to handle various data types, so not all fields make sense in all cases. Before putting the Validator step in action, let's have a look at some of the key characteristics of this step.

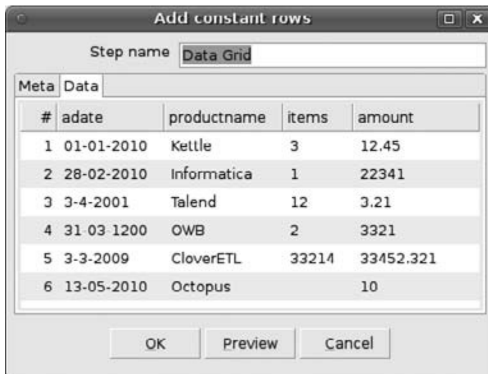
- **Apply multiple constraints to a single column:** Some columns might require multiple checks; there is no restriction in the number of validations that can be applied to the same column.
- **Validate data type:** Especially valuable when text file inputs are used. Conversion masks for dates and numbers make it easy to check for invalid entries.
- **Concatenate errors:** Ability to group all found errors on a row into a single, character-separated field.
- **Parameterize values:** Almost all constraints can be parameterized allowing for one central rule base, which can be maintained outside the ETL tool. Changes to the rule base will then be automatically propagated to the validation steps.

- **Regular expression matching:** Regexes enable very flexible and powerful matching structures and allow for any pattern to be matched, not only the available default options such as `starts with` or `ends with`.
- **Lookup values:** Allowed values can also be read from another step; using multiple steps for multiple validations in a single validation step is also allowed.

The validator itself works much like a highly configurable filter, comparable to the Filter Rows step. All the data that satisfies the various validation rules is sent to the main output stream; all the data that doesn't is reverted to the error stream. Unlike the "Filter rows" step, a secondary output is not required. Nevertheless, we highly recommend using one because only filtering out incorrect data without at least storing these rows somewhere for later reference destroys valuable insights such as why exactly the validation failed or how many rows were rejected.

Applying Validation Rules

To explain what the validation step can do, we've created a small sample set of data using a Data Grid step, which is displayed in Figure 7-9.



#	adate	productname	items	amount
1	01-01-2010	Kettle	3	12.45
2	28-02-2010	Informatica	1	22341
3	3-4-2001	Talend	12	3.21
4	31-03-1200	OWB	2	3321
5	3-3-2009	CloverETL	33214	33452.321
6	13-05-2010	Octopus		10

Figure 7-9: Data to validate

This data has to comply with the following rules:

- None of the fields may contain a NULL value.
- Dates cannot be before January 1, 2000.
- Names outside the official names list are not allowed.
- Items must be between 1 and 10.
- Price per item cannot exceed 1000.

The last validation is a trick question actually because it requires a calculation, which is one of the things that the Validator step cannot do directly. There is no option to add on-the-fly calculations or derived fields and validate the outcome of these, so the only way to accommodate for this is to prepare the data for validation prior to pushing it

into the validation. Figure 7-10 shows the transformation including the amount/items calculation, the allowed product list, and the valid and rejected row output streams.

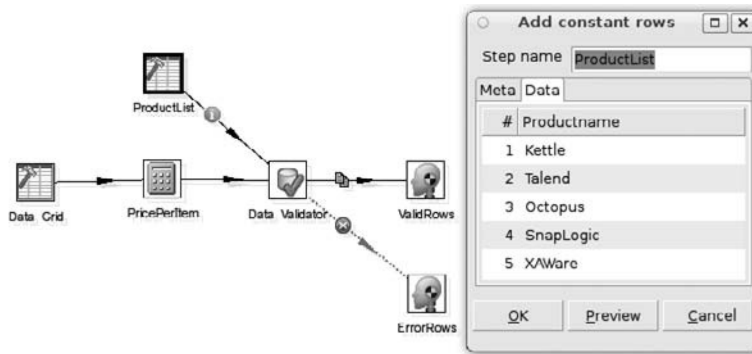


Figure 7-10: Data validation transform

You can now start working on the required validation rules. The first one (no NULL values) already forces you to add a validation for every field in the input. As you do this, you'll notice that the "Null allowed?" checkbox in the Data section is checked by default and needs to be unchecked to enforce the no NULL constraint. The second rule could be added to the already existing validation for the date field, but this will limit the error analysis options later. It is better to create separate validation options for every data error that needs to be trapped in order to process the erroneous data correctly later. After creating separate validations for trapping NULL values and the other validations, the screen will now look like the one displayed in Figure 7-11.

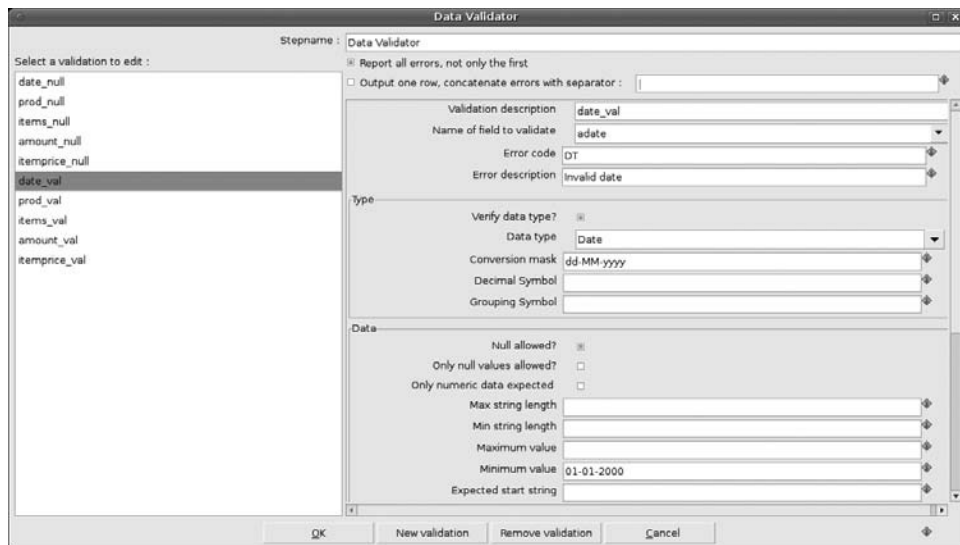


Figure 7-11: Date rule data validation

Figure 7-11 shows a couple of noteworthy items. First, on the left side, all the created validations are visible. The details of the selected validation rule are displayed on the right.

WARNING **For date validations, it's important to specify not only the data type, but also the conversion mask (the date format). If this format is not specified, the default system date format will be used to read the dates. If your system date format is** `yyyy/mm/dd` **but the input date is of the format** `dd-MM-yyyy`**, Kettle will throw an error.**

The second constraint, stating that dates must be at least January 1, 2000, is now correctly enforced. The third one looks pretty straightforward; just mark the "Read allowed values from another step?" checkbox and select ProductList as the step with Product as the field to read from. In order to have Kettle handle the ProductList as a data reference, the hop between ProductList and the validation needs to be created using the mouse over menu of the validation step. If you move your mouse over the Data Validator step and click on the left (input) icon, and then move the mouse to the ProductList step and click on that, a list is displayed with an `optional reference data for validation <validation_name>` entry for all the validations. This is where the option for the validation `prod_val` must be selected. Kettle will now create a special kind of hop that can be recognized by the circled "i" information symbol on the hop, as you can see in Figure 7-10.

The fourth rule, items must be between 1 and 10, is an easy one, too: Enter 1 as minimum and 10 as maximum value and you're done. The last rule is a no-brainer as well and only requires a maximum value of 1,000 to be entered. To check whether the validation works, you can now run a preview on both the `ValidRows` and `ErrorRows` Dummy steps, as shown in Figure 7-12.

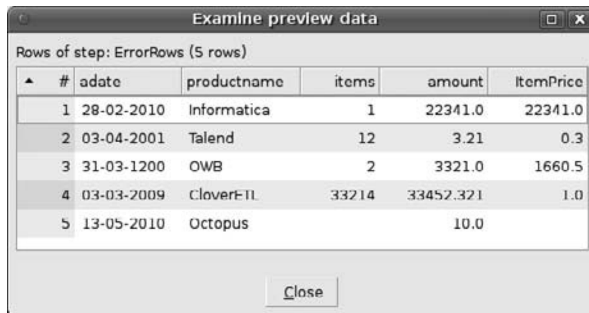


Figure 7-12: Data validation results

Data validation can also be done on the metadata or property level, not on the data itself. For instance, think about the date a file was created or modified, and a rule that specifies that if there's a file older than 10 days the transformation should abort. In order to build such a transformation, let's consider one of the hidden features of Kettle, namely the "Get file names" step. This step does a lot more than just retrieve file names for processing. If you place a "Get file names" step on the canvas, click on the right

mouse button, and select "Show output fields," you'll notice that all the file attributes are displayed. One of those attributes is the `lastmodifiedtime` field, which can be used to calculate the maximum age of all the files in a folder. How this is done exactly is described in the "Scripting" section at the end of this chapter.

NOTE **The `lastmodifiedtime` field only returns usable data if the underlying file system supports tracking this time.**

Validating Dependency Constraints

There are two forms of dependency constraints, very similar to the dependency profiles you'd run as part of a data profiling effort. The first form is the dependency between two or more columns of the same table, also known as *intra-table* dependencies. The second form of dependencies occurs when one or more columns in a table have a dependency on columns in other tables, also known as *inter-table* dependencies. We already showed several examples of this because it is basically a lookup validation problem and not very hard to do with the various steps that Kettle offers for this. Intra-table dependencies are a bit trickier because they usually require extra preparation steps in a transformation.

The deduplication section later in this chapter will show examples of address validation, which is actually a case where inter-table and intra-table dependencies coincide, at least when external address reference tables are being used. Another challenging example of intra-table dependencies is the one between name and gender; if the data contains both first name and gender, it is potentially possible to validate the gender based on the name, and vice versa. This can never be fool proof, however, because many names can be used for both men and women.

Error Handling

The purpose of error handling is obvious: You want your ETL jobs and transformations to gracefully handle any errors that may occur during processing. There are different classes of errors, however:

- **Process errors:** These occur whenever the process cannot continue because of technical reasons. A file might not be available, a server could be switched off (or crashes during a job), or a database password might have been changed without informing the ETL team. And no, this last example isn't fictitious, unfortunately.
- **Data (validation) errors:** Some of the data doesn't pass validation steps. Depending on the impact of the error, the transformation process could still complete, which isn't the case with a process error.

- **Filter errors:** These are not actually errors, but a "Filter rows" step requires two output steps: one for the rows that pass the filter, and one for the rows that don't. In many cases, it suffices to use a Dummy step for the latter category
- **Generic step errors:** Most of the data transformation and validation steps available in Kettle allow you to define error handling to redirect any rows that cannot be handled to a separate output step or to trigger another action such as sending an e-mail to an operator.

The following sections will cover various examples of the described error classes.

Handling Process Errors

Process errors are, at first sight, the most severe of the error classes described in the section introduction because they break the data transformation process entirely. This doesn't mean that there are no other errors that might turn out to be even more serious. An ETL team's worst nightmare is probably the unspotted error: a process that runs flawlessly for several months and then it turns out that the data warehouse data is out of sync with the source data. Early cross-validation of, for instance, record counts and column totals can prevent this from happening; that's why it's so important to use these techniques.

In Chapter 4, you used process error handling in the `load_rentals` job. Each transformation in this job had an error hop pointing to a Mail step, but inside the transformations there's nothing specifically that would cause a transformation failure that would trigger the error. How does this work then? It's actually rather simple. When a transformation runs as designed and all the steps complete successfully, the transformation returns an implicit "Success" signal. If anything goes wrong inside a transformation that causes an error, an implicit "Failure" signal is returned. You'll get this out-of-the-box without any special settings or configuration; it's the way Kettle has been designed to work. Figure 7-15 shows an example of a job with multiple transformations where the Success and Failure notifications are easily recognizable.

The execution of the job can also be followed in a visual way; successfully completed parts of a job are identified with a green checkmark, running tasks have a blue dual-arrow indicator, and failures are indicated with a red stop sign. Figure 7-13 shows what a running job looks like, while Figure 7-14 displays the screen after an error has occurred. Note that there are two locations where the visual indicators are displayed. The success/fail indicators on the hops are always visible, while the indicators for a started job will be placed on the top right corner of the steps that are executed.

Although the job shown in Figure 7-14 might look like a failed job, this isn't actually the case which might be somewhat surprising. The job itself completed successfully, as is clearly visualized in the job metrics displayed in Figure 7-15.

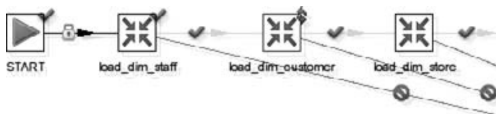


Figure 7-13: A running job

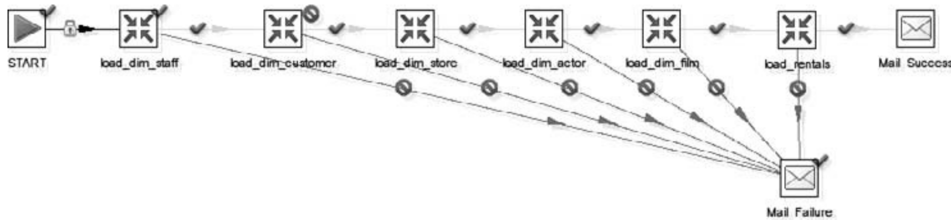
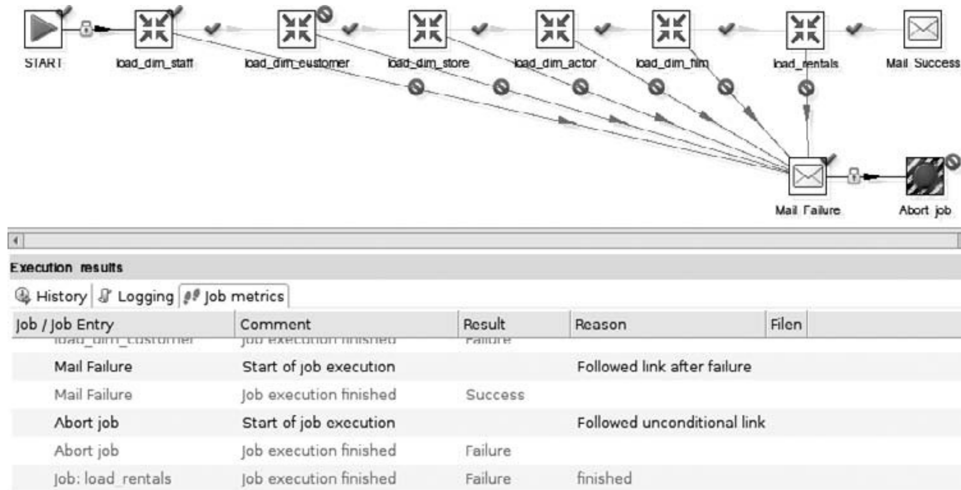


Figure 7-14: A failed job

load_dim_staff	Job execution finished	Success	
load_dim_customer	Start of job execution		Followed link after success
load_dim_customer	Job execution finished	Failure	
Mail Failure	Start of job execution		Followed link after failure
Mail Failure	Job execution finished	Success	
Job: load_rentals	Job execution finished	Success	finished

Figure 7-15: Failed job metrics

Only when the Mail Failure step fails does the job itself return a failed result. What this tells you is that the last executed part of the job determines whether it succeeds or fails. In this case, the Mail step is the last one and if that succeeds, the job returns a successful status. If this weren't the main job but one of the subprocesses, it would look like everything was all right at first glance. To make it clear that this job failed, it's better to have it return an explicit error after sending the error mail message. The "Abort job" utility does just that and can simply be added after the Mail Failure step in the job, as shown in Figure 7-16.



Job / Job Entry	Comment	Result	Reason	Filen
load_dim_customer	Job execution finished	Failure		
Mail Failure	Start of job execution		Followed link after failure	
Mail Failure	Job execution finished	Success		
Abort job	Start of job execution		Followed unconditional link	
Abort job	Job execution finished	Failure		
Job: load_rentals	Job execution finished	Failure	finished	

Figure 7-16: Forcing a Failure return

The important thing to remember here is to use an unconditional hop from the Mail Failure to the "Abort job" step because it's always possible that the mail step itself fails. In this particular case, you would still get the correct outcome of a failed job but that doesn't have to be the case in other circumstances.

Transformation Errors

Jobs execute their constituting parts in a sequential order so it's easy to abort a job at a specific location when an error occurs. That's not the case with transformations because all the steps are started simultaneously. Suppose you need a transformation to abort when a certain condition in the data is not met—consider what this would look like. Figure 7-17 shows a solution most novice Kettle users would create.

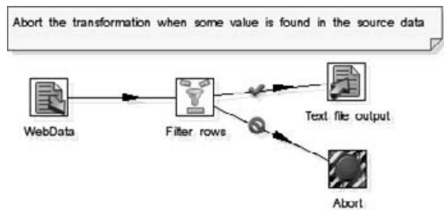


Figure 7-17: Forcing a transformation failure

If the source data volume of the transformation in Figure 7-17 is small enough or the condition is met in one of the first rows, this would actually (almost) work because the output file will be created as soon as the transformation starts. Given enough records in the data source, however, chances are that a considerable number of records will already have been written to the output file before the Abort step kicks in. In these cases, where some condition must be met before the data can be processed, the transformation shouldn't write any data at all. The same applies to validations: if meeting all required validations is mandatory, two transformations need to be created and combined in a job, as shown in Figure 7-18.

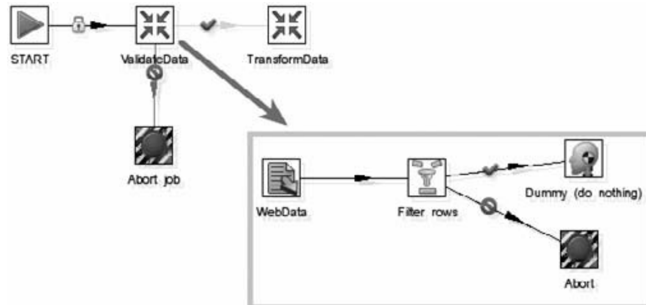


Figure 7-18: Aborting a job

The job in Figure 7-18 will do two things: First the validation transformation (displayed in the right lower corner) is started, and only after it returns a success indicator does the job continue. If the transformation fails, the job is aborted without data being transformed at all. Similar checks could be created for the availability of a server, file, database, table, or column using one of the Conditions steps in a job.

Handling Data (Validation) Errors

So far, we've only covered the use of the Data Validator step as an enhanced type of filter, but it can do a lot more than that. This step can also tell you the exact reasons for rejecting a certain row, enabling you to take appropriate action. First, let's explain the first two checkboxes of the Data Validator step (refer back to Figure 7-11):

- **Report all errors, not only the first:** It is possible that there are multiple validations that fail for a record. If this box is checked, Kettle will perform all validations and output all errors that were found.
- **Output one row, concatenate errors with separator:** When this option is selected, only one row is written to the error stream with all errors in a single field, much like a `group_concat` function in MySQL would work. If the "Report all errors" option is checked and this is not, every failed validation will result in a separate error row.

The next two options to note are the "Error code" and "Error description" fields as these values will determine, for each of the validation rules, how the error is reported. If you need to process the data further after a failed validation, using a conformed set of error codes will help in executing the right cleansing transforms.

Before error handling actually works, you need to enable it. Defining the error codes and defining an error handling hop doesn't do much, other than passing the error rows to a separate step. In order to add the error codes and descriptions to these rows, you need to right-click on the Data Validator step and select Define Error Handling. This opens the "Step error handling settings" panel, as shown in Figure 7-19.

Figure 7-19: Define error handling

In this screen, you can instruct Kettle as to how errors should be handled. Some of the entries here have a close relationship to the error settings in the Validate Data step:

- **Nr of errors fieldname:** The name of the error row column that will store the number of errors in that particular row. If the option "Output one row" is unchecked, this field will always contain 1.
- **Error descriptions fieldname:** The name of the field that will contain the descriptions as entered in the "Error description" field of the validation step.
- **Error fields fieldname:** Displays the name of the field that caused the error.
- **Error codes fieldname:** Displays the code that was entered in the "Error code" field of the validation step.
- **Max nr errors allowed:** If this value is filled in, the transformation will throw an exception (failure code) when the number of errors exceeds this threshold.
- **Max % errors allowed:** Similar to the previous entry but then with a relative instead of an absolute value.
- **Min nr of rows to read before doing % evaluation:** Holds off the % calculation until a certain number of rows has been read. If this value is omitted and the max % is set at 10, Kettle will stop the transformation if there's an error in the first nine rows.

TIP **Here are a couple of tips for handling validation errors:**

- **If you don't explicitly specify an error code and description in the validations, Kettle will create one for you. Codes are in the form** `KV###` **where** `###` **is a number. Although a text description like** `During validation of field 'items' we found that its value is null in row <[13-05-2010], [Octopus], [null], [ 10.0], [null]> when this is not allowed.` **is very descriptive, you might want to specify your own descriptions for brevity and clarity.**
- **It is tempting to use as many checks as possible in a single validation, but it's not always clear what caused the error if you have multiple checks. A best practice here is to use multiple validations per field, for instance one that checks for NULL values and one that checks whether the values are within a certain range.**

Armed with this knowledge, you can now start to create data cleansing flows to handle the data that didn't pass the validation. First, let's look at the result of the validation so far. The results of the error output stream are displayed in Figure 7-20.

As you can see, there seems to be a lot wrong with this data. Look at the bottom two rows: Both `items` and `itemPrice` (a derived value) have a NULL value and this is correctly listed in the error output. In order to get this result, the NULL validations were added separately, while the "Null allowed" checkbox was checked for both existing value validations. If you don't follow this best practice and try to validate this data using just a single validation you'll get the results in Figure 7-21, which are a lot more confusing.

Examine preview data

Rows of step: Errors (9 rows)

#	adate	name	items	amount	ItemPrice	error_desc	error_field	error_code
1	28-02-2010	Informatica	1	22341.0	22341.0	Invalid Product	name	PN
2	28-02-2010	Informatica	1	22341.0	22341.0	Too expensive!	ItemPrice	PR
3	03-04-2001	Talend	12	3.21	0.3	Nr of items outside range	items	ITM
4	31-03-1200	OWB	2	3321.0	1660.5	Wrong date	adate	DT
5	31-03-1200	OWB	2	3321.0	1660.5	Invalid Product	name	PN
6	31-03-1200	OWB	2	3321.0	1660.5	Too expensive!	ItemPrice	PR
7	03-03-2009	CloverETL	33214	33452.321	1.0	Nr of items outside range	items	ITM
8	13-05-2010	Octopus		10.0		Items has NULL value	items	NULL
9	13-05-2010	Octopus		10.0		Item price has NULL value	ItemPrice	NULL

Close

Figure 7-20: Validation errors

7	03-03-2009	CloverETL	33214	33452.321	1.0	Nr of items outside range	items	ITM
8	13-05-2010	Octopus		10.0		Nr of items outside range	items	ITM
9	13-05-2010	Octopus		10.0		Too expensive!	ItemPrice	PR

Figure 7-21: Confusing error descriptions

Figure 7-20 also illustrates why unique and brief error codes are such a good idea: They make it very easy to split the data for further processing using a Switch / Case step. One word of caution here: Because we created error rows for every possible error condition, the error stream now contains duplicate entries. The following example provides one possible method of correcting the date and items field and merging the result back into the main output stream. For practical reasons we'll ignore the other errors for this row here.

A Switch / Case step can be added to the transformation to send the rows to different steps according to their error code. As this step is never an end point, subsequent steps must be created for handling the various codes. Remember that a Switch / Case step can also have multiple entries send the rows to the same output step. The rows with error codes for items and item price (ITM and PR, respectively) can therefore be directed to the same step. Validating data and making corrections to it in the same transformation like this are entirely possible, of course, as shown in the example in Figure 7-22.

The transformation in Figure 7-22 shows how the output of a validation can be directed to a Switch / Case step, which in turn redirects the rows to other steps based on the error code of the row. After the corrections to the `adate` and `items` field have been made, the rows are merged with the rows that were valid in the first place.

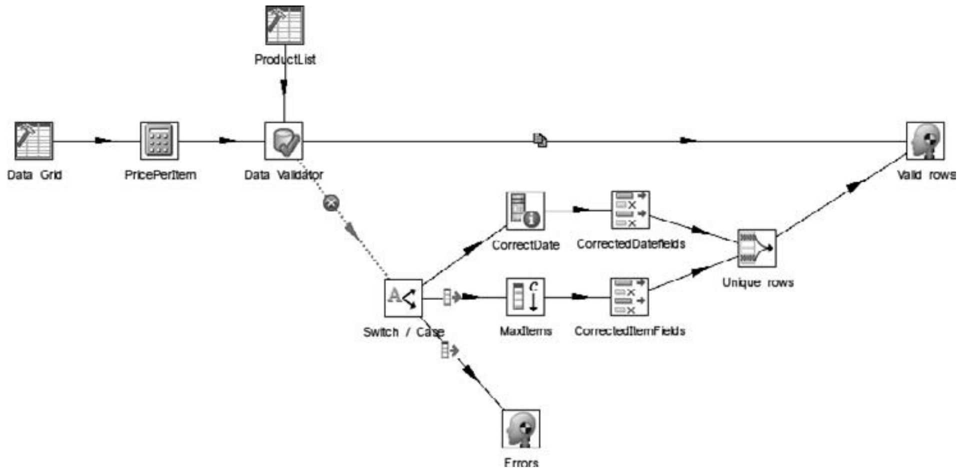


Figure 7-22: Validation and correction

One thing to be aware of is that a Switch / Case step only works based on a list of fixed values or parts of a string when the `Use string contains comparison` option is checked. In programming languages or SQL, a statement like `CASE WHEN somevalue < 5000 THEN 'Pass' ELSE 'Fail'` is fairly common, but the Kettle Switch / Case step cannot be used in this way. If you want this step to handle a case statement like the one in the previous sentence, you need to first translate this formula into the corresponding fixed values and add them to the stream. To do this with numeric values, you use the "Number ranges" step, which makes this a very straightforward task. Figure 7-23 shows an example transform where the aforementioned case statement is translated into a Kettle solution.

Figure 7-23: Preparing case statements

Auditing Data and Process Quality

Auditing the quality of the data that is transformed regularly is, in fact, the third step in a data quality improvement initiative. First there's profiling, then there's validation and error handling, and when these results are stored in separate audit or log tables, this information can then be used to report on and analyze. The final step, which we won't cover here, is the correction of the data that takes place based on the findings of the validation and auditing activities. Figure 7-24 shows a Data Quality Lifecycle, which makes it clear that this is a never-ending process.

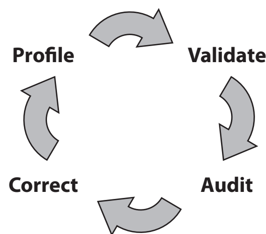


Figure 7-24: Data quality lifecycle

There's a second kind of auditing as well: the process itself. If you read Chapter 4, "Cleaning and Conforming" in Ralph Kimball's *The Data Warehouse ETL Toolkit*, there are two cleaning deliverables defined: an error event table and an audit dimension. The former is modeled as a star schema with a date dimension and a snowflaked "screen" dimension. Translated into Kettle terminology, a "screen" is a transformation, and more specifically, a data validation transformation. Creating an error event schema like this using Kettle is relatively straightforward. We've already shown you how to validate data and create error records for each failed validation. This data can be augmented with system information about the current batch and transformation using the "Get System Info" step, as shown in Figure 7-25.

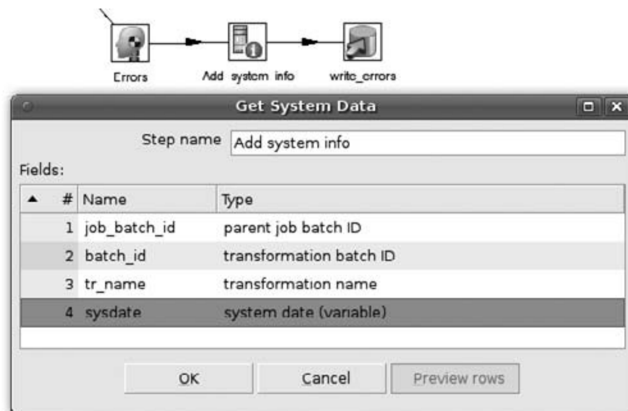


Figure 7-25: Creating error event data

This will store the detailed error data in a separate table. As you'll see in Chapter 14, Kettle provides extensive logging and auditing capabilities, with several options for storing job, transformation, and step-level log-information. The key fields in these logging tables are the same (unique) job and transformation batch IDs used in the example in Figure 7-25. Together, these tables make for a very powerful tool to analyze the quality of the processed data.

The second cleaning deliverable defined by Kimball is the *audit dimension*. The purpose of this table is twofold:

- **Provide a fact audit trail:** Where did the data come from? When was it loaded, by which job/transformation and on which server?
- **Provide basic quality indicators and statistics about the data:** How many records were read, rejected and inserted, whether the fact row is complete (i.e., all dimension foreign keys reference a known entity, not an N/A or unknown record), and whether the facts are within certain bounds.

The observant reader probably has figured out while reading the definition of an audit dimension table that this comes very close to the already mentioned Kettle log tables. In fact, most of the required data is already automatically inserted when logging is enabled, so the only thing left to do is to add the `batch_id (=audit key)` to the fact table load. The changes needed are displayed in Figure 7-26.

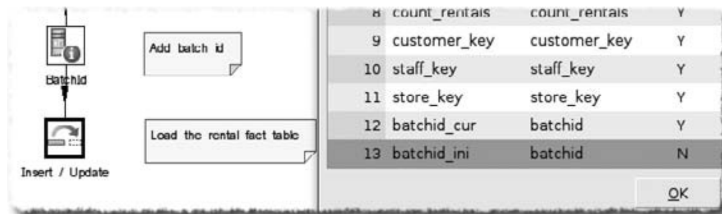


Figure 7-26: Adding batch ID columns

You might wonder why batch ID is listed twice here. This is because it's an updatable fact table and the original batch ID should not be overwritten. In order to accomplish this, one version is updatable (the current batch ID, called `batchid_cur`) and the other one is a static version (called `batchid_ini` in this case) that keeps the original batch ID that caused the insert of the record. Adding the job and transformation batch IDs doesn't have to be limited to the fact records either; every dimension table can be extended in a similar way. By doing this, all the records in the data warehouse can be linked to both the log and the error event tables. The result is a very robust and auditable solution where every piece of data can always be traced back to the process that created it.

Deduplicating Data

Deduplication is in many ways a challenging task. Many large customer tables contain duplicate records; think for instance about CRM (Customer Relationship Management)

systems. Data in these applications is entered by call center agents who need to work fast and don't always have the time to check whether a calling customer is already in the system. In case of doubt, they just create a new customer record and possibly misspell name or address information. This causes problems when the CRM data is used some time later to create mailing lists for marketing campaigns. As a result, many CRM initiatives contain projects to clean up customer data to make sure every customer is listed in a system only once. The problem, of course, is: How do you detect duplicate records? And even more challenging: How do you detect duplicate records when you know there are no shared keys (like postal codes), or the shared keys have the risk of being misspelled as well? The disappointing answer to these questions is that there is no 100 percent fool-proof method or piece of software that completely solves these problems. However, by using "fuzzy" matching logic in Kettle, you can come a long way in getting your customer data in order.

Handling Exact Duplicates

Kettle contains a very simple method to remove duplicate records which is implemented in two similar steps: the "Unique rows" step, and its direct sibling the "Unique rows (HashSet)" step. They both work in a similar way and are easy to use, but the former requires a sorted input while the latter is able to track duplicates in memory. The steps can recognize only exact duplicates and allow the restriction of the duplicate detection to certain fields. You can put this step to good use, for example, when an organization is preparing a direct mailing but only wants to send one mail package per address. The input data would have to contain customer ID and address information and needs to be sorted on address.

NOTE **Example prerequisite: At least two customer records with the same address ID.**

The ReadSource step contains the following query:

```
SELECT   customer_id, last_name, address_id
FROM     customer
ORDER BY 3
```

The `last_name` field isn't strictly necessary but gives a little more information about the duplicate entries found. Figure 7-27 shows an example transformation where the duplicate records are redirected to the error stream of the step.

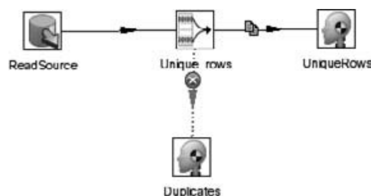


Figure 7-27: Unique rows

The “Unique rows” step works in a pretty straightforward way, as shown in Figure 7-28. The duplicate rows redirection option is selected in order to push the duplicate rows to the error stream of the step, and the only comparison is on the field `address_id`.

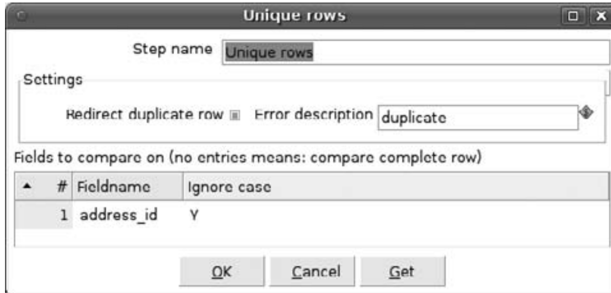


Figure 7-28: “Unique rows” step

The two dummy steps are for easy testing; when a preview is issued on the dummy step with the name Duplicates, the duplicate rows will be displayed. For this example, you get the output shown in Figure 7-29.

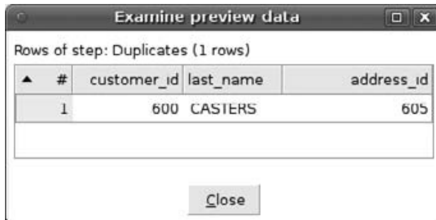


Figure 7-29: Unique rows preview

The Problem of Non-Exact Duplicates

As explained in the introduction to this section, most data quality problems arise in customer and product data sets. For the discussion here and the examples later on we’ll limit our scope to customer data because it is easy to understand. Let’s introduce some typical examples of duplicate records based on names. Suppose the CRM system only stores first and last name, e-mail address, city, and country, as displayed in Table 7-2.

Anyone who can read will immediately see that these two records probably point to the same person, but what if there are 3, 30, or even 100 million records in the table? It’s impossible to correct very large data sets by just browsing through the data.

Table 7-2: Duplicate Entries Example

ID	FIRSTNAME	LASTNAME	E-MAIL	CITY	COUNTRY
3	Roland	Bouman	roland.bouman@gmail.com	Leiden	NL
433	Rolan	Bowman	rolan.bowman@gmail.com	Leiden	NL

The first thing needed is a check at the data entry point; things such as City, Postal code, and Country don't have to be typed in and most modern applications don't allow you to enter invalid address/postal code/city combinations. Whatever the checks are, you'll need some columns to contain accurate data; otherwise, there will be nothing that links the records together. In Table 7-2, it's the city and country that you know are correct (or at least cannot be misspelled). If there is no way potential duplicates can be tied together, deduplication is not possible at all.

The next thing needed to solve duplication errors is ample computing power. Finding possible duplicate records means searching the complete table for every available record. With 1 million records, that means a million searches for each of the million records (1,000,000 * 1,000,000). When doing this multiple times for multiple fields, it gets worse. This is one of the reasons why the specialized tools on the market that can do this very efficiently are still very expensive. The other reason is the built-in "knowledge" of these tools that enables them to match and correct customer and address information automatically.

Finally, you need algorithms to identify potential duplicate entries in your data. Some fields will contain exactly duplicated information, as in Table 7-2, and some fields will contain misspelled or differently spelled entries. Using plain SQL to identify those entries won't cut it: Equality checks will fail, and using "like" operators won't work either because you'll need to specify the search string in advance. The only way to find possible duplicates is to use so-called *fuzzy logic* that is able to calculate a similarity index for two strings, as we will show in the following sections.

Building Deduplication Transforms

Based on the information in the previous sections it shouldn't be too hard to figure out a deduplication approach. We'll propose an approach consisting of four steps here, but you are of course free to add more steps or augment the demo solution. The step that will be doing most of the magic is the already mentioned "Fuzzy match" step. This step works as follows:

1. Read an input field from a stream.
2. Perform a lookup on a field in a second stream using one of the match algorithms.
3. Return matches.

Figure 7-30 shows the available options on the main tab of the step. As you can see, there is a main input step (ReadSource) of which the stream field `src_lastname` is used, and a second "Table input" step that provides the lookup data. The available options in the Settings part of the step properties depend on the chosen algorithm. If you select one of the phonetic algorithms (Soundex, Metaphone, or their improved versions) none of the options can be set, and the "Case sensitive" checkbox is only available for the Levenshtein algorithms.

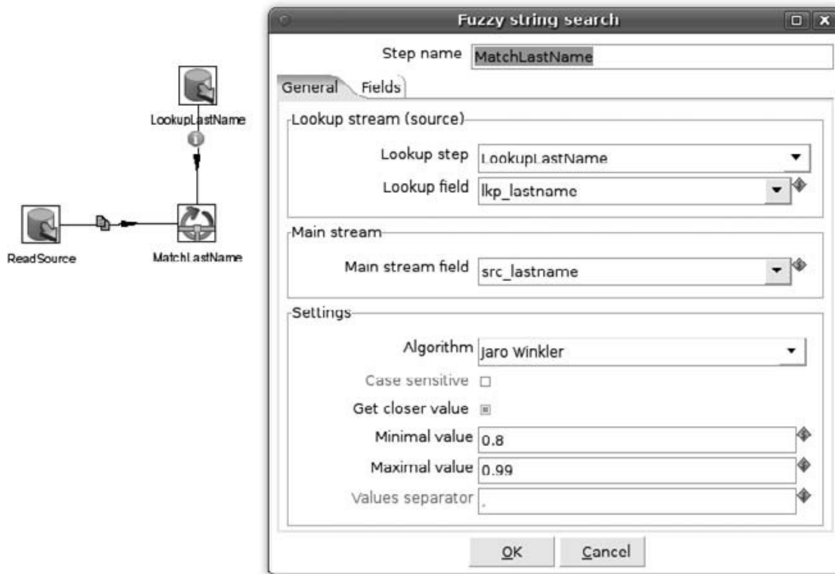


Figure 7-30: Fuzzy match basics

The other options can be used to fine-tune your matching, and also enable you to retrieve one or more possible matches. The "Get closer value" setting is an important one here as this will determine what data is returned and whether the result can be used for deduplication. If the "Get closer value" box is not checked, you'll notice that the Values separator can be set. As a result, all matches that satisfy the Minimal and Maximal value setting will be returned as a separated list. Figure 7-31 shows the result of the preview of a fuzzy match on the last name column in the sakila database with the settings as shown in Figure 7-30 and the "Get closer value" box unchecked.

Examine preview data

Rows of step: MatchLastName (600 rows)

#	src_custid	src_lastname	src_email	match
9	9	MOORE	MARGARET.MOORE@sakilacustomer.org	MORALES,MORRELL
10	10	TAYLOR	DOROTHY.TAYLOR@sakilacustomer.org	
11	11	ANDERSON	LISA.ANDERSON@sakilacustomer.org	SANDERS,GUNDERSON,HENDERSON,ANDREWS,ANDREW,PATTERSON,PEARSON,LARSON,HANSON,NELSON
12	12	THOMAS	NANCY.THOMAS@sakilacustomer.org	THOMPSON
13	13	JACKSON	KAREN.JACKSON@sakilacustomer.org	JACOBS
14	14	WHITE	BETTY.WHITE@sakilacustomer.org	WHITING,WHEAT,HITE
15	15	HARRIS	HELEN.HARRIS@sakilacustomer.org	HARDISON,HARKINS,RICHARDS,HART,ARTIS,HARDER,HARPER,HARRISON
16	16	MARTIN	SANDRA.MARTIN@sakilacustomer.org	MCCARTNEY,MARTEL,ARTIS,MARK,MARTINO,MARTINEZ

Close

Figure 7-31: Preview multiple match data

The name of the match column in the preview can be set on the Fields tab of the step. These results are not very useful for further processing. The main problem is not the number of values, but the lack of a link to the lookup table in the form of a key or other reference. For deduplicating data, it's not enough to have the possible matches—you also need to know exactly which records provide those matches. Hence it is better to check the "Get closer value" box. This will only return a single result (the one with the highest similarity score), but at the same time enables you to retrieve additional column values from the lookup table, as shown in Figure 7-32. The Match and Value field names speak for themselves, and in the Fields section you can specify which additional fields you want to read from the lookup stream.

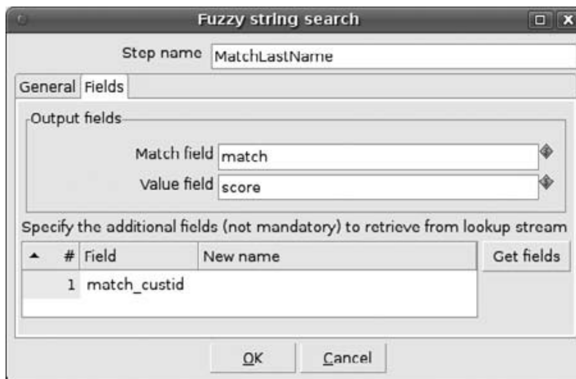


Figure 7-32: "Fuzzy match" return fields

Based on the knowledge you've obtained so far, you can start building a deduplication transform using the "Fuzzy match" step. In order to get some results, you need to modify some data. The simple example we're going to build uses the last name field to search for duplicate entries and the e-mail field as a "reference." Therefore, we need at least two records with the same e-mail address and last names that are somewhat (or exactly) similar.

NOTE **In this example, it would of course be a lot easier to simply count e-mail addresses and filter duplicates, but that's not the point here.**

Let's start by exploring the four-step process.

Step 1: Fuzzy Match

In this example, fuzzy matching is indeed the first step. In real-life scenarios, it's probably preceded by some other steps to get the data as clean as possible using regular expressions or one of the other techniques covered in this chapter. For instance, if a source data file contains only a single name field, which contains the full person name, a regular expression could be used to split the field into first name, middle initial, and last name for U.S. names, and possibly other splits for names from other countries.

One of the authors of this book, for instance, has a name that's quite common in the Netherlands but is usually misspelled in the United States because it includes "van" (with a lowercase *v*) which is mistakenly treated as a middle name.

The initial fuzzy match is the same one that you saw in Figure 7-30 and uses a table input step that retrieves the following data:

```
SELECT customer_id AS src_custid
,      last_name   AS src_lastname
,      email       AS src_email
FROM   customer
```

For matching the incoming data with reference data, a second table input step is needed to 'feed' the "Fuzzy match" step. The following data is retrieved:

```
SELECT customer_id AS match_custid
,      last_name   AS match_lastname
FROM   customer
```

The details of the "Fuzzy match" step are the same as the ones displayed in Figure 7-30.

Step 2: Select Suspects

Because we are using a lower boundary of 0.8 for the Jaro-Winkler algorithm, not all records will get a possible match using the fuzzy lookup. These must be filtered out before proceeding, which involves a very straightforward use of the "Filter rows" step with the condition `match_custid IS NOT NULL`. All records that comply with this condition are passed to the next step of the process; the other records are discarded

Step 3: Lookup Validation Value

In this example, you use e-mail addresses as an extra reference, but this could of course be any other address field as well. What happens here is that the `match_custid` found in the fuzzy lookup step is used to retrieve the e-mail address of the possible duplicate record. As a result, you now have records that contain two customer IDs, two names, and two e-mail addresses. Figure 7-33 shows the partial results.

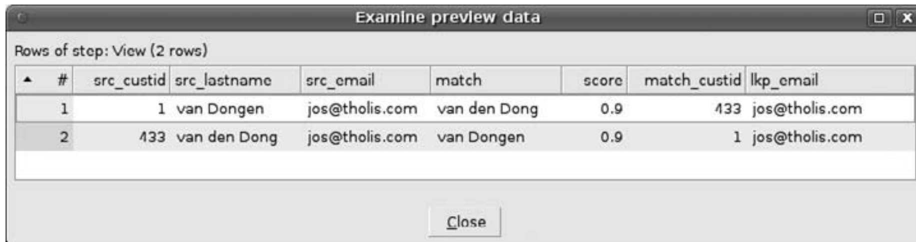
Rows of step: Lkp_Email (475 rows)

#	src_custid	src_lastname	src_email	match	score	match_custid	lkp_email
16	17	THOMPSON	DONNA.THOMPSON@sakilacustomer.org	THOMAS	0.9	12	NANCY.THOMAS@sakilacustomer.org
17	18	GARCIA	CAROL.GARCIA@sakilacustomer.org	GARZA	0.9	224	PEARL.GARZA@sakilacustomer.org
18	19	MARTINEZ	RUTH.MARTINEZ@sakilacustomer.org	MARTIN	1	16	SANDRA.MARTIN@sakilacustomer.org
19	20	ROBINSON	SHARON.ROBINSON@sakilacustomer.org	ROBINS	1	472	GREG.ROBINS@sakilacustomer.org
20	21	CLARK	MICHELLE.CLARK@sakilacustomer.org	CLARY	0.9	525	ADRIAN.CLARY@sakilacustomer.org
21	22	RODRIGUEZ	LAURA.RODRIGUEZ@sakilacustomer.org	RODRIQUEZ	1	289	VIOLET.RODRIQUEZ@sakilacustomer.org
22	23	LEWIS	SARAH.LEWIS@sakilacustomer.org	WILES	0.9	455	JON.WILES@sakilacustomer.org
23	25	WALKER	DEBORAH.WALKER@sakilacustomer.org	WALTERS	0.9	269	CASSANDRA.WALTERS@sakilacustomer.org

Figure 7-33: Possible duplicates

Step 4: Filter Duplicates

Selecting the final duplicates is now pretty simple. You again use a "Filter rows" step with the condition `src_email = lkp_email` to get your possible duplicate records, which are displayed in Figure 7-34.


Examine preview data

Rows of step: View (2 rows)

#	src_custid	src_lastname	src_email	match	score	match_custid	lkp_email
1	1	van Dongen	jos@tholis.com	van den Dong	0.9	433	jos@tholis.com
2	433	van den Dong	jos@tholis.com	van Dongen	0.9	1	jos@tholis.com

Close

Figure 7-34: Possible duplicates

We refer to the records in Figure 7-34 as *possible* duplicates for a reason. Although you can be pretty sure in this case that you're talking about the same human being here based on the information at hand, it is still entirely possible that you're looking at a married couple sharing one e-mail address. It can get much worse, too. For example, two records pointing to the same person, both records with valid addresses where one is a street address and the other is a post office box. Also consider the possibility that most people have more than one e-mail address. The list of possibilities goes on and on.

Another problem with deduplication is the question of validity: Even if duplicates are detected with different addresses or phone numbers, how can you tell which address or phone number is correct? Not all source systems keep a detailed change log at the field level, and even if they did, there's still the possibility of human error because someone must enter and update the information.

Merging information from two or more records into one should be handled with care. First, there's the issue of which data should overwrite the other. Second, addresses should be treated as a single block. Merging the city name of one record into a second one with an empty city column usually doesn't make a lot of sense. With all these cautions in mind, and a basic version of a deduplication transform described here, you can start tackling those tricky deduplication efforts. As a reference, the completed transformation is displayed in Figure 7-35.

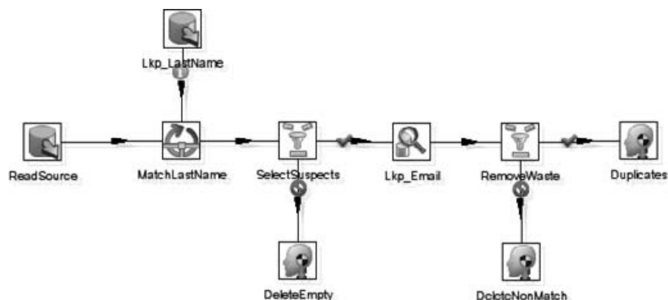


Figure 7-35: Possible duplicates

Scripting

Most ETL developers have a love-hate relationship with scripting. You strive to develop easy-to-maintain solutions where you don't need to revert to custom coding, but as a last resort, you can always solve even the most daunting problems with a script. Historically, the Java Script step has been the "duct tape" of Kettle, where a lot of the complex transformation work took place. Over the years, however, more and more standard steps have been introduced to replace most of the functions that previously needed a Java Script step. As a general rule of thumb, you should try to avoid using scripting altogether, but there will always be some requirements that simply cannot be met with one of the regular steps.

At the time of this writing, the Scripting transform category contains seven different scripting steps and there is at least one other in the making.

Because there are very few use cases for the SQL script steps, let's concentrate on the other available steps: Formula, Modified Java Script Value, User Defined Java Expression, User Defined Java Class, and Regex Evaluation. Before we cover each of them in more depth, we'll provide a brief overview of what they have to offer:

- **Formula:** This isn't a real scripting step but it does allow for more flexible formulas than with the predefined calculations in the Calculator step. The formula language is the same as the one used in OpenOffice, so if you're familiar with formula expressions in Calc you'll feel right at home
- **Modified Java Script Value:** This step offers the full breadth of JavaScript to be used in a transformation. JavaScript enables you to read files, connect to databases, output information to pop-up screens, and so on—it has many more capabilities than you'll generally need.
- **User Defined Java Expression:** This step lets you use Java expression directly, with the advantage that these are translated into compiled code when the transformation is started. Of the available scripting steps, this is the highest performing one.
- **User Defined Java Class:** Instead of just a single expression, this step lets you define a complete class, which basically allows you to write a Kettle plugin as a step. For an example of how to use this step, go to Matt Casters' original blog post at `http://www.ibridge.be/?p=180`
- **Regex evaluation:** Designed to parse regular expressions ranging from short and simple to very large and complex, and is also capable of creating new fields from *capture groups* (parts of the regular expression).

The choice of which step to use for which purpose is always a tradeoff between ease of use, speed of development, and speed of execution. An excellent overview of the advantages of one step over the other is available in a blog post by one of the authors and can be found online at `http://rpbouman.blogspot.com/2009/11/pentaho-data-integration-javascript.html`.

Formula

The Formula step is the Calculator's step direct sibling and the two of them can cover most of the things that previously required JavaScript. Working with the Formula step is especially useful when conditional logic must be applied to the data, or calculations need to be made that are not available (yet) in the Calculator step. Date arithmetic is a good example: the Calculator step lets you calculate the number of days between two days, but not the number of months or years. With a Formula step this can be done but there's a catch: As soon as a date is in a different month, even if there's only one day between the first and second date, the `datedif` function will calculate this as one month.

A formula in a Formula step is also different from a formula in the Calculator step in another sense: It cannot work with the fields defined in it. In order to create a condition based on a calculated field, two Formula steps are needed. Finally, the fields that need to be referenced in a Formula step must be typed in manually; there is no drop-down list to choose from, as there is in the Calculator step. As an example, look at the file age calculation we promised earlier in the chapter. The first three steps get file names, add the current date to the stream and select the file name, sysdate, and last modified date. You can use any set of files to work with; this example uses the files with a `.conf` extension in the `/etc` folder on a Linux system. First you create a Formula step that creates the new field `age_in_months`; then add a second Formula step to determine whether the file is too old. The transformation is displayed in Figure 7-36.

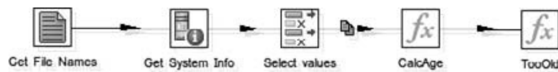


Figure 7-36: Determine file age using Formula steps

The first formula in this transformation is `datedif([lastmodifiedtime];[today];"m")`; the second one is `if([age_in_months]>24;"Too Old";"OK")`. The output of a transformation like this is displayed in Figure 7-37.

Examine preview data

Rows of step: TooOld (42 rows)

#	filename	lastmodifiedtime	today	age_in_months	TooOld
9	/etc/fuse.conf	2009/03/05 18:42:52.000	2010/05/17 00:00:00.000	14	OK
10	/etc/gai.conf	2008/03/26 18:44:50.000	2010/05/17 00:00:00.000	26	Too Old
11	/etc/gssapi_mech.conf	2009/06/08 12:24:44.000	2010/05/17 00:00:00.000	13	OK
12	/etc/hdparm.conf	2009/10/06 22:36:37.000	2010/05/17 00:00:00.000	17	OK
13	/etc/hesiod.conf	2009/04/20 16:07:21.000	2010/05/17 00:00:00.000	13	OK
14	/etc/host.conf	2008/12/23 19:53:36.000	2010/05/17 00:00:00.000	31	Too Old
15	/etc/idmapd.conf	2009/10/20 18:30:31.000	2010/05/17 00:00:00.000	17	OK

Close

Figure 7-37: File age output

JavaScript

With all the powerful new transformations added to Kettle in version 4, the need for JavaScript has declined considerably. Nevertheless, there are various use cases where you need JavaScript. Chapter 10 contains an example where a loop is performed over the data stream, something that cannot be done with another step. One other example that we'll provide here is based on the Rentals table in Sakila. This table contains a rental and a return date, and we want to flag customers who returned films more than a week after the rental date. The most obvious strategy would be to look at the Calculator or Formula step for this, but neither of these is capable of calculating the number of weeks between two dates. With a Java Script step, however, this is easy. Let's start with an input table with the following SQL:

```
SELECT rental_id
,      customer_id
,      rental_date
,      return_date
FROM   rental
```

After adding the Java Script step, you need to determine the number of weeks between the rental date and return date using the `dateDiff` function. Adding something like `var age = dateDiff(rental_date,return_date,"w");` will do the trick.

NOTE **Unlike SQL, JavaScript is case-sensitive so make sure to use the correct casing for functions;** `dateDiff` **is something entirely different than** `datediff`**.**

This gives you the week number but not yet the Late flag for records that have a "weeks" value of 1 or more. There are several approaches you could take now: Use a Value mapper, a Formula, or add a little code to the JavaScript step. If you want to use as little JavaScript as possible, go for the Value mapper. The reason is very simple: While a Formula would work in this case, the conditional logic will get messy when the solution needs to be refined further—for example, to distinguish between early returns (< 1 week), late returns (> 1 week but < 2 weeks), and very late returns (> 2 weeks). In this example, we've augmented the JavaScript code to calculate the requested output directly, as shown in Figure 7-38.

As the JavaScript in Figure 7-38 shows, only the `status` field is added to the output stream, while the `age` field is only used to determine the status.

User-Defined Java Expressions

The Java Expressions step looks very similar to the Formula step covered earlier, the only difference being that instead of formulas, you're able to use Java Expressions. The Pentaho wiki has a good introduction at `http://wiki.pentaho.com/display/EAI/User+Defined+Java+Expression`.

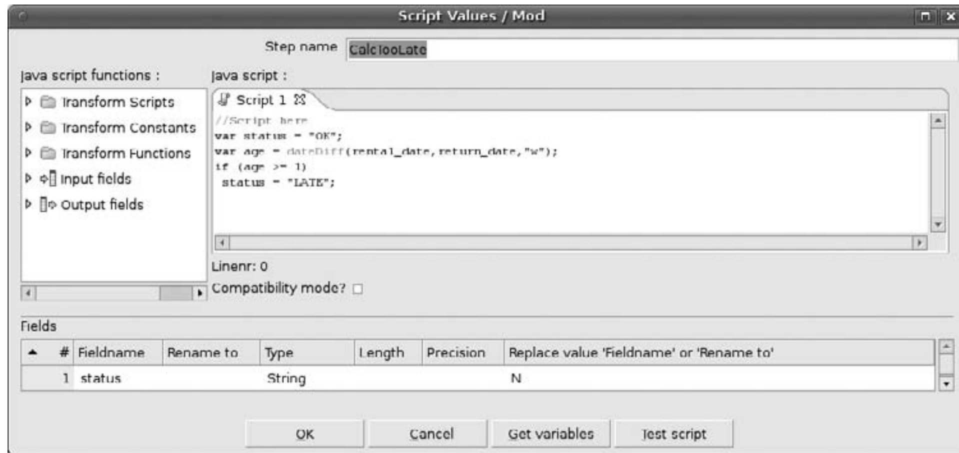


Figure 7-38: JavaScript for week calculation

One small example of using Java Expressions in a cleansing process is from the Mozilla Corporation, which makes extensive use of Kettle for processing weblogs. Because the amount of data transformed is already a stunning several gigabytes per hour (and still growing), they want to squeeze every bit of performance possible out of Kettle. Figure 7-39 shows one of their data-cleansing transformations in which three Java Expression steps are used.

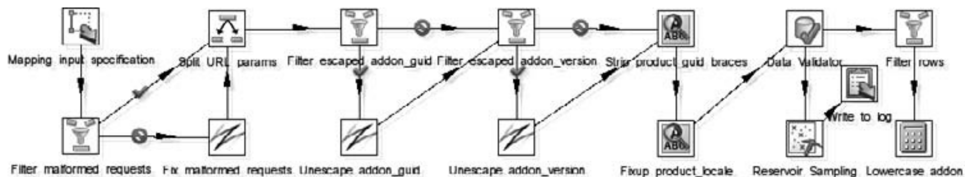


Figure 7-39: Java Expressions in data cleansing

The first Java Expression cleansing step uses `url.replaceAll("([^&])(appID|appVersion)", "$1&$2")` to replace all regex matches inside the first string with the replacement string specified. The two "unescape" expressions use the `Java.net.URLDecoder.decode(<urlpart>)` function to decode (part of) an encoded URL and will, for instance, return `Roland, Jos & Matt` from the input `Roland,+Jos+%26+Matt`. This transformation is available from the book's website as a reference.

Regular Expressions

Throughout Kettle there are many steps where regular expressions, or regexes, can be used. They are commonly used in the locations where wildcards can be used such as the "Text file input" or "Get File Names" steps, or to search for a string using a "Replace in string" step. JavaScript can also be used to evaluate regular expressions using the `str2RegExp` function, but the real regex power is delivered by the Regex Evaluation step.

> **NOTE** **The original version of this step was only capable of returning a Boolean indicating whether the regular expression matched the target field. Later, a Pentaho community member enhanced the step to support capture groups for creating new fields using values matched in the target field. This enhancement was contributed back to the open source project with an extensive sample transformation that ships with Kettle. This sample is the transformation** `Regex Eval - parse NCSA access log records.ktr`**, which can be found in the** `/samples/transformations` **directory under the main Kettle install directory. The Kettle wiki contains an extensive explanation of this step (**`http://wiki.pentaho.com/display/EAI/Regex+Evaluation`**) and also has pointers to regex tutorials and reference sites to help you on your way with regular expressions.**

The extensive sample might be a bit overwhelming so let's use a very simple example here. The first thing you'll need is input data. For this example, you'll use the Sakila address table and a Regex Evaluation step to split the street name and house number into two separate fields. The "Input table" step gets the data from the Sakila database using the following query:

```
SELECT address_id
,      address
FROM   address
```

Then the Regex Evaluation step will use `address` as the "Field to evaluate," as shown in Figure 7-40.

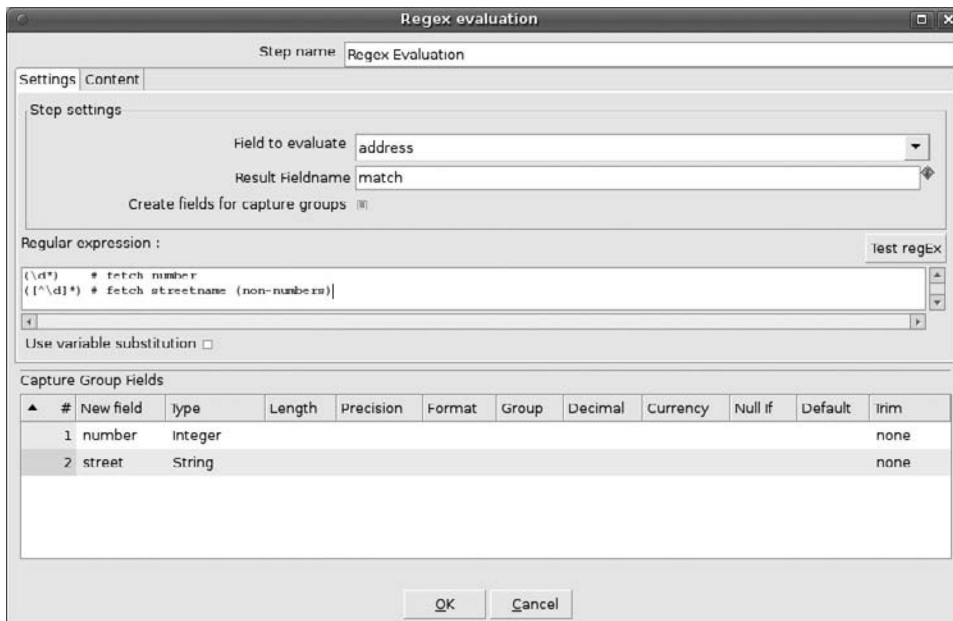


Figure 7-40: Regex Evaluation step

To enable the use of whitespace and comments, the corresponding option for this has to be checked on the Content tab, and to be able to create fields for capture groups the checkbox with that name has to be marked on the Settings tab. A *capture group* is a part of the regular expression between parentheses so in this case `(\d*)` captures the digits that make up the house number, and `([^\d]*)` captures all non-numeric characters from the string. Figure 7-41 shows the partial result of the step.

Rows of step: Regex Evaluation (603 rows)

#	address_id	address	match	number	street
1	1	47 MySakila Drive	Y	47	MySakila Drive
2	2	28 MySQL Boulevard	Y	28	MySQL Boulevard
3	3	23 Workhaven Lane	Y	23	Workhaven Lane
4	4	1411 Lillydale Drive	Y	1411	Lillydale Drive
5	5	1913 Hanoi Way	Y	1913	Hanoi Way
6	6	1121 Loja Avenue	Y	1121	Loja Avenue
7	7	692 Joliet Street	Y	692	Joliet Street
8	8	1566 Inegl Manor	Y	1566	Inegl Manor
9	9	53 Idfu Parkway	Y	53	Idfu Parkway
10	10	1795 Santiago de Compostela Way	Y	1795	Santiago de Compostela Way
11	11	900 Santiago de Compostela Parkway	Y	900	Santiago de Compostela Parkway

Figure 7-41: Regex Evaluation result

Summary

This chapter looked at the various tools and technologies available within Kettle to validate, cleanse, and conform your data. We discussed various types of rules and constraints to which data needs to comply. More to the point, in the chapter you learned:

- The various steps available for data cleansing and several use cases for them.
- How the rich collection of string matching algorithms Kettle contains works, and how they can be applied.
- The use of reference tables for cleansing and conforming data.
- How to apply the Data Validator step and what the options of validating data can be used.
- How to handle data, process and process errors using the error handling capabilities of Kettle.
- How to use the Fuzzy Match step to deduplicate data.
- The various scripting options available and the way they can be applied. Examples for the Formula, Java Script, Java Expressions and Regex Evaluation steps were included.

CHAPTER

8

Handling Dimension Tables

In this chapter, we take a closer look at how you can use Kettle to manage dimension tables. In particular, we'll look at those features in Kettle that are particularly useful to transform and/or generate data to fit the format of typical dimension tables, as well as the actual loading of the data into the tables. Before we discuss the details, let's consider which of the ETL subsystems discussed in Chapter 5 are involved in the management of dimension tables:

- **Change Data Capture (subsystem 2):** There are a few specific issues with regard to change data capture when loading typical denormalized star schema dimension tables, so this is briefly discussed in this chapter.
- **Extraction (subsystem 3), Data Cleaning and Quality Screen Handler System (subsystem 4), and Error Event Handler (subsystem 5):** These three are generic subsystems that apply to both dimension and fact tables. These subsystems were covered in Chapters 6 and 7, and won't be covered here in depth.
- **Audit Dimension Assembler (subsystem 6):** Functionally, the audit dimension is a special dimension that provides data about the ETL process itself, as opposed to information that has a business context. Loading the audit dimension was covered in Chapter 7.
- **Slowly Changing Dimension Processor (subsystem 9):** In many cases, data stored in dimension tables is not fixed and static: it typically changes over time, albeit typically at a much slower pace than any of the fact tables. This chapter describes in depth how to use Kettle to implement type 1, type 2, and type 3 slowly changing dimensions, and covers a few additional and complementary techniques as well.

- **Surrogate Key Generator (subsystem 10):** Loading dimension tables implies generating surrogate keys; this topic is discussed in detail in this chapter.
- **Hierarchy Dimension Builder (subsystem 11):** Although it is not so easy to pinpoint exactly which part of a transformation for loading a dimension constitutes the hierarchy manager, dealing with hierarchies certainly has a lot to do with loading dimension tables. Several aspects of dealing with hierarchies are discussed in detail in this chapter, such as top-down level-wise loading of snowflaked dimensions and implementing solutions to solve hierarchies of indeterminate depth (recursive hierarchies).
- **Special Dimension Builder (subsystem 12):** This is not a really clear-cut subsystem—rather it is a bucket of various types of dimensions. Some of these are discussed in this chapter, such as generated dimensions and junk dimensions.
- **Dimension Manager System (subsystem 17):** This is the actual subject of this chapter.

Managing Keys

A big part of loading dimension tables is about managing keys. There are two types of keys to take into account:

- **Business keys:** These keys originate from the source system, and are used to identify its business entities.
- **Dimension table surrogate keys:** These keys identify rows in the dimension tables of the data warehouse.

NOTE **Arguably, there is another type of key, namely the foreign keys to dimension tables. There may be some confusion about the term *foreign key*. In particular, the term is often confused with a *foreign key constraint*.**

In this chapter, the term *foreign key* is used to convey the idea that a table contains a column that has the express purpose of storing the key values of another table in order to relate the rows from the different tables. In this sense, foreign key just means the column stores key values, but from a key of another (foreign) table.

Foreign keys typically occur in the fact tables where they are used to relate the fact table rows to the context of the facts described in the dimension tables. These are discussed in Chapter 9, which is all about fact tables. But as you will see later in this chapter in our discussion of snowflaked dimensions, foreign keys to dimension tables also occur in dimension tables and bridge tables.

Here's a list of common tasks related to managing keys when handling data warehouse dimensions:

- Matching business keys from the source system to the dimension tables to determine whether to update or insert dimension rows

- Generating a surrogate key value for new dimension rows
- Looking up surrogate keys when loading snowflaked dimension tables (as well as fact tables)
- Deriving smart keys from attribute data for generated dimensions such as date and time dimensions

Kettle offers at least one and sometimes several transformation steps that can be used to perform each of the listed tasks regarding key management. In this section, we describe these key management tasks in general terms. Later in this chapter, we present concrete, detailed examples of these tasks.

Managing Business Keys

Data warehouses are loaded with data coming from one or multiple source systems. Understanding which pieces of data can be used to identify business entities is paramount, in both the source system and the data warehouse. Without a means of identification, you would be unable to keep records pertaining to different real-world objects separate from one another, rendering the entire collection of data structureless, and thereby useless.

Keys in the Source System

In the source systems, the pieces of information that identify the business entities are *keys*. For any given table, a key is each column (or group of columns) such that logically, there cannot be more than one row in that table having a particular value for that column (or a combination of values in case of a group of columns). Typically the source system enforces keys at the database level by either a primary key or unique constraint. This actively prevents any changes to the database that would result in a situation where the table would store multiple rows having the same value(s) for the column(s) of one particular key.

Keys in the Data Warehouse

In the data warehouse, business entities such as products, customers, and so on end up in the dimension tables. These each have their own key, separate from any key coming from the source system. Typically, keys of dimension tables are *surrogate keys*, consisting of a single integer column for which the values do not bear any relationship with the descriptive columns of the dimension table, and they are typically generated in the course of the ETL process.

Business Keys

In order to properly load the dimension tables, there has to be some way to keep track of how the rows in the dimension tables relate to the business entities coming from the source system(s). This is achieved by storing the surrogate key of the dimension table

together with at least one of the keys from the source system. Typically, the primary key from the source system is most convenient for this purpose.

In this context, the key coming from the source system is referred to as the *business key*. Business keys can originate either from natural keys or surrogate keys present in the source system, but in the data warehouse we simply refer to them as business keys.

NOTE A business key is typically not a key of the dimension table. One of the main purposes of the data warehouse is to maintain a history of the changes in the source systems, and the only way to truly do that is to allow for storing duplicates in the business keys. A typical example is discussed later in this chapter in our discussion of type 2 slowly changing dimensions. For reasons of performance, it is usually still a good idea to index the business keys.

Storing Business Keys

The business keys are often stored directly in the dimension table. This is a very straightforward approach that allows for relatively simple ETL procedures. For example, to check whether data changes coming from the source system should result in an update of a existing dimension row or in an insert of an additional dimension row, a simple and efficient check based on the business key is sufficient to see if a relevant dimension row already exists. A similar search based on the business key can be done to obtain the corresponding surrogate key of the dimension table.

Alternatively, the business-key/dimension-key mapping can be kept outside of the dimension table and maintained in the staging area instead. In this case, the design of the dimension tables may be cleaner, at the expense of more complicated ETL procedures.

Looking Up Keys with Kettle

Kettle offers a number of steps that can be used for looking up data. These steps can be used for looking up keys (typically, looking up a surrogate key based on the value of a business key), or for looking up attribute data (which can be used to denormalize data).

In addition to pure lookup steps, there are several steps that are capable of inserting or updating rows in a database, based on a matching key, as well as looking up data. You'll see plenty examples of looking up data by key in this and other chapters.

Generating Surrogate Keys

Data warehousing best practices dictate that dimension tables should, in principle, use an automatically generated meaningless integer type key: a surrogate key. There are cases where it makes sense to deviate from this rule, and some of these cases are discussed later on in this chapter, but in this section, we focus on the genuine surrogate keys.

Kettle offers features to generate surrogate key values directly from within the transformation, as well as functionality to work with key values that are generated at the database level. In this section, we'll take a look at the steps that offer these features.

The "Add sequence" Step

In Spoon, the "Add sequence" step resides beneath the Transform category. This step is designed especially to generate incrementing sequences of integer keys. You first encountered this step in the `load_dim_date` and `load_dim_time` transformations in Chapter 4 (shown in Figure 4-4 and Figure 4-5 respectively).

The sequence step works by passing through rows from the incoming stream to the outgoing stream, thereby adding an extra integer field containing generated integer values. You can configure the step to draw values from two alternative sources:

- A local counter maintained at runtime within the transformation.
- A database sequence. Databases like Oracle and PostgreSQL offer sequence schema objects, allowing tables to draw unique values from a centrally managed incrementing number generator.

In the remainder of this section, we describe in detail how to use the "Add sequence" step to generate surrogate keys for dimension tables.

Using Internal Counters for the "Add sequence" Step

Figure 8-1 shows the configuration dialog of the "Add sequence" step that is set up to generate values at runtime within the transformation from an internal counter variable.

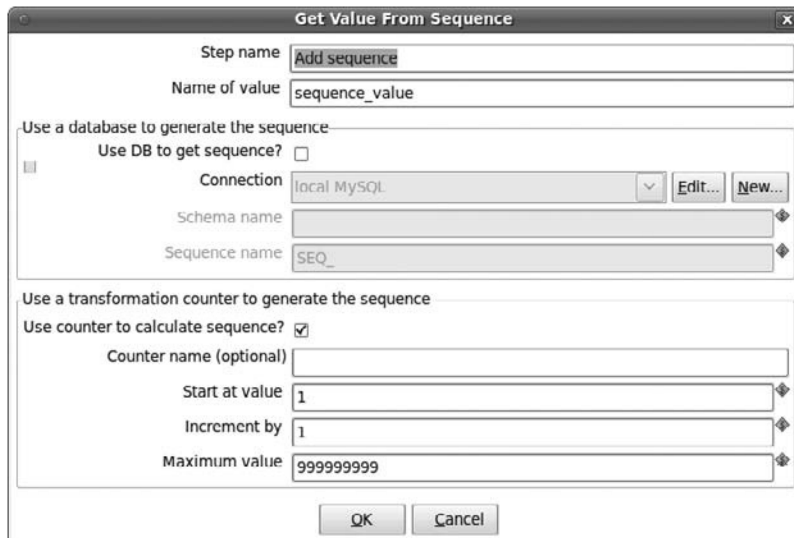


Figure 8-1: Configuring the "Add sequence" step to generate values within the transformation

In the header of the dialog, the "Step name" field is used to give the step a unique name within the transformation. The "Name of value" field is the name of the field that will be added to the outgoing stream to convey the sequence of integer values.

The bottom section of the dialog is labeled "Use a transformation counter to generate the sequence." Within that section, the checkbox "Use a counter to calculate sequence?" is selected, which disables the top section ("Use a database to generate the sequence") and specifies that the values are to be generated by a counter at runtime from within the transformation.

The "Counter name" field is left blank in Figure 8-1. When you have multiple "Add sequence" steps in one transformation, all such steps that specify the same name in this field will draw values from one and the same internal counter variable, ensuring the values drawn from these different steps are unique during one transformation run.

The "Start at value" field is used to specify the offset of the sequence of values. Here, you can either specify an integer literal or a variable reference that evaluates to an integer literal. In Figure 8-1, this is set to 1 but later in this section we discuss an example where we use a variable to initialize the offset of the sequence.

NOTE **The value for the "Start at value" is not remembered across different runs of the same transformation. The "Add sequence" step also does not provide a way to automatically load the offset value based on a database query. So, there is no built-in way to have the sequence pick up where it left off the last time the transformation was run. The sample transformations discussed in the remainder of this section illustrate how to cope with this issue.**

The "Increment by" field also takes either an integer literal or a variable that evaluates to an integer literal. This is used to specify the interval between the generated values. Normally you don't need to modify the default value of 1, but allowing this to be parameterized allows for some extra flexibility, which can be useful in some cases.

NOTE **As an example of using the "Increment by" field, consider a case where you need unique numbers to be drawn across multiple transformations, 1 through *N*. This can be achieved by giving each transformation its own "Add sequence" step, each having its own offset 1 through *N*, and giving them all an "Increment by" value of *N*.**

The "Maximum value" field is used to specify the maximum value of the sequence. If the internal counter exceeds this value, the sequence wraps around and starts again at the value configured for the "Start at value" field.

Generating Surrogate Keys Based on a Counter

The `test_sequence1.ktr` sample transformation uses the "Add sequence" step exactly as it is configured in Figure 8-1 to generate a surrogate key for a sample table called `test_sequence`. Each time the transformation is run, it generates another 100 rows, each getting its own sequential integer surrogate key. The transformation is shown in Figure 8-2.

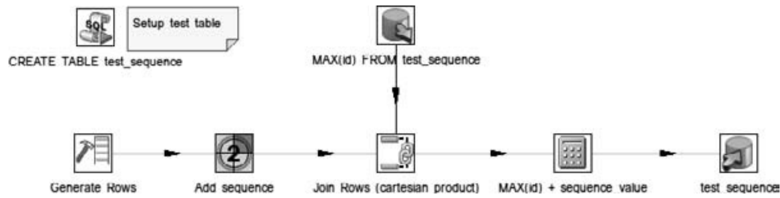


Figure 8-2: Generating surrogate key values using the "Add Sequence" step and internal counters

NOTE **You can download the** `test_sequence1.ktr` **transformation file from this book's website at** `www.wiley.com/go/kettlesolutions` **in the folder for Chapter 8. To run it, be sure to check the database connection properties to match your database. The transformation was built for a MySQL database having the schema called** `test`**. It connects to the database with username** `test`**, using the password** `test`**.**

The transformation shown in Figure 8-2 works by first running the step "CREATE TABLE test_sequence" in its initialization phase. This step is of the Execute SQL type, and sets up a sample table called `test_sequence`, which represents the dimension table, by running this SQL statement:

```
CREATE TABLE IF NOT EXISTS test_sequence (
    id    INTEGER NOT NULL PRIMARY KEY
);
```

After the initialization phase, the transformation generates 100 empty rows using the "Generate rows" step. These rows represent the data set that is to be loaded into the `test_sequence` table. Then the "Add sequence" step, having a configuration exactly as shown in Figure 8-1, adds a field called `sequence_value` to the stream, which contains an incrementing series of integers.

The `sequence_value` field is, of course, a crucial element in generating values for the surrogate key. Unfortunately, it cannot be used as-is because each time the transformation is run, it starts again at 1, and would thus generate duplicate keys immediately after the initial run.

To overcome this problem, we add the maximum key value present in the target table prior to running the transformation. To achieve this, we have a "Table output" step labeled "MAX(id) FROM test_sequence," which uses the following SQL statement to retrieve a single row with one column, holding the maximum key value found in the `test_sequence` table:

```
SELECT  MAX(id)
FROM    test_sequence
```

In order to add this value to the generated sequence values, the streams coming out of the "MAX(id) FROM test_sequence" and "Add sequence" step must first be joined.

This is achieved by the “Join Rows (cartesian product)” step. You first encountered this step when we discussed the `load_dim_time.ktr` transformation in Chapter 4 (shown in Figure 4-5). In the `test_sequence` transformation, it computes a Cartesian product just like in the `load_time_dimension.ktr` transformation, but because the “MAX(id) FROM test_sequence” step always returns exactly one row, the step does not lead to a duplication of records: it simply combines the sequence value along with the maximum key value in a single record.

The next step in the `test_sequence` is labeled “MAX(id) + sequence_value,” which is of the Calculator type. It is used to add the maximum value of the key coming from the “MAX(id) FROM test_sequence” step to the sequence value generated by the “Add sequence” step. This finally yields the value for the surrogate key column in the target table. The configuration of the Calculator step from the `test_sequence` transformation is shown in Figure 8-3.

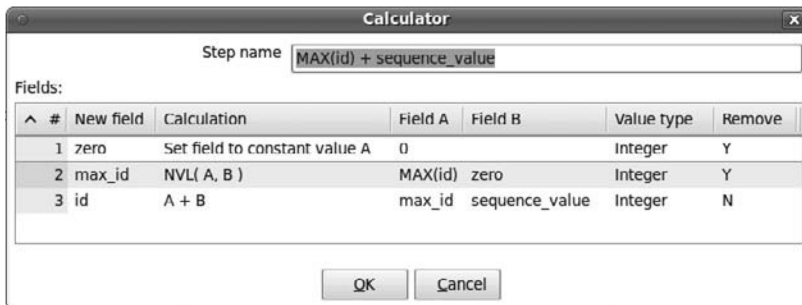


Figure 8-3: Using the Calculator step to actually compute the new key values

Figure 8-3 reveals another calculation to deal with the (rather exceptional) case in which the target table is empty. In this case, the step “MAX(id) FROM test_sequence” returns a `NULL` value, and the second calculation ensures that this value is replaced with the numerical 0 (zero) in order to continue with the addition to generate the actual surrogate key value.

The final step is a “Table output” step to load the `test_sequence` table, which represents the actual loading of the dimension table.

Dynamically Configuring the Sequence Offset

The `test_sequence` transformation shown in Figure 8-2enables you to work around the problem of generating only new unique key values by taking the maximum value of the existing keys prior to running the transformation and adding that maximum to the newly generated sequence values.

Although this method works nicely in a sample transformation, real-world transformations run the risk of becoming cluttered because they involve too many steps that do not directly contribute to the logic of the actual loading of the dimension table. Another potential disadvantage is the fact that the Join Rows and “MAX(id) + sequence_value” steps are executed for each row of the changed data set, slowing down the transformation as a whole.

There is an alternative to ensure the "Add sequence" step will generate only new unique key values. Instead of calculating the key value again and again for each row, it makes more sense to generate only the right key values in the first place. The only thing that is required to achieve this is the correct initialization of the "Start at" field of the configuration of the "Add sequence" step.

As it turns out, it is perfectly possible to dynamically configure the "Start at" property of the "Add sequence" step. The process is not entirely straightforward, and it does require some effort, but it has the advantage of resulting in a cleaner main transformation, which is typically just a little bit faster, too. Figure 8-4 shows the job called `test_sequence2.kjb`.

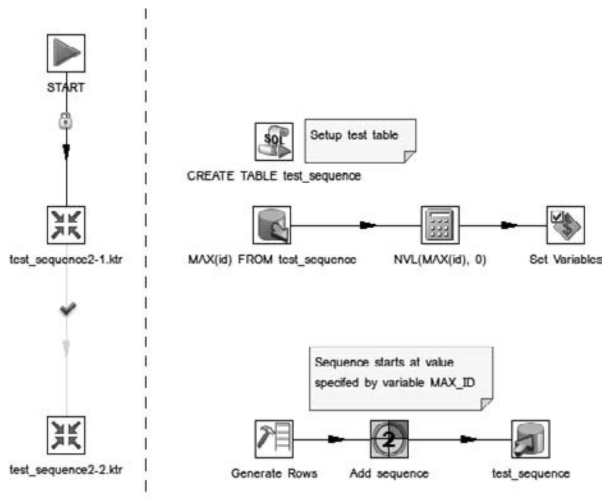


Figure 8-4: Dynamically configuring the offset of the "Add sequence" step

The job shown in Figure 8-4 contains two transformation job entries, `test_sequence2-1.ktr` and `test_sequence2-2.ktr`. These transformations are also shown in Figure 8-4. The first transformation, `test_sequence2-1.ktr`, does the job of setting up the `test_sequence` table and querying for the maximum value of the key. It also contains a Calculator step to supply a default value instead of the maximum key value in case the table is still empty, as discussed in the preceding section.

Transformation `test_sequence2-1.ktr` contains one step that you haven't encountered yet: the Set Variables step. Steps of this type accept one row from the incoming stream and expose the values of specified fields as environment variables, which are accessible outside the transformation that sets the variables. The scope of variable accessibility can be configured per variable. The configuration for the Set Variables step used in transformation `test_sequence2-1.ktr` is shown in Figure 8-5.

In Figure 8-5, you can see that the `id` field of the incoming stream is exposed as a variable called `ID`. The variable has root job scope. Variables with root job scope are accessible inside the root job itself (which is the outermost job executed by Kettle) along with all jobs and transformations that are directly or indirectly called from this job.

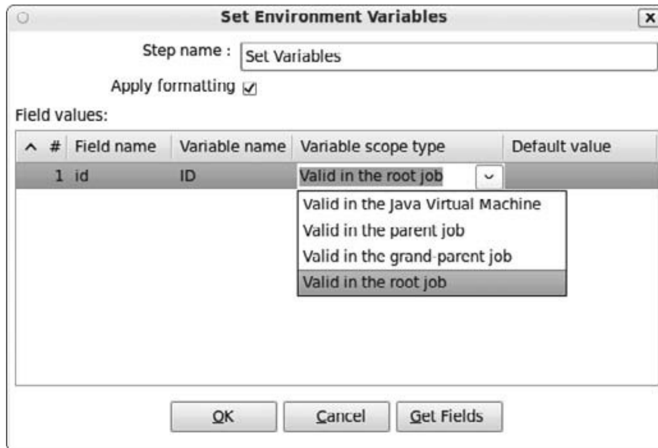


Figure 8-5: The configuration of the Set Variables step

The second transformation in the `test_sequence2` job, `test_sequence2-2.ktr`, does the actual loading of the table. Again you see the Generate Rows step for creating dummy rows, the "Add sequence" step to generate the surrogate key values, and the "Table output" step to actually load the table. In this case, the configuration of the "Add sequence" step is almost identical to the one shown in Figure 8-2, but in this case, the "Start at value" field is set to `${ID}`, which denotes a reference to the `ID` variable that was set in the `test_sequence2-1.ktr` transformation by the "Set Variables" step shown in Figure 8-4. The configuration of this "Add sequence" step is shown in Figure 8-6.

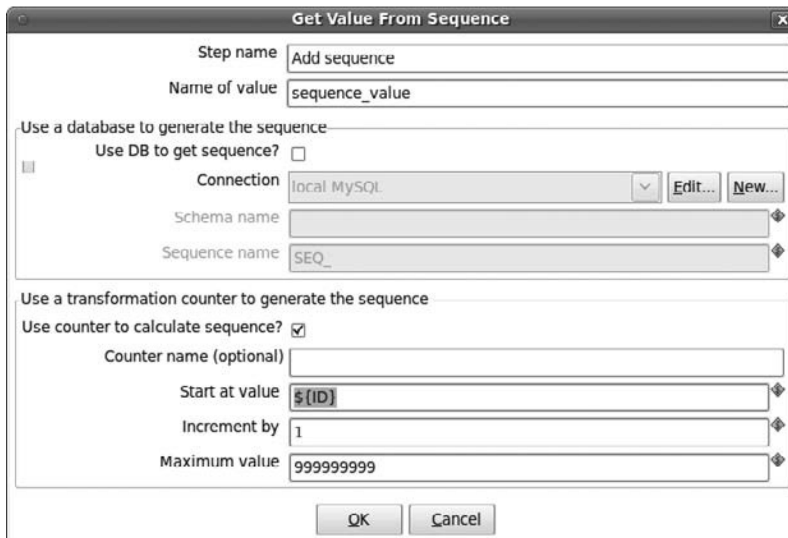


Figure 8-6: Using a variable reference to parameterize the sequence of the "Add sequence" step

Surrogate Keys Based on a Database Sequence

The "Add sequence" step can also be used to draw values from a database sequence. To configure this, select the "Use DB to get sequence?" checkbox in the top section of the "Add sequence" dialog.

To specify the database sequence itself, you need to select a valid connection, which must connect to an RDBMS that supports database sequences. If the database sequence does not reside in the default schema of the connection, you can specify the name of the schema in the "Schema name" field, where you should also specify the name of the database sequence.

That's all there is to it. In contrast, with a sequence based on an internal counter, no special measures have to be taken to initialize the offset of the sequence. The database can be relied upon to keep track of the current value of the sequence.

Working with auto_increment or IDENTITY Columns

Some databases offer sequence support at the column level. Using DDL, one (or sometimes more) table columns are created using a special column attribute or data type that causes the database to automatically supply a unique incrementing integer value for the column whenever a new row is added to the table. The purpose of such a feature is to simplify implementing surrogate keys.

For example, MySQL supports the `auto_increment` column attribute, which may be applied to one column of an integer type, provided that column is the primary key (or under certain circumstances, part of a composite primary key). Microsoft SQL Server has a similar feature but implements this using a special `IDENTITY` data type.

On the one hand, these database features alleviate some of the burden of generating surrogate keys in the transformation. On the other hand, they can also be a complicating factor in case the remainder of the transformation needs access to the database-generated value. This problem does not occur when using database sequences via the "Add sequence" step. In that case, the act of drawing a value always has to occur before adding the row to the table, so by definition the value is always available to the transformation.

The Kettle steps that are designed for adding rows to dimension tables come with configuration options to add the automatically generated column value to the outgoing stream. This is discussed in greater detail later in this chapter.

Keys for Slowly Changing Dimensions

The "Slowly Changing Dimensions" section in this chapter contains a detailed description of the "Dimension lookup / update" step. This step offers built-in functionality to automatically generate surrogate key values for various types of slowly changing dimensions. Although this is built-in functionality, it is still flexible enough to work with database-generated surrogate keys drawn from either a database sequence or an `auto_increment` or `IDENTITY` column.

If you're designing a transformation to load type 1 or type 2 slowly changing dimensions, we recommend using the "Dimension lookup / update" step. When using this step, you do not need to add a separate "Add sequence" step to generate the values.

Loading Dimension Tables

In this section, we look at how you can use Kettle to load dimension tables based on data stored in one or more operational source systems or a staging area. We look at two different but typical scenarios:

- Loading snowflaked dimension tables
- Loading denormalized star schema dimension tables

Snowflaked Dimension Tables

In snowflake schemas, a single dimension is implemented as a series of related dimension tables. The dimension tables are typically in the third normal form (3NF) or Boyce-Codd normal form (BCNF) and are thus said to be *normalized*. Figure 8-7 shows an example dimensional model having one fact table and snowflaked date and product dimensions.

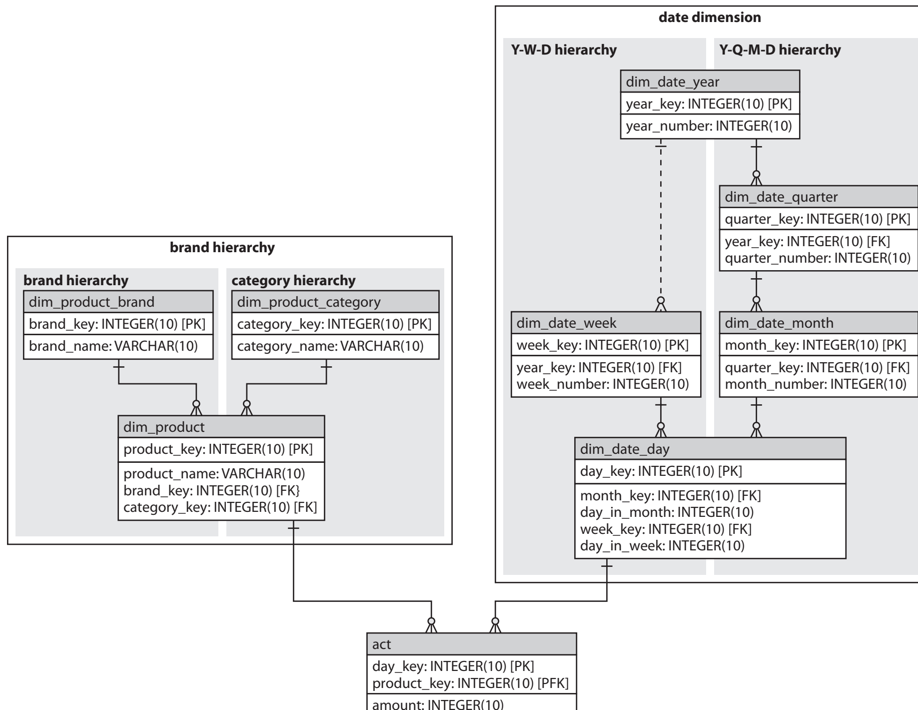


Figure 8-7: Snowflaked dimensions

Each dimension table in a snowflaked dimension represents a level in one of the dimension's hierarchies. The dimension tables are inter-related via one-to-many relationships, where the table that implements the higher aggregation level is at the "one" side, and the table that implements the nearest lower aggregation level is at the "many" side. For example, in Figure 8-7, the date dimension has two hierarchies: Y-W-D (Year, Week, Day) and Y-Q-M-D (Year, Quarter, Month, Day). The `dim_date_year` table implements the Year level, which is the highest aggregation level in both hierarchies. This table has a one-to-many relationship with both the `dim_date_week` and `dim_date_quarter` tables, each of which implements the next lower aggregation level in the Y-W-D and Y-Q-M-D hierarchies.

There's always one dimension table within the dimension that represents the lowest aggregation level: the primary dimension table. The primary dimension table is usually related to the fact table, again via a one-to-many relationship. In Figure 8-7, the tables `dim_product` and `dim_date_day` are the primary dimension tables for their respective product and date dimensions.

Top-Down Level-Wise Loading

In a snowflaked dimension, loading dimension tables at any but the highest level implies doing a lookup in the dimension table at the next higher level. The lookup is required to obtain the key value, which needs to be stored in the dimension table at the lower level to maintain the hierarchical relationship between the two consecutive levels.

Because of the dependency of lower-level dimension tables upon those at the higher level, it makes the most sense to load hierarchies in a top-down fashion, first loading all dimension tables at the highest level, then loading the dimension tables at the level immediately below the highest level, then at the next lower level, and so on, finally loading the primary dimension tables. We refer to this as *top-down level-wise loading.*

NOTE We already encountered another example where the table loading order was controlled by lookup dependencies: In Chapter 4, we discussed the `load_rentals` job, which used a series of transformation job entries to sequentially load the dimension tables before loading the fact table.

A good way to manage top-down level-wise loading of snowflaked dimensions is to build a single transformation to load each individual dimension table. At a minimum, such a transformation should contain the logic to extract the changed data at the level appropriate for the dimension table. In case the dimension table is not at the top level of its hierarchy, one or more lookups have to be performed to fetch the keys from the dimension table(s) at the nearest higher level.

Sakila Snowflake Example

To illustrate the process, we created the `sakila_snowflake` schema. This is another dimensional model based on the sakila sample database schema introduced in Chapter 4. The `sakila_snowflake` schema is identical to the sakila rental star schema, except for

the customer and store dimensions. The respective dimension tables `dim_customer` and `dim_store` have been modified by replacing all columns related to address data with a foreign key to the primary dimension table of a snowflaked location dimension. A diagram of the (partial) `sakila_snowflake` schema is shown in Figure 8-8.

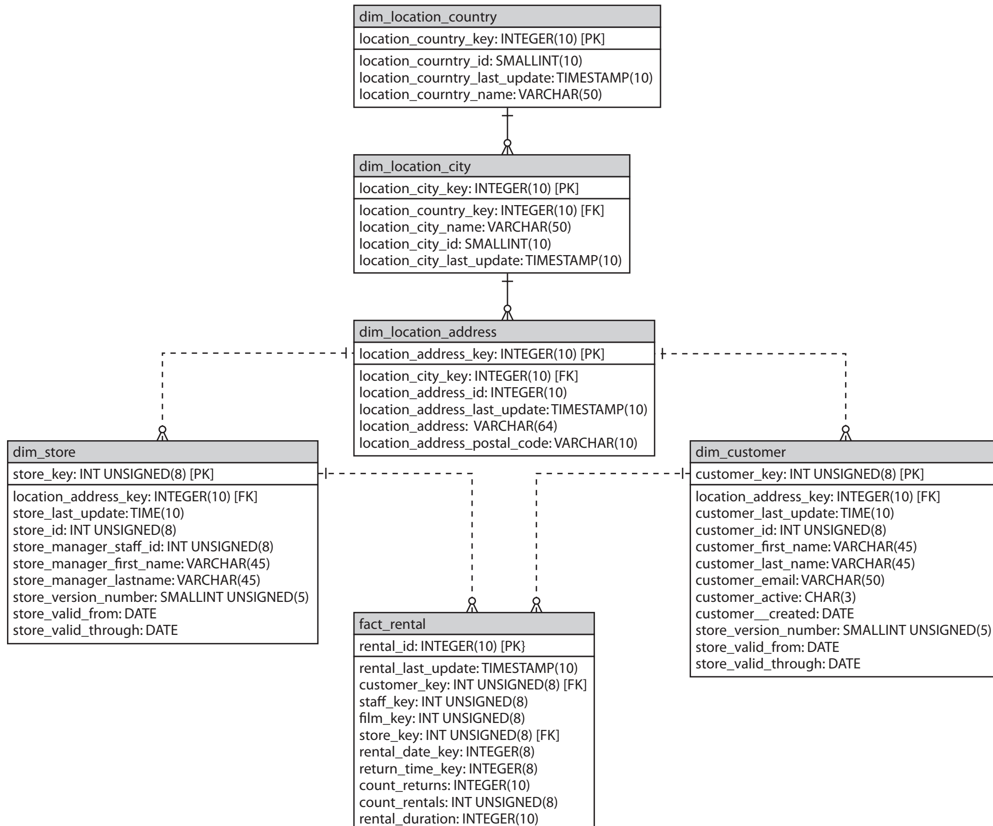


Figure 8-8: The (partial) sakila_snowflake schema

The snowflaked location dimension consists of three tables: `dim_location_address`, `dim_location_city`, and `dim_location_country`. The `dim_location_address` table is the primary dimension table, which represents the lowest level of the location dimension.

NOTE **The setup process for the** `sakila_snowflake` **example is similar to that for setting up the rental star schema, described in Chapter 4. Download the** `sakila_snowflake_schema.sql` **and** `create_sakila_snowflake_accounts.sql` **files from this book's website and run them.**

To examine the load process, download the `load_dim_location_%` **job and transformation files.**

In the `sakila_snowflake` example, the primary dimension table of the location dimension is not related to the fact table, but to other dimension tables. Because the location data has been pulled out of the existing store and customer dimension tables, these become themselves snowflaked. In the *Data Warehouse Toolkit, Second Edition* by Ralph Kimball and Margy Ross (Wiley, 2002), even Kimball deems this a valid case of dimension normalization for a location dimension, and he refers to this construct as a *location outrigger* (although he does not recommend further snowflaking of the location dimension itself). Because the schema itself is still largely a star schema, this can also be called a *starflake*.

Sample Transformation

By way of example, the `load_dim_location_address.ktr` transformation is shown in Figure 8-9.

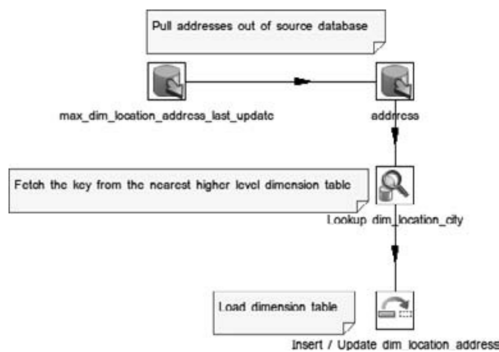


Figure 8-9: Loading an individual dimension table of a snowflaked dimension

In this particular case, the transformation is designed to load the `dim_location_address` table of the location dimension shown in Figure 8-8. The first two table input steps of the transformation are responsible for capturing changed data in a similar way as for the sample transformations described in Chapter 4: You first query the maximum value of the `last_update` column from the dimension table and then use that value to retrieve all rows from the source system that changed since that date.

After the changed data capture, the rows are then fed into the "Lookup dim_location_city" step to find the corresponding key of the `dim_location_city` table. We discuss the configuration of this step in detail in the next subsection.

Finally, the address data plus the foreign key to the `dim_location_city` table are loaded into the `dim_location_address` table using an Insert / Update step. We discuss this step later in this chapter in greater detail in the section "Type 1 Slowly Changing Dimensions."

Database Lookup Configuration

In the `load_dim_location_address.ktr` transformation shown in Figure 8-9, a step of the "Database lookup" type is used to find the value of the surrogate key used in the data warehouse (`location_city_key`) that corresponds to the business key (`city_id`) coming in from the source system.

As you will see later in this chapter, the "Database lookup" step can also be used to retrieve attribute values from the source system, which is useful in preparing the denormalized data required to load star schema dimension tables. In Chapter 9, you also learn how this type of step is useful in obtaining the keys of the dimension tables when loading fact tables.

The configuration of the "Lookup dim_location_city" step from the `load_dim_location_address` transformation in Figure 8-9 is shown in Figure 8-10.

Figure 8-10: The configuration of the "Database Lookup" step

The elements of this dialog are discussed in detail in the remainder of this section.

Connection, Lookup Schema, and Lookup Table

The configuration shown in Figure 8-10 is pretty straightforward: Because you're looking up the surrogate key of the `dim_location_city` table in the `sakila_snowflake` schema, the "Lookup table" and "Connection" fields are filled in correspondingly.

For the Connection field, you can pick an existing connection from the drop-down list. You can also use the Edit button to review or modify the definition of the connection.

In this case, the "Lookup schema" field is left blank because the sakila_snowflake schema is already the default schema of the connection, but you could use this field to specify another schema if required. You can either type the name of the schema, or use the Browse … button to pop up a dialog from which you can choose the appropriate schema.

Unsurprisingly, the "Lookup table" field must be used to specify which table should be used to perform the lookup. As with the "Table schema" field, you can either type the name of the table, or use the Browse … button to open a dialog from which you can pick the table.

Lookup Key

At the bottom of the dialog shown in Figure 8-10 are two grids. The upper grid is labeled "The key(s) to lookup the value(s)" and is used to specify how the fields from the incoming stream should be matched against columns of the lookup table. To that end, each row in the grid defines a comparison operation between a field value (sometimes two field values) from the incoming record stream against a column value of the lookup table.

The first column in the grid is "Table field." Here, you can specify the names of the columns from the lookup table. You can either type the column name or use the drop-down control to pick a column from the list. Typically, the set of columns specified for "Table field" make up a key of the lookup table, so that each row from the incoming stream matches at most one row. For example, the location_city_id column specified in the grid shown in Figure 8-10 constitutes the business key of the dim_location_city_table.

The Field1 column in the grid is used to specify which fields from the incoming stream should be used to look for a record in the lookup table. The Comparator column specifies a comparison operator that determines exactly how the database column and the stream field are to be matched. In the vast majority of cases, the database columns and stream values are tested for equality using the = as comparator.

Sometimes, it's useful to look up a record based on a value range. In this case, you can use the BETWEEN comparator. When using this comparator, a stream field must be filled in for both the Field1 as well as the Field2 columns in the grid to specify the lower and higher bounds of the value range respectively.

To quickly fill the grid, you can press the Get Fields button at the bottom of the dialog. This populates the grid with an equality comparison for each field of the incoming stream. Both the "Table field" and the Field1 columns will be filled with field names coming from the stream. This means that when the column names are not identical to the field names, you will need to manually adjust the "Table field" values.

Lookup Value

You can use the grid labeled "Values to return from the lookup table" to specify which columns of the lookup table are to be retrieved. For each of these columns, a field is added to the outgoing stream.

Use the Field column to specify the columns from the lookup table. You can either type the column names or select them from the drop-down list, just as you can in the grid that specifies the lookup key. The "New name" column in the grid can be used

to control the names of the fields that are added to the outgoing stream. If you don't specify a field name here, then the column names will be used. The Default column can be used to specify a custom literal value that is to be returned if no matching row is found in the lookup table. When specifying a Default, you should use the Type column to specify the data type of the value.

To quickly populate the grid, use the Get Lookup Fields button. This will add a row to the grid for each column in the lookup table. By deleting the entries you don't need, or alternatively, making a selection and pressing Ctrl+K (for keep) to retain only those entries that you do need, you can save yourself a lot of time configuring "Database lookup" steps.

Cache Configuration

The "Database lookup" step can be configured to cache any lookup results. When caching is enabled, Kettle will store records retrieved from the database in an in-memory cache, and then try to use the cache to serve subsequent lookup requests. This can result in an enormous increase in speed if the same lookup has to be performed repeatedly, which is typically the case. If there is little chance the same key has to be looked up repeatedly, enabling the cache will result in some slowdown incurred by the extra work required to search the cache before searching the lookup table.

In Figure 8-10, the "Enable cache?" checkbox is checked to enable caching. The "Cache size in rows" field may be used to specify the size of the cache as a number of rows. If the cache contains the maximum specified number of entries, any subsequent lookup request that is not in the cache will cause one of the older cached entries to be removed. As the label indicates, you can specify that all lookups are to be cached by setting the property to zero.

In Figure 8-10, the "Cache size in rows" field is disabled, indicating that the setting is not used. This is because the "Load all data from table" checkbox is checked, which forces upfront caching of all of the rows in the `dim_location_city` table. When "Load all data from table" is not checked, caching occurs in an "as-you-go" fashion: an attempt is made to retrieve a row from the cache, and if it's not in the cache, an attempt is made to retrieve it from the table. When that attempt is successful, the row is stored in the cache, speeding up subsequent lookup requests by the same key.

When "Load all data from table" is checked, the entire lookup table is scanned only once in the initialization phase of a transformation run, storing all rows in the cache. During the run itself, all lookup requests are done directly against the cache, and no attempt is ever made to do any lookups against the database table. In this case, no time is ever lost to maintain the cache and issue any further database queries, so typically this option offers the best performance. The only reason not to use this option is when the lookup table is very large and it would cost too much in terms of memory to store the lookup table in memory.

Lookup Failure

When using the "Database lookup" step, there is typically an assumption that the lookup will always succeed—that is, that the lookup key is found in the table, and that it matches exactly one row. This may not always be the case, however: The lookup table

may not contain any rows that match the lookup key, or there may be multiple rows that match the lookup key.

Normally, if a row is not found in the lookup table, then all lookup fields will have get the value of the Default column defined in the "Values to return from the lookup table" grid, and NULL in case no default value is configured. In some cases, this may be exactly what you want. In other cases, you can actively block those rows from flowing through the step by checking the "Do not pass the row if the lookup fails" checkbox.

NOTE **In most cases,** NULL **values due to a failed lookup will cause a problem sooner or later, typically resulting in failure of the entire transformation. For example, the** load_dim_location_address **transformation shown in Figure 8-9 would fail because the row would be rejected by the** dim_location_address **table as the** location_city_key **column is declared to be** NOT NULL**.**

Failure of the transformation may actually be a good thing: as we explained before, our strategy of level-wise top-down loading should have ensured that the dim_location_city **table was already completely loaded before attempting to load the** dim_location_address **table so a failure of the lookup probably indicates some logic flaw in either the design of the application, or some data integrity issue in either the source system or the target dimension tables.**

Without any explicit configuration of the "Database lookup" step, Kettle will simply pick the first of the available rows if the lookup key matches multiple rows. You can influence which one of the rows is picked by explicitly specifying the order in which the rows are returned. You can do this by filling in the "Order by" field. The value for this field should constitute a valid SQL ORDER BY clause (but without the actual keywords ORDER BY) for whatever database the connection definition points to.

In many cases, the fact that a key lookup returns multiple records indicates a logical error. In this case, you should select the "Fail on multiple results" checkbox. This will cause the step, and thereby the entire transformation, to fail as soon as a key matches multiple rows.

Sample Job

After building the transformations, a job can be used to organize the transformations sequentially, ensuring the top-down loading order of dimension tables from higher to lower levels. An example is shown in Figure 8-11.

Figure 8-11: The load_dim_location.kjb job ensures top-down level-wise loading of the snowflaked location dimension tables

The job shown in Figure 8-11 is quite straightforward. It simply ensures that the transformations that load the location dimension tables are run sequentially by order of the levels of the dimension.

Star Schema Dimension Tables

In star schemas, each dimension is typically implemented as a single dimension table. (For an example of a star schema, see the rental star schema shown in Figure 4-2.) The hierarchies in which the dimension is organized take the form of a collection of columns, where each column in such a collection represents a distinct level of the hierarchy.

Denormalization

Typically, star schema dimension tables are quite wide, having many columns that are typically, as Kimball puts it, "highly correlated." This is just another way of saying that the values in these columns are functionally dependent on one another. Because the columns are dependent on one another (and thus, not only dependent on the key of the dimension table), these dimension tables are not in the third or even second normal form (3NF and 2NF, respectively). Star schema dimension tables may even have multi-valued columns, containing a list of values representing a multitude of entities in the source system, thus violating even the first normal form (1NF). Regardless of the exact classification, star schema tables are typically characterized as being "not normalized."

In many cases, the data for the dimension tables originates from OLTP source systems of which the underlying database is typically normalized up to the third or Boyce-Codd normal form (BCNF). So, in order to load the star schema dimension tables, a transformation for loading a dimension table has to extract data from multiple related source tables, combine the records, thereby denormalizing the data, and then deliver the data to the target dimension table.

Denormalizing to 1NF with the "Database lookup" Step

In the discussion on loading snowflaked dimensions earlier in this chapter, we described in detail how the "Database lookup" step can be used to look up the surrogate key of a dimension table, based on the business key of a table in the source system. But the "Database lookup" step can be used equally well to denormalize data by combining related rows from the source system, thus preparing the data for loading star schema dimension tables. The "Database lookup" step simply looks up field values based on a key, and does not care about whether you're doing the lookup to fetch a single surrogate key value or an entire record.

In Chapter 4, you witnessed several instances of how the "Database lookup" step type was used in this way to load star schema tables. It was first discussed in the example of the `fetch_address` subtransformation (see Figure 4-14), which featured a lookup-cascade to retrieve address data from the `address`, `city`, and `country` tables in the sakila database, providing a denormalized set of columns with address data for the `dim_customer` and `dim_store` dimension tables in the rental star schema.

We won't repeat the detailed discussion of the "Database lookup" step here, but for completeness and comparison, it is a good idea to take some time to examine how the "Database lookup" steps are used in various transformations discussed in Chapter 4 and compare them to the steps in the snowflaked example earlier in this chapter.

Change Data Capture

In the rental star schema discussed in Chapter 4 and shown in Figure 4-2, each dimension table is based directly on a table in the sakila sample database. For example, the `dim_store` dimension table is based on the `store` table, the `dim_customer` dimension table is based on the `customer` table, and so on. To load the dimension tables, change data was captured for these tables in the sakila schema and then denormalized using the "Database lookup" step until it could finally be loaded into the dimension table.

If you consider the way data changes are captured in the transformations described in Chapter 4 and compare that with the process for the level-wise loading of the snowflaked location dimension, an important difference emerges. The transformations in Chapter 4 only detect changes for those tables that fill the lowest level of each dimension, and look up related rows from there to fetch the data that makes up the higher levels. But what if any changes occur at the higher level? Those changes will not be picked up, resulting in an inconsistency between the source and target system. There are a number of solutions to ensure the changes are captured at all levels of the dimension.

If the dimension is not too large (they often aren't), it may be acceptable not to bother with any sophisticated method to capture the changes. Simply scanning all rows in the table from the source system that corresponds to the lowest level of the dimension and then doing a lookup cascade to obtain the denormalized resultset will automatically pick up any changes in the tables that correspond with the higher levels of the dimension.

If it is a requirement to load only the changes from the source system, loading star schema dimensions can become tricky. If the requirement is in place to reduce the load on the source system, there is a workaround that still allows for a fairly simple loading process of the dimension table—by utilizing a staging area. In this scenario, you would extract only the changes from the source system, as per requirement. You can then store the changes in a normalized staging area that is part of the ETL system. The staging area still allows you to do the brute force approach because the staging area is not part of the source system, and can be burdened as much as necessary to load the dimension tables.

NOTE **It is still possible to load only the changes while avoiding a staging area. However, the transformation to load the dimension tables will rapidly increase in complexity. Loading a denormalized dimension with these stringent requirements concerning change data capture is an advanced topic, and is beyond the scope of this chapter. However, we should point out that Kettle does in fact offer all the building blocks to do it, should it be necessary.**

For example, loading a denormalized location dimension containing data at the address, city, and country level presumes capturing changes at all those levels, too, just as we discussed when loading the snowflaked location dimension. But for a denormalized dimension, you now have to work your way down from the higher levels to see which rows at the lower levels these changes correspond to. Once you have that set, you can use the previously described lookup cascade to prepare the denormalized result set.

For example, the processes of "looking down" from the captured changes at the higher levels to see which rows are affected at the lower levels can be implemented using the "Database join" step, which you first encountered in the `load_dim_film` transformation in Chapter 4 (see Figure 4-15 and the sections on creating the flags for film categories and filling the `dim_film_actor_bridge` table). This may prove to be quite expensive, however, because it will result in many separate queries on the lower level as the rows at the higher level flow through the step. It may be more efficient to spend some time writing a more advanced SQL `JOIN` query in the "Table input" step while capturing the data.

You already encountered yet another approach to fight the loading complexity: By using a normalized or snowflaked dimension, the problem is virtually absent. Another approach to the modeling problem is discussed in Chapter 19, which explores the Data Vault architecture.

Slowly Changing Dimensions

In this section, we examine in detail how to implement various types of slowly changing dimensions (SCDs) with Kettle. For each type of slowly changing dimension, we briefly describe its characteristics and then examine which Kettle steps you could use to load it.

NOTE **This book does not offer a shortcut to data warehousing theory. If you want to know the purpose of slowly changing dimensions, or if you're not aware of the typical techniques that are used to implement them on the database level, you should read up on the subject in a specialized data warehousing book such as *The Data Warehouse Toolkit, Second Edition*, by Ralph Kimball and Margy Ross. Chapter 7 of *Pentaho Solutions* by Roland Bouman and Jos van Dongen (Wiley 2009) also contains a basic explanation of the features and purpose of different types of slowly changing dimensions.**

Types of Slowly Changing Dimensions

Following Kimball, we distinguish three main types of slowly changing dimensions: type 1, type 2, and type 3:

- **Type 1:** Updates in the source system result in corresponding updates in the target dimension.
- **Type 2:** Updates in the source system result in inserts in the target dimension by maintaining multiple timestamped versions of dimension rows. This allows you to find whichever version of the dimension row was applicable at any given point in time.
- **Type 3:** Updates in the source system are stored in different columns in the same row.

We presume the reader is already aware of these distinctions, their purpose, and their applicability.

Type 1 Slowly Changing Dimensions

In a type 1 slowly changing dimension, the dimension data is loaded to always reflect the current situation, overwriting the current record with the changed one. There are a couple of Kettle steps that can be used for type 1 dimensions.

In the section "Type 2 Slowly Changing Dimensions," which follows, we discuss the "Dimension lookup / update" step, which can be used to load a variety of slowly changing dimension types, including type 1. Later in this chapter, we discuss the "Combination lookup / update" step in the "Junk Dimensions" section. The remainder of this section focuses on the Insert / Update step as an example of loading a type 1 slowly changing dimension.

The Insert / Update Step

The Insert / Update step resides beneath the Output category in the left pane tree view in Spoon. As its name implies, this step either inserts or updates a row. Sometimes, this is referred to as an *upsert*. You first encountered the Insert / Update step in the `load_dim_actor` transformation in Chapter 4. Let's take a closer look at the configuration of the `load_dim_actor` transformation (see Figure 8-12).

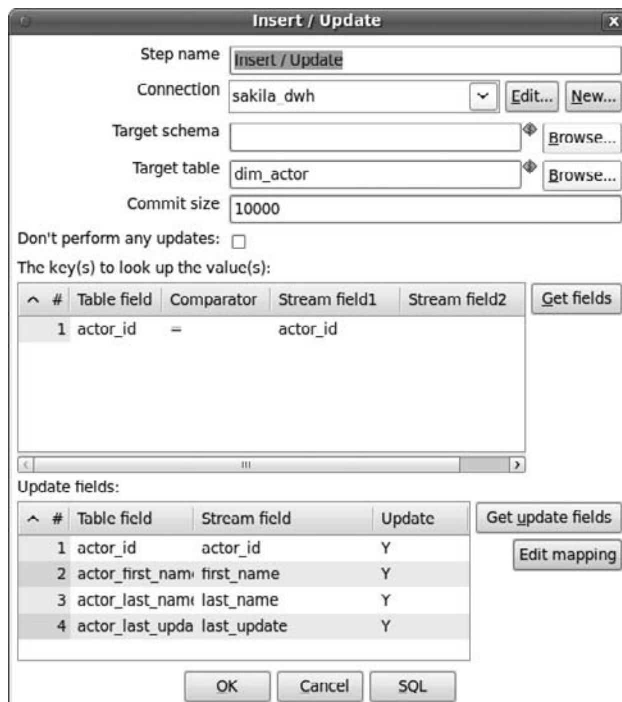


Figure 8-12: The configuration of the Insert / Update step

If you compare Figure 8-12 with the configuration of the "Database lookup" step, shown in Figure 8-10, you will see a number of similarities. Just like the "Database lookup" step, Insert / Update matches fields from the incoming stream against a column or collection of columns of a database table. These columns act as the lookup key. If a row is found, another set of table columns is updated with the values of the fields from the incoming stream. If the row is not found, it is added to the table.

NOTE **Don't use the Insert / Update step if all you really want to do is insert new rows without being bothered with errors resulting from unique or primary key constraint violations. If that's what you want to do, it is more explicit and often faster to simply use a "Table output" step. You can define error handling for the "Table output" step to catch any rows that are rejected by the table. This approach is faster than using the Insert / Update step because it doesn't waste time attempting to look up the row first.**

We will discuss the elements shown in the dialog in Figure 8-12 in the remainder of this section.

Connection, Target Schema and Table, and Commit Size

In the top section of the dialog shown in Figure 8-12, you see a number of fields to configure the database connection, table schema, and table name, as with the "Database lookup" step. The only difference is that this step refers to "Target schema" and "Target table," whereas the "Database lookup" step refers to "Lookup schema" and "Lookup table."

The "Commit size" field is used to specify the size of the commit batch. The size is specified as the number of row operations (in this case, the number of `UPDATE` and `INSERT` statements) that are to be performed before sending a `COMMIT` command to the database. A higher value will result in fewer `COMMIT` commands (and larger batches of changes), which is usually good for performance. However, the value cannot be too large as a large pending uncommitted transaction may consume a lot of resources from the database, which may negatively impact database performance.

It is impossible to provide generic advice on the ideal value for the commit size because it is dependent on many factors, such as the type of the database, the amount of available memory and, of course, the overall activity of the database at dimension loading time. However, the default commit size of 100 is quite conservative, and will usually lead to unsatisfactory performance. In many cases, it shouldn't be a problem to use a commit size of 1,000 or even 10,000. More info for tuning configuration options such as the "Commit size" can be found in Chapter 15.

An additional checkbox labeled "Don't perform any updates" is available in the top section of the dialog. By default, it is not checked, allowing the step to do both updates as well as inserts. If it is checked, no updates will be performed, only inserts.

Target Key

Just like the "Database lookup" step, the Insert / Update step needs a key—that is, a way to match the field values from the incoming stream to the columns of the target table. In this case, if the match is made, the key is used again in a `WHERE` clause to perform an `UPDATE`.

In both the "Database lookup" step and the Insert / Update step, the key is defined in a grid labeled "The key(s) to look up the value(s)." For the Insert / Update step, the grid to specify the key is almost identical to the one that appeared in the "Database lookup" step. The only difference is in the columns used to specify the fields from the stream: instead of Field1 and Field2, the analogous columns are called Stream Field and Stream Field2 in the Insert / Update step. You can use the Get Fields button at the right of the grid to quickly populate it.

As you saw in the "Database lookup" example shown in Figure 8-10, this step uses the business key to match the rows from the source system against the rows already present in the dimension table. So in this case, the `actor_id` column of the `dim_actor` dimension table is matched against the `actor_id` field that originates from the source system.

Update Fields

The lower section of the configuration dialog shown in Figure 8-12 has a grid labeled "Update fields." This is where you specify which fields from the stream should be used to perform the insert or update.

At the right side of the grid, there are two buttons. The "Get update fields" button can be used to quickly populate the grid with fields from the incoming stream. The "Edit mapping" button opens a simple wizard that can simplify the task of assigning the stream fields to the table columns. The mapping dialog is shown in Figure 8-13.

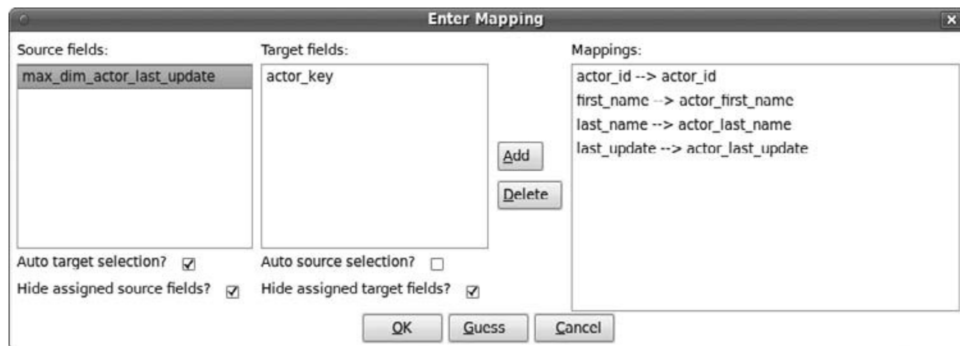


Figure 8-13: The mapping dialog offers a simple wizard to specify from which fields values should be assigned to the database columns.

The dialog has three lists: on the left, the "Source fields" list of field names from the incoming stream; in the middle, the "Target fields" list of column names from the target table; on the right, the Mappings list, which contains the field/column mapping, indicating from which field the value will be assigned to the table column. Selecting the "Auto target selection" checkbox causes automatic selection of a matching column when selecting a field. Similarly, selecting a column name will automatically also select a matching field if the "Auto source selection" checkbox is selected.

Once you select a field name and a column name, you can move that combination to the Mappings grid by pressing the Add button. You can quickly populate the mapping grid

by pressing the Guess button. If the "Hide assigned source fields" and "Hide assigned target fields" checkboxes are checked, then the field or column will be removed from its respective grid after adding a combination that contains it to the Mapping grid. Pressing the Delete button will remove the selected mappings and the field and column will become available again in their grid.

Type 2 Slowly Changing Dimensions

The characteristic of a type 2 slowly changing dimension is that it tracks changes in a dimension over time. Whereas a type 1 slowly changing dimension destructively overwrites the old data when a change occurs, the type 2 slowly changing dimension preserves its history and adds a new row to the dimension table to reflect the current situation. In this manner, a type 2 slowly changing dimension maintains a collection of different versions of the dimension row, which are all tied together by the same business key.

In Chapter 4, you witnessed several instances of type 2 slowly changing dimensions: `dim_customer`, `dim_staff`, and `dim_store`. In several of the transformations discussed in Chapter 4, we used a Kettle step that is especially designed to work with this type of dimension: the "Dimension lookup / update" step. We encountered this step in the transformations `load_dim_customer` (see Figure 4-11), `load_dim_staff` (see Figure 4-8), and `load_dim_store` (see Figure 4-13), where it was used to load the dimension tables. We also used it in the `load_fact_rental` transformation (see Figure 4-18) to look up the keys of these dimension tables. In this section, we describe in detail how to use this step to load dimensions. In Chapter 9, we describe how to use this step to look up dimension keys when loading fact tables.

The "Dimension lookup / update" Step

In Spoon, the "Dimension lookup / update" step type resides beneath the "Data warehouse" category in the left pane tree view. The configuration dialog for this step type is shown Figure 8-14. In this case, we used the configuration used in the `load_dim_customer` transformation (see Figure 4-11).

The "Dimension lookup / update" step can operate in two distinct modes:

- It can be used to add and/or update data in a dimension table. This functionality may be used to maintain type 1 and 2 slowly changing dimensions. This mode is referred to as the *update mode*.
- It can be used as a lookup step to retrieve the surrogate key of a type 2 slowly changing dimension. This functionality is especially useful when loading fact tables, and is referred to as the *lookup mode*.

Because the "Dimension lookup / update" step unites these different functionalities, it superficially resembles a hybrid of the Insert / Update and "Database lookup" steps, each of which implements one of these different functionalities on its own. However, it is more complicated than that. Whereas the Insert / Update and "Database lookup" steps perform only and exactly the task implied by their name, the "Dimension lookup / update" step is aware of the history-preserving characteristics of type 2 slowly changing dimensions. In

the update mode, this allows steps of this type to decide whether it is appropriate to add to the history when loading the dimension table, and to automatically provide appropriate values for those columns that do the history bookkeeping in the dimension table. In the lookup mode, the step is capable of automatically selecting the correct version of the dimension row based on date or datetime fields in the incoming stream without having to explicitly implement the details of the logic to do this.

It is possible to build a transformation that provides the same functionality as the "Dimension lookup / update" step based on other Kettle steps. However, this would be quite an elaborate transformation, which would quickly become impractical because it would have to be rebuilt for each type 2 slowly changing dimension table.

Figure 8-14: The configuration of the "Load dim_customer SCD" step of the load_dim_customer transformation

Specifying the Mode of Operation

To specify the mode of operation, use the "Update the dimension" checkbox. When this is checked, the step can be used to load a dimension table. Unchecking this option enables the lookup functionality. In the remainder of this section, we primarily describe the dimension maintenance features. Chapter 9 explores how to use the lookup functionality when loading fact tables.

General Configuration

The "Dimension lookup / update" step has a number of configuration properties similar to the ones already described in our discussion of the "Database lookup" step (see Figure 8-10) and Insert / Update (see Figure 8-12):

- The Connection, "Target schema," "Target table," and "Commit size" properties have similar meaning to those discussed for the Insert / Update step. Note that the "Commit size" is only applicable in the lookup mode.
- The "Enable the cache," "Pre-load the cache," and "Cache size in rows" properties have the same meaning as the "Enable cache," "Load all data from table," and "Cache size in rows" properties of the "Database lookup" step, respectively. The "Pre-load the cache" option is only available in the lookup mode.

Keys Tab Page

The "Key fields (to lookup row in dimension)" on the Keys tab page is used to map the business key of the dimension table to the stream. This resembles the "The key(s) to lookup the value(s)" grids in the "Database lookup" and Insert / Update steps.

The "Dimension field" is used to specify which columns constitute the business key in the dimension table. The "Stream field" is used to specify the fields from the incoming stream to which these columns are to be matched. The fields are always compared based on equality. For this reason, the grid does not supply a way to specify which comparison operator should be used.

There is an important difference with the "Database lookup" and Insert / Update steps in regard to the way the fields and columns are actually matched. The grid only defines how to match the business key. But in a type 2 slowly changing dimension, the business key does not identify a single row in the dimension table! Because type 2 slowly changing dimension tables keep track of history, there may be multiple rows for one particular business key, each representing a version that was valid at some point in time.

There has to be some policy that determines how to pick just one row out of the collection of rows for the same business key. We explain how this works in the subsection "History Maintenance."

Surrogate Key

The "Dimension lookup / update" step offers a number of properties pertaining to the surrogate key of the dimension table. In the dialog shown in Figure 8-14, these can all be found beneath the tabbed pages.

The "Technical key field" should be used to specify the name of the primary key column of the dimension table. For example, in Figure 8-14, this is set to `customer_key`. The "Creation of technical key" fieldset offers options to control the automatic generation of surrogate key values. These options apply only to the update mode because they are used only when adding new dimension rows. You can choose one of the following methods to generate surrogate key values:

- **Use table maximum + 1:** As the name implies, this option will automatically query the dimension table for the maximum value of the column specified in the "Technical key field" property, add 1 to the value, and use the result as the initial value of the surrogate key for newly added rows.

- **Use sequence:** Here you can specify the name of a database sequence that should be used to draw the values from. You can use this option if your database supports sequences. (See also the section on "Surrogate Keys Based on a Database Sequence" earlier in this chapter.) The name entered here should be an existing database sequence that resides in the default schema of the specified connection, or otherwise, the schema specified in the "Target schema" property.
- **Use auto increment field:** You can choose this option if your database supports `auto_increment` or `IDENTITY` columns and the column specified by the "Technical key field" property is defined using that feature. (See also the section "Surrogate Keys Based on a Database Sequence" earlier in this chapter.)

History Maintenance

The "Dimension lookup / update" step has a number of properties to configure how to deal with the history maintained by type 2 slowly changing dimensions. These can all be found in the bottom section of the dialog shown in Figure 8-14, below the properties to define the surrogate key.

The "Version field" property can be used to specify the name of the column in the dimension table that stores the version number of the row. The combination of the business key and this version number can be used to uniquely identify a row in the dimension table. In the update mode, the "Dimension lookup / update" step will automatically store the appropriate version number whenever the step adds a new row to the dimension table.

The Stream Datefield property can be used to specify the Date field from the incoming stream that provides the chronological context for the change in dimension data. For example, in Figure 8-14, this property is set to `last_update`. You might recall that in the `load_dim_customer` transformation, this field originates from the `last_update` column of the `customer` table. This column contains the timestamp value indicating when the row was last changed in the source system.

NOTE **Although the `last_update` column of the customer table seems like a reasonable choice for the Stream Datefield, it is actually not entirely correct. The reason is that the customer dimension is denormalized and thus contains data from multiple tables. This was mentioned earlier in our discussion on change data capture for star schema dimensions.**

Although the customer table in the sakila database represents the lowest level of the customer dimension, it is not the only table that contributes to the denormalized row in the `dim_customer` dimension table. So a change at a higher level, say the customer's address, will in fact count as a new version of the customer dimension record. The chronological context for a change in the `address` table should be taken from the `last_update` column of the `address` table.

So in order to obtain the correct chronological context for the entire denormalized row, you would need to take the values of all `last_update` columns of all tables denormalized into the `dim_customer` dimension table, and then select one of them. For example, it seems reasonable to pick the value of the most recent of these `last_update` columns because you can be sure that the resulting denormalized dimension table record existed for sure by that time.

In many real-world scenarios, source systems do not record when a record was last changed. In these cases, the chronology must be stipulated. In this case, it makes most sense to use the current timestamp. The rationale behind this is that as far as the data warehouse is concerned, the actual change may have happened at any moment in the past. The only thing you can be sure of is that, at the latest, the change took place before the change was detected and loaded into the dimension. For these cases, you do not need to specify a stream field: Not specifying a value for this property automatically causes the system date to be used as chronological context for the dimension change.

Type 2 slowly changing dimensions should have a pair of columns that specify the period in time to which the dimension record applies. The columns that store the start and end of this period should be specified using the "Date range start field" and "Table daterange end" properties. In Figure 8-14, these properties are set to `customer_valid_from` and `customer_valid_through`, respectively. The date range is required to pick the correct version of the dimension row for a particular business key. This is how the matching process of the "Dimension lookup / update" step differs from that implemented by the "Database lookup" and Insert / Update steps: The "Dimension lookup / update" step compares against the business key, but also requires that the value that is used as chronological context lies within the range specified by the "Date range start field" and "Table daterange end" properties. (As we just mentioned, the chronology can be specified explicitly by configuring the Stream Datefield property; otherwise, the system date will be used.)

When a dimension row is first inserted for a particular business key, the range is initialized using a generated minimum and maximum date based on the values in the "Min. year" and "Max. year" fields. By default, the values in these fields are 1900 and 2199, so the first dimension row for any new business key will get `1900-01-01` and `2199-12-31` for the columns specified by the "Date range start field" and "Table daterange end" properties, respectively. The assumption is that this date range is wide enough to fit all history of the dimension table.

If the dimension table already contains an existing dimension row for the business key, its column values are compared with the fields from the incoming stream. If a change is detected, the column specified by the "Table daterange end" property is updated and set to the value of the field in the incoming stream configured in the "Stream Datefield" property. In addition, a new row is inserted into the dimension table to store the change. For the new row, the date range starts with the value "Stream Datefield" field, and ends with the end of the range of the existing row. The final result is that the existing row and the new row will have consecutive date ranges, making it easy to track the chronology of the changes for each distinct business key.

It may be desirable to use another, more realistic value in the start of the date range used for the first dimension record added for a particular business key. To control the way the start of the range is recorded, check the "Use an alternative start date?" checkbox, and use the list box to the right to pick another sort of date. The list offers a number of predefined date values, such as the start of the transformation, or the system date, or the `NULL` value. It seems reasonable to start the date range based on whatever was configured as "Stream Datefield" but unfortunately this is not a supported option. You can, however, set the type to "A Column value" and specify a column of the dimension table instead.

Lookup / Update Fields

In the Fields tab page, you can specify the mapping between the columns of the dimension table and the fields from the incoming stream. This is somewhat like the "Values to return from the lookup table" and Update fields grids of the "Database lookup" and Insert / Update steps, respectively. The Fields tab page of the "Load dim_customer SCD" step shown in Figure 8-14 is shown in Figure 8-15.

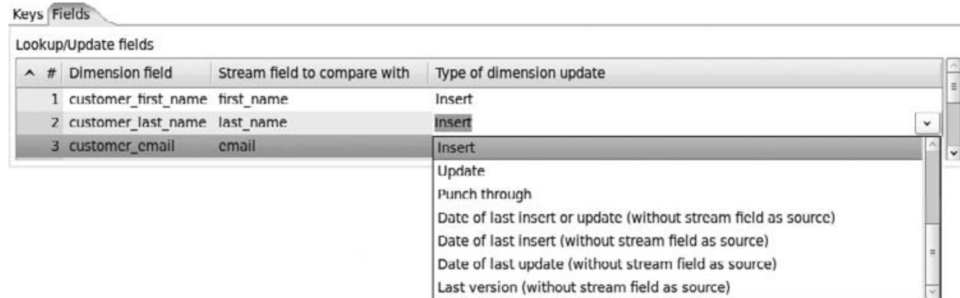


Figure 8-15: The configuration of the Fields tab page of the "Load dim_customer SCD" step

In both the lookup and the update modes, the values of the columns specified the "Dimension field" column will be retrieved from the dimension table.

The "Stream field to compare with" column is used to map the dimension table columns to the fields from the incoming stream. The mapping is relevant in the update mode because the values of these fields are then compared to the values retrieved from the corresponding database columns.

If the values in the dimension table column and the stream field are not identical, then the option specified in the "Type of dimension update" determines the behavior. The nice thing about this is that this enables you to choose a change history policy on a per-column basis, allowing you to manage hybrid slowly changing dimensions, having some columns that conform to a type 1 slowly changing dimension while some columns use a typical type 2 policy or yet another form of history management.

For typical type 2 slowly changing dimension, you should use the Insert option, which inserts a new dimension row for each change.

Other Types of Slowly Changing Dimensions

Types 1 and 2 slowly changing dimensions discussed in the previous sections are by far the most useful and abundant. But other slowly changing dimension types do exist. In fact, in many cases a dimension is not purely type 1 or type 2. Instead, it may be desirable to choose a particular policy depending on the attribute.

Type 3 Slowly Changing Dimensions

Kettle does not provide a specialized step to maintain type 3 slowly changing dimensions. However, you can implement them yourself without too much trouble.

For example, if you're only interested in the current and the immediately previous situation, you can use a Database Lookup step to look up the value of what is now the current value, and store that as previous value along with the new current value using an Update step. If you want to add columns dynamically to maintain more than one version of the past, you can write a job and use the "Columns exist in a table" to check if you need to modify the table design, and a SQL scripting step to execute the required DDL to add new columns.

Hybrid Slowly Changing Dimensions

It should be stressed that the policies for managing change do not have to act on a dimension table as a whole. Because changes occur at the column level, you could apply a different policy for any group of columns. Dimensions that employ multiple policies for tracking changes are referred to as "hybrid" slowly changing dimensions.

When thinking about which policy to choose for a particular column, you may discover that it is not so much the column itself that determines which policy is appropriate; rather, the cause of the change is what's really interesting, and this is what should be considered when choosing when to apply which policy.

For example, consider a customer's birth date. There can be little argument as to what it means when we detect a change in the customer's birth date in the source system: It can be interpreted only as a correction of a previously inaccurate value. But what about a customer's last name? A change in last name can occur to correct a spelling error, but could also indicate that the customer got married and adopted the name of the spouse.

The "Dimension lookup / update" step, which was described in detail in the "Type 2 Slowly Changing Dimensions" section supports a per-column mapping policy for handling change. You can configure the policy by choosing one of the predefined policies per dimension column / stream field mapping in the Fields tab page of the "Dimension lookup / update" step. This is shown in Figure 8-15. The available options are:

- **Insert:** This option implements a type 2 slowly changing dimension policy. If a difference is detected for one or more mappings that have the Insert option, then a row is added to the dimension table.
- **Update:** This option simply updates the matched row. It can be used to implement a Type 1 slowly changing dimension
- **Punch through:** The punch through option also performs an update. But instead of updating only the matched dimension row, it will update all versions of the row in a Type 2 slowly changing dimension
- **Date of last insert or update (without stream field as source):** Use this option to let the step automatically maintain a date field that records the date of the insert or update using the system date field as source.
- **Date of last insert (without stream field as source):** Use this option to let the step automatically maintain a date field that records the date of the last insert using the system date field as source.

- **Date of last update (without stream field as source):** Use this option to let the step automatically maintain a date field that records the date of the last update using the system date field as source.
- **Last version (without stream field as source):** Use this option to let the step automatically maintain a flag that indicates if the row is the last version.

More Dimensions

There are still more types of dimensions to consider. Some examples are discussed in the subsections that follow.

Generated Dimensions

For certain types of dimensions, such as date and time dimensions, the data can be generated in advance. Other examples of generated dimensions include typical mini-dimensions such as a demography dimension.

Date and Time Dimensions

In Chapter 4, we described the `load_dim_date.ktr` (see Figure 4-4) and `load_dim_time.ktr` (see Figure 4-5) transformations, which load the `dim_date` and `dim_time` dimension tables in the sakila rental star schema. The construction of these dimensions was already discussed in some detail there, and will not be repeated here.

You can find more details about generating the data for the localized date dimension at `http://rpbouman.blogspot.com/2007/04/kettle-tip-using-java-locales-for-date.html`.

Kettle also ships with a few example transformations for constructing a date dimension. To review these examples, look in the `samples/transformations` directory right beneath the Kettle home directory for the following transformations:

- `General - Populate date dimension AU.ktr`
- `General - Populate date dimension.ktr`

Generated Mini-Dimensions

In many cases, data for mini-dimensions such as a demography dimension can be generated in advance. The typical pattern for a transformation like this resembles the one used in the `load_dim_time` transformation shown in Figure 4-5 in Chapter 4.

The pattern is to set up multiple Generate Rows steps, one for each independent type of data stored in the dimension table. By adding an "Add sequence" step, you can then generate a number to uniquely identify the individual rows in a stream. For example, see the Generate Hours and "Hours sequence" steps in the `load_dim_time` transformation.

Often, the next step is to use the sequence value as input for some function that generates descriptive labels based on the sequence value input. By function we mean some step that generates meaningful output derived from the sequence number input. Many steps can be used for this purpose, depending on what output data you need, such as the Calculator, Number Range, "Value mapper," and of course any flavor of the scripting steps such as Formula, Modified Javascript Value, or User Defined Java Expression. An example of this is the "Calculate hours12" step in the `load_dim_time` transformation.

The next step is to make combinations of all the streams using a "Join rows (cartesian product)" step. The result is one stream that contains all combinations of the input data. The only thing required now is the generation of a key. Depending on the nature of the dimension table, you might feel that it is appropriate to generate a smart key. This is what is done in the Calculate Time step of the `load_dim_time` transformation. Alternatively, you can use an "Add sequence" step to supply a surrogate key, or leave the generation of the key to an `auto_increment` or `IDENTIY` column in the database. The final step is to store the generated data in the dimension table using a simple "Table output" step.

NOTE **For an example of generating a mini-dimension, take a look at the** `dim_demography.ktr` **transformation described in Chapter 10 of *Pentaho Solutions*. The transformation itself can be freely obtained from the download area at that book's website at** `http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470484322.html`**. Use the tabs to navigate to the download area, and download the archive containing all materials for Chapter 10. For your convenience, the transformation is shown in Figure 8-16.**

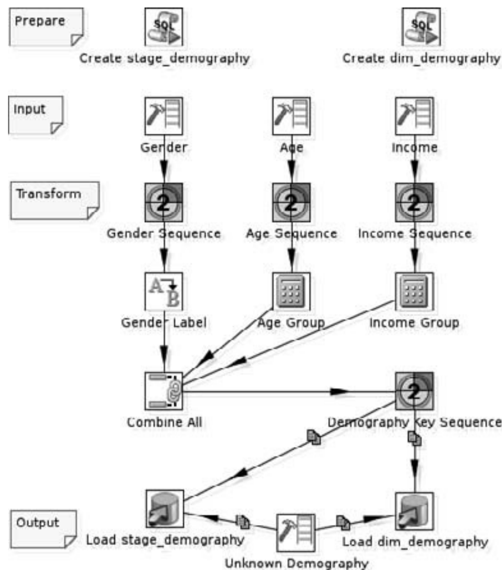


Figure 8-16: Generating a demography mini-dimension

Junk Dimensions

Mini-dimensions are typically junk dimensions: they consist of many different and unrelated attributes that can still be useful for analysis, but cannot (yet) be classified with another dimension. Often, junk dimensions are pulled out of a monster dimension. For example, a customer profile dimension may contain several demographic attributes along with some attributes that describe payment or ordering behavior.

Whereas a mini-dimension like demography can often be generated, this may not be possible or convenient in all cases. Either the total number of possible combinations of the junk attributes may be inconveniently large, or the value space for one of the attributes may not be known beforehand.

The “Combination lookup/update” step is an excellent choice to implement these types of junk dimensions. In Spoon, the “Combination lookup/update” step resides beneath the Data Warehouse category in the left pane of the tree view.

NOTE You encountered the “Combination lookup/update” step type in Chapter 4 where it was used in the `load_dim_film` **transformation (see Figure 4-16) to load the** `dim_film` **dimension table. However, the** `dim_film dimension` **table is not a junk dimension, which is why we don’t discuss that particular example here.**

When you open the configuration dialog of the “Combination lookup/update” step, you will recognize many options that are also present in the “Database lookup,” Insert / Update, and “Dimension lookup / update” steps:

- Connection, table, and schema properties
- A grid for mapping the dimension table columns to fields from the incoming stream
- Options to specify the dimension surrogate key and control the behavior for generating surrogate key values

The “Combination lookup/update” step can best be compared to the Insert / Update step: It looks up rows according to the mapping of dimension table columns to stream fields. If a row is found, it is updated; if no row is found, it is added. The surrogate key column is added to the outgoing stream, so any surrogate key value that is generated in the process can be used in the transformation downstream of the step.

The main difference between the “Combination lookup/update” and Insert / Update steps is that the former does not distinguish between the key fields and the lookup fields: Junk dimensions are characterized by having all kinds of combinations of unrelated attributes, and almost by definition, there is no real natural key for junk dimension tables. Rather, the natural key consists of the combination of all attributes. For this reason, the lookup and update fields are merged into one: The “Combination lookup/update” step is typically used to match all dimension columns (except its surrogate key) to the fields in the stream.

Recursive Hierarchies

Some hierarchies are recursive by nature. For example, the relationship between an employee and his boss or the departmental organization of a company is recursive: the boss of the employee is itself also an employee, and a department is itself a component of an organizational unit at a higher level. Typically, relationships like this are implemented by adding a foreign key to the dimension table that refers to the primary key of the dimension table itself (so-called adjacency list). The foreign key column can be referred to as the *parent key* because it points to the row that is directly hierarchically related. Similarly, we can refer to the original key of the table as *child key*.

ROLAP servers such as Mondrian can use a dimension design like this to drill down from higher levels to the lower levels. However, they can do so only one level at a time: After discovering the children of a particular level, each of these children may itself have children, and another query is required to retrieve those. The problem with that is that there is no convenient way to roll up a higher level and calculate an aggregate of all descendant levels because all levels would have to be queried one by one, until no more children are found.

There is a solution to this problem: You can define a so-called *transitive closure table* or simply *closure table*. Figure 8-17 shows a simplified Entity-Relationship diagram with an employee table with a recursive relationship along with its closure table.

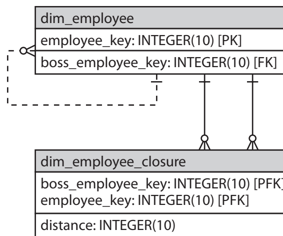


Figure 8-17: A table with a recursive relationship and its closure table

The closure table has at least two columns that derive from the table with the recursive relationship: one for the original key, and one for the self-referencing foreign key. The closure table is filled by querying all descendants for any given row from the original table, and storing the descendant keys along with the key of the original ancestor. This allows the entire set of descendants to be retrieved based on one ancestor key and then allows SQL queries to aggregate.

Kettle provides a Closure Generator step to populate these transitive closure tables. This step type resides in the Transformation category in the left pane of the tree view. The configuration of this step is shown in Figure 8-18.

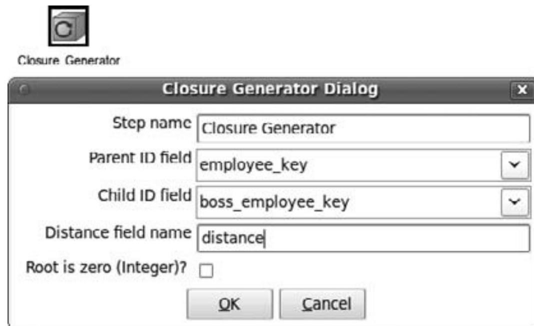


Figure 8-18: The configuration of the Closure Generator step

The Closure Generator step operates only on the incoming stream and generates the closure records to the outgoing stream. To configure it requires only that you specify the field that acts as parent key and the field that acts as child key. The step will also calculate the "distance" between the related rows. The distance is simply the number of levels between the ancestor and descendant pair that is generated by the step. You can specify the name for the field that will be used to store the distance in the "Distance field name" property.

Summary

In this chapter, we discussed in detail how to use Kettle to manage data warehouse dimension tables. The main points covered include:

- How the Kimball's theoretical framework of ETL subsystems ties into the act of maintaining and loading dimension tables in practice
- Business keys and surrogate keys, and what role they play in the data warehouse
- Generating values for surrogate keys with Kettle's "Add sequence" step, and how this relates to using database features such as sequence objects, and auto_increment or IDENTITY columns
- Using Kettle's "Database lookup" step for looking up keys and for loading denormalized dimension tables
- Using Kettle for level-wise top-down loading of snowflaked dimensions
- Loading denormalized dimension tables for star schemas, and the sometimes complicated implications for proper change data capture
- Different types of slowly changing dimensions, most notably Type 1 and Type 2
- Using Kettle's Insert/Update step to load Type 1 slowly changing dimensions

- Using Kettle's "Dimension lookup/update" step, both for loading Type 2 slowly changing dimension tables as well as looking up the key in a Type 2 slowly changing dimension
- Techniques for generating data for special dimensions like the data and time dimension, and mini-dimensions
- Using the "Combination lookup/update" step to load junk dimension tables
- Using the closure generator to generate a transitive closure table to flatten recursive hierarchies

CHAPTER

9

Loading Fact Tables

Fact tables are the storage-hungry collections of data where the analysis details are stored. In a typical data warehouse environment, it's the fact tables that take up the most space, which is measured in gigabytes, terabytes, and sometimes even petabytes. In order to get this data from a source system into the data warehouse, you need a fast loading mechanism, consisting of several components, the basics of which are covered in this chapter. First we discuss bulk loaders, special Kettle plugins that allow you to take advantage of the bulk-loading capabilities of the various types of databases.

NOTE **Although the subject of this chapter is loading fact tables, the Kettle bulk loading features can be used for other bulk load tasks as well, for instance to load flat files in staging tables.**

Before you can load the data into its final destination it needs to undergo other operations as well, such as looking up the correct dimension surrogate keys. We cover the techniques available in Kettle for doing this in this chapter; in Chapter 19 we'll show another technique that's purely based on using SQL.

In the final section of this chapter, we cover three different types of fact tables, as identified by Ralph Kimball. We also introduce a fourth type of fact table, the *state-oriented fact table*, as described by Dutch mathematician and data modeling guru Dr. Harm van der Lek. The chapter ends with a short introduction in the use of aggregate tables and how to load them using Kettle.

Before we start, let's take a step back and look at where the topics in this chapter fit in with respect to the 34 subsystems. This chapter actually completes the previous one

in covering the Data Delivery section of the subsystems. More specifically, the following subsystems are covered here:

- Fact table loader (subsystem 13) is used to generate the load for the various kinds of fact tables.
- The surrogate key pipeline (subsystem 14) is one of the key concepts in a dimensional data warehouse and will be covered in depth, including the various mechanisms Kettle can apply for handling dimension key lookups.
- Multi-valued dimension bridge table builder (subsystem 15) is used for building so-called bridge tables. Chapter 4 contains an example of this type of table for bridging films to actors.
- Late arriving data handler (subsystem 16) covers loading late arriving dimension and fact data. The latter needs some special consideration but don't pose a big problem because looking up the dimension key that was valid at the transaction time of the fact is pretty straightforward. It's the early-arriving facts (or late arriving dimension data) that cause the headaches. We'll describe possible solutions for these issues.
- The aggregate builder (subsystem 19) is needed to precalculate summarized data to speed up analysis performance. To date, none of the open source databases offer an automated aggregate navigation mechanism as Oracle and other commercial databases do with materialized views. Still, you can benefit from aggregate tables because they can be used within a Mondrian schema or when you need to develop standard reports that require high-level overviews and speedy performance. The trick with aggregate tables is to have them automatically refreshed when new facts are added. With Kettle this is actually pretty straightforward, as we demonstrate at the end of this chapter.

Remember that loading facts is usually the final stage in an ETL process. First, data is extracted and written to a staging file or table. Then all dimension tables need to be updated, and finally the updated dimension information can be used to update the fact tables. Although this is the preferred way of processing data, it's not always possible to do so. This could result in either dimension or fact data arriving "late," and the late arriving data handler subsystem should take care of that.

Loading in Bulk

Loading data into a database table using standard *Data Manipulation Language* (*DML*) operations works fine if your data size is limited to a few thousand or even a few hundred thousand rows.

When loading millions or even billions of rows, however, insert statements just don't cut it anymore. Why is that? Simple: All DML operations such as insert, update, and delete are logged by the database management system. This means that in addition to the operation to insert a row into a table, there's a second operation that writes the fact that the record is inserted into a database transaction log. This is not the only thing

that slows down standard insert statements: All constraints on the table columns are also checked as rows are inserted into the table. Both logging and constraint checking can cause considerable delays when large datasets need to be loaded, so for optimal performance the process of *bulk loading* was invented. Basically, there are two ways to load data in bulk:

- **File-based:** Data is available in a flat file and must be read from disk. Usually the database requires a bulk loader definition (the *control file*) telling what fields and delimiters are being used. There are two possible scenarios in which flat files play a role. The first one is when the file is already available and can directly be loaded into the database. The second scenario is that you want to bulk load data that results from a transformation. In the latter case there are both advantages and disadvantages to using a file based bulk loader. The major disadvantage, of course, is that an intermediate file needs to be created, which is then read by the bulk-loading process. This adds two extra I/O streams to the data processing workflow, which increases the data volume threshold for choosing between regular inserts and bulk loading. There are advantages as well: An intermediate file adds an extra layer of reliability to a solution. If the bulk load fails, it's fairly easy to resolve possible issues and restart the load. And sometimes a ready-to load-file is delivered by another system that cannot be accessed directly.
- **API-based:** There is no intermediate file storage. The data is loaded directly into the database from the ETL tool by using a bulk loader API or STDIN as the data source. In theory, this is the easier and faster way of bulk loading because there's less disk I/O and no need for specifying and creating control files. Unfortunately, Kettle currently doesn't contain bulk loaders that can call, for example, the Oracle or Microsoft SQL Server bulk load APIs.

Each database has its own way of handling bulk loading and the methods and requirements vary wildly, from the simple `LOAD DATA INFILE` command used in MySQL to the elaborate SQL*Loader specification of the Oracle database. Not every database, however, has a bulk loader that can read directly from the input of another program. To benefit from the direct path loading capabilities of Oracle's SQL*Loader, calls to the Oracle Call Interface (OCI) must be sent from the external program issuing the bulk load.

STDIN and FIFO

Some databases, most notably PostgreSQL and MonetDB, allow for STDIN to be used as the source for the bulk load. STDIN is shorthand for *standard input* and simply refers to the data stream going into a program. When you type text using a keyboard, the keystrokes are the STDIN for the word processor that receives this input. In a similar way, the data output of a Kettle step can be used as the STDIN for the receiving bulk loader process. For the transfer of the data, Kettle can use a FIFO.

FIFO is shorthand for *first in, first out*, to indicate that the order of the data going in is the same as the one coming out. An alternative name for a FIFO is a *named pipe*. To most people in the Linux and UNIX world, a pipe is a very familiar concept. Take

for instance the command `ls -l | grep 06`. This will direct the output of the `ls -l` command (a detailed listing of all files) to the `grep 06` command (showing only the lines that contain the character combination `06`) using a pipe (|). The data that's exchanged between the two commands only exists in memory for the duration of the chained command and cannot be accessed by any other means. This changes when you use a *named* pipe to exchange this data. A named pipe is a special type of file that is also displayed by the `ls` command but actually exists only in memory. To create a named pipe, use the `mkfifo` command, after which `fifo` can be used to direct output to and read data from. As a simple example, open two terminal screens and type the following commands in one terminal:

```
mkfifo mypipe
ls -l > mypipe
```

Then go to the second terminal and type

```
cat < mypipe
```

As you will discover, the command in the first terminal appears to do nothing until the command in the second terminal is executed. This is the expected behavior because both sides of the pipe need to be connected before the flow of data can start. If you run the `ls` and `cat` commands again but in reverse order (first `cat < mypipe` on terminal 2 and then `ls -l > mypipe` on terminal 1), you'll notice that it doesn't matter which end of the pipe is opened first.

The reason we explain this in such detail is the fact that named pipes are the default mechanism for the MySQL Bulk Loader transformation step, and for the majority of people starting with an open source database, MySQL is the default choice.

Kettle Bulk Loaders

As mentioned before, every database has its own way of offering bulk loading capabilities. This means that for every database a different bulk loader component needs to be created. Kettle offers bulk loader steps for the most common databases such as Oracle, SQL Server, MySQL, and PostgreSQL, but also for more exotic products such as Greenplum (experimental), MonetDB, and LucidDB. While we can't cover each one in detail in this chapter, we offer a quick summary of the main considerations to keep in mind for each database and point you to other references to find more in-depth coverage of the specific database loader functionality. You may notice that DB2 is missing so if you're a DB2 shop, you need to either create a CSV file first and load it using the DB2 load utility, or write your own plugin for it.

You might already have noticed that both the Job and the Transformation step collections contain bulk loaders. There's a very good reason for this: All the file management operations belong in a job as these are distinct operations executed sequentially. The bulk loaders that are in the "Bulk loading" section of the Job design tab all require a physical file to be present for the bulk load, whereas the bulk loader steps that can be used in a transformation are capable of reading data directly from the stream. The next subsections highlight some of these transformations.

MySQL Bulk Loading

MySQL might be the most widely used database around but it's probably not the best solution for large data warehouses. Nevertheless, the bulk loader support within Kettle is extensive. MySQL is also the only database for which bulk loading from the database to a file is available, which is actually a bulk extracter instead of a loader. And there are two options for bulk loading into MySQL: one as part of the job steps that's using files as input, and one as a transformation step. We've already explained the difference. The job version expects a file to process; the transformation version can handle data directly from a stream and load it into a MySQL table using a FIFO file. You can find more information on the load facilities of MySQL at `http://dev.mysql.com/doc/refman/5.1/en/load-data.html`.

LucidDB Bulk Loader

Previous versions of Kettle had a FIFO-based stream loader for LucidDB, but since Kettle 4.0 it has been replaced by a new version that can push data directly from Kettle into the LucidDB column store. Since the complete documentation including examples is available on `http://pub.eigenbase.org/wiki/LucidDbPDIStreamingLoader` there's no need to repeat that information here. Three things are important to note, however:

- The LucidDB bulk loader is smarter than most loaders; it allows for intelligent updating via `MERGE/UPDATE` in addition to the traditional bulk `INSERT`.
- LucidDB bodes well with all components in the Pentaho BI suite, not only with Kettle. For instance, the integration with Mondrian is more than excellent.
- As a column-based data warehouse database, it offers at least a 10-fold performance increase over classic row-based databases such as MySQL or PostgreSQL.

For more information about LucidDB, please visit `http://luciddb.org/`.

Oracle Bulk Loader

The Oracle loader step is one of the more overwhelming components within Kettle because of the enormous number of options that can be set and the number of file locations to enter. Oracle's SQL*Loader (the utility that provides the loading capabilities) has been around for several decades and is still one of the most widely used loading utilities in existence. If you've never worked with SQL*Loader before, it's best to start with the Wiki entry at `http://wiki.pentaho.com/display/EAI/Oracle+Bulk+Loader`. For more background information about SQL*Loader, you can go to the Oracle documentation at `http://www.oracle.com/pls/db111/homepage` and search for "SQL*Loader Concepts."

There are good and bad things about this loader. The bad is easy to see: There are many options and it requires a lot of preparation and setup before you can start loading data. The good news, however, is the fact that the tool is extremely robust and lets you specify exactly how your data should be handled, including possible errors and discarded records. Kettle also saves you the trouble of having to create a control file because that's being done on-the-fly based on the metadata of the incoming data stream.

PostgreSQL Bulk Loader

The PostgreSQL bulk loader is still an experimental step that lets you copy data from a Kettle stream directly into the database using PostgreSQL's standard `COPY` command. The manual offers the following as the main command, with several other options available for specifying a delimiter, escape character, and so on:

```
COPY tablename [ ( column [, ...] ) ]
    FROM { 'filename' | STDIN }
```

As you can see, both `filename` and `STDIN` are available as input options, but Kettle uses only the latter. The Wiki entry at `http://wiki.pentaho.com/display/EAI/PostgreSQL+Bulk+Loader` is a good starting point for the explanation of the PostgreSQL bulk loader, but beware that there are still many issues with this loader—hence its "experimental" status.

NOTE In order for the PostgreSQL bulk loader to work, you'll have to define a trusted connection to your server. See the Pentaho Wiki entry at `http://wiki.pentaho.com/display/EAI/PostgreSQL+Bulk+Loader` **for detailed instructions.**

Table Output Step

This might be a surprising entry here, since the Table Output step is not a bulk loader at all but uses the standard insert SQL calls to a database. As we explained in the introduction of this section, loading several thousand or maybe even more rows using a Table Output step might work perfectly. You can even increase the throughput for this step by running several of them in parallel, as we show in Chapters 15 and 16. There are a few compelling reasons for sticking with the Table Output step too. It is easy to use, has good field mapping options compared to some of the bulk loaders, and it will work with any database.

General Bulk Load Considerations

You should take a couple of things into account when using bulk loading, especially when intermediate files are used. You should, of course, make sure that there is ample space in the location specified for the files. And as stated before, the additional file I/O might be costly, performance wise. Luckily, general advancements in memory and storage technology and the constantly dropping price points can help you to overcome this potential bottleneck. You can consider the following techniques if performance is an issue:

- Use a fast array of SSD drives to write and read the files needed for bulk loading operations.
- Create a RAM disk if you have enough available memory.
- Use one of the parallelization and clustering techniques covered in Chapter 16.

As always, there's a speed-versus-reliability tradeoff; data on a RAM disk isn't stored physically, and the fastest RAID option is raid 0, which is also the least secure RAID alternative.

Another thing to be aware of is that, in general, there are two types of bulk loading, Insert and Truncate. The `Insert` operation appends rows to an existing table, leaving all other data unchanged, whereas the `Truncate` operation removes all existing data from the table. These options also allow you to optimize your solution. A staging table should, in general, be truncated before data is loaded, but you probably don't want to truncate and completely reload the data warehouse fact tables. For those, several strategies can be used, especially when the tables are very large. These strategies, however, depend on the type of fact table. Different fact table types are covered later in this chapter. We'll discuss the appropriate bulk loading strategy for each type there as well.

Dimension Lookups

One of the key operations to perform during a fact table load is the lookup of the correct surrogate keys of dimension tables. In the previous chapter, we covered the generation of surrogate keys and explained why you need surrogate keys, especially when preserving history (type 2 SCD) is important. Another reason for using single-column integer keys is because they allow for very compact fact tables that take up considerably less space than original key values from the source system. When storing hundreds of millions of rows, each byte saved in a fact table row helps to cut down storage requirements and improve query speed.

Maintaining Referential Integrity

For a dimensional data warehouse, each foreign key in a fact table should correspond to a primary key in the corresponding dimension table. In a regular transactional database, this referential integrity is enforced by using foreign key constraints. These constraints prevent the following, related problems from happening:

- Deleting a dimension record, leaving all the related fact table records with an unknown dimension key
- Inserting a foreign key in the fact table for which there is no corresponding dimension primary key

It is not always feasible to use foreign key constraints in a data warehouse, especially while loading large amounts of data. The database must check the constraints for every inserted row, slowing down the overall load process. Not having foreign key constraints isn't a big problem, however: As you may recall, dimension records are never deleted so a dimension key will always exist for a loaded fact record. And by using a dimension key lookup to prepare the fact records for loading, the existence of a foreign key is guaranteed as well. At least, that's the theory, and for many people the daily practice too.

There are other approaches with respect to foreign key constraints as well: an alternative to not using foreign keys is to enforce them, but only after the fact table has been loaded. This means using an "Execute SQL script" step before the load starts to drop all existing foreign key constraints, do the fact load, and after that execute a second "Execute SQL script" step to create them again. When this last step fails you'll immediately know that there's a referential integrity problem in your data warehouse, which should cause a massive alarm.

As with many things in data warehousing and ETL, it's hard to tell which approach is best. In 99 percent of the cases, using a well designed surrogate key pipeline together with proper CDC handling will work perfectly. Combined with the test techniques presented in Chapter 11, you can guarantee a correct load of the data warehouse and avoid the extra development and load time required for dropping and re-enabling the foreign key constraints. Still, there can be very good reasons to use them. We already mentioned one (detecting faulty loads), but query performance is another one; most databases will use the foreign key constraints to optimize the query execution plan and thus yield better response times.

The next section covers the techniques available for looking up the correct key values.

The Surrogate Key Pipeline

The general principle for finding the correct dimension keys is fairly straightforward and is often pictured as a pipeline, as you can see in the simple example in Figure 9-1 where three "Database lookup" steps are used. For reasons of simplicity and compatibility we've used a "Table output" step instead of one of the bulk loaders.

Figure 9-1: The Surrogate Key Pipeline

Chapter 4 already contained a more elaborate example (load rentals) using the Sakila example database where the correct dimensions keys were retrieved based on the rental date. If you translate this lookup function into a more generic example in a kind of pseudo SQL, you get the following:

```
SELECT      dimkey
FROM        dimtable
WHERE       fact_businesskey = dimtable.businesskey
AND         valid_from <= fact_date
AND         valid_to   >  fact_date
```

What happens in the preceding example is that the dimension key that was valid on the day the incoming transaction took place is retrieved. Because there are multiple conditions that need to be evaluated, this is not the fastest way of retrieving dimension keys. If there is a regular load process for the data warehouse and there haven't been

any delays in delivering both dimension and fact data, it is sufficient to retrieve the latest version of the dimension keys.

Using In-Memory Lookups

The fastest way to retrieve information is from the computer's RAM. So the fastest way to find a surrogate key for a certain business key is to store all the dimension information needed for the lookup in memory. This is achieved by selecting both `Enable cache` and `Load all data from table` in the "Database lookup" step. Be careful, however, because you can quickly run out of memory with large dimension tables, especially if you chain all key lookups together in a single transformation. Also, make sure to limit the columns selected to the ones that are actually needed for doing the lookup. It doesn't make sense to load an entire dimension table into memory when all you need are the dimension and business key, and optionally the valid from and to dates.

> **TIP** **The maximum amount of memory available to Kettle is limited by the** `-Xmx` **setting in the startup scripts (Spoon, Carte, Kitchen, and Pan), or if using Kettle.exe on Windows in the Kettle ini file** `kettle.l4j.ini`**.**

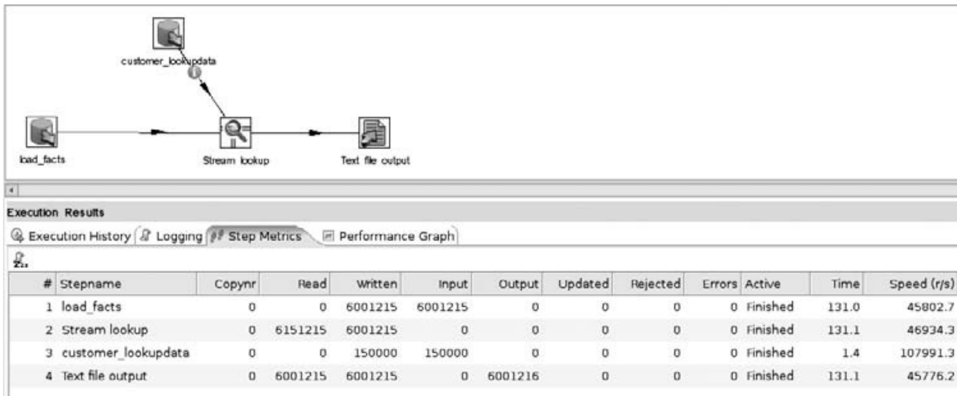
Stream Lookups

In most cases, the "Database lookup" step will work just fine. It doesn't solve every lookup problem, however. What if the lookup values are coming from an external source, or aren't stored in a database at all? That's where the "Stream lookup" step can help. This step doesn't have all the options available in the "Database lookup" step and only allows for basic key lookups based on a single lookup value. It does, however, allow you to use any available step as input to get the lookup values from. For the typical case where dimension and fact data arrive at the same time, using a table input step to retrieve the latest version of the dimension records and using that as the input step for the stream lookup is a very efficient way of doing lookups on large dimension tables.

The efficiency is established by the fact that only the actual records are available in the lookup set, so if a customer dimension contains 3 million rows with all history preserved but the current records are a third of that (which is not an uncommon assumption), only 1 million rows have to be searched in the lookup step using the stream lookup. Even with the same number of rows, the "Stream lookup" step is a little faster, as Figures 9-2 and 9-3 will show. For this exercise, we used a TPC-H dataset of 1GB (see `http://www.tpc.org` for details about the TPC-H benchmark database), which gives us some data volume to play with. A 1GB database has 150,000 customer records, 1.5 million orders and roughly 6 million lineitem rows. The customer dimension created from this data is an almost 1 to 1 copy, except for an added `customer_id` column using the Add Sequence step.

Figure 9-2 shows a stream lookup where the lookup data is pre-fetched using a "Table input" step with the following query in it:

```
select customer_id, c_custkey from dim_customer where current_
flag = 1
```



#	Stepname	Copynr	Read	Written	Input	Output	Updated	Rejected	Errors	Active	Time	Speed (r/s)
1	load_facts	0	0	6001215	6001215	0	0	0	0	Finished	131.0	45802.7
2	Stream lookup	0	6151215	6001215	0	0	0	0	0	Finished	131.1	46934.3
3	customer_lookupdata	0	0	150000	150000	0	0	0	0	Finished	1.4	107991.3
4	Text file output	0	6001215	6001215	0	6001216	0	0	0	Finished	131.1	45776.2

Figure 9-2: Using a "Stream lookup" step

NOTE **Note that the example in Figure 9-2 does use a "Table input" step as the data source for the "Stream lookup" step, which might be confusing. It is, however, just an example: any step that provides data can be used to feed the data into a stream lookup.**

The stream lookup itself is displayed in Figure 9-3. As you can see, it's a fairly straightforward step where the input step for the lookup data is selected in the Lookup step drop-down list.

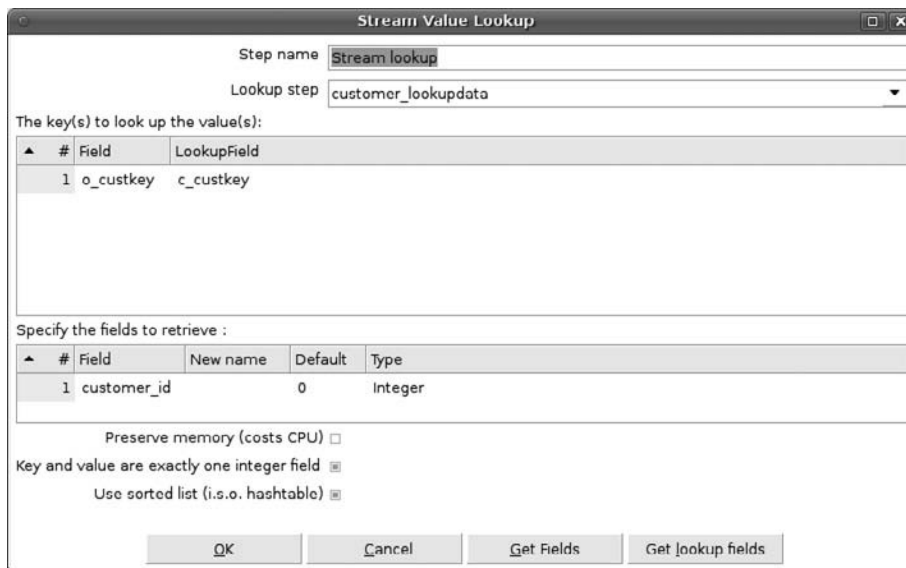
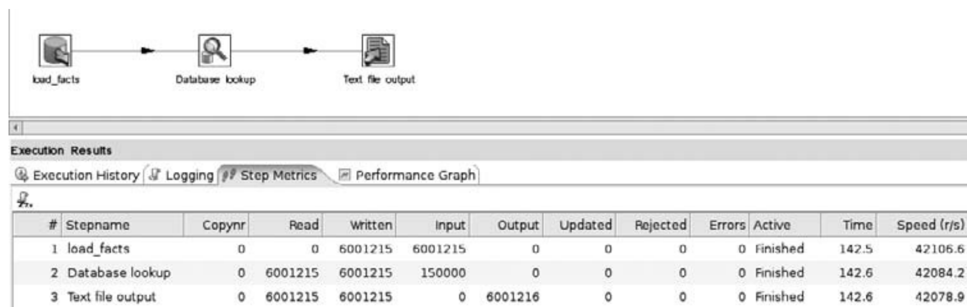


Figure 9-3: Inside the "Stream lookup" step

It's also possible to use multiple keys, but only with an implicit "equals" operator. In this case, the incoming customer key from the orders table (`o_custkey`) is compared with the lookup field `c_custkey` from the dimension table. The field to retrieve (again from the dimension table) is `customer_id`. Don't forget to specify the type of the field; the default is a hyphen (-), which causes errors within subsequent steps. The three checkboxes at the bottom serve to tweak the way Kettle compresses and sorts the data.

NOTE **For an excellent explanation/discussion about the memory options available in the "Stream lookup" step, go to** `http://forums.pentaho.org/showthread.php?t=68058`**.**

As a comparison, you can run the same transformation but then with a regular "Database lookup" step; the results are displayed in Figure 9-4.



#	Stepname	Copynr	Read	Written	Input	Output	Updated	Rejected	Errors	Active	Time	Speed (r/s)
1	load_facts	0	0	6001215	6001215	0	0	0	0	Finished	142.5	42106.6
2	Database lookup	0	6001215	6001215	150000	0	0	0	0	Finished	142.6	42084.2
3	Text file output	0	6001215	6001215	0	6001216	0	0	0	Finished	142.6	42078.9

Figure 9-4: "Database lookup" step

If you look closely, you can see that the stream lookup is a little bit faster, even with the same amount of data in the lookup set. Whether to use the "Stream lookup" in every case is hard to tell; as you can see from the examples it always takes at least one extra step, and there's only a simple key comparison available for the lookup. In all cases where more advanced lookup conditions are required, for instance when late-arriving facts need to be handled, a database lookup step is required. The next section explains how to handle late arriving data

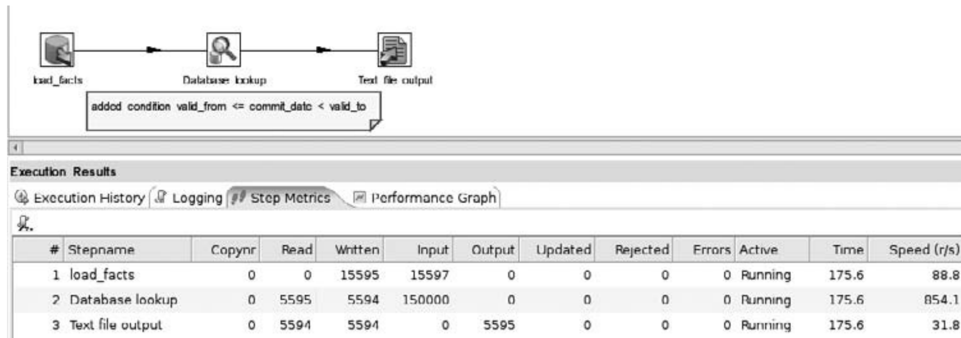
Late-Arriving Data

Under normal circumstances, data is extracted from source systems at regular intervals and processed with Kettle. As we explained in the chapter introduction, the standard order is to first handle the dimensions, and then the fact tables. This order is important because you need the surrogate keys in the fact tables, and these keys have to be generated first (if necessary) using the dimension loading process. Now what happens when the data delivery is out of sync and fact or dimension data arrives with a delay of several days or even weeks? This can cause irregularities in the data warehouse; it's one

thing to have customers without any orders (which could be true of course), but orders shipped to an unknown customer makes reporting on this data quite hard. Still, there are resolutions for these problems as we describe in the following sections.

Late-Arriving Facts

Facts arrive late when transaction data arrives at the ETL process long after the transaction took place. During the delay, the related dimension data might have gotten new versions so a simple lookup on the current dimension records might return false results. The problem is easily resolved by using the `valid_from` and `valid_to` timestamps to find the correct version of the record and the accompanying surrogate dimension key. But *easily* doesn't necessarily mean *fast*. Just look at Figure 9-5 where we added the date conditions to the "Database lookup" step we used earlier. Now compare the process counters with the values in Figure 9-4, and you'll see a dramatic difference.



#	Stepname	Copynr	Read	Written	Input	Output	Updated	Rejected	Errors	Active	Time	Speed (r/s)
1	load_facts	0	0	15595	15597	0	0	0	0	Running	175.6	88.8
2	Database lookup	0	5595	5594	150000	0	0	0	0	Running	175.6	854.1
3	Text file output	0	5594	5594	0	5595	0	0	0	Running	175.6	31.8

Figure 9-5: Late-arriving facts performance

The message here is that when late-arriving facts occur, it's best to isolate these records from regular loads because it might take a long time to process all records using the `valid_from` and `valid_to` conditions, even when indexes are created for these columns.

Late-Arriving Dimensions

The late arrival of dimension data is the reverse situation where the fact records have been processed (or need to be processed), but the dimension records are not yet available. Unfortunately, this is a lot harder to solve than the late-arriving fact issue because you need to be prepared for this to happen. The problem is that a key exists in the incoming transaction (let's assume it's a new customer), but there's no dimension record yet. As a result, the dimension lookup will fail. A simple solution would be to add an *unknown* record to the dimension table with an ID that's reserved for these situations. Usually the ID 0 is used, but some people prefer `-999` or another special value. In

the "Database lookup" step, this is achieved by adding a Default value, as shown in Figure 9-6. This value gets assigned when no corresponding ID can be found based on the lookup keys and criteria.

Figure 9-6: Default lookup return value

Now when a report is run, there is an `unknown` category in the data showing all transactions with an unknown customer or unknown product.

Does this solve the problem? Well, not really. There's no way of ever getting the correct dimension key assigned to the fact row after assigning the `unknown` value. So instead of talking about "unknown," it's better to speak of "not yet known," which leads to a better solution for this problem. First let's explain the workflow and then show how the solution works in Kettle:

1. A fact row comes into the ETL process and one of the lookups fails. Let's assume it's a missing customer with a source system customer key ABC123.
2. Instead of assigning an `unknown` ID, send the row to a sub-process.
3. The sub-process uses the unknown customer key ABC123 to create a new dimension record. All values in this record (name, address, and so on) are set to `N/A` or `unknown` because the only information we have is the customer source system key.
4. Submit the failed fact rows to the original process again; the customer lookup will now find the newly created dimension ID.
5. As soon as the real customer information is available, the dimension is automatically updated with the new information.

The last step of this process will probably create a new version of the customer record with another dimension ID than the one that was used to load the fact record. This means that this fact record still points to the dimension record with all columns set to `N/A` or `unknown` It's a business decision whether this situation should be maintained or not. Keeping the original reference shows that something was wrong with the data at the time of loading, but it would also make sense to do only an update of the dimension record when the correct customer record is received by the dimension load process, and not create a new version.

EXAMPLE: USING KETTLE TO HANDLE LATE-ARRIVING DIMENSION DATA

To illustrate the process using Kettle we've created a very simple example consisting of a single customer dimension with only a few fields, and a single fact table with `customer_id`, `sale_date`, and `sale_amount` as the only available columns. There are six customers, added by using a Data Grid step, as shown in Figure 9-7.

Customers → Add sequence → Add constants → dim_customer_init

Add constant rows

Step name: Customers

Meta | Data

#	cust_key	cust_name	cust_country
1	1	Doug	USA
2	2	Richard	USA
3	3	James	USA
4	4	Matt	BE
5	5	Roland	NL
6	6	Jos	NL

Figure 9-7: Default lookup return value

Note that there are six customers in this table, which are all loaded into the `dim_customer` table. Next we need to create a few sales records, of which one will get a customer key not present in the dimension table. Figure 9-8 shows the sales data; note that `cust_key` 7 does not exist in the dimension table yet.

Salesdata

Add constant rows

Step name: Salesdata

Meta | Data

#	cust_key	sale_date	sale_amount
1	1	21-05-2010	23
2	1	30-06-2010	15
3	3	11-05-2009	76
4	6	15-02-2010	82
5	7	03-09-2010	45

Figure 9-8: Sample sales data

This data is read and passed to a "Database lookup" step to find the correct `customer_id` values for the customer keys of the source records. The lookup will find the customer IDs for the first four fact rows, but not the record for the `cust_key` with value 7. In order to handle failed lookup data, we need to add an error output step for the lookup step to catch all the fact rows that cannot be processed. This data can be stored in a file, a database, or even in a temporary space using the "Copy rows to result" step. In this example, we'll use a simple text file so the complete workflow looks like the one in Figure 9-9.

EXAMPLE: USING KETTLE TO HANDLE LATE-ARRIVING DIMENSION DATA

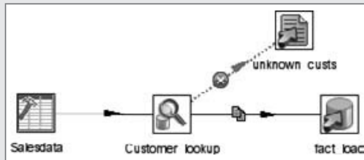


Figure 9-9: Output error fact rows

Now that we've isolated the unknown customer keys, we can process them separately and create new customer dimension records. This is a very straight-forward process, which can consist of the steps displayed in Figure 9-10.

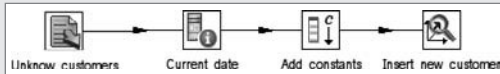


Figure 9-10: Add unknown customers

First the unknown records are read; a "Get System info" step is then added to retrieve today's date. Using an "Add constants" step, we can add the missing values to the stream, as shown in Figure 9-11.

Add constant values

Step name: Add constants

Fields :

#	Name	Type	Length	Precision	Decimal	Value
1	cust_name	String	35			N/A
2	cust_country	String	20			N/A
3	current_flag	Integer				1

OK Cancel

Figure 9-11: Adding N/A values

Finally a "Dimension lookup / update" step takes care of generating the new `customer_id` and adding the record to the customer dimension table. (See the previous chapter for an elaborate explanation of the "Dimension lookup / update" step.) Figure 9-12 shows the resulting records in the dimension table.

Continued

EXAMPLE: USING KETTLE TO HANDLE LATE-ARRIVING DIMENSION DATA

customer_id	cust_key	cust_name	cust_country	valid_from	valid_to	cur
7	7	N/A	N/A	2010-04-08 00:00:00	2999-12-31 23:59:59	1
6	6	Jos	NL	1900-01-01 00:00:00	2999-12-31 23:59:59	1
5	5	Roland	NL	1900-01-01 00:00:00	2999-12-31 23:59:59	1
4	4	Matt	BE	1900-01-01 00:00:00	2999-12-31 23:59:59	1

Figure 9-12: New customer added

The final step is to process the rejected fact records again, which will now be correctly loaded with the newly found `customer_id`.

Fact Table Handling

If you recall the ETL subsystems from Chapter 5, you already know that there are three types of fact tables according to the Kimball Group. The first one is the standard transactional fact table where each row represents a fact such as a product sale, a banner click, or an e-mail coming in. Each of these events occurs at one point in time and a single timestamp can be linked to the transaction. The nature of this data makes it fairly easy to summarize it by some dimension hierarchy level such as year, product group, or country. So far, all the examples in this book have been based on transactional fact tables.

Periodic and Accumulating Snapshots

The second type of fact table is the *periodic snapshot* table, which is used to store the status of certain information periodically. One of the reasons for using snapshots is that storing each individual transaction in a fact table might be overkill. Think, for example, of a large distribution warehouse where thousands of items are stored and constantly moved. For reporting purposes, it wouldn't make a lot of sense to store each individual movement of each inventory item. On the other hand, a warehouse manager might be interested in tracking total product inventory by day or by month, and this is exactly where a transactional snapshot table is useful.

Good examples of periodic snapshot tables are inventory counts and bank account or insurance policy status summaries. One thing to keep in mind with periodic snapshots is that the measures in these tables are *semi-additive*. In a regular fact table, you can sum sales amount by period, customer group, and product line. In a periodic snapshot table, you cannot summarize on all dimensions, especially not on the time dimension. Remember that a periodic snapshot shows a total by period. Calculating the sum of all bank accounts for all periods is therefore meaningless; calculations that can be used are minimum, maximum, or average totals, hence the word semi-additive. There is also *non-additive* data, meaning that you cannot summarize the data over any dimension. A good example for this is room temperature. The temperature in different rooms varies

over time, and you can take a periodic snapshot of this measure, but summarizing the temperature for all rooms is meaningless, just like summarizing the temperatures for each hour.

WARNING **A word of caution about periodic snapshots is in order here. Suppose you need to store the daily account status for a large bank with millions of customers. This means that each day, a record is added to the snapshot table for every customer the bank has, even though there hasn't been a change in the account status. These tables can therefore become extremely large.**

The third type of fact table is the *accumulating snapshot* table, which is actually a special case of a regular transactional fact table. The term "accumulating" denotes the fact that this type of table gets updates over time, usually when steps in a process have been completed. The term "snapshot" might be a bit misleading and shouldn't be confused with the periodic snapshot. Calling this type of table an accumulating *fact table* would probably have made more sense, but the industry convention is snapshot, so we'll use that as well. What is actually meant here with the term "snapshot" is that each record shows the stage of completion in a process. A record might therefore have only an order date filled in, meaning that the status is currently "ordered."

A good example is a typical sales process that consists of an order, picking, shipping, invoicing, and payment step, each with its own date. In fact, the `fact_rental` table introduced in Chapter 4 is an accumulating snapshot table with a rental date and return date.

Introducing State-Oriented Fact Tables

When you look closely at the three types of fact tables covered previously and perhaps are familiar with insurances or other data that change very infrequently, you'll see that something is missing. Let's say I've insured my house and bought a fire policy for it. There are a few relevant points in time to consider:

- The creation of the policy
- Changes in yearly due amounts (typically these amounts increase over the years)
- Changes in the insured value
- Execution of the insurance (in case of the house burning down)
- Expiration of the policy

Basically, these are the dates that something changes in our policy. It is quite usual for these kinds of policies to not change at all for several years in a row. Yet the only way to use this kind of data in a data warehouse environment is to create a periodic snapshot to store the monthly or yearly status of every policy. This seems a bit of a waste, and also doesn't reflect the changes made in between the period ends when the snapshots are taken. Think for instance of a policy that changes at the 20th of the month but snapshots are taken on the 1st day of the next month; the fact that the change was made on the 20th is lost. It would be better to store the *state* of the policy and only change the data when an actual change in the source system takes place, much like

a transaction fact table. This is why we need a fourth type of fact table, called a *state-oriented fact table*. The concept of state orientation was first recognized by Dutch data warehouse specialist Dr. Harm van der Lek. The state-oriented fact table material and examples here are reused with kind permission from Dr. van der Lek.

A *state-oriented fact table* is a table where each row represents the state of an object during a span of time in which this state didn't change. Changes in higher level objects are to be considered changes in the low-level object at hand. State-oriented fact tables might fit the bill if a combination of the following business requirements exist:

1. There are low-level objects relevant to the business, which we need to analyze.
2. The objects change in an unpredictable way.
3. These changes have to be captured on a very detailed level, including changes in parent objects.

Here's a simple example: The lowest level objects are saving accounts. We already have one slowly changing dimension of type 2, the customer dimension. In Table 9-1, you see that a customer apparently moved from New York to Boston on July the 21, 2009.

Table 9-1: A Slowly Changing Dimension

CUSTOMER VERSION_KEY	CUSTOMER CODE	FROM DATE	TO DATE	CITY
5	CG067	1900-01-01	2009-07-21	New York
8	CG067	2009-07-21	9999-12-31	Boston

Table 9-2 shows an example of a state-oriented fact table referring to the customer dimension (among others) and containing the semi-additive measure balance.

Table 9-2: A State-Oriented Fact Table

ACCOUNTNR	FROM DATE	TO DATE	CUSTOMER VERSION_KEY	BALANCE
3200354	1900-01-01	2009-07-10	5	$ 100
3200354	2009-07-10	2009-07-21	5	$ 200
3200354	2009-07-21	9999-12-31	8	$ 200

On July 10, 2009, the balance changed from $100 to $200, so a new state was inserted (the second row). The move of the customer has to be reflected is this table as well so we again add a new row pointing to the new version of the customer and repeating the balance because that was not changed. This technique allows us now to retrieve the total balance per city on a particular moment in time, for example at the end of July 2009. It should be clear from this example that the balance of this CG067 customer should be added to the Boston total because that was the valid state at that moment in time.

The really neat thing about storing states in this way is that you can still define periodic snapshots based on a state-oriented table. Because we have the complete history available, a snapshot can be derived fairly easily. First we need to create an additional table with all the points in time for which a snapshot is required. As an example, let's assume this is a monthly snapshot so we'll store our periods in a `Month` table, as displayed in Table 9-3.

Table 9-3: Month Table

MONTHNR	LASTDAY
200906	2009-06-30
200907	2009-07-31

We can now use this information to create the periodic snapshot, either as a view or (as we'll do later in the chapter) in a physical table using the following SQL statement:

```
SELECT Month.MonthNr
     , SOFactTable.AccountNr
     , SOFactTable.Customer_version_key
     , SOFactTable.Balance
FROM   SOFactTable
     , Month
WHERE  Month.LastDay >= SOFactTable.From_date
  AND  Month.LastDay < SOFactTable.To_Date
```

The result of this query, based on the example data, is displayed in Table 9-4.

Table 9-4: Periodic Snapshot View

MONTHNR	ACCOUNTNR	CUSTOMER VERSION_KEY	BALANCE
200906	3200354	5	$ 100
200907	3200354	8	$ 200

Loading Periodic Snapshots

There is nothing inherently complex about loading periodic snapshots, but there are some issues involved with preparing and getting the data. First, a periodic snapshot is like a photo taken at a specific point in time. If you translate this to the ETL and data warehouse world, this means that you need to get the state of the data at an exact point in time, and this point in time should be exactly the same for each period. In situations where periodic snapshots are the rule instead of the exception, this is usually not a big

problem. For instance, in a banking environment, periodic closings are already a standard procedure and the data won't change between two closing periods. As a result, it doesn't matter if you load the data at exactly 00:00 hr of the last day of the month, or a couple of hours or even days later.

Something else is in play here as well: The calculation of the monthly account balances can be an extremely complex process and is better left to the operational systems that are designed to support this. Trying to reproduce these calculations in an ETL process is a daunting task that will likely fail or at least cost a huge amount of time and money. Yes, even with Kettle!

The loading of the periodic snapshot itself can be done by using a simple bulk load using the `append` option because all you do is add an additional set of rows to an existing table. The only thing to take care of is a snapshot period ID to include in the fact rows.

Loading Accumulating Snapshots

Accumulating snapshot loading means using updates. For this reason, you might think that bulk loading is out of the question. But is it? Not necessarily, as we'll explain shortly. Chapter 4 showed you how to use an Insert / Update step for loading an accumulating fact table so we won't cover that here. The problem with using inserts and updates is that when the fact table gets huge, and there are many updates, processing the data might take a very long time. There is a very elegant way to avoid having to do insert/update statements on a large fact table, but it requires partitioning support in the database. However, you can achieve something similar with two tables and a view to access them as a single table.

Logically the solution looks like the diagram in Figure 9-13.

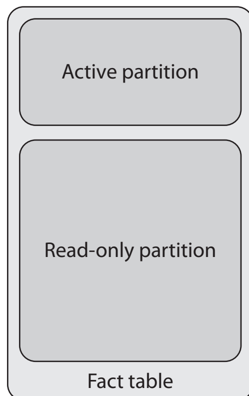


Figure 9-13: Partitioned Accumulated Snapshot Table

This works as follows:

- All the inserts and updates take place in the Active partition, which is relatively small.
- As soon as the accumulation is complete (all steps in the process have a final date and all other fact columns have their final values inserted as well), the record is deleted from the Active partition and moved to the Read-only part of the table.
- The latter can be set up as a periodic batch process to clean the Active partition from completed records. These records can then be appended to the read-only partition using a bulk loader.

Another solution that completely eliminates the use of insert/update on the database uses the same logical division of active and read-only records and works as follows:

- Read all active records into a staging area and truncate/delete the Active partition.
- Retrieve new and changed fact records from the source system and merge these with the records from the previous step.
- Load the incomplete records into the Active partition and the completed records in the Read only partition.

This second approach eliminates the need for a periodic cleaning process because the records are already assigned to the correct partition by the load process. It also eliminates the use of DML statements, which means that this approach can also be used with a product like the community edition of InfoBright, which doesn't allow for DML operations at all.

NOTE **Not all databases support truncating a partition; some require you to delete and recreate the partition to delete all the data from it.**

Loading State-Oriented Fact Tables

You might wonder whether a state-oriented fact table (SOF) is actually a real and different type of fact table because it has many similarities with both dimension and periodic/accumulating snapshot tables. This might be the first impression, but there are some notable differences:

- A SOF table contains measures, and dimension tables don't.
- A SOF table contains foreign keys to dimension tables so in a pure star schema model, it has to be a fact table; otherwise it would be a snowflake.
- The number of states in a SOF table is unknown beforehand; in an accumulating snapshot, all states are hard-wired in the table.

There are some similarities as well: The measures are semi-additive (cannot be summarized over time), just like in a periodic snapshot table. The records have a `valid_from` and `valid_to` date, just like dimension tables. And, just like in a dimension table, a `current_flag` could be added to indicate the current active record. Furthermore, the

technical design of the table could benefit from a surrogate key because this enables you to avoid having to define a multi-column primary key. This leads us to the conclusion that we have all the ingredients available to efficiently load SOF tables using Kettle! A sample of the workflow needed is displayed in Figure 9-14.

Figure 9-14: Example SOF loading workflow

The only difference with other fact table workloads is that we use a "Dimension lookup/update" step for the final SOF insert/update operation. And unfortunately, there's no way around this because there's generally no way of telling that the "end state" is reached. Exceptions are, of course, the already mentioned fire insurance policy, which could expire or be terminated, or a bank account that's closed. The larger part of the data will, however, be apt to change because we're dealing with long-running contracts and infrequent changes to them.

Loading Aggregate Tables

Aggregate tables add an extra layer of complexity and overhead to an ETL solution and the data warehouse but can offer big performance advantages. An aggregate table contains summary data at a higher grouping level. When a query tool or OLAP engine is capable of using these aggregate tables, it doesn't have to go through all details of a fact table but can retrieve the summarized data directly. Consider the following: the fact table in the example data warehouse in Chapter 4, `fact_rental`, contains 16,044 records. The lowest level of detail in this table is the individual movie in a rental, which means that when an overview of the rentals per customer country per year is required, all 16,044 records need to be scanned. Indexing can, of course, speed up this process. However, when an aggregate table is added with only the number of rentals per year per customer country, this table contains only 169 rows, which is an almost 100-fold decrease in number of records to evaluate.

Basically there are three ways to load aggregate tables:

- Extend the fact load transformation with extra steps to look up the required summary fields `customer_country` and `year4`, and then use a "Memory Group by" step to create the aggregate records and a "Table output" step or one of the bulk loaders to write the data to the aggregate table.
- Use a "Table input" step with the following SQL to retrieve the data and load it directly in an aggregate table:

```
SELECT     c.customer_country, d.year4,
           SUM(count_rentals) as count_rentals
FROM       fact_rental  f
INNER JOIN dim_customer c ON c.customer_key = f.customer_key
INNER JOIN dim_date     d ON d.date_key = f.rental_date_key
GROUP BY   c.customer_country, d.year4
```

- Use an "Execute SQL script" step with an extended version of the preceding SQL. The first line of the script will then read something like `insert into agg_table(customer_country, year4, count_rentals` (in case the table already exists), or `create table agg_table as` (in case you want to create it on the fly, but make sure to drop it first if it's already there). The syntax of these statements will of course depend on the SQL dialect of your database. The statements used here are for MySQL. The advantage of using this approach is that the statement is completely executed inside the database without the need to process it within Kettle.

Within a Pentaho solution, the most obvious use for aggregate tables is within a Mondrian schema. It is beyond the scope of this book to elaborate on this, but our book *Pentaho Solutions* explains how to use the Mondrian Aggregate Designer to create aggregate tables and how to augment the Mondrian schema to use them.

Summary

This chapter covered the various load options and fact table variations using Kettle. The first section covered the bulk-loading options available in Kettle and explained the difference between file- and API-based bulk loading. In between those two options is the use of named pipes. We illustrated this concept using a simple example with basic Linux commands. The first part of the chapter also covered the bulk loaders for MySQL, LucidDB, Oracle, and PostgreSQL.

One of the most important aspects of preparing fact data for loading is performing the dimension lookups, also described as the *surrogate key pipeline*. We covered both database and stream lookups, and provided an in-depth explanation of how to handle late-arriving dimension and fact data.

The last part of the chapter explained the different types of fact tables and the issues involved with loading and updating the data. The following well-known fact table types were covered, including an explanation of how to load data for each of them using Kettle:

- Transactional fact tables
- Periodic snapshot tables
- Accumulating snapshot tables

Next, we introduced a new type of fact table first described by Dr. Harm van der Lek, the *state-oriented fact table*, and explained how Kettle supports this type of fact table out-of-the-box. We concluded the chapter with a brief coverage of the different approaches that are available to load aggregate tables.

What we didn't cover in this chapter are all the performance options available in Kettle to handle large data volumes and speed up the transformation and load process. These options are covered in Part IV of this book where performance tuning, parallelization, clustering, and partitioning are described in depth.

CHAPTER 10

Working with OLAP Data

OLAP, or Online Analytical Processing, has a special position as subsystem 20 in the 34 ETL subsystems. Although handling OLAP data stores is only one of the 34 ETL subsystems, the topic is so important that we have devoted this entire chapter to it.

The term *OLAP* was introduced in 1993 by database legend E. F. (Ted) Codd, who came up with 12 rules for defining OLAP. The rules are nicely summarized at `http://www.olap.com/w/index.php/Codd's_Paper`.

The most important notion in Codd's definition is the multi-dimensional nature of OLAP data—the two terms, OLAP and multi-dimensional, are now used almost as synonyms. It's not that Codd invented OLAP, but he did give it a name, and since then a multi-billion–dollar industry emerged around this concept. The first OLAP products already existed when Codd came up with his rules, with Cognos Powerplay and Arbor Essbase probably being the most familiar. These products still exist today, Powerplay as part of the IBM-Cognos BI offering, and Essbase as part of the Oracle BI offering. The biggest player in the OLAP market, however, started its life in Israel under the wings of a small software company called Panorama. In 1996, the company sold its OLAP server technology to Microsoft and the rest is history, as Microsoft Analysis Services is now the OLAP solution with the biggest overall market share and the largest number of production deployments. Microsoft did some other good things for the OLAP market as well: it created the multi-dimensional query language called Multi Dimensional eXpressions, or MDX for short. MDX is now the de-facto standard multi-dimensional query language and a very powerful way to express analytical queries.

This chapter will give you enough background information on OLAP technology, configuration, and challenges but it is not an introduction course in MDX. If you're not

familiar with MDX you can still benefit from this chapter because we'll be covering an OLAP database that lets you read and write data without using MDX queries.

NOTE **If you need a quick introduction in OLAP and MDX, you might want to start with a look at Chapter 15 of our earlier book, *Pentaho Solutions*. For in-depth coverage of the MDX query language, several books and online resources are available, three of which we'd like to mention here:**

- **The "MDX Essentials" series on *Database Journal*, by William Pearson:** `http://www.databasejournal.com/features/mssql/article.php/10894_1495511_1/MDX-at-First-Glance-Introduction-to-SQL-Server-MDX-Essentials.htm`
- ***MDX Solutions: With Microsoft SQL Server Analysis Services 2005 and Hyperion Essbase*, by G. Spofford, S. Harinath, C. Webb, D. Hai Huang, and F. Civardi (Wiley, 2006).**
- ***Professional Microsoft SQL Server Analysis Services 2008 with MDX*, by S. Harinath, M. Carroll, S. Meenakshisundaram, R. Zare, and D. Lee (Wrox, 2009).**

OLAP Benefits and Challenges

What's so special about OLAP? Remember that usually we're dealing with data in rows and columns, whether that data is in a flat file, an Excel sheet, or a database table. In an OLAP cube, there is no such thing as a row or a column, only *dimensions*, *hierarchies*, and *cells*. In order to read data from or write data to a cube, you need to know where the data is located. This location is indicated by the intersection of the dimensions involved. First, let's take a look at a typical cube, such as the one on the left in Figure 10-1.

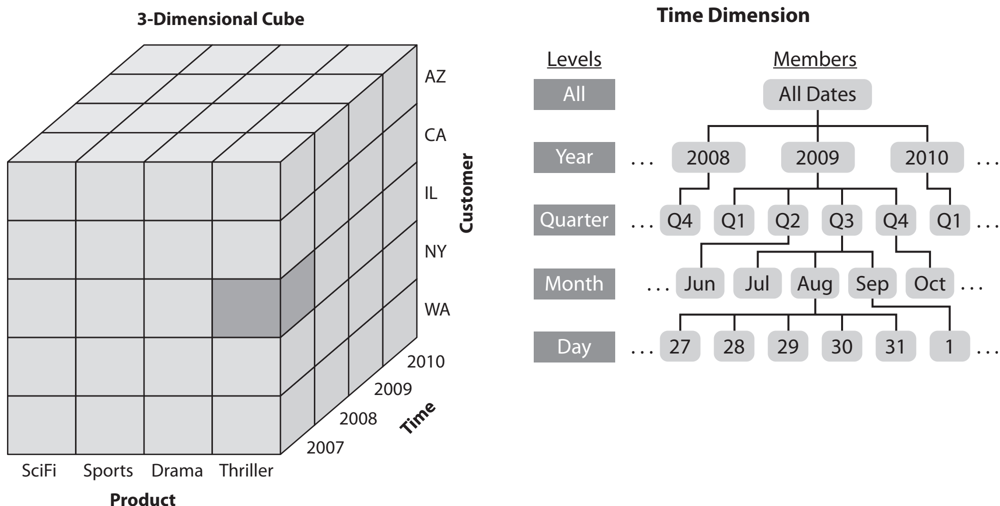


Figure 10-1: 3-dimensional cube and time dimension

Figure 10-1 shows a typical example of an OLAP cube consisting of the dimensions Time, Product, and Customer. The darker colored cell is the intersection of all days in the year 2007, the product category "Thriller," and all customers in the state of Illinois. So if the value in this cell is called Revenue, we'd have the revenue for the Thriller category for all customers from Illinois in the year 2007. This is a highly aggregated value, however, as you can tell from the time dimension diagram on the right in Figure 10-1. A year consists of quarters, quarters consist of months, and months consist of days. This is called a *hierarchy*, and a dimension can contain multiple hierarchies, or ways to roll up values. The same breakdown shown for the time dimension is probably also available for customer (state ➪ city ➪ area ➪ customer) and product (category ➪ group ➪ product).

The value of using an OLAP cube for analysis is not necessarily delivered by this multi-dimensionality per se, but if you add the notion of pre-aggregation to it, you'll get an idea of what's happening. Revenue in an OLAP cube is not only available at the lower levels in the cube cells (e.g. revenue for a specific customer and a specific day), but is also pre-calculated at the higher levels as well. This means that an OLAP cube is ideally suited for interactive, ad-hoc analysis of data. The real drivers for the success of OLAP tools were the following:

- **Unprecedented analysis performance:** The precalculated and aggregated summary cells provide immediate response times.
- **Ability to drill down or drill up:** Move up and down through the hierarchies to go to a more detailed or more grouped level.
- **Ability to slice and dice:** Select other dimensions, select a specific value on-the-fly, or switch row and column entries.
- **The fact that multi-dimensional models are intuitive:** Most people easily grasp the concept of analyzing data using dimensions with hierarchies, which is also why the dimensional model for relational data warehouses became so popular.

These benefits, plus the very disruptive pricing model of Microsoft's Analysis Services OLAP server, pushed a broad adoption of OLAP tools and technologies. Several OLAP servers also allow users to *write back* data or create different versions of specific *slices*, (e.g., budget A, budget B), which makes these databases perfect for planning, budgeting, or forecasting.

So far so good; but what about integrating data from an OLAP cube? That's where the challenges of OLAP start. Unlike analysis tools, ETL tools aren't designed to work with multi-dimensional, hierarchically stored data. Instead, we need to transform the data structure into something that's row- and column-based in order to work with it. We need a way to "flatten" the cube. Another challenge is getting the data at the aggregation level needed. MDX lets you do all those things as we'll show later, but just getting the data out of the cube is probably only the start of the process. In most cases, the data needs to be pivoted or unpivoted to conform the structure to other data sources we're processing. Last but not least, every OLAP data provider returns data in its own specific way, so having solved a problem for Mondrian doesn't necessarily mean that you've solved the same problem for Microsoft Analysis Services as well.

OLAP Storage Types

In order to work with OLAP data stores, it's important to know the different types of OLAP storage that exist. Basically there are two distinct types and a third somewhere in between the first two:

- **MOLAP:** Multi-dimensional OLAP where all the data resides in the OLAP solution, both for storage and for analysis. Examples are Jedox PALO, IBM-Cognos Powerplay or TM1, and (to a large extent) Microsoft Analysis Services.
- **ROLAP:** Relational OLAP where the data resides in the underlying relational database and the OLAP engine serves only as a caching and MDX to SQL translation mechanism. Examples are Pentaho Mondrian and (also to a large extent) Microstrategy.
- **HOLAP:** Hybrid OLAP where the detail records reside in a relational database and the summary information in the OLAP database. Examples are actually all of the above-mentioned products. Mondrian has the ability to cache data in memory, effectively acting like a HOLAP solution then. Other products such as Microsoft Analysis Services let you decide what amount of data should be pre-aggregated in the cube and what data can remain in the underlying detail database.

These differences in approach also have an effect on the available ETL options. For MOLAP stores, you'll have to use MDX queries, unless there is another way of getting at the data. This is slightly different for ROLAP stores. Because the data for a ROLAP solution is also available in a relational database, you could in theory extract the data directly from there using SQL instead of using MDX statements fired against the OLAP store. This isn't always practical or achievable; ROLAP is a technology used for very large data warehouses and accessing the data directly by circumventing the ROLAP engine might cause severe performance issues. Furthermore, the database that acts as a ROLAP foundation usually contains multiple predefined aggregate tables to help speed up the OLAP performance. If the ETL developer is not aware of these aggregate tables, the ETL process could place an undesirable heavy burden on the data warehouse. The conclusion here must be that for extracting data from an OLAP cube, whether ROLAP, MOLAP, or HOLAP, the safest and fastest way is to query the cube using MDX and not try to get the data from a relational data store.

Writing data is a different story; not all OLAP engines allow for write back operations or are capable of handling something like an insert statement. For Mondrian, this means that if there is new data, it must be inserted or updated in the underlying database. After doing that, the cache must be rebuilt in order to actually see the new data in an analysis front end.

Positioning OLAP

In most situations, an OLAP database will be used as a data mart; sometimes organizations use the OLAP solution as the corporate data warehouse but this is the exception, not the rule. A typical layered architecture using OLAP technology is displayed in Figure 10-2.

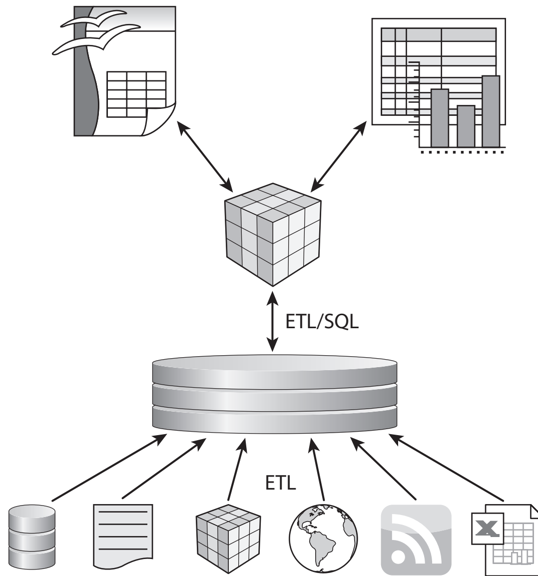


Figure 10-2: Positioning OLAP

Figure 10-2 shows, from bottom to top:

- Various data sources that are loaded and transformed using an ETL process
- The central data warehouse
- The OLAP solution (visualized as a single cube but usually there are more than one)
- Front-end tools for analyzing, visualizing, and modifying the data in the cube

In the diagram, you'll immediately spot the fact that there's also a cube displayed as one of the possible data sources. There's nothing wrong with that; as we've mentioned before, OLAP technology is an ideal solution for planning and budgeting. Many organizations rely on Excel for those purposes, but then lack the centralized management and security options offered by OLAP tools such as Palo. So in these cases, the OLAP cube serves two purposes: as a centrally managed datastore that can be used to enter or manipulate information, and as a fast analysis database. Both roles are depicted in Figure 10-2, including the different front ends needed to work with the cubes in these different roles. Products like Palo offer add-ins for Microsoft Excel and Open Office Calc to enable users to work with the multi-dimensional database using a familiar front end.

Kettle OLAP Options

In addition to being able to access nearly every relational database on the planet, Kettle can now also extract information from nearly every available OLAP database and even write data to some of them. The following three options are available:

- **Mondrian Input step:** Allows Kettle to retrieve data from a Mondrian cube using MDX statements. There is no special connection needed because this step uses

the same connection as the Mondrian schema itself. In addition to the connection, the Mondrian schema is required to be able to tell Kettle how the cube is structured.

- **OLAP Input step:** A generic plugin that allows Kettle to communicate with all XML/A compliant OLAP servers. Nearly all OLAP products support this standard, making this a very powerful option. The magical stick that enables this has two components: XML/A and Olap4J. The latter is a generic open source library developed to communicate to OLAP servers from a Java environment using MDX statements. XML/A allows a client to communicate with an OLAP server over HTTP using SOAP services and is not limited to reading data, but can be used to write data or fire processing tasks as well.
- **Palo add-in:** Perhaps the most versatile and easy-to-use option for working with OLAP data. The add-in contains four steps to read or write data, update or refresh dimension information, and create new dimensions in a cube.

The following sections cover these three options using the sample databases that are installed with the different products. This means that there's very little setup work to follow along with the examples if you have a running installation of Mondrian, Microsoft SQL Server Analysis Services, or Palo.

Working with Mondrian

Mondrian is the default OLAP (or actually, ROLAP) server in any open source BI solution because Pentaho, Jaspersoft, and SpagoBI all base their analytic solution on Mondrian. As a result, the chances that you'll run into a Mondrian cube are increasing every day. To date, however, Mondrian does not have writeback capabilities, so updating data in a Mondrian database is still a matter of updating the data warehouse itself and refreshing the Mondrian cache. Getting data out of Mondrian is something else; it's extremely easy, as we'll show here.

In order to run the following examples on your own computer, we assume you have a Mondrian server up and running. The Foodmart sample database is used because that is part of a sample Mondrian installation. If you don't have Mondrian or the Foodmart database set up but would like to do so, just follow the instructions at `http://mondrian.pentaho.org/documentation/installation.php`. The first step in a transformation using Mondrian data is, of course, the Mondrian Input step. As you can see in the online documentation (`http://wiki.pentaho.com/display/EAI/Mondrian+Input`), the step comes with an example query included. The example query uses the Sales cube from the Foodmart schema and we'll do the same here. There are three essential elements for the Mondrian Input step to work:

- The **Connection**, which is a normal database connection to the relational database that Mondrian uses to get its data from
- The **Catalog location**, which is the schema file created with the Schema workbench
- The **MDX Query** to define the result set and retrieve the data

Figure 10-3 shows the completed input step for this example.

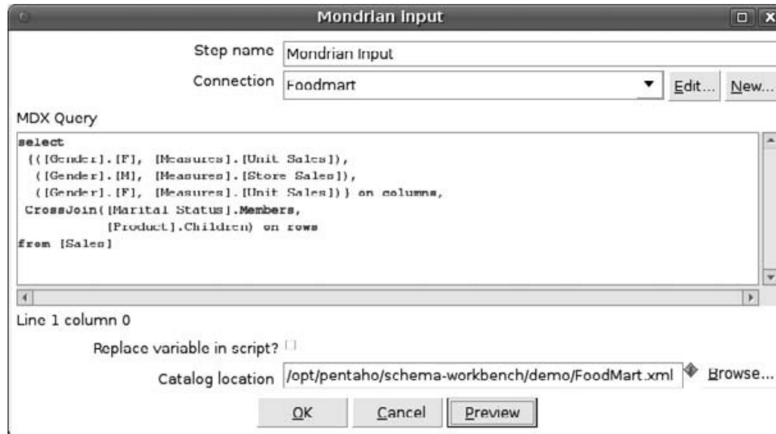


Figure 10-3: Mondrian Input step

You can immediately view the resulting output by selecting the Preview option of the step, which will show the output as displayed in Figure 10-4.

Examine preview data

Rows of step: Mondrian Input (24 rows)

#	[Time]	[Store Type]	[Measures].[Unit Sales]	[Measures].[Store Sales]
1	[Time].[1997].[Q1]	[Store Type].[All Store Types].[Deluxe Supermarket]	20977.0	43864.84
2	[Time].[1997].[Q1]	[Store Type].[All Store Types].[Gourmet Supermarket]	3822.0	8203.89
3	[Time].[1997].[Q1]	[Store Type].[All Store Types].[HeadQuarters]		
4	[Time].[1997].[Q1]	[Store Type].[All Store Types].[Mid-Size Grocery]	3096.0	6439.32
5	[Time].[1997].[Q1]	[Store Type].[All Store Types].[Small Grocery]	1457.0	3037.78
6	[Time].[1997].[Q1]	[Store Type].[All Store Types].[Supermarket]	36939.0	78082.52
7	[Time].[1997].[Q2]	[Store Type].[All Store Types].[Deluxe Supermarket]	16371.0	34486.06
8	[Time].[1997].[Q2]	[Store Type].[All Store Types].[Gourmet Supermarket]	5837.0	12597.15

Close

Figure 10-4: Mondrian output preview

You can probably see right away that this output is not really useful for reporting purposes. The column names are the least of your problem here because a few other things need further processing. You should:

- Clean up all the brackets.
- Split the Year and Quarter fields.
- Extract the Store Type.

As with many things in Kettle, you can accomplish this in several ways. One option, shown in Figure 10-5, doesn't require any coding and consists of a series of steps to clean up the result set.

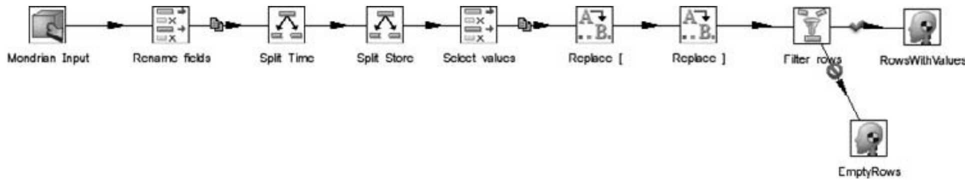


Figure 10-5: Completed Mondrian transformation

The first step is just to rename the columns in order to ease working with them downstream. The two "Split field" steps separate the values in a single field based on a split character. New field names can then be defined for the results, as shown in Figure 10-6 where the Split Time step is displayed.

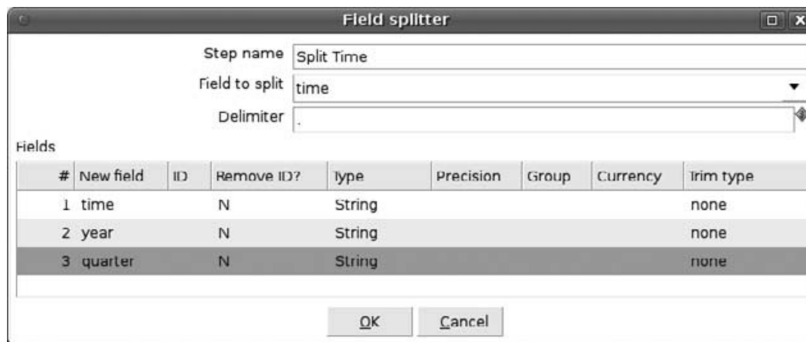


Figure 10-6: Split Time step

You split the Store field with the same step type. You can, of course, use other options to get the same result. With a "Strings cut" step, you could take the fields and just get a substring from the field values. This is, however, quite risky: It might work now, but if someone decides to rename the `All Store Types` level to just `All Types`, you're in trouble. The Split Fields step is therefore a more robust solution here. The subsequent "Select values" step enables you to get rid of the superfluous columns, and the two "Replace in string" steps take care of removing the two brackets from the field value. For completeness, we added a "Filter rows" step because there is no revenue information for the store type HeadQuarters. You can find the completed transformation in the Chapter 10 examples on the book's companion site at `www.wiley.com/go/kettlesolutions`.

Alternatively, you can solve everything in a single Java Script step. In addition to the fact that you don't really need to write code here, the JavaScript option would require constructs such as `var year = substr(time,indexOf(time,".")+2,4);`, just to

get the year value from the time string. Getting the Quarter value (Q1, Q2, etc.) from the Time field needs a nested `indexOf` to find the second dot in the string, which makes matters even worse. Besides, why should you? The results obtained using the regular steps are displayed in Figure 10-7, and this is exactly the output we wanted.

Examine preview data

Rows of step: RowsWithValues (20 rows)

#	year	quarter	store_type	unit_sales	store_sales
1	1997	Q1	Deluxe Supermarket	20977.0	43864.84
2	1997	Q1	Gourmet Supermarket	3822.0	8203.89
3	1997	Q1	Mid-Size Grocery	3096.0	6439.32
4	1997	Q1	Small Grocery	1457.0	3037.78
5	1997	Q1	Supermarket	36939.0	78082.52
6	1997	Q2	Deluxe Supermarket	16371.0	34486.06
7	1997	Q2	Gourmet Supermarket	5837.0	12597.15

Close

Figure 10-7: Cleaned Mondrian output

For Mondrian, there is another option as well. As you may know, Mondrian is also an XML/A-compliant server, meaning that you could access it with the OLAP Input step, which is the subject of the next section.

Working with XML/A Servers

XML/A is short for XML for Analysis, and is an industry-standard protocol to communicate with OLAP servers over HTTP. It defines a SOAP web service that allows clients to obtain metadata and to execute MDX (multi-dimensional expressions) queries. XML is used as the data exchange format. The SOAP (Simple Object Access Protocol) envelope still contains the actual query that is usually written in MDX (although the standard also allows for DMX and SQL statements). You can find more information about XML/A at `www.xmla.org`.

In order to work with XML/A servers, Kettle offers an OLAP Input step that uses Olap4J under the hood. The OLAP Input step, however, takes care of all the XML/A and Olap4J intricacies, so the only thing left to get the solution working is knowing how to enter the correct connection string and formulate a working MDX query. This opens up a world of opportunities because almost every OLAP analysis database, whether open or closed source, is accessible using XML/A. Examples are SAP BW, Oracle-Hyperion Essbase, IBM-Cognos TM/1, Mondrian, and Microsoft Analysis Services.

To use XML/A, you need the OLAP database enabled to accept requests over HTTP. For most products this is not something that's available out-of-the-box. The examples in this section are based on Microsoft SQL Server 2008 Analysis Services (MSAS) and the server used to access the OLAP database using XML/A needs some setup before it

can be used. It is beyond the scope of this book to explain in-depth how to configure XML/A access to MSAS, but here are a few steps and tips:

- For configuring HTTP access to SQL Server 2005 Analysis Services on Microsoft Windows Server 2003, go to `http://technet.microsoft.com/en-us/library/cc917711.aspx`.
- For the 2008 versions of both Analysis Services and Windows Server, an updated version is available on `http://bloggingabout.net/blogs/mglaser/archive/2008/08/15/configuring-http-access-to-sql-server-2008-analysis-services-on-microsoft-windows-server-2008.aspx`.
- For SQL Server 2008 running on Windows 2003, just use the first option.
- You can test whether you can connect to the MSAS instance over HTTP simply by typing the URL into the new application you just created. If you followed one of the previous examples, try typing `http://`*`<server address>`*`/olap/msmdpump.dll` where *`<server address>`* needs to be replaced by the actual name or IP address. A response like `XMLAnalysisError.0xc10e0002Parser: The syntax for 'GET' is incorrect` means that the HTTP access is working.
- One of the trickiest parts of implementing a solution like this is setting up the correct access rights for the catalogs and cubes; a useful guide for this is available at `http://www.activeinterface.com/b2008_12_29.html`.
- If you just want to play around to check out how XML/A works in combination with MSAS and you don't require a secure setup, you can enable Anonymous Access in the IIS security settings for the OLAP web application. Next, make sure that you add a role to the MSAS Database you want to access and assign the proper authorizations for the role. The role needs to have at least one (Windows) user added to it; if you just want to be able to access the cube, add the user that is used for the anonymous access to the role. Usually this will be the `IUSR_`*`<machine name>`* account, where *`<machine name>`* needs to be replaced with your own server name.

You can now try to access the OLAP server from Kettle. Start a new transformation and select the OLAP Input step from the toolbox. Enter the XML/A URL you created for the database; if you've used Anonymous Access, you can leave Username and Password empty. The last item needed for accessing the correct database is the Catalog name. This is the MSAS Database name as shown in SQL Server Management Studio, including spaces. Then you need to decide which data is needed from the OLAP cube; we used a fairly simple data set for this example with Year and Product Category on the rows, and Sales Amount and Gross Profit on the columns. Figure 10-8 shows the complete step to retrieve the data

It should now be possible to do a Preview of the result set, as shown in Figure 10-9.

WARNING **At the time of this writing, the OLAP Input step automatically triggers a** `process` **command forcing the cube to rebuild first before returning results, which takes several minutes each time.**

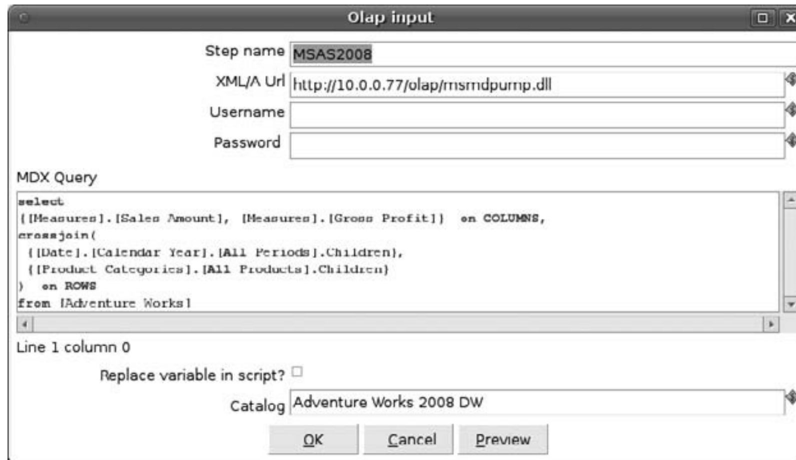


Figure 10-8: OLAP Input step

Examine preview data

Rows of step: MSAS2008 (20 rows)

#	Column0	Column1	[Sales Amount]	[Gross Profit]
1	CY 2001	Accessories	$20,235.36	$8,171.48
2		Bikes	$10,661,722.28	$1,580,237.72
3		Clothing	$34,376.34	($1,911.73)
4		Components	$615,474.98	$54,035.47
5	CY 2002	Accessories	$92,735.35	$28,351.25
6		Bikes	$26,486,358.20	$2,413,803.00
7		Clothing	$485,587.15	$102,113.92
8		Components	$3,610,092.47	$425,983.77
9	CY 2003	Accessories	$590,242.59	$283,673.27
10		Bikes	$34,910,877.69	$3,050,962.76
11		Clothing	$1,010,112.16	$154,607.64
12		Components	$5,482,497.29	$414,931.73
13	CY 2004	Accessories	$568,844.58	$314,271.16
14		Bikes	$22,561,568.03	$3,470,093.14
15		Clothing	$587,537.80	$114,026.17
16		Components	$2,091,011.92	$138,015.51
17	CY 2006	Accessories		
18		Bikes		

Close Show Log

Figure 10-9: XML/A query result set

If you run the same query directly in the Analysis Services query browser, you'll notice among other things that the periods haven't been deduplicated, as shown in Figure 10-10.

Messages | Results

		Sales Amount	Gross Profit
CY 2001	Accessories	$20.235.36	$8.171.48
CY 2001	Bikes	$10,661,722.28	$1,580,237.72
CY 2001	Clothing	$34,376.34	($1,911.73)
CY 2001	Components	$615,474.90	$54,005.47
CY 2002	Accessories	$92,735.35	$28,351.25
CY 2002	Bikes	$26,486,358.20	$2,413,803.00
CY 2002	Clothing	$485,587.15	$102,113.92
CY 2002	Components	$3,610,092.47	$425,903.77
CY 2003	Accessories	$590,242.59	$283,673.27
CY 2003	Bikes	$34.910.877.69	$3.050.962.76
CY 2003	Clothing	$1,010,112.16	$154,607.64
CY 2003	Components	$5,482,497.29	$414,931.73
CY 2004	Accessories	$568,844.58	$314,271.16
CY 2004	Bikes	$22.561.568.03	$3.470.093.14
CY 2004	Clothing	[illegible]	[illegible]

Figure 10-10: MSAS query result set

This would make further processing in Kettle a lot easier, so in order to use this data you need a few extra steps of pre-processing. As you can see, the repeating values from the cross join have been deleted from the result set. Because there is no standard step available that takes care of this for you, you need to use an alternative resolution, which is relatively simple to create. Before we begin the explanation, take a look at Figure 10-9 again and decide what you need to do to clean up the data for further processing:

- Rename the fields to get more meaningful descriptions than `Column0` and `Column1`.
- Remove the brackets from the field names.
- Remove the $ sign.
- Replace the opening parentheses with a minus sign.
- Remove the closing parentheses.
- Fill out the empty year values.

This looks like a complicated transformation but you actually need only two steps to accomplish this. (We could have used only one, but this solution makes the code easier to read and maintain.)

The first two bullet points are the easy ones: Just add a "Rename fields" step and rename all the fields. The remainder of the list requires the following little bit of JavaScript (part of the `MSAS_Example.ktr` sample file from the downloads for this chapter):

```
var prevyear;

if (gross_profit != null)  ¬
var gross_profit = gross_profit.replace("($","-").
replace("$", "").replace(")", "");

if (sales_amount != null) ¬
var sales_amount = sales_amount.replace("($","-").
replace("$", "").replace(")", "");
```

```
if (year != null) var year = year.replace("CY ","");

if (year != null)
 {
    prevyear = year;

 }
else
 {
   year = prevyear;
 }
```

The code starts by providing a placeholder variable (`prevyear`) to store the previous year value. The next three lines replace parts of the field values using the `string.replace(search_string, replace_string)` function. As you can see, you can add multiple replace calls to a single field, making this a very flexible way of transforming string values. The last part might need some more explanation.

The thing to remember here is that the JavaScript step works row by row. This means that you can set a variable and as long as the condition to change it doesn't change, the value of the variable doesn't need to change either. Be careful, however: In this case, we knew what our input data looked like, and also knew that the first row contained a value for `year`. Based on that knowledge of the incoming data and the sort order, we can assign the value of the `year` field to the variable `prevyear` because the condition `year != null` will be satisfied at the first row. When the second row is processed, the condition `year != null` fails (the field is empty) so the `else` branch is executed. Because the `prevyear` variable didn't change and still holds the value `2001`, this value is assigned to the field `year`. The same occurs with the two following rows. Row number 5 again contains a year value, which is then assigned to the variable `prevyear`, and so on until all rows have been processed. If you retrieve more columns from the OLAP Input step, resulting in more fields with empty values, you can still use this approach to fill out all the field values. Figure 10-11 shows the cleaned output, ready to be used in the data warehouse.

Examine preview data

Rows of step: Fill years _clean amounts (20 rows)

#	year	article_group	gross_profit	sales_amount
1	2001	Accessories	8,171.48	20,235.36
2	2001	Bikes	1,580,237.72	10,661,722.28
3	2001	Clothing	-1,911.73	34,376.34
4	2001	Components	54,035.47	615,474.98
5	2002	Accessories	28,351.25	92,735.35
6	2002	Bikes	2,413,803.00	26,486,358.20
7	2002	Clothing	102,113.92	485,587.15
8	2002	Components	425,983.77	3,610,092.47

Close

Figure 10-11: MSAS cleaned results

Using XML/A and the OLAP Input step is probably the most flexible solution for this example and also the one to standardize on. Almost every OLAP database can be accessed this way, which means that you can use one standard way of transforming result sets obtained from these servers. As always, standardization is a good thing because it helps you build ETL solutions that are easier to maintain.

Working with Palo

Palo is the open source multi-dimensional in-memory database developed by German software firm Jedox (`www.jedox.com`). (If you're wondering about the origins of the product's name, just reverse the name and you'll know.) We assume here that you have a Palo server running; to get access to it from another machine, read the following tip. For installation and configuration instructions, just visit the Jedox website and download the open source version of Palo.

TIP **To make the Palo server accessible from another machine, you'll need to add an `http` entry in the `palo.ini` file that's accessible in the Palo Data directory. The default is `http "127.0.0.1" 7777`, which means the server listens for local connections only. You can change this entry in the machine's IP address or simply add another entry. If you use multiple entries, make sure to use different port numbers. For instance, the `palo.ini` file on the server we use has the following entries:**

```
http "127.0.0.1" 7777
http "10.0.0.71" 7778
```

The samples in this chapter have been created using Palo version 3.1 and the accompanying Demo database that was released in April 2010. There are several options to browse the Palo cubes. The default *Palo for Excel* installation for Windows contains the Excel add-in that's needed to work with Palo cubes. If you're a Windows or Linux user working with OpenOffice, you can use the PalOOCa add-in provided by Tensegrity Software (`http://www.jpalo.com/en/products/palo_open_office.html`). Figure 10-12 shows the Palo Modeler that's part of the Open Office add-in.

In the figure, the Demo database with the Products dimension is opened, which also shows that this dimension has a hierarchy with three levels: All Products, Product Group, and Product. This information is needed later when working from Kettle because there is no explore functionality for Palo cubes in Kettle.

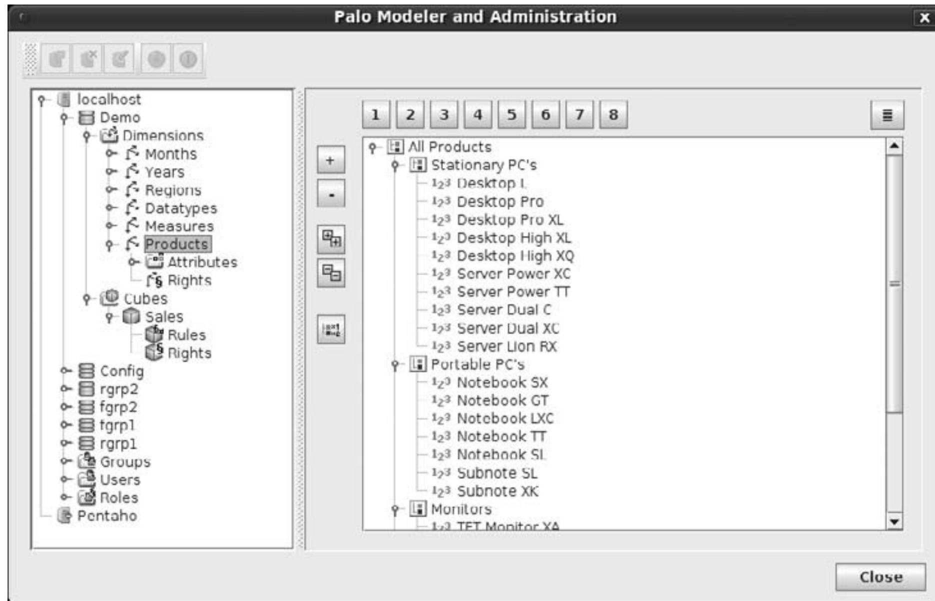


Figure 10-12: Palo Modeler with Demo database

Setting Up the Palo Connection

The first step is getting the Palo client connection software, which is developed by jPalo (`www.jpalo.com`). The client is called Palo Java API and can be obtained by either going to the download and filling in your e-mail address and getting a download link, or by downloading the file directly from `http://www.jpalo.com/en/products/start-download.php?product=API&lang=en`.

The jPalo software can also be obtained from SourceForge (`http://sourceforge.net/projects/jpalo/`) but that repository usually contains a slightly older version. And, of course, you can always download the latest source code version from *trunk* and compile it yourself.

NOTE **The term *trunk* refers to the last version of a product in development. For more information see** `http://en.wikipedia.org/wiki/Trunk_(software)`**.**

After the download, unpack the `.zip` file and copy the `jpalo.jar` file from the `lib` directory to the Kettle `libext` directory.

The Palo steps can now be used to connect to a Palo cube. The following steps are available, two in the Input and two in the Output step folder:

- Palo Cells Input
- Palo Dimension Input
- Palo Cells Output
- Palo Dimension Output

Before moving on, it's a good idea to check whether you can make a connection to the Palo server. If no job or transformation is opened yet, just create a new one or open one of the existing jobs/transformations. Open the View tab of Explorer and right-click on Database connections to create a new one. You can either use the wizard (also available from the Wizard menu options) or jump right to the Database Connection screen. The host name, database name, port number, user name, and password need to be filled in. If you used the default install on a local machine, the values for these settings are as follows:

Host name:	localhost
Database Name:	Demo
Port Number:	7777
User Name:	admin
Password:	admin

When you click on Test, Kettle should display a message telling you that the connection succeeded. If not, check your connection, or your firewall settings if you're running the server on a different machine.

Palo Architecture

Before starting to read data from or write data to a Palo cube, you need a little background about the inner workings of the database. Although Palo is an in-memory OLAP database, this doesn't mean that there is no persistency layer available. The data is stored on disk, just like with any other database. There is some similarity between Palo and Mondrian in the sense that both need to read the data from disk into their memory cache when they start up, but that's about where the similarity ends. The Mondrian cache is partial, whereas the Palo database is fully loaded into memory. The Palo data on disk is stored in plain-text files, as shown in Figure 10-13. The figure shows a default Linux install where the data is stored in a subdirectory of the Palo server (`ps`), and displays the first part of the database definition file.

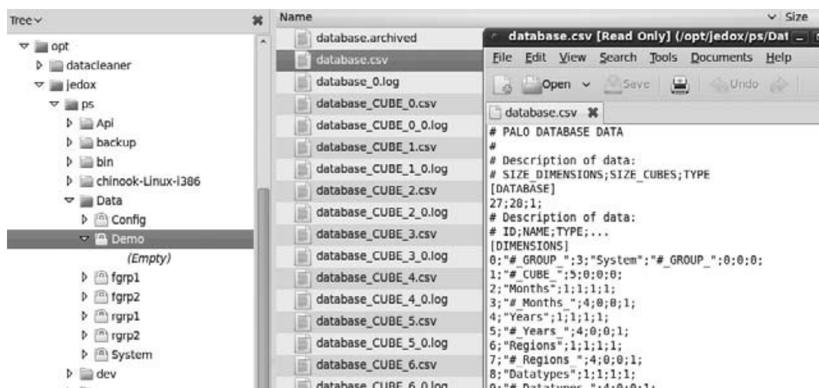


Figure 10-13: Palo Data files

The fact that the database is just a plain folder with text files also means that making backups or database copies is very simple. Stop the Palo server, compress or save the directory, and move it to wherever you want to store it. If you want to move a database to another Palo server, just stop the Palo service on the destination machine, place the complete directory in the Data folder, and start the service again. The copied database is then immediately available from the Modeler.

There are a few features that are specific to Palo and might help in understanding the best way to use Kettle to read or write data. First, there are no predefined dimensions or dimension types. Most OLAP servers make a distinction between a regular and a time dimension, and also treat measures differently. Palo doesn't make this distinction: Every dimension has the same options. Palo also has only two data types for an element: `String` and `Number` (an element is the basic building block of the cubes and forms the lowest detail level in a dimension). When retrieving data from a Palo cube using Kettle, you'll have to specify whether the content for the elements you're reading or writing are of type String or Number but be careful here: In Palo, you'll see that, for instance, the Month elements are of type Number, but when you specify Number as the data type in the Palo Dimension input step in Kettle, you'll get a conversion error. Again, no dates, so creating a date hierarchy can be done in every imaginable way. If you take a look at the date dimension, you'll see that there is a separate Year and Month dimension, but it could have been a single dimension as well. Using a separate Year and Month dimension enables you to create crosstabs with Year on the X axis, and Month on the Y axis, which you cannot do with a single time dimension.

A Palo database can also contain *attributes*. An attribute can be used to store an extra piece of information for a certain element. The Demo database, for instance, contains the attribute Price per Unit in the Product dimension. This means that for each product, the price can be stored. To date, there is no way for Kettle to get to this information directly. If you need to retrieve this information, you can use the Palo add-in to paste the values in a spreadsheet and subsequently read the data using a Kettle transformation.

The last feature to be aware of is Palo's fine-grained authorization. Each user belongs to a group, and each group can have one or more roles attached to it. Roles determine what a user can do on the server (create other users, delete cubes, define rules, and so on); groups are used to determine the access level for the data itself. If you can connect to the database using the admin account, this won't be a problem, but in a production situation this is rarely the case so you need to be aware of possible limitations for the account information you use for ETL access.

Reading Palo Data

The two input steps for Palo data in Kettle are Palo Cells Input and Palo Dimension Input. The former will get you the detailed data at the lowest levels of the Palo cube (the facts); the latter retrieves the corresponding dimension information, but only for one dimension at a time. If you want to retrieve the complete cube and all the dimensions and hierarchies, you'll need to create a Dimension input step for each dimension in the cube, plus a Cells input step to retrieve the cube content. Let's start by retrieving the Product information using a Palo Dimension Input step so we can show what it does.

Create a new transformation and drag a Palo Dimension Input step to the canvas. Select the connection you created earlier and in the Dimension drop-down list, select Products. Now you can retrieve the dimension level information using the Get Levels option. This will create three rows showing the Level Names Level 0, Level 1, and Level 2, with the corresponding level numbers. This is all the metadata Kettle can get from the Palo database. The Field values are the names you want to give to the columns in the Kettle transformation and need to be entered manually. The Type values need to be selected using the drop-down list. There are only two options: Number and String. Although these values correspond to the types found in the Palo cube, these are actually the types that Kettle uses internally to cast the data to. In this case, the Palo metadata lists all the product elements as type `Number` (see the gray "123" indicators in Figure 10-12), but in Kettle, they need to be defined as String because the descriptions contain alphanumeric data. Figure 10-14 shows the completed input step.

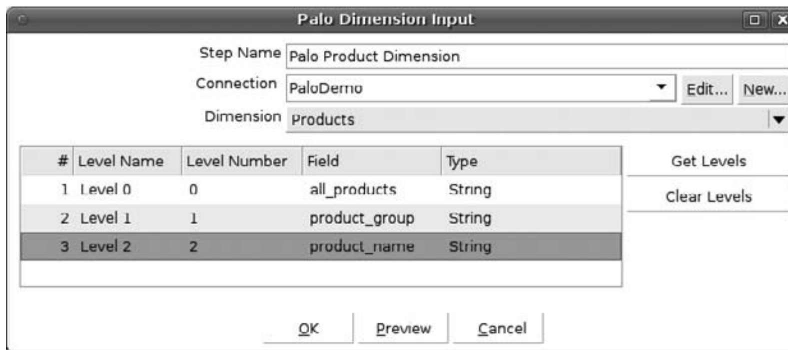


Figure 10-14: Palo Dimension Input configuration

Figure 10-15 displays the data preview, which is, in essence, a three-level dimension table now, but without any additional keys and dates yet.

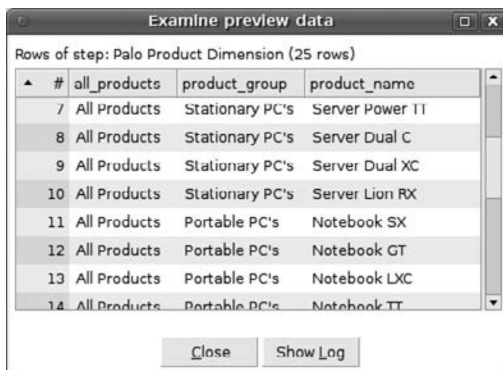


Figure 10-15: Palo Dimension Input preview

Given the fact that the product_name column is the dimension key, the only data that can change here is the content of the all_products or product_group columns in order to be able to recognize the changes with a Dimension lookup step. To transform this Palo Dimension Input step result into a relational product dimension, you could add a surrogate key using an "Add sequence" step, and other fields such as date_from, date_to, and is_current using an "Add constants" step.

The next task is loading the actual fact data from a Palo database. In order to do this, you need to drag a "Palo Cells Input" step to the canvas and specify which cube you want to extract data from. If you're using the admin user account, the Cube drop-down list will show all the system cubes as well, but we're only interested in the first one called Products. The next options in the screen need some explanation. As we explained in the chapter introduction, a cube contains values at various intersection points of dimension and dimension hierarchy levels. In Palo, the data is entered at the lowest cell level and aggregated at run-time. There is no way to retrieve data at an aggregated level using Kettle.

Ultimately, a numeric value will be available. In the case of the Sales cube in the Demo database, these are the values stored at the intersection of the six dimensions at the lowest detail level. These numeric values should be given a name because they don't have a specific name in Palo; that's what the Measure Name field is for. The Measure Type will be a Number, and with the Get Dimensions button you can retrieve the dimension information. You'll notice that the Field names are automatically copied from the Dimension names, but you can alter them if needed. As with the Dimension Input, you'll need to define the Dimension Type. String will always work, but if you're sure all the values in a dimension are numeric (such as Year in this case) you can use Number, too. Figure 10-16 shows the filled out Input step with the preview results next to it.

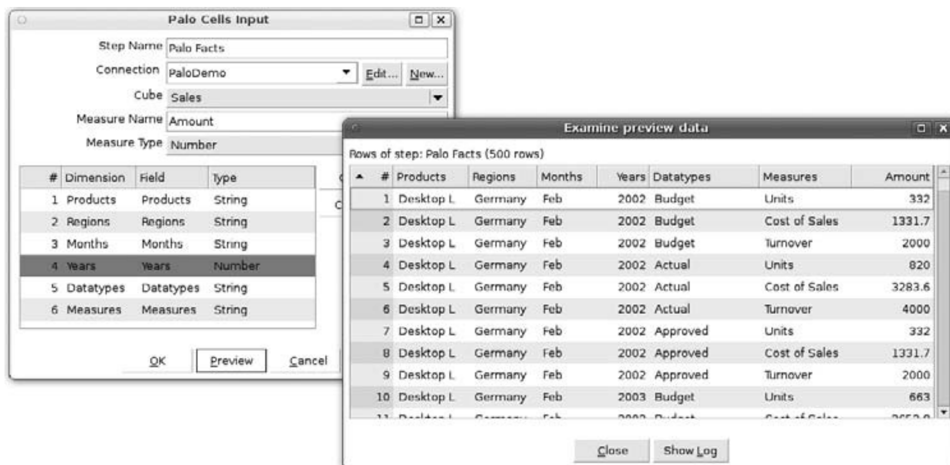


Figure 10-16: Palo Cells Input definition and preview

As you can see from the figure, only the lowest level dimension entries are retrieved (for instance, product Desktop L). In Figure 10-16, you can also see that this data cannot be used easily for reporting as it is. The Measures column containing the values Units,

Turnover, and Cost of Sales needs to be pivoted first so that for each of these measures a separate column is created. In Kettle terms, you need a "Row denormaliser" step, which will do the magic for you. A Denormaliser requires a key field (the *pivot* field), which in this case is Measures. The Group fields are all columns except the Amount (that will be used as the Value field) and the Measures because that's the key field. The Target fields need to be defined manually and you can name them whatever you want. The Value fieldname and Key value need to be typed in as well, so be careful to spell the names exactly as they are returned by the Palo Cells Input step. In Figure 10-17, you can see the completed Denormaliser step.

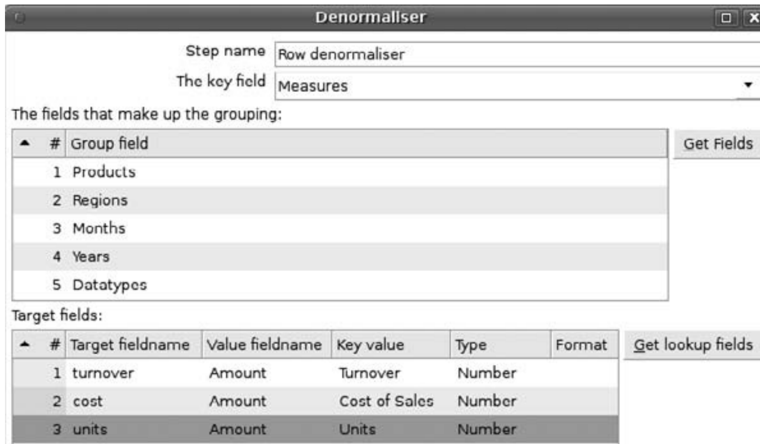


Figure 10-17: Palo data denormalised

Because all the data is already at the lowest level, you don't need to use aggregation here. To test whether this will get you the required results, you can do a preview for which the output is shown in Figure 10-18.

Examine preview data

Rows of step: Row denormaliser (100 rows)

#	Products	Regions	Months	Years	Datatypes	turnover	cost	units
1	Desktop L	Germany	Feb	2002	Budget	2000	1331.7	332
2	Desktop L	Germany	Feb	2002	Actual	4000	3283.6	820
3	Desktop L	Germany	Feb	2002	Approved	2000	1331.7	332
4	Desktop L	Germany	Feb	2003	Budget	3500	2652.8	663
5	Desktop L	Germany	Feb	2003	Actual	17500	17270.3	4317
6	Desktop L	Germany	Feb	2003	Approved	3500	2652.8	663
7	Desktop L	Germany	Feb	2004	Budget	21000	18441.6	4610
8	Desktop L	Germany	Feb	2004	Approved	21000	18441.6	4610
9	Desktop L	Germany	Feb	2005	Budget	31500	21865.8	5466
10	Desktop L	Germany	Feb	2005	Approved	31500	21865.8	5466

Close Stop Get more rows

Figure 10-18: Denormalised result set

This data can now be further processed using dimension lookups and loaded into a fact table. As explained before, Palo is an excellent tool for budgeting, planning, and analysis. Using Kettle, you now have the ability to retrieve this data and load it into the data warehouse where you can report on it in any way, using any tool you like.

Writing Palo Data

The two output steps to write data to a Palo database using Kettle are, not surprisingly, the Palo Dimension Output step and the Palo Cell Output step. These steps are used not only for inserting data to existing cubes, but the Dimension Output step can also be used to create new dimensions, or add new elements to existing ones. The Cell Output step is a bit more strict in the sense that the structure of the data should adhere to the existing cube definition. What we'll do here is first add a new product to the product dimension, and then load data for this new product. You can't just upload new data using the Cells Output step, unless all the dimension values used already exist. This is why adding new data is always a two-step approach: First insert the dimension data, then the cell data. If you just want to update existing data, then this problem doesn't exist.

WARNING Back up the Palo database before you start updating or loading new data into it; trying to write measures to unknown dimension values could corrupt your database.

To give you an idea of the possibilities of the Palo output steps, we'll end this chapter with a couple of examples. First we'll update an existing cell in the database. We need to know the correct values of the dimension elements so having Calc or Excel open with the Palo add-in is very helpful. Then it's simply a matter of creating a data source containing the new values. In Figure 10-19, the input consists of a Data Grid step with one row of data that will update the Budget Units value for the product Desktop L in January 2002 in Germany. If you look at the demo database you'll notice that the value for this cell is 1135, and we'll change it into 99 using the Palo Cells Output step.

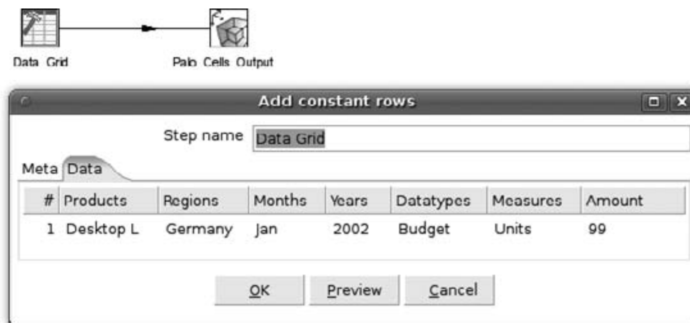


Figure 10-19: Updating a Palo database

The proof that this actually works can be seen in Figure 10-20, which shows a screenshot with the output. The updated row is clearly visible.

Figure 10-20: Update results in Open Office Calc

TIP **If new dimension entries don't show up immediately in OO Calc, just disconnect from and reconnect to the database.**

The next example involves adding a new product category and a few products to the database using a Palo Dimension Output step. Note that this step allows you to create new dimensions, as well as clearing the dimension before you load it. Be careful with this last option in an existing database because all the values that reference the dimension data will become obsolete. We can again use a Data Grid step to create the data and feed this into the Palo step, as shown in Figure 10-21.

Figure 10-21: Adding a new Product category and Products

In Figure 10-21, you're not only looking at the data feed in the grid, but also at the resulting updated product dimension with the new product group and products. After adding another value (2010) to the Year dimension, we can upload new budget values for our new product line, as shown in Figure 10-22.

Figure 10-22: Adding budget values for new products

Figure 10-22 shows both the Data Grid we used for the data input, including the data added to the cube, and the results in Open Office Calc. As you have seen in this section, working with Palo databases is very easy with the four steps available in Kettle. Palo can be a great asset for your organization's BI efforts.

Summary

This chapter covered everything related to working with multi-dimensional data, also known as OLAP cubes. The first part of this chapter was a general introduction to the benefits of using OLAP technology and how it fits in a business intelligence or data warehouse solution. Next, we provided three sets of examples, illustrating how the different OLAP steps in Kettle can be used. We covered the following products and techniques:

- **Reading data from a Mondrian cube using the Mondrian Input step with MDX:** The data retrieved was cleaned up for further processing with the additional steps "Split fields," Select Values, "Replace in string," and "Filter rows."
- **Reading data from a Microsoft SQL Server 2008 Analysis Services cube using the OLAP Input step:** The retrieved data was cleaned up for further processing using a single JavaScript step that showed how to loop through a data set to enter missing values.
- **Reading data from and updating data in a Palo database using the four available Palo steps:** We showed how to de-normalize data extracted from a Palo cube using the "Row denormaliser" step, and showed how to update data in an existing cube, extend a cube with new dimension information, and add new detail data for the newly created dimension values.

As we've demonstrated in this chapter, Kettle is one of the very few available ETL products that can work with nearly every OLAP database on the planet, including Hyperion Essbase, SAP BW, Microsoft SQL Server Analysis Services, IBM-Cognos TM/1, Mondrian, and Palo.

Part III

Management and Deployment

In This Part

Chapter 11: ETL Development Lifecycle
Chapter 12: Scheduling and Monitoring
Chapter 13: Versioning and Migration
Chapter 14: Lineage and Auditing

CHAPTER

11

ETL Development Lifecycle

In previous chapters, we introduced several parts of Kettle and showed how they fit into the 34 ETL subsystems identified by Ralph Kimball. This chapter is broader in scope in that we cover the total development lifecycle, not just individual pieces. Of course, we dive into some specific topics in detail as well.

Developing ETL solutions is an important part of building and maintaining a data warehouse and, as such, should be considered as part of a process, not an individual project. A *project* has one or more pre-defined goals and deliverables, and has a clearly defined start and end point. A *process* is an ongoing effort with periodically repeating activities to be performed. Creating ETL solutions is usually conducted as part of a project; monitoring, maintaining, and adapting solutions is part of an ongoing process. Adapting existing ETL jobs can, of course, be done in a more project-oriented setting. This chapter focuses on the initial part of the lifecycle where a new solution is being built. ETL solutions go through analysis, design, build, test, documentation, and delivery stages, just like any other piece of software. The challenge, of course, is to go through these phases as quickly as possible at the lowest possible cost and with preferably zero rework due to errors. As you will see in the subsequent sections, Pentaho's Agile BI tools are ideally suited to support this.

Solution Design

Any solution should be based on a design; jumping into coding without a plan is a common mistake made by developers, not just ETL developers. Even if you just start

out by sketching the transformation steps needed on the back of an envelope, you're already improving the quality of the solution. At the other extreme, you could go totally overboard by spending countless hours in specifying and designing solutions using workflow design tools and whatnot. We think the proper way of doing this lies somewhere in the middle between these two extremes.

Best and Bad Practices

An ETL design effort can only be started when the initial data warehouse design is done. Sometimes developers work their way toward a data warehouse from the source systems using their ETL tools. In fact, as you can see in Figure 11-1, Kettle supports such a way of working: whenever you augment a transformation that connects to a destination table, one click on the SQL button (highlighted in the figure) will show the change script to adapt the table to the new output specification.

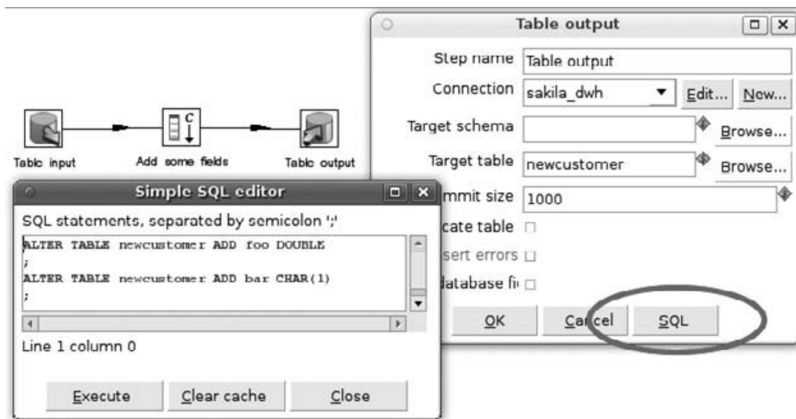


Figure 11-1: Table output alter script

Although this is a great piece of functionality for prototyping, we strongly advise you against using it for building production data warehouses. A good data warehouse should be based on a solid and well thought-out design. That design should be based on business requirements, not on what data is available in a source system. Since this is a book about building ETL solutions using Kettle and not about data warehouse design, we won't go into the details of data warehouse design. An excellent book on this topic is *Mastering Data Warehouse Design* by Claudia Imhoff et al. (Wiley, 2003); our previous book, *Pentaho Solutions* (Wiley, 2009), can also serve as a starting point for your work with data warehouse design.

Even if we won't cover the actual design of the data warehouse and data marts, the first best practice you should adhere to is take the time to create a proper design for these essential parts of your business intelligence solution. Other best practices that help improve the quality of your solution include:

- **Peer/expert reviews:** During the project you should plan for regular peer/expert review sessions to make sure that the solution will meet all criteria set out at the beginning. At a minimum, you should go over the overall ETL design before the actual implementation starts. A customer or manager might complain that this will increase the total project cost, but actually it does the opposite by minimizing the risk of rework and ensuring the best balance between development effort and the quality of the final result.
- **Standardization:** Developing standards and templates at the beginning of a project makes the effort of building the ETL solution a lot more predictable. Initially it takes time to set up those standards, but that effort will pay off many times over during the rest of the project lifecycle.

The next two subsections will cover two other best practices in more depth: data mapping, and naming conventions (which is a part of the general standardization best practice mentioned above). We conclude with a list of "bad practices" or common pitfalls that you need to avoid.

Data Mapping

Assuming you already have the data warehouse design in place, the initial challenge will be to find out where the data should come from. Chapter 6 covered the data profiling and extraction work involved. But as soon as your sources and target are known, the fun begins. How do you map the data from source to target? Which transformations are needed and how do you split up the workload into logical and manageable chunks? The latter is the subject of the next section; we'll cover the data mapping part here.

First, you should try to find the source system meta documentation, or at least get access to the metadata. If there are a lot of tables and columns involved, you don't want to enter them by hand. As a best practice, create a spreadsheet with all the target columns on one side, grouped by the table they belong to. Then, define the source for each of these columns. If fields need to be combined into a single target column, add additional rows to the spreadsheet to accommodate this. Each column on both the source and target side should also have the data type specified. Then you can use one or more columns in the spreadsheet between the source and target fields to specify the transformation. When no source column is available (for example, for a dimension version number or surrogate key), specify the process, function, or Kettle step that will deliver the correct value for the field.

TIP **Almost any database system will let you extract the metadata for the tables and columns. With Kettle, you can easily read this data and write it to a spreadsheet—no typing required and no chance of forgetting anything. Just add an Input Table step and an Excel Output step to a new transformation. Add the required metadata extraction query to the Table Input step and write the wanted columns to the spreadsheet. In the example in Figure 11-2, you can see the query for the** `sakila_dwh` **tables and columns, and the resulting XLS file with the metadata information opened in Open Office Calc.**

Figure 11-2: Sakila_dwh metadata

Naming and Commentary Conventions

Many people fall into the trap of using too many naming conventions, while others use none at all. Especially if your team consists of more than one person, or your work needs to be transferrable to a third party, naming standards can greatly improve the maintainability of the solution. Each object in Kettle can be given a specific name and, when combined with the icons used for jobs, transformations, and steps, this makes for easy and understandable solutions.

One important caveat is to avoid using the same names for jobs and transformations in one solution. Using `load_data` as a job name where one of the transformations is also called `load_data` is unnecessarily confusing. It's not a bad idea to use the prefix `jb_` for all your jobs and `tr_` for the transformations. Furthermore, using names for the transformations based on the type of work they are performing or table they are loading is a good convention as well. An extraction transformation for the customer table will then be called something like `tr_e_customer`, or when the extraction and staging takes place in a single transformation, it might be `tr_stg_customer`. Loading the corresponding dimension table can then be performed with a transformation called `tr_dim_customer`, and so forth. Some organizations also use the source system code in the prefixes, so if you're reading some customer data from an SAP system and some from a Siebel CRM application, you'll get a `tr_stg_sap_customer` transformation and a `tr_stg_crm_customer` transformation.

With steps it's a different story; they should at least get a meaningful name that reflects what action the step is performing. Last but not least are the notes. Notes allow you to add inline comments to your jobs and transformations, which is a great help to anyone trying to understand your work. To further improve the transparency of your solution, you should also use the description field of jobs, transformations, job entries, and steps. In contrast to notes, these description fields are "bound" to the corresponding object, so even if you copy and paste a step, the documentation gets copied as well.

An excellent example of a well-documented transformation can be found in the standard Kettle samples and is called `GetXMLData - Different Options.ktr`.

Within the transformations, the fields or column names are, of course, in use. As a naming convention, you might consider using the `t_` or `tmp_` prefix for all fields in a transformation that are only there for the purpose of the transformation but won't be loaded into the final output. Especially when working with large tables with many fields, this practice makes it easier to distinguish between the temporary and the final column names. You can also use Select Values steps to "clean up" the fields that won't be needed for subsequent steps.

Common Pitfalls

Other than avoiding the "just start coding" trap, there are a few other pitfalls or bad practices to consider:

- **Not talking to end users:** As an ETL developer, you might think that dealing with end users is for the people involved with project management or developing front-end tools. Nothing could be further from the truth! It's the business user who can tell you exactly if the data you're delivering is correct or not, and who can tell you what the meaning of the data in the source system actually is (which might be different from what the manual or the metadata tells you). The "Agile Development" section later in this chapter will show you how you can work jointly with your end users to develop great solutions, fast.
- **Making assumptions based on other projects:** Maybe your last project was for a different client that had the expensive online backup tool and where there was a 12-hour batch window to run ETL jobs. That doesn't mean that this project will be the same.
- **Ignoring changing requirements—**In many cases people start building a solution based on open or unsettled requirements and don't deal with changing requirements during the project life-span. The recommendations in the "Agile Development" section later in this chapter can help you avoid this pitfall.
- **Ignoring production requirements:** It is quite common to build your ETL solution in a development environment using a limited set of data. Make sure you have the ability to run your solution against a full production-size data set using production hardware in an early stage of the project. Usually your production environment will be more powerful, but we've seen situations where a process ran within 30 minutes on a development system but took several hours on the designated production machine.
- **Over-engineering the solution:** Kettle will save you a lot of time compared to other ETL tools, and there are always ways to make a solution even better than it is. Don't fall into this trap, however; when the jobs run within the designated batch windows, do what they need to do, and handle errors correctly they're good enough. Maybe not perfect, but good enough will do in most cases. At some point, the extra work stops adding value.

- **Skipping testing:** If the job runs on your development environment, it can be moved right into production, can't it? Wrong. A technically error-free process doesn't mean that the outcomes are correct. More on this later in this chapter
- **Failing to back up your data:** Lest we forget: make backups regularly! Although it is rare, your repository can become corrupt and thus unusable.

ETL Flow Design

An ETL solution is actually a process, which also means it can be designed as one. The process can be kicked off automatically at a designated interval, or can be waiting for a system event such as the availability of a file or system. Then there are inputs for the process: files, records and messages; and transformations that operate on these inputs. Roughly speaking, it's not that different from a business process such as the ones you'll find in a claims department or a car factory. Based on this similarity, you could be tempted to work with the various business process diagramming tools out there to design the ETL solution, but we have a better idea: Why not just use Kettle for this as well? Consider the example in Figure 11-3.

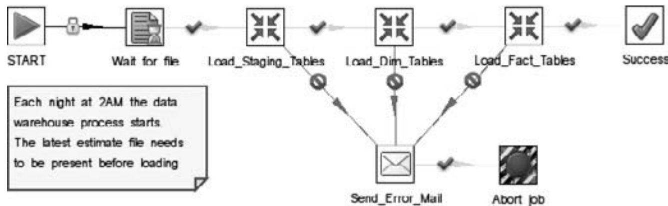


Figure 11-3: Job design

Figure 11-3 shows a typical main job that kicks off other jobs or transformations. This is a diagram that can be used to give a high-level overview of the ETL process to a business user, and it takes only a couple of minutes to create. If that seems quick, think about this: The job and the transformations shown don't do anything yet. They are basically just placeholders for the things to come, and serve as a good way to check with a domain expert about whether you're forgetting anything. Also don't forget to add items like the details put in the note—the plan to start the job each night at 2 a.m. It could be that the backup of the source systems is still running at that time, or that the required files won't be available until 6 a.m. By showing and discussing the design early in the development process, you'll avoid unpleasant discussions later on.

Reusability and Maintainability

One of the most important principles of structured programming is dividing your code into generic, reusable blocks. With Kettle, you can adhere to this same principle. The available building blocks, jobs, and transformations already allow for easy division of the steps in an ETL process. The capabilities to nest jobs in jobs and transformations

in jobs, and to reuse transformations as sub-transformations via a mapping step offer an endless list of possibilities. A good example of reusing functionality appeared in Chapter 4, where the `fetch_address` transformation was used in both the `dim_customer` and `dim_store` transformation via a mapping.

Reusability can also be accomplished by consistently using variables, especially for repeating elements such as error e-mail addresses or database connections. Remember that there are two ways of reusing elements within Kettle: copy/paste and real reuse. Whenever you find yourself using copy/paste, consider whether you'd be better off by placing the copied objects in a separate job, transformation, or mapping that can be called from other parts of the process. Using the Mail job entry poses another challenge: Many fields need to be filled in (sender, recipient, message, SMTP server, and so on) so it's tempting to just copy it to other locations after having done the data entry once. You could indeed do this, but then only when you've used variables for all the fields. When you use variables, it doesn't matter how many copies are being made; the value of the variables needs to be changed only once—unless, of course, someone decides to alter the variable names. The basic question you'll always have to ask yourself is this: If I copy this step/transformation/job and I need to make a change, in how many spots do I need to make this change? If the number in your answer is greater than 1, try to think of a better solution.

Back to the Mail job entry mentioned earlier. Remember that there are two Mail components in Kettle: one for jobs and one for transformations. Here we are discussing the Mail job entry. When you want a single Mail job entry to send an e-mail in case an error occurs during processing, create a new job called `jb_ErrorMail` or something similar, with just a single Mail step in it. This `jb_ErrorMail` job can be used as many times as you like, and whether a variable or anything else changes, the change needs to be applied in only one place.

Another good example of reuse is using variables for all database connection information. This enables you to migrate jobs and transformations from development to test to acceptance to production without the need to change anything inside your transformations. We cover migration in depth in Chapter 13

Agile Development

The hype du jour in the BI world seems to be about "agile development." We mentioned this briefly in Chapter 1 and want to elaborate a bit more on the subject here. There's a very good reason for doing that, too: Pentaho in general, and specifically Kettle, is not only positioned as an agile solution but also equipped with many features that support an agile way of working. There are many agile development methodologies, with Scrum and XP (Extreme Programming) probably the most well known. With Scrum, artifacts (deliveries such as code blocks or transformations) are posted in a backlog, and each development phase (or *sprint*) aims at clearing part of the backlog. Both the backlog and the final product are subject to constant changes. This doesn't mean that projects will run endlessly (there are time and financial constraints as well), but does mean that agile projects have a much more dynamic nature. These dynamics are reflected in the Kettle ETL design tool, Spoon, as well.

Starting with Kettle 4, Spoon will be the cornerstone of the Pentaho BI suite for developing BI solutions. Originally, Spoon was purely an ETL development tool. Now, it also allows you to work with so-called *perspectives*, with the modeling and visualization perspectives already available. These new additions to the Spoon IDE make it possible to design a multidimensional model and visualize the data in the solution in one easy-to-use workflow.

Spoon has adopted the *perspective* idea from Eclipse so that you can work on different types of solutions, all from within the same familiar interface. The Community Edition contains the Data Integration, Model, and Visualize perspectives, while the Enterprise Edition adds a fourth perspective called Scheduling. The workflow for which the Agile BI initiative offers the foundation is depicted in Figure 11-4, where you can see the three activities: Design ETL, Build Model, and Visualize.

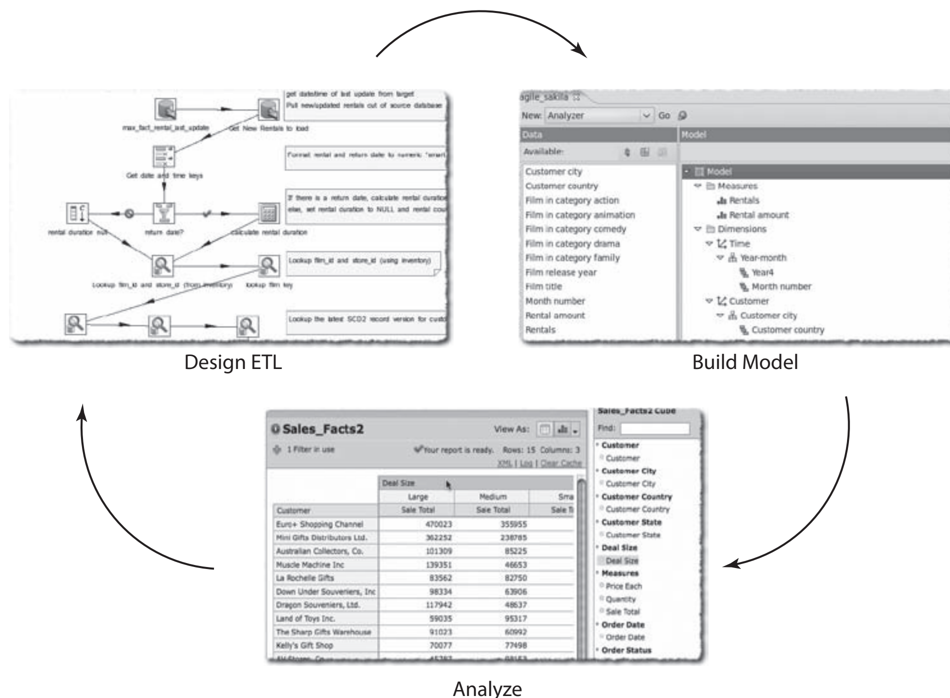


Figure 11-4: Agile BI initiative

The data integration capabilities should be very familiar already, but modeling and visualization are newly added capabilities to Kettle 4. With these related activities combined in a single toolset, it becomes very easy to sit down with an end user and interactively design the initial prototype for a data mart by showing the data as it will look like to a user in the final solution. Be careful, though; everything we mentioned earlier in this chapter about creating a proper design for your data warehouse, data marts, and ETL solution still holds. The Agile BI tools are great for prototyping, not (yet) for designing an enterprise data warehouse!

NOTE At the time of this writing, the Model and Visualize steps in Kettle were restricted to a single output table. The single output table might seem like a limitation but for quick prototyping it's not a real handicap.

EXAMPLE: WORKING WITH THE AGILE BI TOOLS IN KETTLE

In order to quickly analyze the data in your data warehouse or data mart from Spoon, you'll need two things:

- A data set that can be put into a single output table
- A multidimensional model built on top of this output table

At first it might seem awkward to put everything in a single table, but as you'll discover shortly, this doesn't have to be a problem. In fact, it makes the modeling very simple and straightforward. If the data is already available in a star schema database, it's very easy to create a flat model out of it that can be used with the *modeler* available from the Modeling perspective. The database used for this example is the same sakila_dwh database that was created in Chapter 4. It contains a single star schema with movies, customers, stores, staff, films, and a fact table with movie rental information. You need to do an extra denormalization step to create a single view of the dimensions and facts you want to analyze initially.

There's even a name for such a flat data model: the *One Attribute Set Interface* (OASI) model, as described by Dutch data warehouse expert Dr. Harm van der Lek in his Dutch book *Sterren en Dimensies (Stars and Dimensions)*, ISBN 90-74562-07-8, only available online at `http://array.nl/Boeken`. An OASI model contains all the non-key attributes available in a star schema. The following query will serve as the denormalization step you need to create an OASI model that you can use to build a multidimensional model:

```
SELECT
      c.customer_country AS customer_country,
      c.customer_city AS customer_city,
      d.year4 AS year4,
      d.month_number AS month_number,
      d.year_month_number AS year_month_number,
      f.film_title AS film_title,
      f.film_release_year AS film_release_year,
      f.film_in_category_action AS film_in_category_action,
      f.film_in_category_animation AS film_in_category_animation,
      f.film_in_category_comedy AS film_in_category_comedy,
      f.film_in_category_drama AS film_in_category_drama,
      f.film_in_category_family AS film_in_category_family,
      sum(r.count_rentals) AS rentals,
      sum(a.amount) AS rental_amount
FROM  dim_film f
INNER JOIN fact_rental  r ON f.film_key     = r.film_key
INNER JOIN dim_customer c ON c.customer_key = r.customer_key
```

Continued

EXAMPLE: WORKING WITH THE AGILE BI TOOLS IN KETTLE *(continued)*

```
INNER JOIN dim_date     d ON d.date_key     = r.rental_date_key
INNER JOIN amounts      a ON a.rental_id    = r.rental_id
GROUP BY
      customer_country,     customer_city,
      year4,
      month_number,
      year_month_number,
      film_title,
      film_release_year,
      film_in_category_action,
      film_in_category_animation,
      film_in_category_comedy,
      film_in_category_drama,
      film_in_category_family
```

Note that this is not the complete set of non-key attributes, but a subset that will demonstrate the principles behind the Model/Visualize concepts. Because the purpose of data analysis in this phase of a project is to get a good grasp of what a user can expect, it doesn't make a lot of sense to load all the data in the target output table. A nice way to randomly select rows from a large data set is to use the Reservoir Sampling step and set the number of rows to be sampled at a fixed number. In this case, you'll read 1,000 rows into the target table, as displayed in Figure 11-5.

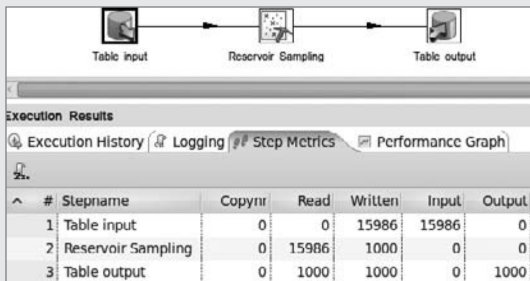


Figure 11-5: Reservoir Sampling

For the Modeler to work on the output table, it needs to be physically available in the database and loaded with the input data, so this transformation needs to run before you can start the Modeler. When you right-click on the output table, you'll see the Model and Visualize options at the bottom of the available option list. You can also click the Model button on the shortcut bar. The Modeler screen that is displayed contains three panels; from left to right you can see the available data items, the model itself, and the properties of the selected model element. Figure 11-6 shows an example model created on top of the data selected in the previous step.

EXAMPLE: WORKING WITH THE AGILE BI TOOLS IN KETTLE

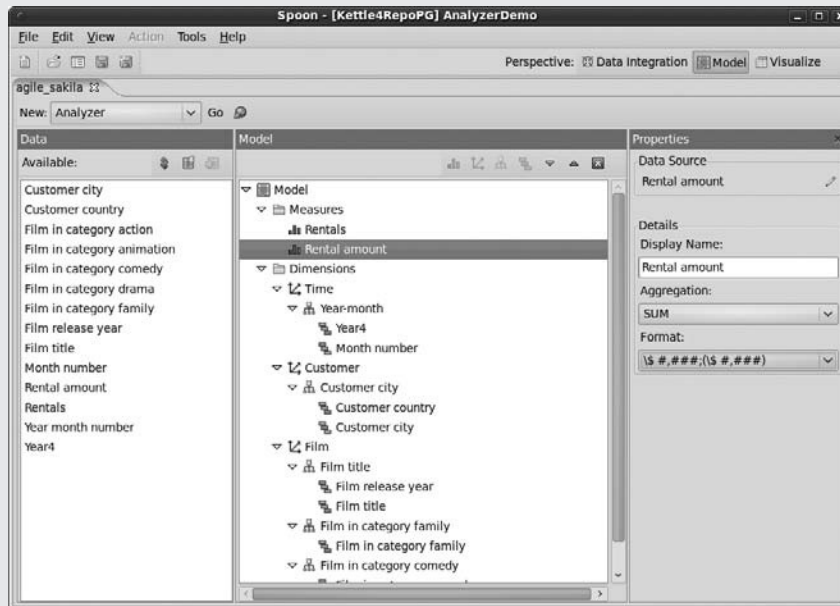


Figure 11-6: Example Analyzer model

Creating this model is pretty straightforward, and the modeler will automatically apply some basic rules to generate names from the query results. Figure 11-6 shows the data elements on the left, and as you will notice the return columns from the query have either been used as is or, in case the names contain underscores, the underscores have been replaced by spaces. This is all done automatically without the need to create a separate metadata model. There is one thing to be aware of, though; it's a multidimensional model. In Figure 11-6, you can see what this means for single attributes such as "Film in category drama" or "Film in category comedy." These attributes must be put in a separate hierarchy because there is no dependency between those fields. This is different for Year-Month or Country-City, which are real hierarchies and can be defined as such. For an easy overview of the available objects within their respective hierarchies, you can select View ➪ by Category, as displayed in Figure 11-7. What you can also see in this figure is why this tool is called Pentaho Analyzer: With a few clicks of the mouse you can filter the top 10 countries based on revenue, and do an analysis on the comedy category to see which countries score best in this category.

Continued

EXAMPLE: WORKING WITH THE AGILE BI TOOLS IN KETTLE *(continued)*

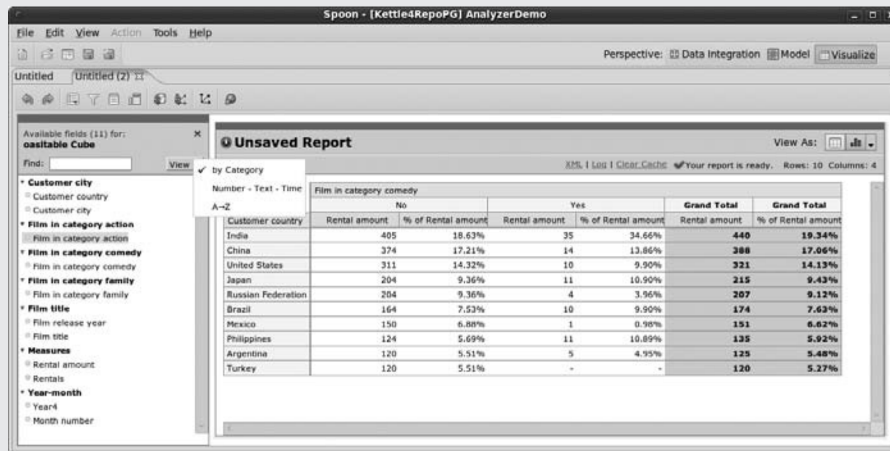


Figure 11-7: Example Analyzer report

A second option, which we won't show here, is the Report Wizard, which can be used to create a standard report. The magic behind the scenes is performed by the metadata layer (the Model), which can also be published to the Pentaho server and be used in the web portal.

It's not so much the technical wizardry that can be accomplished using the Agile BI toolset; it's the communication with end users that adds the real value. Working in an iterative way with immediate visualization options makes it possible to get answers right away when you're not sure what exactly is meant by things like "cost," "price," or "date." Being able to quickly replace "order date" with "delivery date" saves a lot of frustration afterwards.

Testing and Debugging

Although testing and debugging are often considered two sides of the same coin, there are some notable differences between the two. Debugging is an activity that follows testing in case an unpredicted or unforeseen error occurs, so if the testing phase runs smoothly, debugging isn't even necessary. Historically there was a distinction between *black box testing* where no knowledge of the internal working of the tested object was presumed, and *white box testing* where this knowledge was required. Nowadays, the terms *functional* or *behavioral* testing are more widely used for black box testing, although behavioral testing does allow some prior knowledge of internals of the tested object. Common terms for "white box testing" are *structural* or *clear-box testing.*

The problem with all these test methodologies is that they have been designed to test software, and what's more, software that has a user interface and predefined use cases.

For an order entry system for instance, a functional test could be that each customer record should have a phone number and, if not, the user should get a warning. The test is then conducted by trying to save a customer record without a phone number. The correct outcome should be that an error message is displayed and the record isn't saved. If that's the case, the test succeeded. If not, it failed. How exactly the system verifies the empty field and handles the exception is irrelevant.

An ETL process is different; it doesn't have a user interface and it's also not triggered by a user but usually automatically scheduled. The functional test has therefore other characteristics than when testing a typical software package. The structural test is also different and even simpler: You'll use the same tools for both developing and testing the solution. As you'll discover later, Spoon offers many ways for easy testing of your ETL solution.

Test Activities

There are complete libraries available about software testing so we won't go into too much detail here, but we do want to explain some of the terms you might not be familiar with.

We already mentioned functional/behavioral testing and structural/clear-box testing. *Static testing* is about reviewing documentation, code inspection, and walkthroughs, and is done without using the software. In the case of a Kettle solution, static testing can be done by going over the different parts of the solution based on a design document and checking whether the solution makes sense. Typically this is the kind of work a senior developer does when parts of a solution have been built by a less experienced ETL developer to get a general impression of the quality of the solution.

Dynamic testing involves actually using the software to find out whether it adheres to the required functionality. The activities we'll cover during the remainder of this chapter are all part of dynamic testing.

Three other familiar terms in testing are *unit tests*, *integration tests*, and *regression tests*. The names of these activities speak for themselves: with a unit test, an individual component (a single transformation or job) is the object of inspection. An integration test is performed on the completed solution, usually consisting of many jobs and transformations. Regression testing is the activity where a change is made to one component and aimed at proving that the total solution still works as expected after committing the change.

The ultimate test is the UA, or *User Acceptance test*. UA testing might seem like a strange activity for an ETL solution because end users won't be working with the system directly. They do, however, work with the data that is loaded into the data warehouse, and should be involved in testing the completeness and reliability of the data. Usually it is the end users who immediately spot an error made in a calculation, or notice missing data before the ETL team does. Data warehouse UA testing also involves working with the complete BI solution, including the front end reporting and analysis tools. Testing the data warehouse this way is beyond the scope of this book so we'll have to restrict ourselves to the ETL testing itself.

When working with Kettle, unit testing should be part of the regular development work. The next section highlights the types of tests that should be conducted and, as you'll see, Kettle performs some of these tests automatically.

ETL Testing

Organizing the test activities of ETL jobs and transformations is a challenging task, if only for the single fact that the number of possible use cases is virtually unlimited. Testing requirements will also differ based on the kind of environment the solution is meant for. In organizations where Sarbanes-Oxley (SOX) compliance is mandatory you need to have a fully auditable process. Article 302 of the Sarbanes-Oxley Act in particular has consequences for the way your ETL process is designed and tested (see `http://en.wikipedia.org/wiki/Sarbanes-Oxley_Act` for more information). These harsh legal requirements could trigger you to mandate 100 percent true and tested ETL solutions, but if you try to accomplish this you'll find out soon that it's impossible to test every exception that might occur during data transformation. What can be done, however, is to prove that under normal circumstances the system completely and correctly transforms a given set of input data. Note the terms *normal circumstances* and *a given set*. By the latter we don't mean *any* given set of input data!

Test Data Requirements

In order to test ETL processes, you need test data. This data should be as close to the actual production data as possible, in both volume and content. It's sometimes hard to get to production-like data, and in some cases it's not even permitted due to legal constraints. Some factors to consider when preparing data for testing or getting access to systems include:

- **Volume:** The sheer volume of data might prohibit conduction unit tests early on in the project. No matter what, at some point you do need to run a full integration/stress test using a full set of production data. For first stage unit tests, try to get a reasonable size of test data, which is a true random subset of the production data.
- **Privacy:** Production data containing financial account details, medical records, or other data containing sensitive private information is not usually available for testing. Sometimes it is possible to obtain an obfuscated set from these systems, which is better than getting a limited and often incomplete data set from the source systems test environment. In any case, make sure that the obfuscated data contains the same data types and data lengths as the data you'll ultimately be transforming.
- **Relevance:** The data should represent the business cases you have to deal with in your production environment. This means that if you need to generate test data manually, you need to make sure it covers real scenarios that represent the data in the production system as closely as possible.

It can be hard to create and maintain a good set of test data but it's an invaluable asset to an ETL project. The problem with using "regular" sources of data is that it constantly changes. Target data changes constantly, too, as a result of running development and test jobs—not an ideal situation for testing purposes as you need a controlled and isolated environment. Next to having test data at hand, it's also important to have a separate test environment. Reversing a database to a previous state after a failed or even successful run is a tedious task and something you'd rather automate. There are several ways of doing this: creating backups with different versions of the database is one option; and there are several tools available for database testing that can be used for this purpose too. One solution you might want to have a look at is DbUnit (`http://www.dbunit.org`), an open source extension for the JUnit testing framework.

Testing for Completeness

One of the first tests to be conducted is the completeness of the data. You need to make sure that all the data that goes into the process also comes out. There are a couple of easy-to-use techniques for this:

- **Record counting:** Count row totals for rows read, transformed, written, updated, and rejected. This is a no-brainer when using Kettle because these values are directly visible in the step metrics when running a transformation.
- **Hash totals:** Calculate totals or hash totals for critical columns in both source and destination system. Use this to determine whether the total sales amount in the source system matches the total sales amount in the data warehouse.
- **Checksums:** Kettle can calculate checksums based on one, some, or all columns in a transformation. After processing the records, this value can be compared to the checksum in the destination table records.

These numbers can be calculated on-the-fly and visually inspected, but a better way is to have them written to a log table in a database. This enables easy reporting against these tables and also provides a system of record of the ETL activities. We'll use the `load_rentals` transformation from Chapter 4 as an example. The transformation has a couple of challenges when testing for data completeness:

- Does the initial load work correctly and load all records from the source database?
- Does the CDC option work correctly, meaning that no records are skipped?
- Are all records read during the extraction loaded into the target table?

In order to have these tests run automatically, you need to capture the record counts in a logging table each time the transformation is run. We cover logging and auditing in-depth in Chapter 14 but we cover some basics here to help you start testing.

1. First, open the transformation `load_fact_rentals` and open the Transformation properties screen with CTRL+T or Edit ➪ Settings. In the third tab of the screen, Logging, you can define several logging options, but we'll keep it simple and just specify the transformation log.

2. To write to a log table, you need a connection to a database. For this example, we used the sakila_dwh connection. (In a real-life scenario, you might want to use a separate schema or database for the logging tables.)
3. If you enter a table name in the field "Log table name" and click the SQL button, the log table create script will be generated, ready for execution. Before you do this, make sure all the required log fields have been checked.

NOTE After checking additional log fields, make sure to alter the table with the SQL option; otherwise, your job/transformation will fail.

4. In the "Fields to log" section of the screen, the `Step name` column enables you to specify for which step the calculations need to be stored in the log table. Figure 11-8 shows the completed screen. Because you are only interested in totals for the transformation it suffices to look at the first and last steps in the transformation.

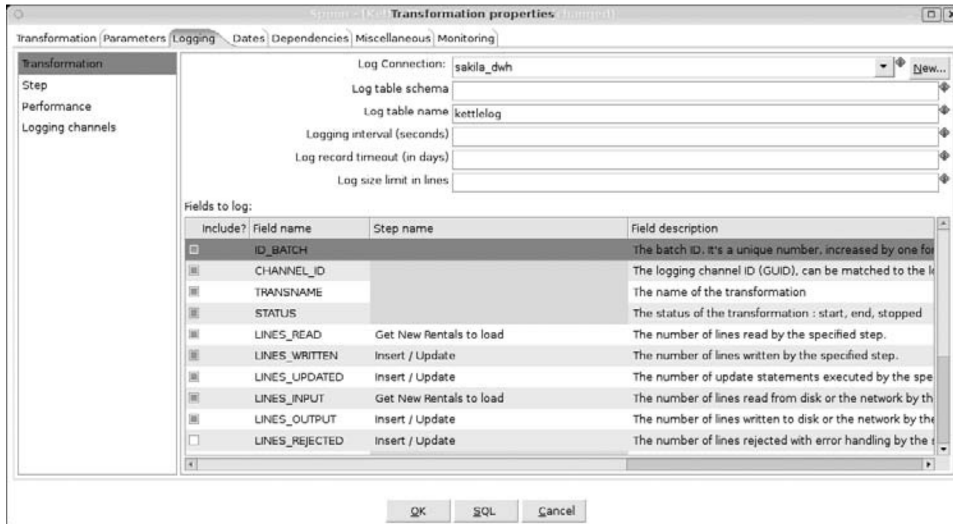


Figure 11-8: Transformation log settings

The first completeness tests can then verify the following test cases:

1. All data from the source system is correctly loaded during an initial load.
2. The CDC mechanism correctly identifies new records in the source system that were inserted since the last data warehouse load.
3. The CDC mechanism correctly identifies records in the source system that were updated since the last data warehouse load.
4. New records from the source system are correctly inserted in the fact table.
5. Updated records lead to an update of the correct rows in the target table.

Executing these tests is simple: For the first case, do an initial load on an empty `fact_rental` table and compare the counts in the `kettlelog` table (you'll of course see them while running the transformation as well). The second test requires at least one newly inserted record in the rentals table, while the third test requires at least one update. Cases 4 and 5 are for testing the correct delivery of the inserts and updates to the fact table.

Testing Data Transformations

As the data is read from various source systems and flows through the solution you've built, data is transformed in one way or another. Regardless of the logic and processing involved, there's one thing that can (and should) always be tested: expected outcomes given a certain input. Ideally you'll have a (complete) list of different scenarios at hand and a set of input and expected output values for each scenario. Try to avoid making this a manual effort; even in a simple solution with one or two source systems there are already many different possible cases.

The most commonly used tool for listing scenarios and input plus expected output values is the omnipresent spreadsheet. Kettle has no problem reading data from XLS files but for initial testing of newly developed transformations there's an even better way: use the Data Grid. The Data Grid allows you to easily specify and insert input data and is a great way to quickly test whether, for instance, a date will be correctly formatted by a transformation.

WARNING **A Data Grid step is a very convenient way to check whether your transformation behaves as expected, but it's only a first step, usually performed during development. For the actual test the normal input steps should be in place.**

Again you can use the `load_rentals` transformation because it offers many possible tests. For instance: What if a store ID is loaded that's not available in the store dimension? Is it specified how Kettle should handle this? A closer inspection of all the lookup steps shows that they all have the option "Do not pass if the lookup fails" selected. There's also a calculation step that calculates the `rental_duration`. Does this step generate the expected result? What happens when the rental return date is empty, and what is the effect if the return date is before the rental date? All these scenarios should have been taken care of already by adding data validation and error handling routines, but these have to be tested, too.

Test Automation and Continuous Integration

Running tests should optimally be an automated task so that once the proper tests are set up, the routines will run at a regular interval. This can be accomplished very easily by adding a schedule for the main job or jobs on the test environment. Whenever a new or altered part of the ETL solution is added to these main jobs it will automatically become part of the continuous integration test.

Upgrade Tests

Kettle is still being actively developed, meaning that every few months a new release becomes available. New releases not only contain bug fixes and new features but improvements and optimizations as well. For these reasons it's a good idea to keep up with the release schedule and upgrade your Kettle environment regularly. Of course you cannot just upgrade your existing production environment and assume everything will just keep working. There's always a chance that a new version contains a bug that wasn't there before. In order to prevent breaking the production system by carelessly performing an upgrade, you need to set up a separate test environment and test repository for performing upgrade tests.

A separate repository is needed because new features and functionalities usually require a repository upgrade. The repository upgrade is also an important part of the upgrade test. The correct order to do this is the following:

1. Create a new test repository using your current Kettle version.
2. Export your current production repository and import it into the test repository you just created.
3. Install the new Kettle version.
4. Upgrade the test repository using the new Kettle version.

When this process can be executed without problems you have your base upgrade test environment ready. Now you can perform all other tests on using this environment to find out whether an upgrade would cause problems. Note that you need to repeat the installation and upgrade process again for each new Kettle release.

Debugging

Debugging is the process of finding and fixing errors in a software program. In Kettle, debugging is in large part already enforced by the program itself. Many errors will have been solved during the development process itself, simply because in order to have Kettle run a complex job from start to finish means that all the connections are working, data can be read, exceptions and errors during transformations are handled, and data is delivered at the designated target. If one or more of these parts of the process fail, Kettle will simply throw an error telling you where you made a mistake. This is, in fact, already a form of implicit debugging and the resolutions are not always obvious. The explicit form of debugging in Kettle is provided by the Preview and Debug options.

The Preview and Debug options allow you to set arbitrary breakpoints in a transformation. Figure 11-9 shows the location of the two starting points for Preview and Debug.

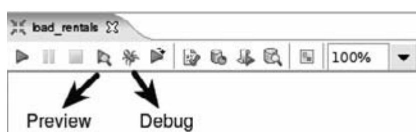


Figure 11-9: Preview and Debug buttons

There's a slight difference between the two, although they both open the same screen. When you have opened the Transformation debug dialog, you'll see all the steps listed on the left of the screen; the active step (the one for which a Preview or Debug will be issued) is selected or highlighted. The right side of the screen shows a large condition panel (similar to the condition builder in the "Filter rows" step) and the two checkboxes in the upper right. These checkboxes determine how the Debug dialog will behave when a Quick Launch is selected. Selecting "Retrieve first rows (preview)" will cause Kettle to pause the transformation, showing the number of rows entered in the "Number of rows to retrieve" box. Selecting to "Pause transformation on condition" will trigger Kettle to pause the transformation based on the condition(s) entered. Selecting both will just do a preview.

The breakpoint also handles the "Number of rows to retrieve" option, but then in reversed order. By setting a breakpoint and entering 10 as the number of rows, Kettle will pause based on the condition but also display the ten rows that entered the step prior to the one that triggered the condition. Figure 11-10 shows a debug screen with ten rows to retrieve where the transformation is paused when `customer_id` 500 is encountered.

ey	rental_datetime	return_datetime	customer_id	inventory_id	last_update
	2005/05/25 18:57:24.000	2005/06/02 20:44:24.000	500	2153	2006/02/15 21:30:53.000
	2005/05/25 18:45:19.000	2005/05/28 17:18:19.000	227	3725	2006/02/15 21:30:53.000
	2005/05/25 18:43:49.000	2005/06/03 18:13:49.000	19	4108	2006/02/15 21:30:53.000
	2005/05/25 18:40:20.000	2005/05/29 20:39:20.000	503	1044	2006/02/15 21:30:53.000
	2005/05/25 18:30:05.000	2005/05/30 19:40:05.000	242	3289	2006/02/15 21:30:53.000
	2005/05/25 18:28:09.000	2005/06/03 22:46:09.000	317	2833	2006/02/15 21:30:53.000
	2005/05/25 18:18:19.000	2005/06/04 00:01:19.000	196	3627	2006/02/15 21:30:53.000
	2005/05/25 17:54:12.000	2005/05/30 12:03:12.000	108	794	2006/02/15 21:30:53.000
	2005/05/25 17:46:33.000	2005/05/27 15:20:33.000	310	4281	2006/02/15 21:30:53.000
	2005/05/25 17:30:42.000	2005/06/03 22:36:42.000	384	3343	2006/02/15 21:30:53.000

1. 2. Close Stop Get more rows 3.

#	Stepname	Copynr	Read	Written	Input	Output	Updated	F
1	max_fact_rental_last_update	0	0	1	1	0	0	
2	Get New Rentals to load	0	1	16044	16044	0	0	
3	Get date and time keys	0	16044	16044	0	0	0	
4	Lookup dim customer key	0	423	422	601	0	0	
5	Lookup dim staff key	0	324	323	3	0	0	
6	Lookup dim store key	0	224	223	5	0	0	
7	Lookup film_id and store_id (from inventory)	0	593	592	589	0	0	
8	lookup film key	0	522	521	1000	0	0	
9	return date?	0	16044	16044	0	0	0	
10	calculate rental duration	0	9784	9783	0	0	0	

Figure 11-10: Using breakpoints

In the figure, Circle 1 identifies the `customer_id` we used for the breakpoint condition. Circle 2 shows the available options to proceed with the transformation: Close, Stop, or "Get more rows." Circle 3 shows that the transformation had already transformed

several rows before the breakpoint condition was met. That means that some caution is needed here: Even selecting Stop won't roll back the transformation to the original starting point because Kettle already might have loaded records into the target table. The Close button will only close the debug screen and keep the transformation in a paused condition. This is visualized by means of the green (=selectable) Pause and Stop icons to the left of the Preview and Debug icons, as shown in Figure 11-11.

Figure 11-11: Paused transformation

Selecting Stop does indeed terminate the running transformation, but again, the rows that have been already pushed through the process before the breakpoint condition was met are now loaded into the target table or file. If this is not your intention, you can keep Kettle from loading records immediately by adding a Blocking step before the final destination step. In case of the `fct_load_rentals` transformation, a Blocking Step can be placed directly before the Insert / Update step.

WARNING **The remarks about the stopping issue do not refer only to debugging but to any Preview. As soon as you select Preview, the entire transformation is started in the background and rows are processed. If there is a subsequent step in the transformation deleting records, the ultimate effect of this is that data could get deleted even if you only do a preview of a previous step!**

If you wonder whether Kettle supports row-by-row debugging: yes it does, although it does so a bit differently than some other tools. Just start with a regular debug and when a preview screen is shown, click "Get more rows." This will cause the preview screen to close, but when you click the Pause button in the transformation quick launch bar, you'll see that it's opened again, but now with one additional row. You can repeat this process for all other rows if needed.

Kettle 4 got many new features, but for debugging purposes the new drill-down and sniffing features stand out. When you're developing transformations and jobs, you probably noticed the fact that you can drill into jobs, transformations in a job, and mappings in a transformation by right-clicking on an object and selecting Open job, Open transformation, or Open mapping, respectively. This works not only during development, but during execution as well. What's more, you can also "sniff" the data during execution by right-clicking on a step and selecting "Sniff test during execution." From there you can select to sniff the input, output, or error rows of a step. It is also possible to open more than one sniffer simultaneously, as is shown in Figure 11-12. The simple example in Figure 11-12 shows a transformation that generates rows and uses a Modified Java Script Value step to concatenate the two string fields. Both the output rows of the Generate Rows step and the input rows of the "Text file output" step are shown.

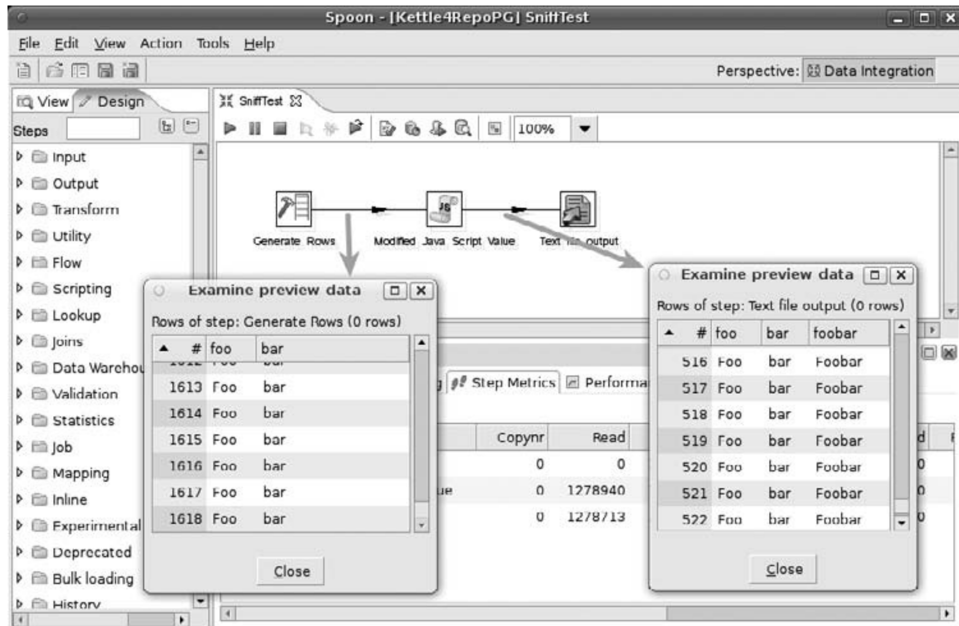


Figure 11-12: Sniffing a running transformation

A live example of the sniffing option is also available on YouTube (`http://www.youtube.com/watch?v=imvpQ8FFo-A`) where Matt Casters demonstrates this feature.

Documenting the Solution

One of the most important and at the same time most ignored aspects of ETL development (or any software development, really) is creating proper documentation. Documentation is an essential deliverable of any IT solution, and ETL is no exception. There are many purposes for documentation. Here are just a few benefits of good documentation:

- By recording the way the ETL design and its requirements are implemented, documentation allows the solution to be validated.
- If done right, documentation can help to train new developers more efficiently, as they can simply read the documentation to help them understand the implementation of the ETL solution.
- In the specific case of ETL, documentation could in principle be used to understand data lineage to facilitate audits.

Why Isn't There Any Documentation?

Although the benefits of documentation may seem obvious, the reality is that in many cases, documentation efforts are very poor or even completely absent. Despite the advantages, actually producing and maintaining documentation is highly unpopular. Developers don't like to write it, they like maintaining it even less, and because there are typically no short term benefits to having documentation, it's usually the project manager's first victim when a software project starts lagging behind its planning. (That is, if time was originally allotted at all for documentation!).

As a testament to its lack of popularity, there are a number of often-heard arguments against documentation:

- Because Kettle jobs and transformations are highly graphical by nature, some ETL developers may argue that they are self-explanatory, and thus need no additional documentation.
- Some developers argue that documentation is actually a bad thing, as it tends to become outdated as the solution develops over time, leaving outsiders with false or incomplete information about the solution.
- Another often-heard argument against documentation is that it's not worth the effort, because it won't be read anyway.

These are classic excuses not to document anything at all, and you may hear them from developers of any kind, be they C/C++ programmers, database developers or ETL developers. These arguments are easily countered, however.

Myth 1: My Software Is Self-Explanatory

Typically, those who claim that some piece of software is self-explanatory are often the ones who actually developed it. But what may seem self-explanatory to whomever built and designed the solution may not be so clear-cut to others who have to maintain and augment the solution in the future.

Also, a job or transformation (or any piece of software, really) can only be self-explanatory to the extent that it is clear as to *what it does*. Although this is certainly a desirable trait, and something you should strive for in your own design, this does not explain *why it is done this way*. Ideally, documentation should not have to explain the *what* so much as focus on the why.

Myth 2: Documentation Is Always Outdated

As for the argument that documentation quickly becomes outdated—this is certainly a valid argument, but one can hardly blame the documentation itself. If the documentation gets outdated too fast, it just means that the act of documenting is not considered part of the development process.

If developers and project managers treat documenting as a part of the development process in its own right, and if the documentation is properly tested (just like the software solution itself should be), there is no way it could become outdated. That said, it

certainly helps if the development environment allows documentation to be supplied as part of the development process to ensure that it is as easy as possible update the documentation while you're updating the actual software solution.

Myth 3: Who Reads Documentation Anyway?

Reading documentation is underrated, just like writing documentation, and often for the same reasons. It is not considered part of the job, and therefore unnecessary. But to be fair, one of the main reasons nobody reads documentation is because there literally is nothing to read; it simply wasn't produced in the first place. And if there actually is documentation, it may have been written by someone who didn't believe it was ever going to be read anyway (with the predictable impact on documentation quality).

These factors all build up to a self-fulfilling prophecy: if there is documentation, it wasn't written very well, because whomever wrote it thought nobody was going to read it anyway. Because the documentation is no good, it isn't going to be read. And so, because nobody reads it anyway, it is not going to get written at all the next time around.

Kettle Documentation Features

It is always possible to document your ETL solution using some external tool like a word processor a wiki website, or a set of static web pages. The advantage of these documentation solutions is that they can be easily distributed, read on a computer screen, or printed out to hard copies.

The disadvantage of maintaining the documentation text in an external solution is that it increases the distance between the documentation and the ETL solution itself. We just argued that ideally, the act of documenting should be part of the development process. This should be taken quite literally; whenever the solution is changed for maintenance, the change should ideally be recorded and added to the documentation immediately. This ensures the documentation will stay up to date, and will be as accurate as possible.

Therefore, it should be as easy as possible to document your ETL solution. While this is no guarantee that documentation will stay up to date, making it harder for developers to document things by forcing them to open an external application to record the change will certainly not help.

Kettle offers a few features to embed human-readable information in jobs and transformations. You can use these features to at least record the intent and design considerations of your jobs and transformations. Currently, Kettle does not offer anything out of the box to turn this information into proper documentation, but in the last subsection of this chapter, we will demonstrate how you extract this information later on to generate proper documentation.

The Kettle features that help you document your solution are:

- **Descriptive identifiers:** Essentially there are no practical restrictions to choosing identifiers for jobs, transformations, steps, job entries, fields, and so on. Of course, identifiers must be unique within their scope; for example, within one

transformation all steps must have a distinct name, but other than that, pretty much anything goes.

- **Notes:** Notes are little panels that can be placed anywhere on the canvas of a job of transformation. They contain arbitrary text and can be used to highlight some feature of the job or transformation. In Spoon, you can add a new note by right-clicking on the canvas and then choosing the "New note" menu item in the context menu. This will open a dialog where you can enter the text and set a few properties such as font style and color.
- **Descriptive fields in the job and transformation settings:** In Spoon, you can right-click on the canvas and choose the "Job settings" or "Transformation settings" item. This opens a dialog with a number of tabs. The Job or Transformation page offers a number of properties can be used for documentation. The Description and Extended Description fields are useful for entering a textual description of the job or transformation. There is a Status field where you can mark the job or transformation as either Draft or Production. The version field may be used to enter a version number. The "Job properties" dialog is shown in Figure 11-13.
- **Parameters:** Parameters can also be edited in the job or transformation settings dialog on the Parameters tab. Parameters support a description field, which you can use to record useful information about the parameter, such as its purpose, the expected data type and format, and maybe some example values.

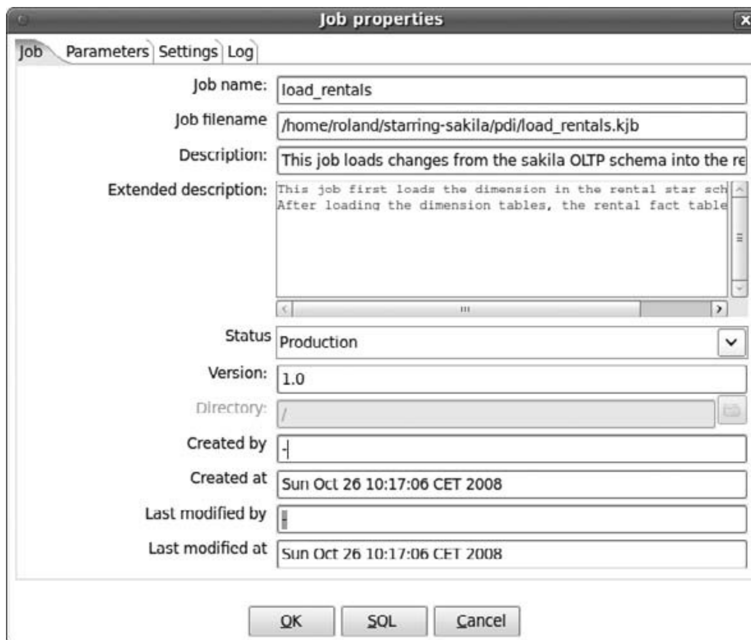


Figure 11-13: Using descriptive fields to document a job

While these features are quite limited, they allow you at least to record essential information about your jobs and transformations as you are developing them. Because the descriptive data is stored inside the job or transformation, it is always available when you're developing or maintaining your ETL solution.

There are two important things that are not provided by these built-in Kettle features. First, you need to have the discipline to actually use these features. Second, merely recording descriptive data is a prerequisite for documentation, but it is not the documentation itself. Iin the final subsection of this chapter, you will see how you can extract the descriptive data you entered into your jobs and transformations to generate the actual human-readable documentation.

Generating Documentation

Currently, Kettle does not offer any built-in functionality to generate documentation. At the time of writing, Pentaho is developing an auto-documentation feature, but this is slated to appear in a future version of the Kettle Enterprise Edition.

Fortunately, there is a community project started by Roland Bouman, one of the authors of this book, that allows you to generate HTML documentation from a collection of transformation and job files. The project is called kettle-cookbook, and is currently hosted at Google code at `http://code.google.com/p/kettle-cookbook`.

The kettle-cookbook solution provides a couple of Kettle jobs and transformations that scan a directory and its subdirectories for `.ktr` and `.kjb` files. The `kjb` and `ktr` files store the definition of Kettle jobs and transformations in an XML format, and kettle-cookbook applies an XSLT stylesheet to the files to generate HTML documents which can be read using a standard web browser. Finally, kettle-cookbook generates a table of contents that allows you to navigate through the jobs and transformations.

To generate documentation using kettle-cookbook, follow these directions:

- Collect all jobs and transformation files you want to document and place them in a directory. This is what is referred to as the *input directory*. A simple way to collect jobs and transformation files in such a way that any dependencies between files are preserved is to use Kettle's export feature. You can find this feature in Main menu ➪ Export ➪ Linked Resources to XML.... This feature does all the hard work of tracking the dependencies, and then places all files it finds in a single `.zip` archive. You can then unzip this archive to the input directory. The export method also works for jobs and transformations stored in the repository.
- Create an empty directory where the documention is to be generated. This is referred to as the *output directory*.
- Run the `document-all` job. This is located in the `pdi` directory beneath the kettle-cookbook main directory. This job takes on two parameters: `INPUT_DIR` and `OUTPUT_DIR`, which have to be set to the location of the input directory and output directory respectively.

When the `document-all` job is finished, the output directory will contain the generated HTML documentation. You can start reading it by opening the `index.html`

document in your web browser. The kettle-cookbook project website shows an example of the generated output.

Summary

The previous part of this book was all about the basics of building ETL solutions. This chapter was the opening for Part III, "Management and Deployment," and provided a broad overview of ETL solution development. We covered the following topics that together form the foundation for a successful ETL project:

- **Solution design:** You learned the best and some of the worst practices in designing ETL solutions. We addressed the importance of proper data mapping and naming conventions, and we listed several common pitfalls in ETL projects. Finally, we covered flow design and reusability.
- **Agile development:** You learned about the available tools and techniques within Kettle to support Agile BI projects. A large part of this section was devoted to a sample case of working with the Model and Visualize tools.
- **Testing and debugging:** You discovered what testing in an ETL setting means and the challenges involved. The last part of the "Testing and Debugging" section showed you how the debug tools in Kettle can be used for inspecting data in detail at each stage of a transformation.
- **Documentation:** You learned about the benefits of documentation and about a number of features that allow you to embed descriptive information inside your jobs and transformations. Finally, you learned how to generate human-readable HTML documentation from an ETL solution using kettle-cookbook.

CHAPTER

12

Scheduling and Monitoring

In this chapter, we take a closer look at the tools and techniques to run Kettle jobs and transformations in a production environment.

In virtually any realistic production environment, ETL and data integration tasks are run repeatedly at a fixed interval in time to maintain a steady feed of data to a data warehouse or application. The process of automating periodical execution of tasks is referred to as *scheduling*. Programs that are used to define and manage scheduling tasks are called *schedulers*. Scheduling and schedulers are the subject of the first part of this chapter.

Scheduling is just one aspect of running ETL tasks in a production environment. Additional measures must be taken to allow system administrators to quickly verify and, if necessary, diagnose and repair the data integration process. For example, there must be some form of notification to confirm whether automated execution has taken place. In addition, data must be gathered to measure how well the processes are executed. We refer to these activities as *monitoring*. We discuss different ways to monitor Kettle job and transformation execution in the second part of this chapter.

Scheduling

In this chapter, we examine two different types of schedulers for scheduling Kettle transformation jobs and transformations:

- **Operating system–level schedulers:** Scheduling is not unique to ETL. It is such a general requirement that operating systems provide standard schedulers, such

as `cron` on UNIX-like systems, and the Windows Task Scheduler on Microsoft Windows. These schedulers are designed for scheduling arbitrary programs, and can thus also be used for scheduling Kettle command-line programs for running jobs and transformations.

- **The Quartz scheduler built into the Pentaho BI Server:** Kettle is part of the Pentaho BI stack, and many Kettle users are likely to also use or be familiar with the Pentaho BI Server. The scheduler built into the Pentaho BI Server can be used to run an action sequence for executing Kettle transformations and jobs.

Operating System–Level Scheduling

All major operating systems provide built-in features for scheduling tasks. The tasks that can be scheduled by operating system–level schedulers take the form of a command line: a shell command to execute a particular program, along with any parameters to control the execution of that program. So in order for you to use an operating system–level scheduler for scheduling Kettle jobs and transformations, we first need to explain how you can run Kettle jobs and transformations using a command line. This is described in detail in the following section.

Executing Kettle Jobs and Transformations from the Command Line

Kettle jobs and transformations can be launched using the command-line tools Kitchen and Pan, respectively. Pan and Kitchen are lightweight wrappers around the data integration engine. They do little more than interpret command-line parameters and invoke the Kettle engine to launch a transformation or job.

Kitchen and Pan are started using shell scripts, which reside in the Kettle home directory. For Windows, the scripts are called `Kitchen.bat` and `Pan.bat` respectively. For UNIX-based systems, the scripts are called `kitchen.sh` and `pan.sh`. As with all the scripts that ship with Kettle, you need to change the working directory to the Kettle home directory before you can execute these scripts.

You may find that you need to modify the Kitchen or Pan scripts. There may be several reasons for this: perhaps you need to include extra classes in the Java classpath because your transformation contains a plugin, a user-defined Java expression, or user-defined Java class that requires it. Or maybe you're experiencing out-of-memory errors when running your job and you want to adjust the amount of memory available to the Java Virtual Machine. Chapter 3 includes a section called "The Kettle Shell Scripts" that explains the structure and content of the Kettle scripts. You may find that section useful if you need to modify the Kitchen or Pan scripts.

NOTE **Note that the scripts for UNIX-based operating systems are not executable by default—they must be made executable using the** `chmod` **command.**

Command-Line Parameters

The Kitchen and Pan user interface consists of a number of command-line parameters. Running Kitchen and Pan without any parameters outputs a list of all available parameters.

The syntax for specifying parameters is:

```
[/-] name [[:=] value]
```

Basically, the syntax consists of a forward slash (/) or dash (-) character, immediately followed by the parameter name. Most parameters accept a value. The parameter value is specified directly after the parameter name by either a colon (:) or an equals (=) character, followed by the actual value. The value may optionally be enclosed in single (') or double (") quote characters. This is mandatory in case the parameter value itself contains white space characters.

NOTE **Using the dash and equals characters to specify parameters can lead to issues on Windows platforms. Stick to the forward slash and colon to avoid problems.**

Although jobs and transformations are functionally very different kinds of things, there is virtually no difference in launching them from the command line. Therefore, Kitchen and Pan share most of their command-line parameters. The generic command-line parameters can be categorized as follows:

- Specify a job or transformation.
- Control logging.
- Specify a repository.
- List available repositories and their contents.

The common command-line parameters for both Pan and Kitchen are listed in Table 12-1.

Table 12-1: Generic Command-Line Parameters for Kitchen and Pan

NAME	VALUE	PURPOSE
norep		Don't connect to a repository. Useful to bypass automatic login.
rep	Repository name	Connect to repository with the specified name.
user	Repository user name	Connect to repository with the specified username.
pass	Repository user password	Connect to repository with the specified password.

Continued

Table 12-1 *(continued)*

NAME	VALUE	PURPOSE
`listrep`		Show a list of available repositories.
`dir`	Path	Specify the repository directory.
`listdir`		List the available repository job/repository directories.
`file`	Filename	Specify a job or transformation stored in a file.
`level`	`Error` | `Nothing` | `Basic` | `Detailed` | `Debug` | `Rowlevel`	Specify how much information should be logged.
`logfile`	Filename for logging	Specify to which file you want to log. By default, the tools log to the standard output.
`version`		Show the version, revision number, and build date of the tool.

Although the parameter names are common to both Kitchen and Pan, the semantics of the `dir` and `listdir` parameters are dependent upon the tool. For Kitchen, these parameters refer to the repositories' job directories, or to transformation directories in the case of Pan.

Running Jobs with Kitchen

In addition to the generic command-line parameters, Kitchen supports a couple of specific ones. These are shown in Table 12-2.

Table 12-2: Command-Line Parameters Specific to Kitchen

NAME	VALUE	PURPOSE
`job`	Job name	Specify the name of a job stored in the repository.
`listjobs`		List the available jobs in the repository directory specified by the `dir` parameter.

The following code provides a few examples of typical Kitchen command lines.

```
#
# list all available parameters
#
kettle-home> ./kitchen.sh

#
# run the job stored in
```

```
# /home/foo/daily_load.kjb
#
kettle-home> ./kitchen.sh \
           > /file:/home/foo/daily_load.kjb

#
# run the daily_load job from the
# repository named pdirepo
#
kettle-home> ./kitchen.sh /rep:pdirepo \
           > /user:admin \
           > /pass:admin \
           > /dir:/ /job:daily_load.kjb
```

Running Transformations with Pan

The Pan-specific command-line parameters are completely equivalent to the Kitchen-specific ones. They are shown in Table 12-3.

Table 12-3: Command-Line Parameters Specific to Kitchen

NAME	VALUE	PURPOSE
`trans`	Job name	Specify the name of a job stored in the repository.
`listtrans`		List the available jobs in the repository directory specified by the `dir` parameter.

Using Custom Command-Line Parameters

When using command-line tools to execute jobs and transformations, you may find it useful to use command-line parameters to convey configuration data. For the command-line tools Kitchen and Pan you can use Java Virtual Machine properties to achieve the effect of custom command-line parameters. The syntax for passing these "custom" parameters is:

```
-D<name>=<value>
```

The following example illustrates how such a parameter might appear in a Kitchen command line.

```
kettle-home> kitchen.sh /file: -Dlanguage=en
```

In transformations, you can use the Get System Info step to obtain the value of the command-line parameters. You can find this step in the Input category. The Get System Info step generates one output row containing one or more fields having a system-generated value. The Get System Info step is configured by creating fields and picking a particular type of system value from a predefined list, as shown in Figure 12-1.

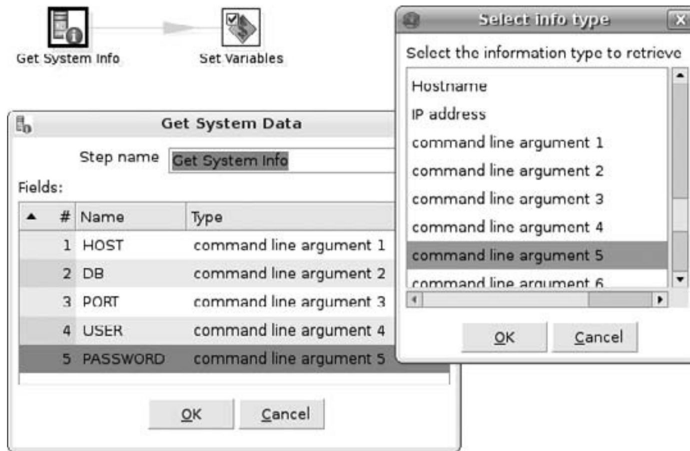


Figure 12-1: Capturing command-line parameters with the Get System Info step

Among the predefined system values, you'll find "command line argument 1" up to "command line argument 10". Fields with one of these types automatically take on the value of the corresponding `D<name>=<value>` command-line parameter.

UNIX-Based Systems: cron

The `cron` utility is a well-known job scheduler for UNIX-like systems. You shouldn't have to install anything to get it to work.

Actually scheduling tasks with `cron` is done by simply adding entries for cron jobs to a special file called the `crontab` (for cron table). The `crontab` file is usually located in `/etc/contab`, but there may be differences depending upon the UNIX flavor. In many cases, a `crontab` utility program is available, which facilitates maintaining `cron` entries.

The actual `cron` entries consist of the `cron` string, which denotes the actual schedule and recurrence, followed by the operating system command that is to be executed. For executing Kettle jobs or transformations, the command line is simply a command-line invocation of `kitchen` (for Kettle jobs) or `chef` (for Kettle transformations), or a shell script containing such a `kitchen` or `chef` invocation.

The `cron` string defines five fields of a date/time value, separated by white space. From left to right, the date and time fields are:

- minutes: 0–59
- hours: 0–23
- day of month: 1–31
- month: 1–12
- day of week: 0–6, where 0 denotes Sunday, 1 Monday, and so on

This is best illustrated with some examples:

```
0  1  ?  *  5  run_kettle_weekly_invoices.sh
```

In the preceding example, the `cron` string reads from left to right `0`: at zero minutes and `1`: at one o'clock a.m. The `?` means regardless of the day of month, `*`: each month, and 5 means Friday. In English, this reads "every Friday at 1:00 AM."

NOTE **For more options on using `cron` and `crontab`, refer to your operating system documentation. `man crontab` is usually a good start. There are also many online resources that offer good examples and explanations concerning cron.**

Windows: The at utility and the Task Scheduler

Windows users can use the `at` utility or the Task Scheduler.

The `at` utility is available from the command line. Here's a simple example that illustrates how to schedule execution of a batch job to run each day at midnight:

```
at 00:00 /every:M,T,W,Th,F,S,Su "D:\pentaho\pdi\dauly_job.bat"
```

Instead of providing a long command directly at the `at` command line, it is usually better to write a batch file (`.bat` file) and have `at` execute that. (This technique can, of course, be applied also on UNIX-like systems where you would write a `bash` or `sh` script.)

Windows also offers a graphical interface for scheduling. You can find the Windows Task Scheduler in the Control Panel or in the Start menu by navigating to Start ➪ Programs ➪ Accessories ➪ System Tools ➪ Task Scheduler.

NOTE **For more information on the `at` command or the Task Scheduler, go to `http://support.microsoft.com/` and search for "at command" or "Task Scheduler."**

Using Pentaho's Built-in Scheduler

If you are using the Pentaho Business Intelligence suite, you can use its built-in scheduler for scheduling Kettle transformations (but not jobs) as an alternative to operating system–level scheduling.

NOTE **A detailed discussion of the Pentaho BI Server and its development tools is outside the scope of this book. For readers interested in installing, configuring, and using the Pentaho BI Server (and the other components of the Pentaho BI stack) we recommend the book *Pentaho Solutions* by Roland Bouman and Jos van Dongen.**

The Pentaho BI Server provides scheduling services through the Quartz Enterprise Job Scheduler, which is part of the Open Symphony project. For more information on Quartz and the Open Symphony project, visit the Open Symphony website at `http://www.opensymphony.com/quartz/`.

Using Pentaho's built-in scheduler for running Kettle jobs or transformations requires two things:

- An *action sequence*, which is a container for one or more tasks (such as a Kettle job or transformation) that can be executed by the BI server
- A *schedule* that determines when and how often some action sequence should be executed by the Pentaho BI Server

In this context, the actual scheduling is done by attaching the action sequence to an existing schedule so it will be automatically executed at the right moment(s) in time.

Creating an Action Sequence to Run Kettle Jobs and Transformations

Before you can schedule Kettle transformations, they must first be made executable by the Pentaho BI Server. This is done by creating an action sequence containing a process action of the `Pentaho Data Integration` type. With this process action, you can execute Kettle transformations from either a `.ktr` file or from the Kettle repository.

If you want to use a Kettle repository other than the default repository, you must edit the `settings.xml` file located in the `kettle` folder that resides in the `system` Pentaho solution. The contents of this file are as follows:

```
<kettle-repository>

  <!-- The values within <properties> are
       passed directly to the
       Kettle Pentaho components. -->

  <!-- This is the location of the
     Kettle repositories.xml file,
     leave empty if the default is used:
     $HOME/.kettle/repositories.xml -->

  <repositories.xml.file></repositories.xml.file>
  <repository.type>files</repository.type>
  <!--  The name of the repository to use -->
  <repository.name></repository.name>

  <!--  The name of the repository user -->
  <repository.userid>admin</repository.userid>

  <!--  The password -->
  <repository.password>admin</repository.password>

</kettle-repository>
```

To configure the repository to use, you'll need to provide a value between the `<repository.name>` and `</repository.name>` tags, and this must match the name of a repository defined in the `repositories.xml` file, which was discussed in Chapter 3.

If you like, you can use a non-default `repositories.xml` file by specifying its location as a value for the `repositories.xml.file` tag. The values for `repository.userid` and `repository.password` must be valid credentials for that repository, and will be used when accessing its contents.

You can create and modify action sequences using the Pentaho Design Studio or the action sequence plugin for Eclipse. You can download these tools from the Pentaho project page on `sourceforge.net`, or from the Hudson server at `ci.pentaho.com`. Again, a detailed discussion of these tools is outside the scope of this book, but you can find more information on these subjects in *Pentaho Solutions*.

Kettle Transformations in Action Sequences

You can incorporate a Kettle transformation into an action sequence using a `Pentaho Data Integration` process action. In Pentaho Design Studio, you can find this process action beneath the Get Data From category.

The `Pentaho Data Integration` process action can accept input parameters from the action sequence, and returns a result set to be used later on in the action sequence. In addition to the result set, the action sequence can also receive diagnostic information, such as the transformation log.

The Pentaho BI Server comes with a number of examples that illustrate this process action. You can find these examples in the `etl` folder of the `bi-developers` solution. The `PDI_Inputs.xaction` is most illustrative, and is shown in Figure 12-2.

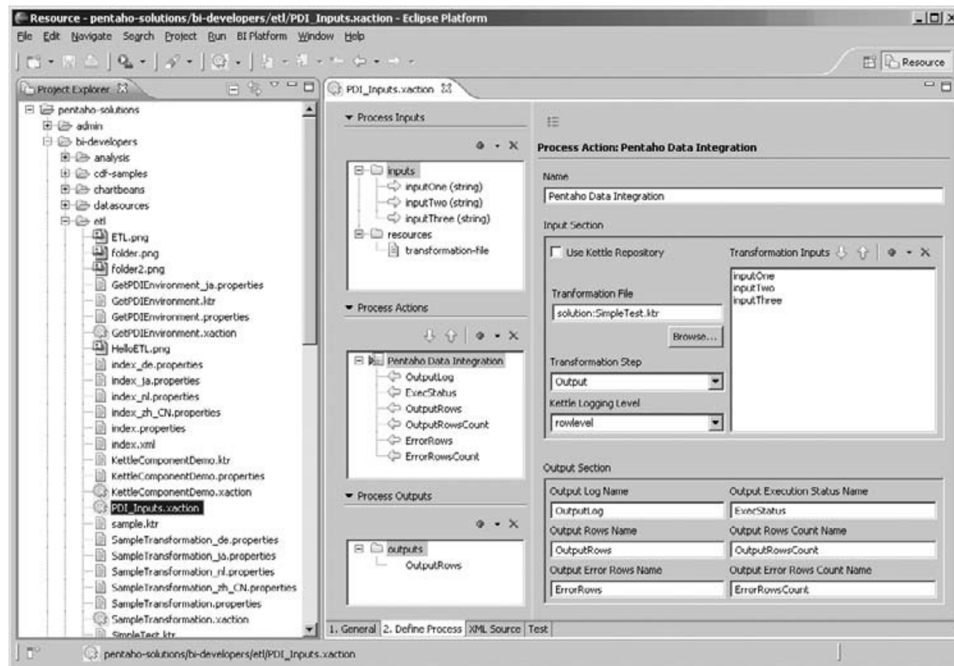


Figure 12-2: The PDI_Inputs.xaction sample action sequence

The following list covers a few of the most important configuration options for the `Pentaho Data Integration` process action.

- To run a transformation stored in the Kettle repository, select the Use Kettle Repository checkbox. Also, see the note on configuring the Kettle repository for usage within the Pentaho BI Server at the start of this section. When this box is selected, the action sequence editor will allow you to enter a directory name and a transformation name to identify the transformation.
- In the Transformation File text box, enter the name of the transformation. You can use the Browse button to point to a `.ktr` file located somewhere on the machine where the Pentaho server resides. In this case, the value for Transformation File takes the form of a regular `file://` URL. You can also use a transformation stored in the Pentaho Solution Repository using the special prefix `solution:`. This is the option shown in Figure 12-2, where `SimpleTest.ktr` is a transformation file that resides in the same solution folder as the `PDI_Inputs.xaction` action sequence itself.
- In the Transformation Step text box, enter the name of the step inside your transformation that is to return the result set to the action sequence. Note that although the transformation could yield multiple result sets, you can only receive one of them in the action sequence.
- You can use the Transformation Inputs list to feed parameters from the action sequences into your transformation. These parameters can be read by the Kettle Transformation using a System Info step configured to read command-line arguments. You encountered this step earlier in this chapter (see Figure 12-1) when we discussed passing command-line parameters to transformations.
- The Kettle Logging Level list box lets you choose at what level Kettle should log the execution of the transformation. Logging levels are explained in detail later in this chapter.
- In the bottom section of the action sequence editor, you can map diagnostics such as the log and the number of returned rows to yet more action sequence parameters.

Creating and Maintaining Schedules with the Administration Console

You can create and maintain schedules using the Pentaho Administration Console. This is included in the Pentaho BI Server.

To work with schedules, navigate your web browser to the Administration Console web page. Enable the Administration page and click on the Scheduler tab. You can see a list of all public and private schedules maintained by the Pentaho BI Server, as shown in Figure 12-3.

By default, there is one schedule called `PentahoSystemVersionCheck`. This is a private schedule that is used to periodically check if there is a newer version of Pentaho.

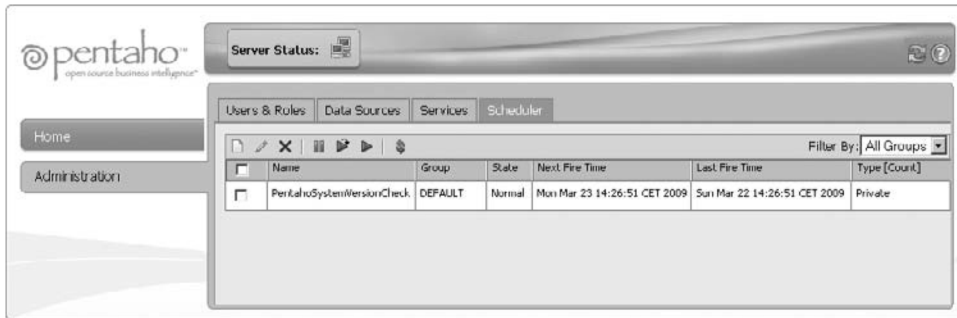


Figure 12-3: The Scheduler tab in the Pentaho BI Server Administration Console

NOTE **A *public schedule* ensures that all users can see the schedule and enables them to use this schedule for subscriptions. A *private schedule* is not available to end users, and is intended to schedule system maintenance operations by the Pentaho BI Server administrator**

On the Scheduler tab, click the first toolbar button to create a new schedule. The Schedule Creator dialog, shown in Figure 12-4, appears.

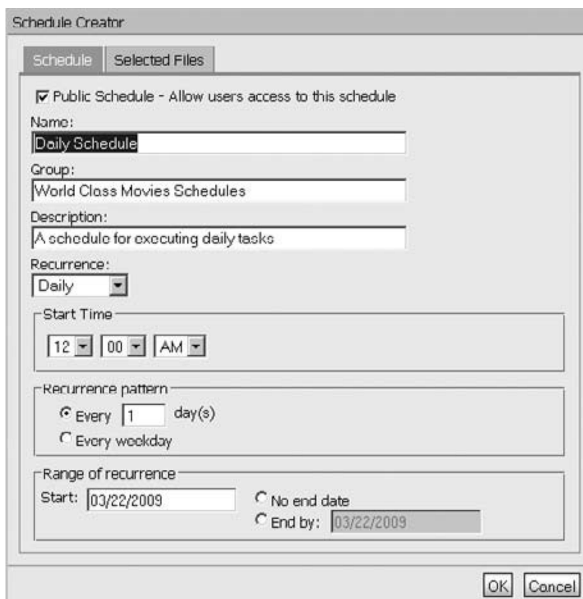


Figure 12-4: The Schedule Creator dialog

The options in the Schedule Creator dialog are described in the following list:

- To create a public schedule, select the Public Schedule checkbox.
- Use the Name field to enter a unique name for the schedule. Note that for public schedules, this name will be presented to end users whenever they have to pick a schedule. Use names that are both clear and concise.
- Use the Group field to specify a name that describes what kind of schedule this is. For example, you can create groups according to department (for example, warehouse, human resources), location (Edmonton, Los Angeles), subject area (Sales, Marketing), or timing (daily, weekly).

  The toolbar on the Scheduler tab includes a list box that enables the administrator to filter for all schedules belonging to the same group. This convenience feature enables the administrator to quickly work with all schedules belonging to the same group.

NOTE Even if you feel you don't need to organize schedules into groups, you are still required to enter one. The same goes for the `Description` field: although it is entirely descriptive, it is still required.

- The `Description` field allows you to enter descriptive text. You should use this field to briefly document the purpose of the schedule. Rather than summing up any properties that have to do directly with the schedule itself (such as timing), this field should describe the kinds of reports that are run according to this schedule and the audience for which the schedule is intended.
- The Recurrence list box contains a number of options to specify how often the schedule is triggered. You can choose from a broad range of intervals, all the way from Seconds up to Yearly. There's also a Run Once option to schedule one shot actions. If the flexibility offered by these options is still not sufficient, you can choose the Cron option to specify recurrence in the form of a `cron` string. The syntax of these strings is similar to that of the standard `cron` strings. You can find more information on cron expressions in Quartz at `http://www.opensymphony.com/quartz/wikidocs/CronTriggers Tutorial.html`.

  For all the available choices of the Recurrence option (except Cron), you can specify a Start Time by choosing the appropriate value in the hour, minute, and second list boxes.

  For all Recurrence options except Run Once and Cron, you can specify how often the schedule should be triggered. The appropriate widget to specify the value appears onscreen as soon as you select an option in the Recurrence list box. For example, for a Recurrence of seconds, minutes, hours, and days you can enter a number to indicate how many seconds, minutes, and so on are between subsequent executions of the schedule.

  The Weekly and Monthly recurrence options support more advanced possibilities for specifying the interval. For example, the Monthly and Daily Recurrence

options pop up a widget that allows you to define the schedule for each Monday of every month, and the Weekly Recurrence option allows you to specify to which weekdays the schedule applies.

Finally, you can specify a start date or a start and end date range. These dates delimit the period of time in which the schedule recurs.

After specifying the schedule, press OK to save it and close the dialog.

Attaching an Action Sequence to a Schedule

To actually schedule the action sequence, you have to attach it to an existing schedule. This is can be done in the Pentaho Administration Console or from the Pentaho User Console:

- In the Schedule Creator dialog shown in Figure 12-4, activate the Selected Files tab. From there, browse the solution repository and select the action sequence you want to schedule.
- In the Pentaho User Console, right-click the action sequence you want to schedule. In the context menu, choose Properties and the Properties dialog will open. In the Properties dialog, activate the Schedule tab. There, you can choose the appropriate schedule.

Monitoring

Two principal monitoring activities are logging and e-mail notification, as detailed in the following sections.

Logging

Kettle features a logging framework that is used to provide feedback during transformation and job runs. Logging is useful for monitoring progress during job and transformation execution, but is also useful for debugging purposes.

Logging is discussed in detail in Chapter 14. In this chapter, we discuss a few aspects of logging that should help you understand how logging can help you to monitor your Kettle jobs and transformations.

Inspecting the Log

In Spoon, the log can be inspected in the Logging tab in the execution pane below the workspace. The Logging tab is shown in Figure 12-5.

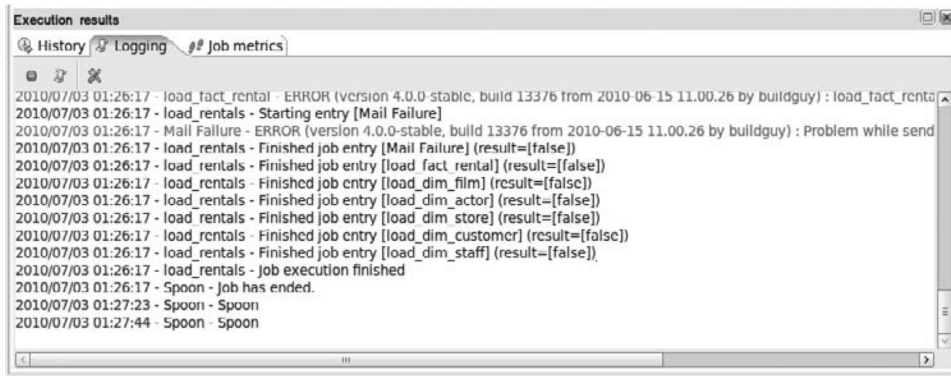


Figure 12-5: Inspecting the log from within the Spoon program

The Logging tab consists mainly of a text area where the actual log lines are shown, and also provides a few toolbar buttons to work with the log. Log lines that indicate an error are colored in red. For a quick inspection of only those log lines that indicate an error, click the toolbar button with the stop sign icon. This will pop up a dialog showing only those lines from the log that contain the word `ERROR`. .Next to the stop sign icon is a button to clear the log, and finally a button to pop up a dialog for setting some options regarding logging. In this dialog you can set the logging level (discussed in the next subsection) and whether to include a date/time at the start of each log line.

The log shown in the execution pane is mainly useful when developing transformations and jobs. It does not apply to scheduled jobs and transformations, because these are launched using command-line tools (Kitchen and Pan), which simply do not have a graphical user interface in which to display a log.

For Kitchen and Pan, you can use the `logfile` parameter to specify that the log should be stored to a particular file. If you do not specify a logfile on the command line, the log will be written to the standard output.

One thing to keep in mind is that Kitchen will only log information pertaining to the root job. If you also want to store the log from the jobs and transformations that are contained within the job, you have to configure that at the level of the individual job entries that make up the outer job.

In Spoon, you can configure the logging behavior on the job entry level by right-clicking the job entry and then choosing "Edit job entry." This opens a dialog that contains a tab called "Logging settings" where you can specify the location for the log file for that particular job entry. This is shown in Figure 12-6.

Note that in Figure 12-6, we're using variables to construct the name of the log file. In this case, we create the file in the directory where the outer job resides using the `${Internal.Job.FileName.Directory}` variable. For the actual file name, we use the `${Internal.Step.Name}` variable, and this will result in one separate log file per job entry, each having the name of the job entry that created it.

Sometimes, it is more convenient to have one big log file for a job and its parts. To achieve that, check the "Append logfile?" checkbox. This ensures the logs for all job entries end up in the file you specified.

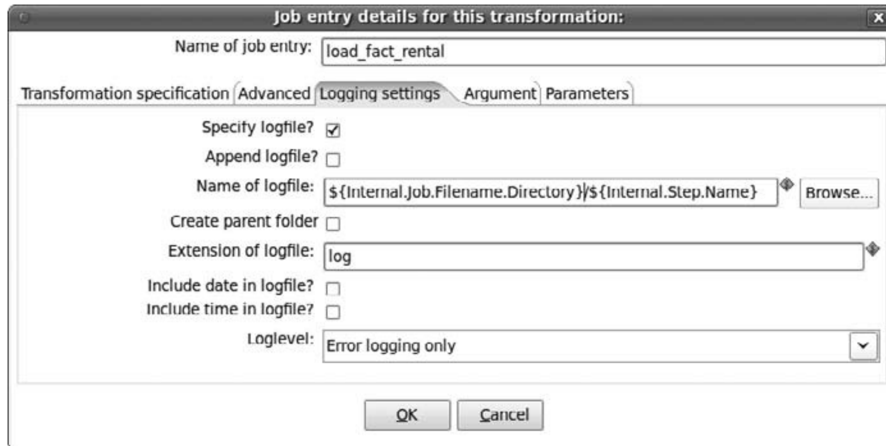


Figure 12-6: Configuring the log file location per job entry

Logging Levels

Kettle supports a number of logging levels to control the verbosity of the log. By increasing verbosity, the levels are:

- **Nothing:** Don't show any output.
- **Error:** Show errors only.
- **Minimal:** Use only minimal logging.
- **Basic:** This is the default basic logging level.
- **Detailed:** Give detailed logging output.
- **Debug:** For debugging purposes, very detailed output.
- **Rowlevel:** Logging at a row level, this can generate a lot of data.

As you can see, there are quite a number of logging levels. In practice, the most useful ones are:

- **Error:** Useful and appropriate in production environments for short-running jobs and transformations.
- **Basic:** Useful and appropriate in production environments for transformations and jobs that take some time to run. In addition to errors (which are also reported by the Error level and higher levels), this level also logs some information on the progress of the process, as some steps like the Text Input step use it to periodically log the number of rows read.
- **Rowlevel:** This is the most detailed log level. This level is only suitable for development and debugging purposes.

You should be aware that logging is not free: As the logging becomes more verbose, performance decreases. So you can't just select the most verbose level to be on the safe side; you should consider in advance which level of logging is required for the task at hand.

For jobs as well as transformations, the log level can be controlled by choosing the appropriate log level in a list box in the execution dialog. A screenshot of the execution dialog with the log level list box is shown in Figure 12-7.

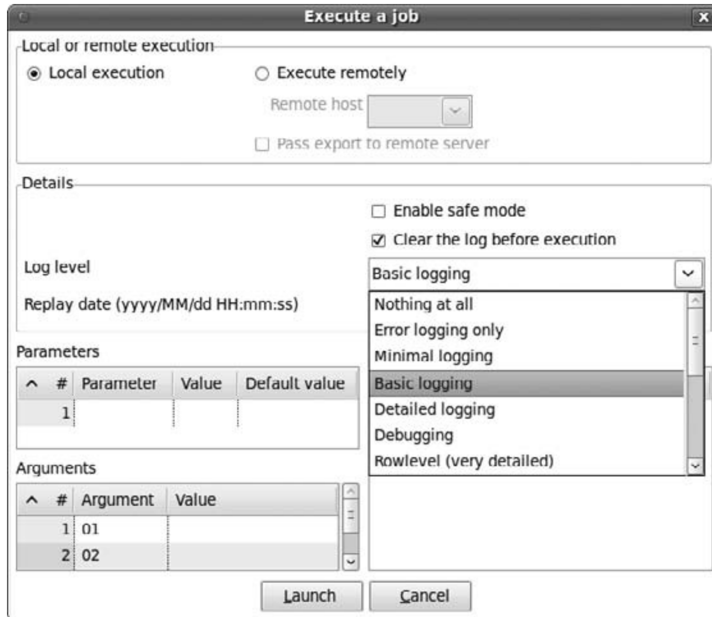


Figure 12-7: Controlling the log level in Spoon

For the command-line tools Pan and Kitchen, you can use the `level` command-line parameter to set the desired logging level.

Writing Custom Messages to the Log

You can write directly to the log from within a transformation or job using the "Write to log" transformation step or job entry respectively.

E-mail Notifications

A very simple method to get basic information about execution, completion, and errors is to build e-mail notifications into your Kettle jobs using job entries of the Mail type. You can find an example of basic usage of Mail job entries in the `load_rentals.kjb` job discussed in Chapter 4 and shown in Figure 4-6.

This load_rentals job contains two job entries of the Mail type: Mail Success, which is executed upon successful execution of the final `load_fact_rental` transformation,

and Mail Failure, which is executed in case any of the transformations that make up the job fails.

NOTE **You may wonder why the job includes two distinct e-mail entries, and not just one that is sent only in case of failure. The reason is very simple: If no e-mail notification is sent in case of success, it would be unclear what it means exactly when one does not receive any e-mail. It could be that the process completed successfully, or it could mean the job failed, but no mail was sent because the mail server was down, or there was a network error of some kind that prevented mail from being sent. By ensuring the job will always send mail, a missing e-mail automatically means something went wrong (perhaps the job didn't run at all, or perhaps the mail system is down).**

You could perhaps get by with one Email job entry, but because the properties and content of the e-mails are very likely to differ depending on whether the job failed or not, it is more convenient to configure a separate Email job entry for each case.

Configuring the Mail Job Entry

Configuration of the Mail step is not particularly difficult, although the number of configuration options may be a bit daunting at first. The configuration dialog contains four tabs.

Addresses Tab

In the Addresses tab you must specify at least one valid e-mail address in the "Destination address" property. Optionally, you can also configure CC and BCC addresses. In addition to the destination address, you must specify the "Sender name" and "Sender address" properties. These data are required by the SMTP protocol. You can optionally specify a "Reply to" address and some additional contact data such as the name and phone number of the contact person. For typical success/failure notifications, you would send notifications to the IT support staff, and specify details of a member of the data integration team as the sender. Figure 12-8 shows the Addresses tab page.

Server Tab

You must specify the details of the SMTP Server in the Server tab page, shown in Figure 12-9.

You are required to provide at least the host name or IP address of your SMTP server. Optionally, you can provide the port to use. By default, port 25 (default for SMTP) is used. In most cases, SMTP servers require user authentication. To enable authentication, select the "Use authentication?" checkbox and provide the user name and password in the "Authentication user" and "Authentication password" properties, respectively. More and more often, SMTP servers require secure authentication using a protocol such as SSL (Secure Sockets Layer) or TLS (Transport Layer Security). You can specify secure authentication by selecting the "Use secure authentication?" checkbox and choosing the appropriate protocol in the "Secure connection type" list box. Note that network communication for a secure authentication protocol generally employs another port.

For SSL, the default port is 465. Contact your local network or system administrator to obtain this information.

Figure 12-8: The Addresses tab page in the configuration dialog of the Mail job entry

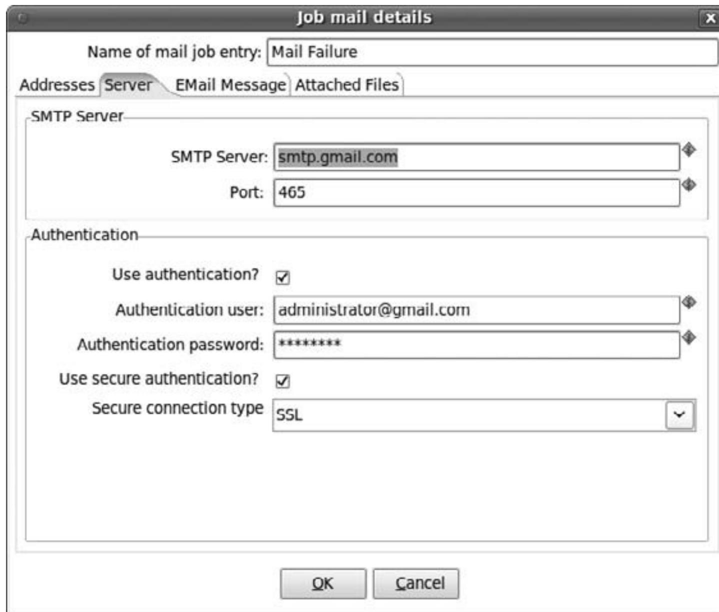


Figure 12-9: The Server tab page in the configuration dialog of the Mail job entry

EMail Message Tab

You can specify the actual message content on the EMail Message tab page, shown in Figure 12-10.

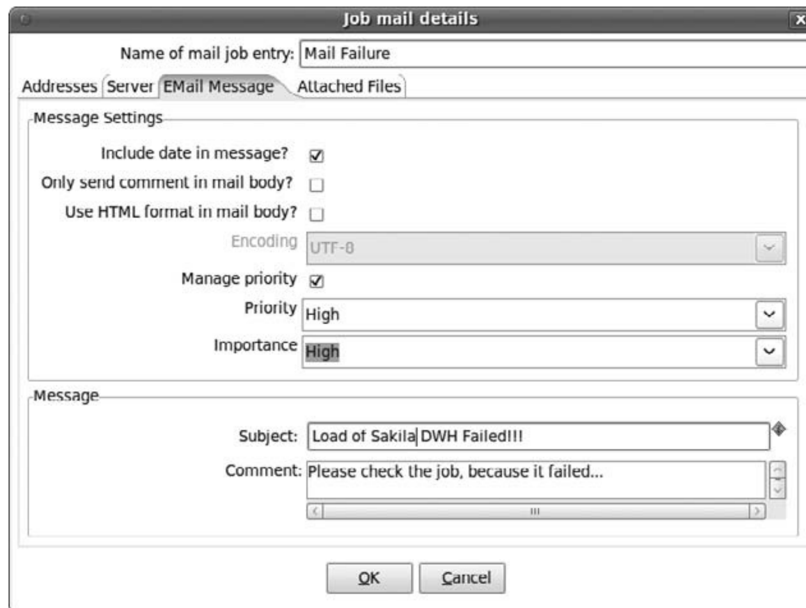


Figure 12-10: The EMail Message tab page in the configuration dialog of the Mail job entry

The message subject and body are specified in the Subject and Comment properties respectively. You can use text and freely include variable references for these properties. By default, Kettle includes a brief status report of the transformation in the message body, right after the content provided in the Comment property. To prevent this status report from being included, select the "Only send comment in mail body" checkbox. Optionally, you can check the "Use HTML in mail body" to send HTML-formatted e-mail. Some e-mail clients use message priority headers. If you like, you can select the "Manage priority" checkbox to enable this. When enabled, you can set the Priority and Importance properties.

Attached Files Tab

In the Attached Files tab, you can control whether files should be attached to the e-mail message and, if so, which ones. A screenshot of this tab is shown in Figure 12-11.

To enable attachments, check the checkbox labeled "Attach file(s) to message." You can then select one or more of the predefined items in the list labeled "Select file type." To select a particular type of file, simply click it. To select multiple file types, hold the Ctrl key and click all items you want to include. Note that when choosing for Log file, you need to configure logging for the individual job entries (as shown in Figure 12-6).

Figure 12-11: The Attached Files tab in the configuration dialog of the Mail job entry

Check the "Zip files to single archive?" checkbox to have Kettle store all files of the selected types into a single archive. If you choose this option, you must provide a filename for the resulting zip file in the "Name of zip archive" field.

Summary

In this chapter, we focused on two important aspects when running ETL jobs and transformations in a production environment, namely scheduling and monitoring. In this chapter, you learned about:

- Kitchen and Pan, the command-line interface for running Kettle jobs and transformations
- Operating system schedulers, such as `cron`, `at`, and the Windows Task Scheduler
- Scheduling Kettle transformations and jobs using the Quartz scheduler built into the Pentaho BI Server
- Controlling log file and verbosity in Spoon, Kitchen, and Pan
- Using e-mail job entries to send notification
- Including the Kettle log into e-mail notifications

CHAPTER

13

Versioning and Migration

In earlier chapters we covered designing, building, and deploying ETL solutions and showed how to use Kettle as a single user development tool. In reality, there are usually multiple developers working on a project, which calls for means to manage the different deliverables using a version control system. Another requirement in most projects is a separation of the development, test, acceptance, and production environments. The following ETL subsystems cover these requirements:

- **Subsystem 25:** Version Control System
- **Subsystem 26:** Version Migration System from development to test to production

In this chapter, we discuss the various reasons behind the use of version control systems, and take a close look at a few popular open source version control systems. After that, we discuss what Kettle metadata actually is and in what formats it can be expressed. Then we explain how you can do versioning and migration with Kettle metadata.

Version Control Systems

When you are developing a data integration solution by yourself, or maybe with a small team, it's easy to keep track of what's going on. It's also fairly easy to find out what changed and who did it. However, things change drastically when the stakes go up and a data integration solution goes into production. Things change even more when

you are working with a larger team or with a team that is geographically distributed. In those situations, you really want to keep your data integration services running smooth and stable. For that to happen, it's important to know when even a tiny change is made to any part of the solution.

Whenever a system runs into a problem (and sooner or later all systems do), the question is asked, "What did you change recently?" The answer is invariably "Nothing! Honest!" Changes occur in a solution in all shapes and forms, and can lead to problems in the system. It's possible that this happens because people are malicious or incompetent but you have to believe that this is the exception. Usually changes occur in response to requests from peers, users, project management, or the IT infrastructure in general. The answer to the question, however, is extremely important. If you know the answer, you can fix the problem with great speed and efficiency. If you don't know the origin of the problem, you find yourself debugging an unknown situation.

To prevent such problems, you need a system that can automatically keep track of all the changes in a project. In general, these systems are called *version control systems* (VCS) or *revision control systems*. In the past, VCS software was only available to the data integration development teams in large enterprises because the cost, thousands of dollars per seat, usually excluded usage in other situations. However, VCS is another area of software development where open source has made a big difference. Nowadays, you can find many varieties of version control systems. In the sections that follow, we list a few of the most popular systems. You can group them into two main types: file-based version control systems and content management systems (CMS).

File-Based Version Control Systems

File-based version control systems operate on individual files that are organized in a directory structure on a central server. You can access these files by checking out the whole directory. Because files on the server are updated, you need to execute update operations on your local file copies to get to the last versions. Individual files are updated leading to a new revision number for each file.

Organization

It's important to be able to label a whole project or directory structure and not just individual files. Because of this, it's common for a VCS to allow users to create branches or tags of a main development directory tree. This observation leads to the typical tree-style directory organization that you see in many projects:

- `trunk/`: The main development tree of the project.
- `branches/`: A branch is a set of files that belong to a single version of the project.
- `tags/`: A tag is a snapshot of the project files. It's simply a label applied for convenience.

If you look at the Kettle project source tree itself (`http://source.pentaho.org/svnkettleroot/Kettle/`), you will notice that the files there are organized in exactly

the same way. At the time of this writing, `trunk/` contains the recent development version of Kettle (the first milestone version of 4.1.0) while `branches/` contains all the stable versions that were ever released as open source from version 2.2.2 all the way to 4.0.0. The folders under `tags/` are usually temporary snapshots for development tests or quality assurance purposes.

Leading File-Based VCSs

The following list describes a few of the most popular open source file-based version control systems:

- **CVS**—One of the first open source version control systems to become popular was the Concurrent Versions System, or CVS. CVS, licensed under the GNU Public License (GPL), started as just a bunch of scripts almost 25 years ago. For a long time, it was the default choice for projects that needed a version control system. It allowed large groups of developers to work together efficiently.

  The main drawbacks of CVS include the lack of atomicity, limited Unicode support, limited binary files support, and expensive branching and tagging operations. It is also not possible to rename any file without the loss of the full history. In particular, the lack of atomicity could lead to a corrupted CVS repository on occasion. The more users work on a repository, the higher the chance is that something goes wrong and the higher the impact is of a corruption. Despite its shortcomings, using CVS beats not using any versioning system at all. Because it is known by many developers across the world it is still maintained and available on almost all operating systems. As such, CVS is still available as a choice on popular free hosting platforms such as SourceForge.net.

- **Subversion**—The shortcomings of CVS led to the development of a few new versioning systems. One of the most popular is currently Apache Subversion, (`http://subversion.apache.org/`). When the project started in 2000, the main project aim was to create a mostly compatible version of CVS that does not have the limitations of CVS. The developers achieved this for the most part, and that has led to a fast rise in popularity. At the time of this writing, Subversion is probably the most popular version control system with broad support of client tools, people, and operating systems.

  The main shortcomings of Subversion include a less than perfect implementation of the file rename operation. Like most file-based version control systems, Subversion uses the name of a file to handle most of the basic VCS operations. The Kettle project itself uses Subversion to handle the source code. Installing a Subversion server is well-documented and straightforward, so we recommend its use when you want to deploy your own VCS in your organization or for your data integration project. At the time of this writing, there is no integrated support for Subversion in the Kettle tools. However, because transformations and jobs can be saved as XML files, you can simply check your changes into a Subversion repository when you're done editing them in Spoon.

NOTE **When working with a Kettle file-based repository, saving as XML is the default option. With a database repository, you need to export to XML explicitly by selecting File ➪ Export ➪ To XML. Note that jobs get a** `.kjb` **extension and transformations get a** `.ktr` **extension, not** `.xml`**.**

- **Distributed version control systems**—The last few years saw the appearance of a number of new so-called *distributed version control systems* (DVCS). The main difference from client-server systems such as CVS or Apache Subversion is that with DVCSs there can be many repositories. Usually tools are available for the project lead developers to merge code from the various repositories. Open source projects that implement a DVCS include Git, Mercurial, Bazaar, and Fossil. Examples of large projects that use Git include the Linux Kernel and Android; Mercurial is used by Mozilla and OpenOffice.org. Examples of projects that use Bazaar are Ubuntu and MySQL. As you can see, this list includes projects with a large developer group and codebase.

Content Management Systems

Content management systems (CMSs) were primarily designed for server-based systems. Examples of these systems are Documentum, Sharepoint, Alfresco, Magnolia, Joomla, and Drupal. While it's certainly possible to download content from the server, the primary use is to allow people to work together. That usually includes workflow or business process management functionality. A lot of Business Process Management (BPM) and Workflow Management Systems (WMS) also have a CMS on board to manage the underlying content. Many CMSs make use of internal identifiers instead of filenames to operate. This allows for renaming and even the reorganization of complete directory structures without any problem. It's usually also possible to add extra information to stored documents such as descriptions, icons, file type information, and so on. All this makes most content management systems highly suitable for handling Kettle metadata.

Kettle Metadata

Before we dive into the available options for version control and migration when working with Kettle, it's helpful to explain what Kettle metadata is and why it is so well suited to use with the tools that'll be discussed later.

As mentioned earlier in this book, Kettle was designed to execute jobs and transformations described by the ETL metadata using an engine. Let's take a closer look at what we mean by *ETL metadata*. Metadata is a very broad term that in general means *descriptive data* or *data about data*. In the case of ETL metadata, we're describing *tasks* that an ETL tool needs to perform. Here are a few examples of things we describe in Kettle:

- The detailed layout of an input file
- The username to connect to a relational database
- The URL of a web service

- The name of a relational table where we want to store data
- The operations needed to read, transform and write the data
- The flow of execution of the various operations

As you can tell from the preceding list, all functionality you can perform is described in the form of metadata. The transformation or job engines inside of Kettle will interpret this ETL metadata at runtime and execute upon all the defined tasks.

ETL metadata can take many forms in Kettle. It usually starts to take shape when you enter the required information in Spoon, the graphical user interface. At that point, the metadata is graphical in nature, easy to interpret for ETL developers. The metadata is also present inside the Kettle software in the form of information stored in the Kettle Java API (see also Chapter 22). When you are done designing, your transformation or job can then be saved as XML or in another form in a metadata repository.

Kettle XML Metadata

XML was chosen to be used as a form of Kettle metadata because it is an excellent interface that supports Unicode. Because XML is, in general, not easily readable by human beings, it is not good as a programming language. On the other hand, Kettle XML files can be understood by those familiar with XML and the Kettle components because they directly reflect the various options and elements that are present in the user interface. Even so, manually editing an XML file is not something that people find enjoyable because of the often cryptic nature of the format. The good news is that the Spoon user interface is capable of defining 100 percent of the possible parts of the Kettle ETL metadata, so it's quite safe to leave Kettle XML generation in the realm of the software. The structure of transformations and jobs is fairly simple. By default, Kettle uses no attributes, only elements.

The two main metadata file types in a Kettle solution are Job files and Transformation files. These are easily recognizable by their extensions which are `.kjb` and `.ktr`, respectively. These are both XML files, meaning that they have a nested collection of elements that make up the job or transformation. Since you'll usually construct transformations first and later build the jobs to stitch the process to execute them together, we'll first cover the transformation XML, followed by the job XML.

Transformation XML

An XML file always has a single root element at the highest level of the nested hierarchy. The root element in a `.ktr` file is always called `<transformation>`. If it's not, it's not a Kettle transformation. The next level of the XML file, right beneath the `transformation` element, contains the following subelements (opening and closing angle brackets left off for clarity):

- `info`: Contains the transformation details such as the name and the description.
- `trans-log-table`: Contains the transformation log table settings.
- `perf-log-table`: Contains the performance log table settings.

- channel-log-table: Contains the channel log table settings.
- step-log-table: Contains the step log table settings.
- notepads: Contains all the notes that are shown in the user interface.
- connection, slaveservers, clusterschema, partitionschema: Describes (in order) database connections, the slave servers, the cluster schemas, and the partition schemas. More than one element of each can be present, for instance a transformation can contain multiple connections and slave servers.
- order: Contains the metadata about how the hops connect the steps in a certain order.
- hop: contains the connections between the different steps. Hops are a sub-element of order.
- step: Contains step-specific metadata. More than one step element can be present. While the other elements in the transformation XML are always the same, this element is always going to have a different structure. That is because all steps are different and because steps can be plugged into Kettle. However, for any given step type (tag is type) the format is always the same.
- step_error_handling: Contains the error sub-elements that specify the source and target step for the error handling, and the other error attributes like maximum number of allowed errors.

Note that most of these elements are optional. For instance, if there is no transformation log table defined, the element trans-log-table won't be available in the .ktr file.

Job XML

The root element of the XML file that defines a Kettle job is always <job>. The structure of a job file is slightly different than the structure of a transformation, but many sub-elements are very similar:

- name, description: Unlike a transformation, a job file doesn't contain an info element but lists name and description as first level elements.
- job-log-table: Contains the job log table settings.
- channel-log-table: Contains the channel log table settings.
- jobentry-log-table: Contains the step log table settings.
- notepads: Contains all the notes that are shown in the user interface.
- Connection and slaveservers: Describe the database connections and the slave servers. More than one element can be present.
- hops: Contains the metadata about how the job entry hops connect the job entry copies in a certain order.
- entries: Contains job entry–specific metadata. The entry elements listed here are always going to have a different structure. That is again because all job entries are different and can be plugged into Kettle. However for any given job entry type (tag is type) the format is always the same.

For the average Kettle user, there are not a lot of reasons why you might want to look at the Kettle XML, let alone modify something in the XML document. However, there are a few situations where you might want to modify or generate the XML yourself. The following sections provide a few examples that might give you some ideas.

Global Replace

Suppose you have a few hundred Kettle transformations and jobs stored in the form of XML. The extensions of these files are `.ktr` and `.kjb` respectively. Changing a single parameter in all these files by opening them up in Spoon is bound to be a highly repetitive task. It will also be easy to overlook occurrences of the parameter, making the task error prone. For example, suppose you found out after development that the production system is using a different database schema. Unfortunately, this value was hardcoded as `dwh` in all transformations. In that situation, you could write a shell script on UNIX, Linux, OS X, or even on Windows (using Cygwin for example) that replaces all occurrences of the corresponding `<schema>` tag in all transformations:

```
# Prevent loss of information : stop after an error!
#
set -e

# Loop over all the transformation XML files:
#
for file in *.ktr
do

  # File names can contain spaces: quote them!
  #
  < "$file" \
    sed 's/<schema>dwh<\/schema>/<schema>${DWH_SCHEMA}<\/schema>/g' \
  > TEMPFILE

  rm "$file"
  mv TEMPFILE "$file"

done
```

The preceding script (`globalreplace.sh`, available in the download files on the book's website), replaces all occurrences of `<schema>dwh</schema>` with `<schema>${DWH_SCHEMA}</schema>`. The result of the execution of the script will be that the schema will be configurable with a variable in all transformations. For someone who knows a scripting language such as Shell (bash), awk, Perl, or Ruby, this can be a simple and quick way to correct small problems in your Kettle metadata. The command that does all the magic here is `sed`, short for *s*tream *ed*itor. Sed is one of the most powerful utilities for modifying text files in a .nix environment. An excellent introduction and tutorial can be found on `http://www.grymoire.com/Unix/Sed.html`.

Kettle Repository Metadata

Since version 4 of Kettle, repository types can be plugins. As a direct result of that, Kettle metadata can take any possible form. The `Repository` interface contains all the required methods to serialize Kettle transformations and jobs as well as shared objects like database connections and slave servers. To allow the ETL developers to categorize their objects, Kettle repositories also contain directories. Let's take a look at the characteristics of the current repository type implementations.

The Kettle Database Repository Type

The Kettle database repository type serializes Kettle metadata to a relational database schema. This schema uses tables to contain metadata related to the various components in transformations and jobs. For example, there is a table called `R_TRANSFORMATION` that contains the name, description, and extended description of a Kettle transformation. The steps are stored in a table called `R_STEP`, and so on. Each table contains a unique identifier that allows all objects to be linked together.

The database repository allows ETL users to work together. It does so by allowing a relational database to be used as a central Kettle metadata source. The creation of the repository is done by Kettle in Spoon and this process even supports upgrades. Select Tools ➪ Repository ➪ Connect, to begin creating a repository, as shown in Figure 13-1.

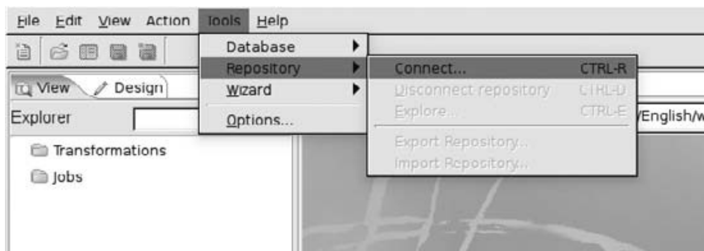


Figure 13-1: Connecting to a repository

This will present the Repository Connection dialog, shown in Figure 13-2, which you can also see when Spoon starts up.

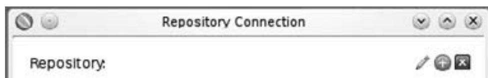


Figure 13-2: Managing repositories

Clicking the + icon will display a list of available repository types that allows you to define new repository connections. For this example, you want to define a Kettle database repository so select that option. This brings up the dialog shown in Figure 13-3.

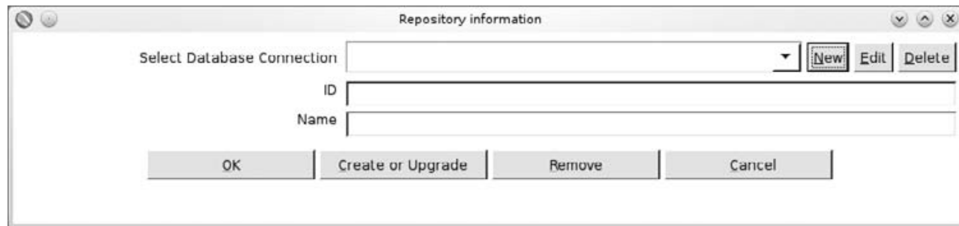


Figure 13-3: The Kettle database repository dialog

In this dialog, you can define the connection to the database schema that contains the Kettle repository. You can create a new database repository on the selected database connection. This will create all the required database tables and indexes to make the repository work.

While the Kettle database repository does the job, it has a few drawbacks:

- It can't store multiple revisions of a transformation or job.
- It relies heavily on the database to properly lock and unlock tables to prevent work getting lost.
- There is no notion of team development, and you can't lock transformations or files for exclusive private use.
- The security system is proprietary and simple.

The Kettle File Repository Type

The Kettle file repository type simply uses the existing Kettle XML serialization to store documents. It uses the Apache VFS driver (see Chapter 2) to access Kettle metadata. That means you are not limited to a local disk. You can, in fact, also use `.zip` archives, FTP, and HTTP locations.

Creating a file repository is easy; simply follow the instructions in the previous section and select the Kettle File Repository type when asked. You can point your file repository to an existing folder with Kettle transformations and jobs to see how it works.

Because the serialization is to XML, this repository type contains the same drawbacks as XML as well:

- You don't know if objects (transformations, jobs, databases, and so on) are referenced in another job or transformation. This means you can't safely delete or rename anything.
- There is no version history.
- Team work is hard, even if you use shared folders, because there is no locking possibility at all.
- There is no security layer except for the file system security of the operating system.

The Kettle Enterprise Repository Type

When the Pentaho team went looking for a repository for its enterprise edition version of Kettle (Pentaho Data Integration EE) they wanted to solve the various drawbacks present in both XML serialization and the existing database repository. The drawbacks of the popular file-based VCS also had to be taken into account. More specifically, the drawbacks with respect to file renaming or file moving make it hard to organize your enterprise metadata repository to your liking. It's also harder to make a decent security layer on top of a VCS.

To address those problems, Pentaho looked in the direction of a content management system for use as an enterprise repository. Currently, the enterprise repository runs on the Pentaho Data Integration Server, which includes an assortment of resources: an Apache Tomcat server with a scheduler on board, a series of web services to act as a Carte replacement, a security layer, and Apache Jackrabbit. Jackrabbit (`http://jackrabbit.apache.org/`) is an open source CMS implementation for Java implementing the Java Content Repository (JCR) specification. This functionality includes version management, security, locking, referencing by ID, metadata, querying, renaming, and reorganization of your documents. This functionality of Jackrabbit is exposed on the data integration server by a set of web services that will allow Pentaho to easily upgrade Jackrabbit itself and possibly add new functionalities in the future without any required changes for the clients that talk to the Pentaho BI server. There are plans to have the Pentaho BI server and other tools in the suite support the repository in the near future. However, at the time of this writing, Kettle (Pentaho Data Integration EE) is the only client for the enterprise repository.

Managing Repositories

As explained earlier, a Kettle repository stores all the metadata that makes up an ETL solution. In most cases there will be more than one repository: one for development, one for test, one for acceptance, and one for production. Usually each developer has his or her own development repository, and a centralized repository is created to assemble the complete solution. Since the Kettle repository plays such a central role during the project lifecycle, you need to make regular backups using the tools for exporting repositories described in the following sections. It's also possible to import a complete repository, which is helpful for migrating solutions from test to acceptance to production.

Exporting and Importing Repositories

The primary use of the export and import functionality, available under the Tools/Repository Export menu in Spoon, is to back up a repository. This functionality works by serializing the contents to an XML file. This means you get a single XML file with the root element `<repository>` followed by a transformations section with multiple transformations and a jobs section containing multiple job elements, representative of the repository's physical contents.

You can also use the "Export repository" job entry for this, as shown in Figure 13-4.

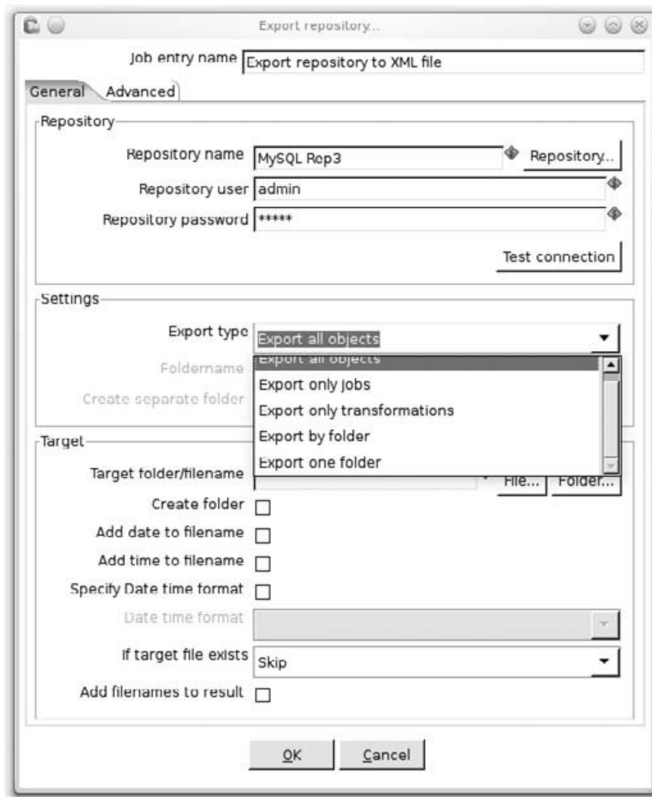


Figure 13-4: Exporting a repository in a job

With this job entry, you can automate the regular backup of your repository from within Spoon yourself. It has the distinct advantage of being able to export individual folders.

Finally, you can also use the Pan utility to export a repository. For example, you can use the following command:

```
sh pan.sh -rep="Production" -user="admin" -password="admin"
 -exprep="/tmp/export.xml"
```

Upgrading Your Repository

When there is more metadata that needs to be stored in a new version of Kettle, you will need to upgrade your repository. The golden rule for doing this is the same for all software upgrades: *Make a backup first!* Preferably you do this with both of the following two methods:

- Get a complete export of the repository to an XML file (see above).
- Back up the repository database itself. In the case of a Kettle Database Repository and the Pentaho Enterprise Repository you should take a complete dump or export of the relational database where the metadata is stored.

Once that is done, you can try to run the upgrade functionality of the Kettle database repository dialog or follow the specific upgrade guide for the enterprise repository. The alternative is to create a completely new repository and perform an import. This has the advantage of leaving the existing repository untouched. See the "Upgrade Tests" section of Chapter 11 for more details.

Version Migration System

One of the critical aspects of the lifecycle of a software project is migrating (all or parts of) a solution from a developer machine to a test, acceptance and finally, production system. And the first question that arises then is: how do you move transformations and jobs from development to production? As you can probably tell by now, this depends on the way you are persisting your Kettle metadata. In the sections that follow, we cover possible ways to work with XML files and repositories for managing your project's lifecycle.

Managing XML Files

As mentioned earlier in this chapter, your best bet for keeping track of your transformation and job files is to check them into your version control system (VCS) of choice. That way, you can nicely do your development on one or more workstations with your team. Put all your development work in the `trunk/` folder as described earlier.

WARNING **If you are doing heavy development with a lot of changes, instruct your team to do frequent updates of the local copies to keep conflicts to a minimum later. By *conflict* we mean that you are trying to check in a file that was already updated earlier by someone else. Since overwriting changes made by someone else is not something you want to do, it's usually something the VCS is going to complain about. These conflicts are unfortunately a direct consequence of having multiple copies of the same file.**

When it comes time to test your data integration solution, you can tag the trunk (copy it to the `tags/` folder). This tag can then be checked out on a test server. When you are done fixing things and you are ready to do user acceptance, you tag it again and check the tag out on the user acceptance server. Finally, when you are ready to go into production you can branch the user-accepted solution and set it in production.

The scenario sketched here is pretty straightforward for moving solutions up the chain. One of the main benefits of using a VCS, however, is the possibility to roll back changes as well. When moving from acceptance to production, there's always a risk that some unforeseen incident happens. Having a previous stable version at hand will save the day in such a case.

Managing Repositories

The easiest way to have development, testing, acceptance, and production environments in place when you are using the Kettle database repository is to create multiple repositories, one for each environment.

WARNING **Unless you use the Enterprise Edition, the Kettle database repository doesn't have locking and versioning capabilities and is not suited for team development. Each developer should use a separate repository.**

There are several ways to advance changes from one repository to another. The first is to simply do an export and an import. That is probably the quickest way to work initially. The second method of passing changes is to open a transformation or job in Spoon, disconnect from the current repository and open a connection to another. Then you can save your file in that repository. If you're working with multiple developers but want to merge different parts of a solution into a single test repository, a good approach would be to appoint one of the team members as the gatekeeper to do all the check ins for the test repository.

Parameterizing Your Solution

The trick to keeping the work of advancing transformations and jobs to production to a minimum is to parameterize your solution. This means that you have to be vigilant about using variables and parameters for all things that are different in another environment or could be different in the future. For example, watch out with database connections: Always make sure to use variables for the values you enter in a database connection. Try to avoid the temptation to quickly hardcode a hostname, a database, or a username because before you know it a transformation with that connection in it is going to end up on a test or production system and seriously foul up things. Also be careful with file locations. It is tempting and convenient to use relative file paths such as `${Internal.Transformation.Filename.Directory}`. Keep in mind, however, that when you are executing a transformation or job remotely or use a repository, this variable has no meaning anymore. Make sure to create separate variables for all meaningful file locations that you might have.

Once your complete solution is using variables, simply define a separate set of values for each environment and you're done. You can use the `kettle.properties` file described elsewhere in this book. You can also create your own file and set the variables in the first job entry of the main jobs of your solution. There are other approaches you can use as well, such as using a parameter table. Working with a separate database table can add extra flexibility, such as having the option to use `valid from` and `valid to` dates for your parameter values. A simple example of a parameter table is shown in Table 13-1.

Knowing how to pass the values from a database parameter table to a Kettle transformation has another benefit as well. Organizations might already have another ETL tool in place and are looking for a replacement. Many of the existing solutions already have a parameter table because not all ETL tools have good support for environment variables and parameters. In this case it might be useful to keep the existing tables and parameters and reuse them in a Kettle solution.

Table 13-1: ETL Parameter Table

ID	ENVIRONMENT	PRM_ NAME	PRM_ VALUE	VALIDFROM	VALIDTO
1	dev	dbhost	sagitta	01/01/1990	12/31/2999
2	tst	dbhost	virgo	01/01/1990	12/31/2999
3	acc	dbhost	scorpio	01/01/1990	12/31/2999
4	prd	dbhost	aquarius	01/01/1990	06/30/2010
5	prd	dbhost	capricorn	07/01/2010	12/31/2999
6	dev	uname	usr123	01/01/2009	12/31/2999
7	prd	uname	usr456	01/01/1990	12/31/2999

So how do you get these values into Kettle variables? That's pretty straightforward, actually. Take a look at the example in Table 13-1 again. The only environment variable you need is `environment`. The value for `environment`, combined with the `sysdate`, suffices to retrieve all parameters with their values for a particular environment. In fact this is a very common scenario where so called key/value pairs are used.

NOTE **Dealing with key/value pairs is covered in depth in Chapter 20.**

For a production run, the Kettle environment variable gets the value `prd` and the "Input table" step needed to retrieve the values uses the following SQL query:

```
SELECT  prm_name
,       prm_value
FROM    kettle_param
WHERE   environment  = '${environment}'
AND     validto     >= sysdate()
```

This will return the two rows of data from the example parameter table that are displayed in Table 13-2.

Table 13-2: Parameter Query Results

PRM_NAME	PRM_VALUE
dbhost	capricorn
uname	usr456

Now you need to find a way to feed this information into the Set Environment Variables step. This step can read the value from a field name delivered by a preceding step and put in into a Kettle variable. The problem, however, is that the data must be in the form displayed in Table 13-3.

Table 13-3: Required Environment Variables Data Structure

DBHOST	UNAME
`capricorn`	`usr456`

To accomplish this, you need one extra step: the "Row denormalizer." This will let you transform the data as displayed in Table 13-2 into the data as displayed in Table 13-3. Figure 13-5 shows what the "Row denormalizer" settings should be in this case.

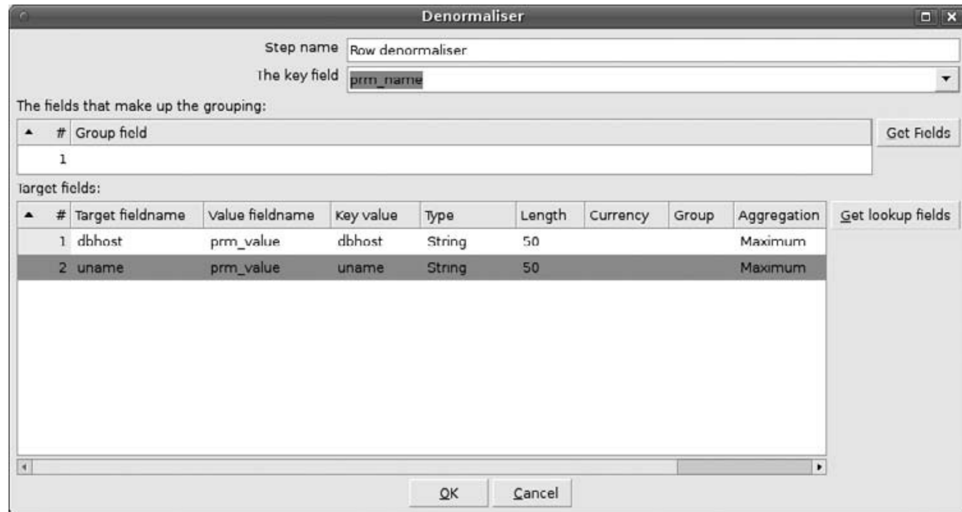


Figure 13-5: Row denormalizer step

As Figure 13-5 clearly shows there's no Group field required to unpivot the data. Figure 13-6 shows the completed transformation with the three steps needed to load parameter values in a database table into Kettle variables.

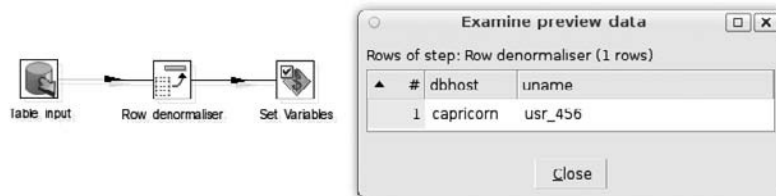


Figure 13-6: Completed set variables transformation

The `table_params` transformation is also available from the book's companion website. Remember that if you want to use a solution like this, you need to keep it as a separate transformation that needs to be executed prior to other jobs or transformations.

Summary

In this chapter we covered two important ETL subsystems: subsystem 25 about the version control system, and subsystem 26 about the version migration system. The following topics were discussed:

- What version control systems are and how to turn them to your advantage for you own data integration projects
- The Kettle metadata for jobs and transformations
- The different repository types that Kettle can work with
- How to set up and upgrade a Kettle repository
- The two methods of promoting solutions from development to test to acceptance to production
- How to parameterize a Kettle solution using properties files and parameter tables

CHAPTER

14

Lineage and Auditing

When you create complex data integration solutions with a lot of Kettle jobs and transformations, you may find it challenging to keep track of the results. At the same time, it is extremely important to keep an audit trail to identify and diagnose problems after they occur. It's important to know what exactly was executed, where errors occurred, and how long it takes to execute a job. In this chapter, we show you how to perform all these tasks and more on the topics of lineage, impact analysis, and auditing.

As you may recall from the ETL subsystem overview in Chapter 5, subsystem 29 covers the lineage and dependency analyzer. Lineage looks "backward" to the process and transformation steps that created the result data set you are analyzing, whereas impact analysis is executed from the start of the process. Roughly speaking, impact analysis is done from the source, and lineage analysis is done from the target.

For both impact and lineage analysis you need metadata, and this chapter begins by showing how you can use a transformation to read the Kettle metadata. This allows you to automate the extraction of lineage information so you can make it part of your nightly batches or even share this lineage information with third-party software.

Next, you'll learn more about the various kinds of lineage information. You will see where field level lineage information and database impact analysis can be obtained in Spoon, and learn how to write a transformation to get the results of a database impact analysis so you can make it part of your batch processes.

Finally, you'll get information about Kettle logging and operational metadata, which actually touches upon more than one ETL subsystem. The subsystems that are partly covered are:

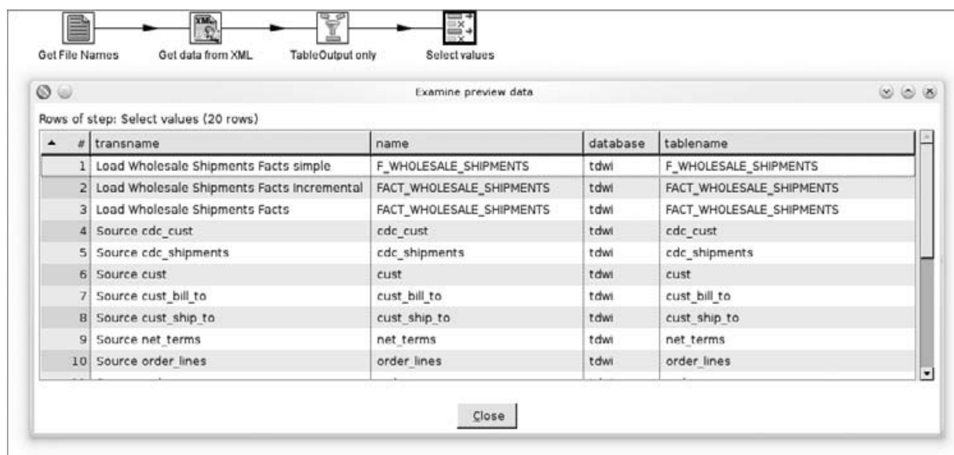
- Subsystem 6, the audit dimension assembler
- Subsystem 27, the workflow monitor
- Subsystem 33, the compliance reporter

Batch-Level Lineage Extraction

Keeping track of all the data streams in an organization becomes more difficult as more systems are added to the infrastructure. With the proliferation of applications backed by relational databases, you automatically get an increase in data transfers taking place. In part, data integration tools such as Kettle help solve this problem by making it easy to track down sources and targets in the user interface. That being said, a data integration tool in a large organization is typically only responsible for a small part of the data transfers. Because of this, it's important that the data integration tool itself can report to third-party systems that keep an inventory of all the data flows. If it can't do that, then it is part of the problem, not part of the solution.

To solve this challenge with Kettle on a transformation level, we offer a small example that searches for all the Table Output steps in a set of transformations in a directory, and then reports the name of the transformation and the step as well as the used database and table name.

Figure 14-1 shows a transformation that does exactly that and some sample output (the `extract-transformation-metadata.ktr` file is included in the download folder for this chapter).



#	transname	name	database	tablename
1	Load Wholesale Shipments Facts simple	F_WHOLESALE_SHIPMENTS	tdwi	F_WHOLESALE_SHIPMENTS
2	Load Wholesale Shipments Facts incremental	FACT_WHOLESALE_SHIPMENTS	tdwi	FACT_WHOLESALE_SHIPMENTS
3	Load Wholesale Shipments Facts	FACT_WHOLESALE_SHIPMENTS	tdwi	FACT_WHOLESALE_SHIPMENTS
4	Source cdc_cust	cdc_cust	tdwi	cdc_cust
5	Source cdc_shipments	cdc_shipments	tdwi	cdc_shipments
6	Source cust	cust	tdwi	cust
7	Source cust_bill_to	cust_bill_to	tdwi	cust_bill_to
8	Source cust_ship_to	cust_ship_to	tdwi	cust_ship_to
9	Source net_terms	net_terms	tdwi	net_terms
10	Source order_lines	order_lines	tdwi	order_lines

Figure 14-1: Extracting lineage information from a transformation

This transformation works by reading a list of transformation filenames from a directory. You can use the wildcard `.*ktr$` for this. Then you parse the content of the transformation XML by looping over the `/transformation/step` elements. For more information on the structure of transformation and job XML, see Chapter 13. The information you need is stored in XML elements (nodes) and can be retrieved with the following absolute and relative XPath expressions:

- **Transformation name:** `/transformation/info/name`
- **Step name:** `./name`

- **Step type:** `./type`
- **Database connection name:** `./connection`
- **Name of the target database table:** `./table`

All that is then left to do for the exercise is to filter out the `TableOutput` step type and remove those fields you don't want.

This simple exercise demonstrates that it's actually quite easy to extract all sorts of lineage information and impact analyses simply by looking at the Kettle metadata in XML format.

NOTE **If you are using repositories, note that there are several ways to export repositories to XML. See Chapter 13 for more information on this topic.**

The act of exposing the metadata in a generic fashion allows you to make it available for all sorts of purposes. For example, the extracted metadata can be stored in a relational database for further analyses with other tools in the Pentaho suite such as reporting and analyses. You could also monitor the quality of the ETL solution itself simply by looking at the various metadata settings. For example, you can verify if a transformation or job has the appropriate logging configuration, if descriptions are available in the transformation, or if there are any notes available.

On a job level, it is also possible to extract interesting metadata. Here are a few suggestions:

- List all the source FTP systems by looking at the FTP job entries. This gives you a list of all the systems where files are being retrieved from.
- Take a look at the settings in the Mail job entries and see if any hardcoded settings such as e-mail addresses are being used.
- List all the jobs where files are being copied or deleted.

All these challenges are answered in almost the same way as demonstrated in the example `extract-transformation-metadata.ktr`. The only differences are that you are reading job XML files and that the information is stored in different nodes.

Lineage

In the context of a transformation, *lineage* means that you want to learn where information is coming from, in which steps it is being added or modified, or in which database table it ends up. This section covers the various lineage features in Kettle.

Lineage Information

In a Kettle transformation, new fields are added to the input of a step in a way that is designed to minimize the mapping effort. The rule of thumb is that if a field is not changed or used, it doesn't need to be specified in a step. This minimizes the maintenance

cost of adding or renaming fields. The row metadata architecture that the developers put in place not only allows you to see which fields are entering a step and what the output looks like, but it can also show you where a field was last modified or created.

Take a look back at Figure 14-1. Now open the transformation (`extract-transformation-metadata.ktr`), right-click on the "Get data from XML" step and select the "Show input fields" option. This will list all the fields that provide input for this step. In this case, the output will list all sorts of information regarding the file names that are retrieved. The Step origin column will in each case indicate the Get File Names step as its source. When you select the "Show output fields" option from the same menu you will get a number of extra fields that originate from the "Get data from XML" step, as indicated in Figure 14-2.

NOTE **You can mouse over a step and press the spacebar to see the output fields for that step.**

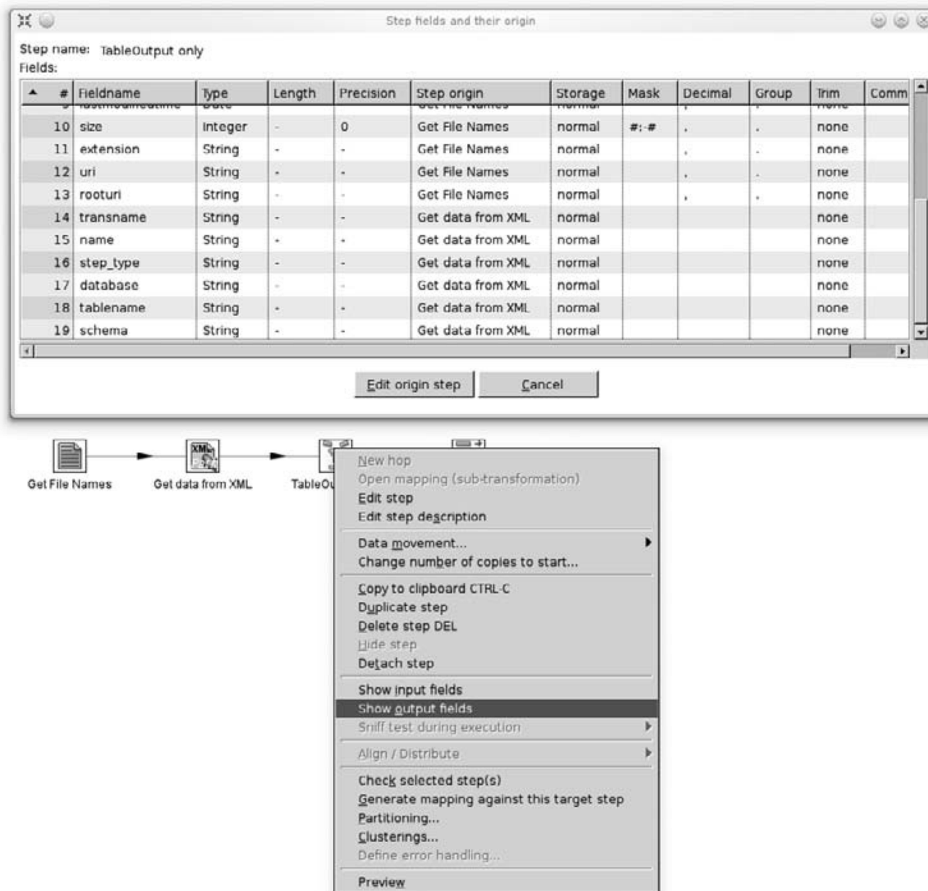


Figure 14-2: Retrieving the output fields of a step

Because no fields are being removed and no data is modified in the "TableOutput only" step, the output fields are exactly the same as for the previous step. The file names from the very first step are in fact passed along until they are explicitly removed by the "Select values" step at the very end.

This simple principle of passing fields along makes it easy to change field names or add steps at the front of the transformation with minimal impact to the other steps. Obviously that does not mean that you don't have to explain to Kettle how fields are used. In the steps themselves, you still have to specify which fields to use as input if this is required. For example, in the "Get data from XML" step you have to indicate that you want to read file names from the `filename` field. In most steps, lists, drop-down buttons, or helpers are available to make this process painless.

Impact Analysis Information

Sometimes it can be interesting to know all the places where a transformation is using a database. For example, you might be wondering if the same database table is used in two different steps in the same transformation. The database impact analysis offered in Spoon on a transformation level lists all that and more. Figure 14-3 shows output of the impact analysis of a transformation that loads data into a fact table.

Figure 14-3: Displaying database impact information

The information listed is extracted from the individual steps in a transformation. Each step that uses a database connection can add rows of database impact information. When possible, the steps will include information down to the field level.

If you want to make this information available outside of the context of the user interface, you can do so because this functionality is implemented on an API level.

That opens up a lot of possibilities. To keep things simple, you can use a few lines of JavaScript (see also the `db-impact.ktr` transformation in the download folder):

```
var transMeta = new Packages.org.pentaho.di.trans.
  TransMeta(filename);
var transname = transMeta.getName();
var impact = new Packages.java.util.ArrayList();
var error = "";

try {
  transMeta.analyseImpact(impact, null);
} catch(e) {
  error = e.toString();
}

for (i=0;i<impact.size();i++) {
  var dbi = impact.get(i);
  var newRow = createRowCopy(
    getOutputRowMeta().size());
  var rowIndex = getInputRowMeta().size();

  newRow[rowIndex++] = transname;
  newRow[rowIndex++] = dbi.getStepName();
  newRow[rowIndex++] = dbi.getTypeDesc();
  newRow[rowIndex++] = dbi.getDatabaseName();
  newRow[rowIndex++] = dbi.getTable();
  newRow[rowIndex++] = "N";
  newRow[rowIndex++] = error;


  putRow(newRow);
}

var ignore = "Y";
```

This script loads the transformation metadata and uses the Kettle Java API to extract all sorts of information such as the type of database impact and the names of the transformation, step, database, and table. The script is even capable of handling any errors that might occur when the impact is analyzed by the transformation.

The script then generates N rows for every file name it receives on input. To make that happen, it uses the `putRow()` function that exposes the method with the same name of the underlying step. You can find more information on `putRow()` in Chapter 23.

The information that is obtained can be put on a report or stored in a relational database for further analysis. That would, for example, make it possible to list all the transformations that make use of a certain database table in one way or another. That information can then be used when changes are made to that table.

Figure 14-4 shows a few lines of output from the example.

Examine preview data

Rows of step: Filter rows (439 rows)

#	filename	transname	stepname	type	db	table
117	/parking/Tl	Source cdc_cust	cdc_cust	Write	tdwi	cdc_cust
118	/parking/Tl	Source cdc_cust	cdc_cust	Write	tdwi	cdc_cust
119	/parking/Tl	Source cdc_cust	cdc_cust	Write	tdwi	cdc_cust
120	/parking/Tl	Source cdc_shipments	cdc_shipments	Truncate	tdwi	cdc_shipments
121	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
122	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
123	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
124	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
125	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
126	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments
127	/parking/Tl	Source cdc_shipments	cdc_shipments	Write	tdwi	cdc_shipments

Close

Figure 14-4: Externalizing database impact analysis

Logging and Operational Metadata

The Kettle developers have always put a lot of effort into making sure that the ETL developers and operators get good logging capabilities. In this section, you can read all about the Kettle logging architecture and how to configure it.

Logging Basics

Most components in Kettle can output logging information in the form of lines of text. For example, when a step finishes, a line is generated to indicate this event:

```
2010/06/18 10:36:29 - Step name.0 -
  Finished processing (I=0, O=0, R=0, W=25, U=0, E=0)
```

You can recognize three main parts in the log lines:

- The date and time
- The name of step followed by a period and the step copy number
- The logging text
- These parts are separated by a space, a dash and a space.

When you execute a transformation or a job, you can choose the logging level at which you want to run. Depending on the level you pick, more or fewer log lines will be generated. Here are the available logging levels in Kettle:

- **Rowlevel:** Prints all the available logging information in Kettle, including individual rows in a number of more complex steps.

- **Debugging:** Generates a lot of logging information as well, but not on the row level.
- **Detailed:** Allows the user to see a bit more compared to the basic logging level. Examples of extra information generated include SQL queries and DDL in general.
- **Basic:** The default logging level; prints only those messages that reflect execution on a step or job-entry level.
- **Minimal:** Informs you of information on only a job or transformation level.
- **Error logging only:** Shows the error message if there is an error; otherwise, nothing is displayed.
- **Nothing at all:** Does not generate any log lines at all, not even when there is an error.

The logging level can be set in the job and transformation execution dialogs in Spoon. It can also be specified on the command line when you use Pan or Kitchen, with the `-level` option. Finally, you can also change the default logging level by going to the Execution Results pane of the transformation or job you are editing in Spoon. From the Logging tab, select the toolbar icon (the crossed wrench and screwdriver) to display the dialog shown in Figure 14-5.

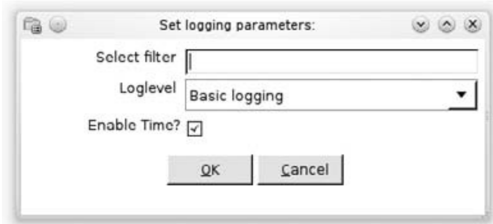


Figure 14-5: Setting the logging parameters

The interesting feature of this dialog is that you can specify the text to use as a filter. If you do so, only those log lines that contain the specified text will be retained. You can use this if you are looking for specific values in a detailed logging level. For example, if you specify the name of a step, only the lines that contain the name of the step will be included in the log.

Logging Architecture

Since version 4 of Kettle, all log lines are kept in a central buffer. This buffer does not simply store the text of the log lines as described in the preceding section. These are the pieces that are stored:

- **The date and time:** This component allows the date-time to be colored blue in the logging windows in Spoon.

- **The logging level:** This allows Spoon to show error lines in red in the logging windows.
- **An incremental unique number:** Kettle uses this for incremental updates of the various logging windows or log retrieval during remote execution.
- **The logging text:** The actual textual information that is generated to help the developer see what is going on.
- **A log channel ID:** This is a randomly generated (quasi) unique string that identifies the Kettle component where the log line originated.

Keeping logging lines in memory has traditionally been one of the most common causes for running out of memory. When you execute a job or transformation with a high logging level, a lot of text is generated. If you then ask Kettle to store this information in a logging table somewhere, you are asking to keep all this in memory and you can run out of memory as a result. At the same time, it is unlikely that any ETL developer is going to look at hundreds of thousands of lines of debugging information. Usually you are only interested in the last thousand rows before an error occurred. Because of this, and also for the reasons described in Chapter 18 on the topic of real-time data integration, you can limit the amount of rows kept in the central log buffer by using the parameters described in the following sections.

Setting a Maximum Buffer Size

The first limit you can put in place is simply to set the number of rows kept in the log buffer. You can do this in the Options dialog in Spoon. Use the option "Maximum nr of lines in the logging windows". Please note that it will affect logging in Spoon only!

Another option is to set the `KETTLE_MAX_LOG_SIZE_IN_LINES` environment variable. You can simply include this variable in the `kettle.properties` file to activate this feature. (See also the "Logging" section in Chapter 18 to read about the real-time data integration implications.)

Finally, you can also specify this limitation in the Carte configuration file. For example, you can include the following lines in the slave configuration XML file to limit the size of the central log buffer to 10,000 lines:

```
<!-- Prevent out of memory by only keeping 10,000
     rows in the central log buffer
-->
<max_log_lines>10000</max_log_lines>
```

Setting a Maximum Log Line Age

A more intelligent way to solve the problem of running out of memory is to specify the maximum age of a log line. Because you know the time when the logging record was added to the buffer, you can calculate the age as well. Similar to setting the buffer size, you can specify this parameter in the Spoon Options dialog under the option "Central

log line store timeout in minutes." Again, remember that this option affects execution in Spoon only.

The environment variable that will define this age for all Kettle tools is called `KETTLE_MAX_LOG_TIMEOUT_IN_MINUTES`. To set this variable in the `kettle.properties` file, from the Edit menu, select "Edit the kettle.properties file" in Spoon version 4 or later.

You can also add these lines to a Carte slave configuration XML file to limit the age of a log line in the central buffer:

```
<!-- Discard log lines if they are in the
     buffer for more than 2 days -->
<max_log_timeout_minutes>2880</max_log_timeout_minutes>
```

Log Channels

Before Kettle version 4, all the logging text was simply written to the logging back end (read about Apache Log4J at `http://logging.apache.org/log4j`). This caused all sorts of interesting problems ranging from the mixture of logging text when you run transformations or jobs in parallel on the same machine to excessive memory consumption when individual components kept logging text in memory for storage in logging tables (see below). Once the conversion to text was made, it was also not always simple to filter out lines pertaining to a certain step or database.

To counter these problems, a single logging buffer was created in Kettle version 4, as described in the earlier section "Logging Architecture." When you now execute a job or transformation, each component (job entry, step, transformation, and database connection) creates a separate log channel with a unique ID that allows you to identify where the logging line originated. Because Kettle also keeps track of the parent log channel ID of the components, an execution hierarchy is kept internally. Because this hierarchy is kept separately it is possible for Kettle to retrieve log lines pertaining to each individual component as well as its children.

For example, suppose you execute a job in Spoon and a long-running transformation is executing. You'll see a small blue icon drawn over the active job entry. You can then open the associated transformation by using the right-click menu and by selecting the "Open transformation" option. Note that this works for sub-jobs as well. Doing this will show the transformation (or job) as if you had started it separately, with all the metrics and logging information available to you. You can then see that the logging information shown comes only from the selected transformation, not from the parent job or anything else. This is made possible by the central logging buffer and the hierarchical logging channel architecture.

Log Text Capturing in a Job

Since version 4 of Kettle, you can find the logging text of a job entry in the result of its execution. With a JavaScript job entry you can extract this log text. While the main goal of this job entry is to simply evaluate a Boolean expression to true or false, you can also

retrieve various aspects of the previous job entry result with it. The following script defines a variable that will contain the logging text of the execution of the previous job entry, for example a transformation or a job:

```
var logText = previous_result.getLogText();
parent_job.setVariable("LOG_TEXT", logText);

true;
```

Our example makes use of the predefined JavaScript object `previous_result`. You can find a complete list of all the information you can extract from it in the "Result" section of Chapter 22. The `LOG_TEXT` variable that is defined in the parent job can be used in a subsequent transformation to write to a database table or an XML file. You could also use it in a Mail job entry to send the log text to an e-mail recipient.

Logging Tables

Because it is not always convenient to start searching for problems in a bunch of logging files on a remote server, the Kettle developers created the option to write logging information to database tables. In the following sections we explain how you can write all sorts of interesting information to the Kettle log tables from the viewpoint of a transformation and a job. We want to emphasize that the use of log tables is recommended and good practice that will allow you to easily track and debug problems.

Transformation Logging Tables

At the transformation level, there are four log tables that can be updated:

- The transformation log table
- The step log table
- The performance log table
- The logging channels log table

The default behavior of a transformation is to write to a configured logging table at the end of a transformation. The exception is the transformation log table, where you will also find a record being written at the start of the transformation. At that time, Kettle also determines the batch ID number. This number is unique for each execution of the transformation and can be used to group information in the transformation logging tables together. In the executing transformation itself, you can use a Get System Info step to retrieve this batch ID in case you want to correlate other information with the logging tables.

If a logging table is configured, you can check the execution results in the Execution History tab in the Execution Results pane, as shown in Figure 14-6. Use the icon immediately to the left of the zoom percentage in the transformation toolbar to show the results pane if it's not visible.

Figure 14-6: The transformation execution history

The transformation log tables can be configured in the Transformation Settings dialog. This dialog is accessible from the Edit ➪ Settings menu in Spoon or when you right-click on the background of a transformation and select Transformation settings.

Using the SQL button of that dialog, you can generate the SQL needed to make the layout of the target logging tables correspond to the configured log tables. Note that at the time of this writing, indexes are not generated. Read on for advice on which indexes to add.

The Transformation Log Table

The transformation log table was created to allow the most important metrics of a transformation to be written to a relational database table. Figure 14-7 shows all the options that you can set for the transformation log table.

Figure 14-7: The transformation log table settings

On the left side of the tab, you'll see the various log tables that you can define for the current transformation. Here is a description of the parameters that you can use to configure it:

- **Log Connection:** The database that contains the log table.
- **Log table schema:** The schema that contains the log table.
- **Log table name:** The name of the log table.
- **Logging interval (seconds):** This optional parameter periodically writes information to the logging table during the execution of the transformation. If you don't specify a value for this parameter, the log table will only be updated at the start of the transformation and when it finishes.
- **Log record timeout (in days):** This optional parameter will remove old log records from the table after it has inserted a new value. It will use the log date field of the table to do this.
- **Log size limit in lines:** This will limit the size of the logging text for databases that don't support very large character fields.

You can also use variables `KETTLE_TRANS_LOG_DB`, `KETTLE_TRANS_LOG_SCHEMA`, and `KETTLE_TRANS_LOG_TABLE` to configure this table for all jobs and transformations, as described in Appendix C.

Below the parameters you can see all the fields that are logged. You can decide to include only certain fields. This feature was created to allow the logging system of Kettle version 4 to be backward-compatible with version 3. Keep in mind that, for consistency if you enable or disable specific columns of a logging table you should also do this in other transformations. Because of this maintenance overhead involved it's probably best to stick to the defaults in Kettle. It is also possible to rename each field in the table even though you rarely have a reason to do this.

In the `Step name` column, you can specify a step for certain metrics like the number of lines written. The goal is to identify steps in your transformation that are representative for a certain metric. For example, suppose you are reading data from a text file in Step A and write that data to a database table in Step B. In that case, you can specify Step A next to the `LINES_INPUT` field, Step B next to the `LINES_OUTPUT`. That way, you get a good idea of how many rows were processed by the transformation. If you do not specify any steps, all values will simply be zero in the transformation logging table.

The `Field description` column describes the purpose of the fields to make sure there is no confusion about the sometimes confusing names of the fields. For example, the `STARTDATE` column does not contain the start date of the transformation. Rather, it is the start of the date range that you can use for incremental data processing. For more information on this topic, see Chapter 6 in the "Change Data Capture" section. There you will get an overview of the various ways to incrementally capture changes in a source system. Read on in this chapter to see how you can use the information in the transformation log table instead of the `cdc_time` table suggested.

The `STARTDATE-ENDDATE` timeframe is a period of time that covers all the new and updated records in a source system from the last time that the transformation ran

correctly until the current time. These two values are calculated and logged at the start of the transformation and they are also available in the Get System Info step.

If you expect the size of the log table to become rather big, you should create a few indexes on the log table. This will speed up lookups that the transformation performs in the log table when the transformation starts or when it performs updates to the log record. The first index is to be created on the `ID_BATCH` column and the second one on the `ERRORS`, `STATUS`, and `TRANSNAME` columns.

The Step Log Table

Because the step metrics in the transformation log table are very high level, it's possible you might have the occasional need to get hard data on the number of processed rows on a step level. To configure the step logging table, revisit the Transformation properties dialog and go to the Logging tab, shown in Figure 14-8.

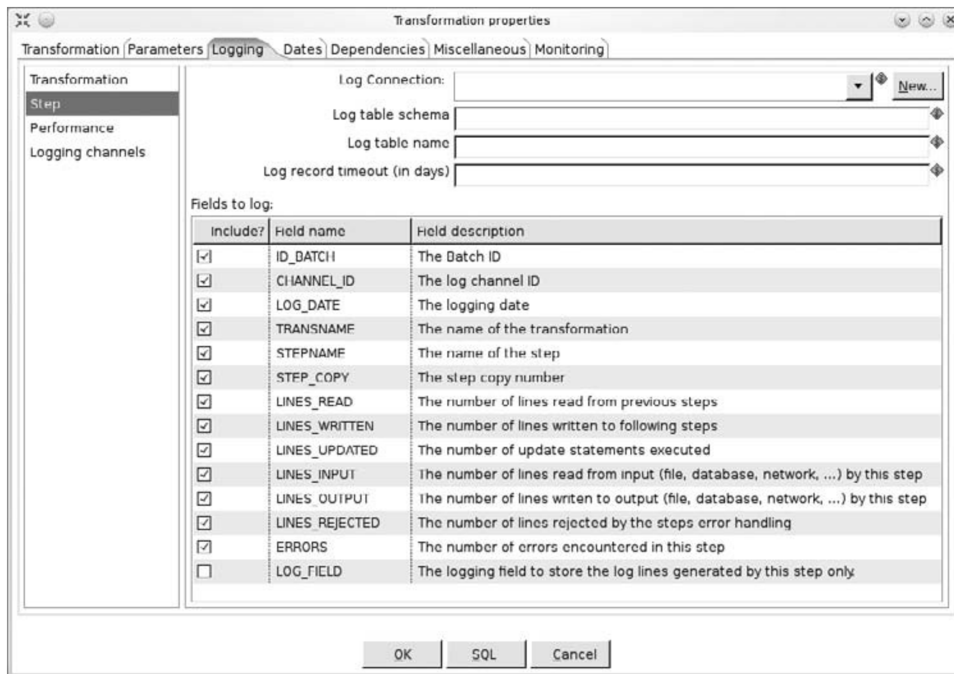


Figure 14-8: The step logging table

With the exception that there is no interval logging option available for this log table, the options are similar to the ones available for the transformation log table. Alternatively, you can use variables `KETTLE_STEP_LOG_DB`, `KETTLE_STEP_LOG_SCHEMA`, and `KETTLE_STEP_LOG_TABLE` to configure this table globally for all executed transformations, as described in Appendix C.

If you are interested in the logging text on a step level, you can choose to include the LOG_FIELD column. Because this is only exceptionally the case, this column is not included in the logging table by default.

The Performance Log Table

Chapter 15 discusses how enabling the step performance monitoring can help you visualize the performance of individual steps. Performance monitoring works by allowing you to periodically capture all the step metrics. This information is then displayed in an updating chart while the transformation is running.

This information is interesting from an analysis standpoint. Performance problems as described in Chapter 15 do not only occur during development when you run in Spoon. It makes sense to allow you to analyze this data after the transformation finished a batch run. This is why Kettle has the option to store the information in a log table. Figure 14-9 shows all the options for the performance log table.

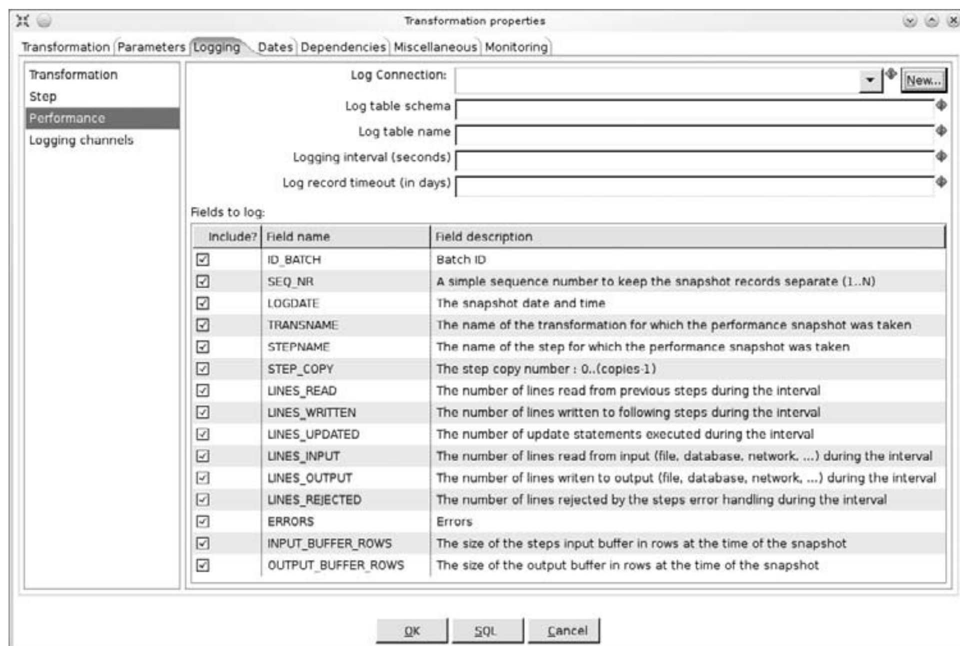


Figure 14-9: Configuring the performance log table

The parameters listed are the same as those described earlier in the transformation log table. This table can also be configured globally for all transformations with the KETTLE_TRANS_PERFORMANCE_LOG_DB, KETTLE_TRANS_PERFORMANCE_LOG_SCHEMA, and KETTLE_TRANS_PERFORMANCE_LOG_TABLE variables as described in Appendix C.

The Logging Channels Log Table

As described earlier in this chapter, Kettle keeps track of the complete hierarchy of executed components such as jobs, job entries, transformations, steps, and database queries in the logging architecture. This information has use beyond the internal bookkeeping of Kettle. Because of this, the logging channels log table came to life. Figure 14-10 shows all the options that are available to configure the logging channels log table.

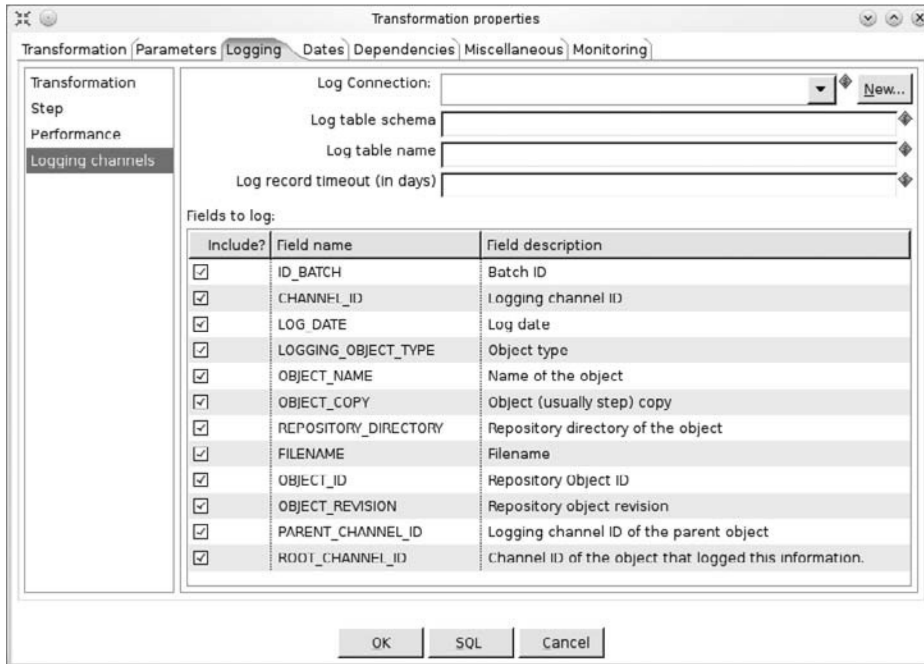


Figure 14-10: Configuring the logging channels log table

Again, the database connection, schema, and table specification are the same as with the other logging tables. This log table can be configured in one go for all executed transformations with the `KETTLE_CHANNEL_LOG_DB`, `KETTLE_CHANNEL_LOG_SCHEMA`, and `KETTLE_CHANNEL_LOG_TABLE` variables, as described in Appendix C.

Here are a few examples of what you can do with this information:

- List the transformations that were executed in a job.
- Check the exact revision of an executed transformation after it failed.
- Find out which transformations used a mapping since it was last changed (for example, if someone made an error in a calculation in a mapping).

Job Logging Tables

Because you don't gather performance-monitoring data in a job, you can only configure three log tables on that level:

- The job log table
- The job entry step log table
- The logging channels log table

Because the logging channels log table is identical to the one described for transformations, we will not cover it again.

To configure the job log tables, open the Job Properties dialog by right-clicking on the background of a job and selecting "Job setting" from the menu.

The Job Log Table

As with transformations, a record is written into the job log table at the start of the job. Kettle does this to indicate that processing of the job has started. By default, that record is also updated when the job finishes.

NOTE Even though the transformation and job log tables are almost identical, you are advised to use separate tables because jobs and transformations have separately calculated IDs.

The job ID that is calculated (column `ID_JOB`) is a unique number that reflects a batch run of the job. Because it is present in all the job log tables, it can be used to link those together. Note that Kettle does not expose the relationship between the job ID and a transformation batch ID because this information is available in the logging channel log table.

The line metrics found in the job log table are different from zero only when one or more transformations are executed in a job. Even then you need to make sure that you specified steps to take the metrics from as described.

You can use the `KETTLE_JOB_LOG_DB`, `KETTLE_JOB_LOG_SCHEMA`, and `KETTLE_JOB_LOG_TABLE` variables to configure this table for all executed jobs, as described in Appendix C.

Just as a precaution, you are advised to create two indexes on this log table as well to speed up queries and updates. First create an index on the `ID_JOB` column. Then create a second one on the `ERRORS`, `STATUS`, and `JOBNAME` columns.

The Job Entry Log Table

In the job entry log table, you will again find a lot of the parameters are the same as for transformations. The main difference is that this table includes information regarding the result of a job entry. For example, the Boolean result flag is included as well as an indication of the amount of result rows and result files that were found.

To configure this log table centrally for all executed jobs, define the following variables as described in Appendix C: KETTLE_JOBENTRY_LOG_DB, KETTLE_JOBENTRY_LOG_SCHEMA, and KETTLE_JOBENTRY_LOG_TABLE.

Summary

Finding out what goes on in complex Kettle jobs or transformations is not always easy. This chapter showed you how to get better insights into problem solving and analysis of the data flows in Kettle. More specifically you learned how to perform the following tasks:

- Extract high-level lineage information for batch-level usage.
- Work with the design time lineage information available in the Spoon user interface.
- Extract database impact information.
- Set up auditing in the form of logging tables for transformations and jobs.

Part

IV

Performance and Scalability

In This Part

Chapter 15: Performance Tuning
Chapter 16: Parallelization, Clustering, and Partitioning
Chapter 17: Dynamic Clustering in the Cloud
Chapter 18: Real-Time Data Integration

CHAPTER 15

Performance Tuning

This chapter provides an in-depth look at the art of performance tuning Kettle. We primarily focus on tuning transformations and briefly look at what can go wrong with the performance in a job.

For readers who are interested in the internals of the transformation engine, the first part of this chapter offers many details with a number of examples. Once you have learned how the transformation engine works, we focus on how to identify performance bottlenecks. Then we offer advice on how to improve the performance of your transformations and jobs.

NOTE **Readers who are new to Kettle may prefer to skip this chapter until they encounter a performance problem. At that point, you can simply turn to this chapter to learn how to identify and solve the problems you're encountering.**

Transformation Performance: Finding the Weakest Link

Performance tuning of a transformation is conceptually quite simple. As in any other network, you search for the weakest link. In the case of a transformation, you search

for the step that is causing the performance of the transformation to be sub-optimal. To better understand why this is important, take a look at a simple example. The following transformation reads customer data from one database and writes it into another, as shown in Figure 15-1. The figure also shows the step performance metrics during execution at the bottom.

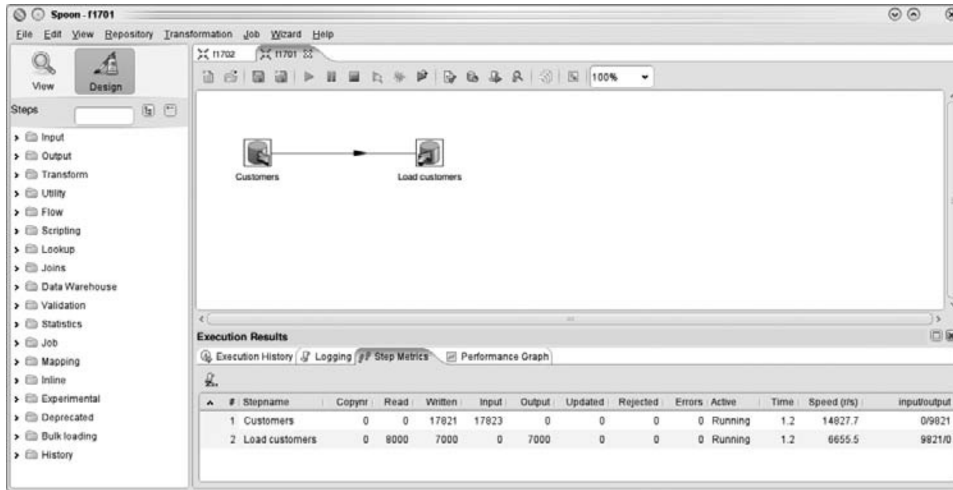


Figure 15-1: Reading and writing customer data

Here is what happens in this example: The Customers step writes rows into a Row Set buffer between the two steps. The "Load customers" step reads them from the buffer. In this example, one of two things can happen to affect the overall speed of the transformation:

- **The "Load customers" step is slow:** This typically occurs because writing to a database is slower than reading or because it needs to write over a slow network. Because of this, the row buffer between the two steps will fill up. Once it has reached its maximum capacity, it won't accept any more rows from the Customers step. This means that the Customers step will have to wait a bit until more room is available. Consequently, the step is slowing down to match the speed of the step "Load customers."
- **The Customers step is slow:** This may occur if the data is read over a slow network or if the source database is slow. In that case, rows are not being written very fast into the row buffer between the two steps. If the "Load customers" step wants to read a row from the input buffer but can't, it will simply wait until a row is available. Again, as a consequence the two steps will proceed at the same speed.

The same principle that applies to these two steps applies to a transformation with any number of steps; the slowest step affects the overall speed of the transformation. Because of that, it's important to figure out what the weakest link is before you can even hope to improve the performance of your transformation.

Finding the weakest link or the slowest step in a transformation can be done in two principal ways: simplifying or measuring. The following sections examine each of these in turn.

Finding Bottlenecks by Simplifying

The first method that is commonly used is the simplification of the problem. Simply put: remove steps from a transformation and see when your performance increase is the largest. In the preceding example, you could try to run the transformation as shown in Figure 15-2.

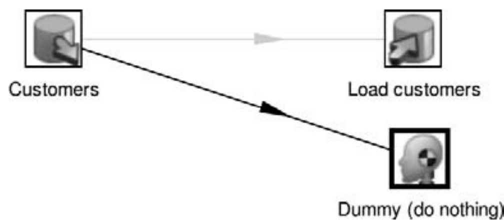


Figure 15-2 : Taking a step out of the equation

If you run this transformation, either of two things can happen:

- If the performance increases, you know that the step "Load customers" is the slower step of the transformation.
- If the performance stays about the same, you know that the Customers step is the slower.

Detecting a bottleneck using this technique is typically done by disabling hops in the transformation. Only steps that are connected with at least one hop are being included in the execution of a transformation. By disabling hops, you can exclude parts of your transformation. That in turn allows you to focus on the performance of individual steps.

Usually you can start to disable steps at the end of the transformation and then see which step degrades the performance the most. Compared to the performance metrics shown in Figure 15-1 performance increased 16-fold, indicating a slow "Load customers" step, as shown in Figure 15-3.

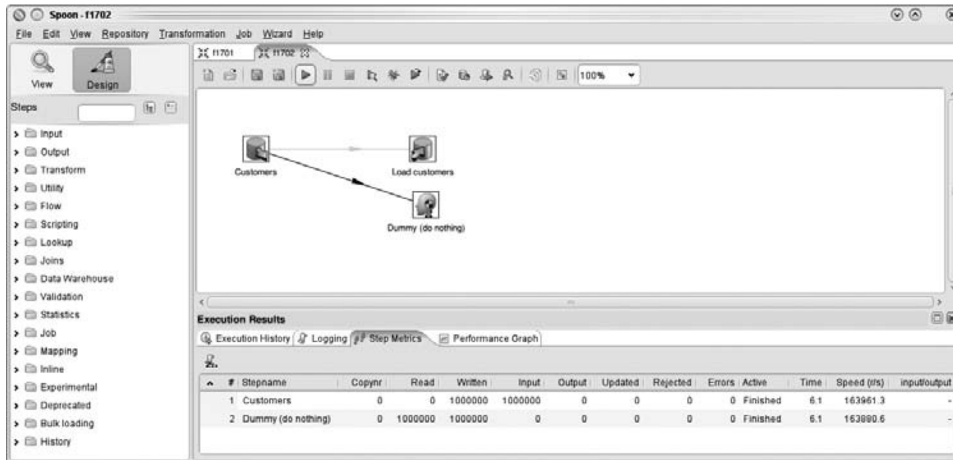


Figure 15-3: The execution results showing improved performance

Finding Bottlenecks by Measuring

Another way to figure out which step is holding back the performance of a transformation is to look at the buffer size during execution. If you execute the first example of this chapter, you get the performance metrics shown in the bottom half of the screen of Figure 15-1.

NOTE **When your transformation runs on a remote system on a Carte instance or on a Pentaho Data Integration server you can see the performance metrics when you consult the Spoon slave monitor. You can do this by clicking right on a defined slave server and by selecting Monitor from the pop-up menu.**

The last column, labeled `input/output`, actually represents the total number of rows in the input and output Row Set buffers relative to the step being monitored. In our sample, the maximum buffer size is 10,000 and we have 9,821 rows in it. This means that the buffer is at 98 percent of its capacity, which means that the Customers step is fast enough to continuously fill up the buffer. Consequently, the "Load customers" step is the slowest link in the network.

You can also see a difference in the performance between the two steps in the performance metrics. This is caused by the buffer between the two steps. For a short while, it allows the two steps to operate at a different speed. Once the buffer is full, the two steps will operate at virtually the same speed.

If you wait a little bit longer, you see the metrics shown in Figure 15-4.

#	Stepname	Copynr	Read	Written	Input	Output	Updated	Rejected	Errors	Active	Time	Speed (r/s)	input/output
1	Customers	0	0	256806	256808	0	0	0	0	Running	22.1	11603.9	0/9806
2	Load customers	0	247000	246000	0	246000	0	0	0	Running	22.1	11160.3	9806/0

Figure 15-4: The step metrics 22 seconds into the execution of the transformation

Despite the benefit of having these metrics, it can still be daunting to find a performance problem in complex transformations, precisely because of the dynamic nature of the performance metrics. In those situations, it can be useful to enable the step performance monitoring feature of a transformation.

NOTE **Step performance monitoring can be enabled on the Monitoring tab of the "Transformation settings" dialog. It allows you to look at various performance graphs per step. The metrics are gathered during execution of a transformation.**

Once you have the step performance metrics, you can take a look at the graph of a step metric, as shown in Figure 15-5. The interesting metrics to look at are the number of rows read or written by a step and the buffer sizes. In Figure 15-5, you can clearly see the fast start of the Customers step followed by the slowdown to arrive at the speed of the second step.

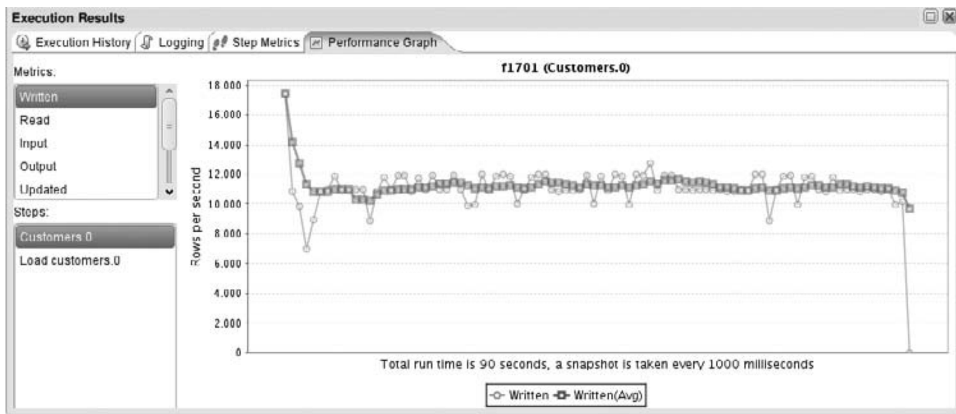


Figure 15-5: The evolution of the speed of a step as a graph

Figure 15-6 shows the evolution of the buffer size during that period of time.

As you can see, the buffer fills up very quickly and stays almost filled until the transformation is finished. This indicates a big speed difference. (The first step is 16 times faster, as mentioned previously.) If the speed difference is smaller, the number of rows in the buffer will increase more slowly over time.

NOTE **Make sure to use as few programs on your computer as possible during performance measurements. Even though it is not always easy to see this, mail and Twitter clients, virus checkers, spam filters, firewalls, and browsers can consume considerable amounts of resources on your system.**

To make sure you don't have any uninvited processes skewing your measurements, run the same transformation more than once to let the data average out. When you compare performance measurements, keep in mind that you should compare values that originate from systems with the exact same characteristics. Even small differences in the speed of processors, disks or network can lead to completely different performance measurements.

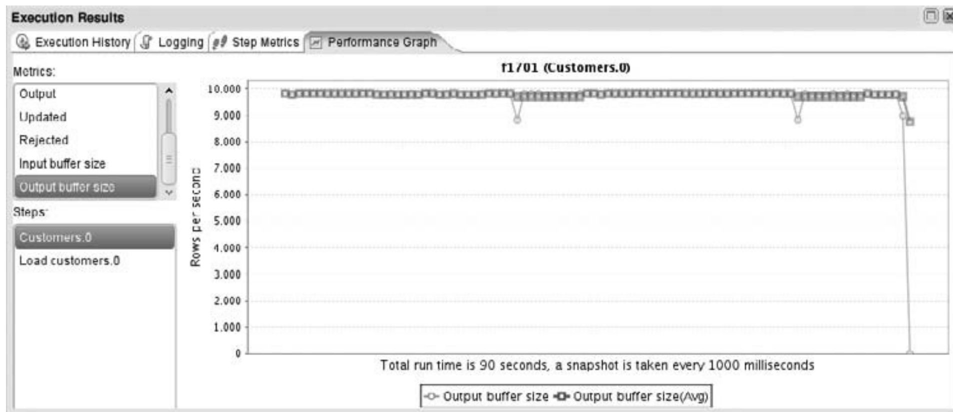


Figure 15-6: The evolution of the output buffer size of a step

Copying Rows of Data

A special performance tuning case is reserved for situations where rows of data are copied to multiple targets. In the sample shown in Figure 15-7, rows of data are copied to a very fast "Dummy (do nothing)" step and to the "Load customers" step as well.

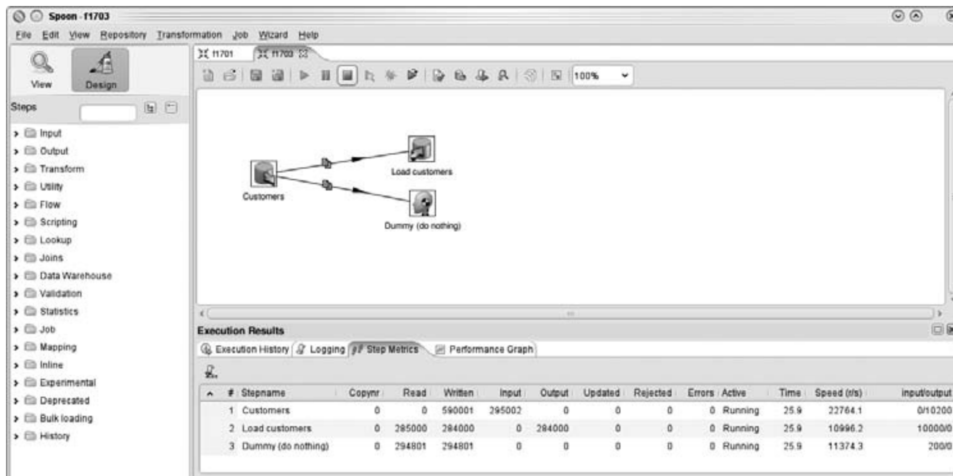


Figure 15-7: Copying rows of data to multiple target steps

Notice that the performance of the "Load customers" step is also slowing down the "Dummy (do nothing)" step, despite the fact its input buffer is almost always empty. The reason for this is that the Customers step will wait a bit whenever there is an output buffer that is full.

A rare but important performance tuning case is one where a step can stop execution and the transformation will completely stall. An example of this case is shown in Figure 15-8.

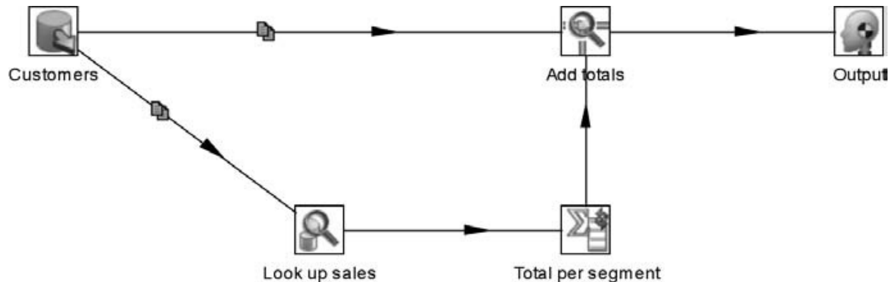


Figure 15-8: A transformation that can stall

In this example an attempt is made to add the total customer sales per segment to the individual customer records, perhaps to calculate a percentage of the sales per segment per customer. The problem with this is that data from the Customers step is being used to supply both the main data and the lookup data of the Stream Lookup step. In this situation, the Stream Lookup step only reads input when all lookup data is read. However, after a short while the Customers step is blocked because its output buffer to the Stream Lookup step is full. That in turn blocks the output to the Dummy step, and the whole transformation is in a so called deadlock state where two or more steps wait for each other

You can obviously solve this problem by increasing the buffer size. However, because you can never be sure of the maximum number of input rows, it's usually better to make sure that the circular reference goes away. You can do this either by splitting up the transformation into two parts or as shown in Figure 15-9, by reading the same source data twice.

You can also solve this problem by cutting your transformation up into two or more transformations that write intermediate information to a temporary file or database table.

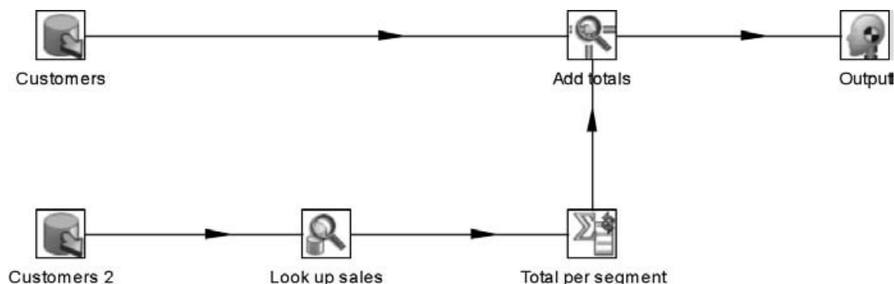


Figure 15-9: Solving a dead-lock by reading data twice

Improving Transformation Performance

Once you know which steps are slowing down the transformation, you can try to do something about it. The following sections describe various ways to make your transformation run faster. First you will learn how to improve the performance of reading and writing text files. Then you will see how you can get information in and out of a database faster. Because the performance of the network has a big effect on databases, we cover that topic as well. At the end of this section you will find various specific ways in specific steps to improve performance.

Improving Performance in Reading Text Files

Every time you read data from a text file, a number of events happen that can cause a performance bottleneck. There are a number of reasons why reading a text file might be slower than you expected. To go deeper into those reasons, we'll take a closer look at what happens while reading.

First, a block of data is read from the disk on which the file resides. If you are reading from a slow disk, the performance of the step that reads the file will also be slow. However, the latency (or seek time) of the disk also plays its role. If you instruct a step such as CSV Input to read in relatively small blocks, it's going to take a tiny bit of time to reposition the heads of the disk to read the next block.

After you read a block of data from the file, it is translated from a series of bytes into a continuous series of characters. These characters form a block of text, also referred to as a *string*. The encoding scheme used to process the bytes, and turn them into characters, must match the original encoding of the file. Depending on the encoding of the file, a different encoding scheme is used. This process is obviously quite important because it is responsible for achieving the correct end result. However, parsing files in various encodings can be quite costly in terms of CPU usage. For certain double-byte encoding schemes like UTF-16, almost half of the CPU cycles are used to convert bytes into characters during the execution of a step.

Once you have individual characters to deal with, you need to split the data up into different fields. In particular, steps such as CSV File Input and Text File Input extract lines of text by looking for carriage return and linefeed characters. Then the lines of text are split up into fields. This is done either by searching for a separator or by counting a certain number of characters if the file is using lines with a fixed width.

Finally, the fields that were extracted from the lines of text are optionally converted into other data types such as `Date`, `Number`, `BigNumber`, or `Boolean`. Date and numeric data conversion is especially costly in terms of burning CPU cycles. In the case of date conversion, all sorts of information is taken into consideration, such as the format, leap years, leap seconds, and time zones. For numeric data, the parsers take into account the format, grouping, and decimal symbols as well as exponential notations.

All this logic comes at a price in the form of using up a lot of processing power. Thus, it would not be very unusual to see a lot of CPU usage while reading a complex text file with a lot of fields that need to be converted to data types other than `String`.

In addition to those basic operations, a field might be trimmed or a default value might be used when there was no original data.

Now that you know the common problems, take a look at a few simple techniques to speed up the reading of your text files.

Using Lazy Conversion for Reading Text Files

At first glance it might look like there isn't very much you can do about the CPU cost incurred by data decoding and conversion. However, in certain cases it might be worth it to enable the Lazy Conversion option in the CSV File Input or Fixed File Input steps. When this option is used, all data conversion for the data being read will be postponed as long as possible. This includes character decoding, data conversion, trimming, and so on. The only tasks that the steps do initially is read the file in binary form (bytes) and split it up into separate fields.

Obviously, if the fields of data are actually needed in other steps the data conversion will still take place. When that occurs, it might actually slow things down. However, you will get better performance using Lazy Conversion in a few specific cases:

- If most fields are simply written to another text file in the same format.
- If the data needs to be sorted and the data doesn't fit in memory. In that case, serialization to disk is faster because you postpone encoding and type conversions.
- When you bulk load data into a database and when there is no data conversion needed, lazy conversion can improve performance. That is because bulk-loading utilities typically read text directly and the generation of this text is faster. This is in contrast with the use of regular steps like Table Output in which the data needs to be converted anyway.

For cases where only a few fields are used in a transformation, note that individual fields can be converted to Normal storage with the aid of the Select Values step on the Metadata tab.

Single-File Parallel Reading

Reading a text file in parallel is possible simply by enabling the "Running in parallel?" option in the "CSV file input" or "Fixed file input" steps. This option will assign a part of the file to each individual copy of a running step. The number of copies to execute of any step is set by right-clicking on the step and selecting the "Change number of copies to start…" option. Another option is to run the steps on multiple slave servers in a clustered run. See Chapter 16 for more information on how to run steps in a clustered environment.

NOTE **Because the parallel reading algorithm in "CSV file input" synchronizes on carriage return or linefeed, it is only possible to read files in parallel if they don't have fields with carriage returns or linefeeds in them.**

Note that this technique will quickly shift the bottleneck from processing power to the read-capacity of the used disk. This is not helped by the fact that reading the same file with multiple threads is going to incur a lot of extra overhead because the disk-heads are going to be repositioning all the time. That being said, the amount of overhead highly depends on the used disk subsystem and the amount of caching the operating system uses.

Multi-File Parallel Reading

Because of the overhead incurred by reading the same file with multiple threads or processes, it's usually better to read a single file with a single step copy if you can. In situations in which you have a lot of similar files, it's therefore better to simply give multiple reader steps different files, as shown in Figure 15-10.

Figure 15-10: Distributing files to multiple step copies

A round-robin system is used to distribute the rows with file names evenly over the four "CSV file input" copies. Each copy will receive a different file name and will process a group of files in parallel.

Configuring the NIO Block Size

The "NIO buffer size" parameter has an influence on the performance and can be set in the "CSV file input" and "Text file input" steps. It determines the amount of data read at once from a text file. It is not possible to give exact guidelines on how to set this parameter. If you make the block size too big, it might take a relatively long time to fetch the data. This could reduce the parallelism in the transformation. However, if you have a really fast disk subsystem, a large buffer size might be speeding things up instead. If you have a disk subsystem with a lot of caching or with a very low latency (seek time), it might be faster to lower the buffer size. In general it will pay off to look at the buffer size, see how much CPU capacity you have left, and try a few values to see what works best.

Changing Disks and Reading Text Files

In those cases where a slow disk is at the root of a performance problem, it usually pays to investigate whether it's possible to change the location of a source file. Especially when data is being extracted from remote databases, you can see text files end up on temporary disks with less than optimal performance. In particular, slow disks that are mounted over a network come to mind. While network-attached storage can be used

without a problem for archival of input files, be careful when using them for high-performance data reading.

Improving Performance in Writing Text Files

All the character encoding and data conversion performance issues that are valid for the reading of text files are also valid for the writing of text files. In writing, the conversion from a date or a numeric data type to text happens first. Then the conversion of the text into the correct binary format of the text file in the correct encoding occurs. The same methods that you can use to speed up the process of reading text files can be used to speed up text file writing.

Using Lazy Conversion for Writing Text Files

For a transformation that involves reading and writing a text file back to another file, the character encoding and data conversion cost is being paid twice. For such cases, the Lazy Conversion option described earlier will help performance.

Parallel Files Writing

Writing to a single text file in parallel is not possible. For example, you can't use multiple step copies of the "Text file output" step to write to the same output file. If you try it, the result is a blended file with lines and fields ending up in the wrong location. This problem can only be countered with advanced locking algorithms that in turn again reduce the degree of parallelism to a single thread.

However, the simple solution is to write to multiple output files. If needed, these files can reside on multiple disks to provide further scaling options.

The easy way to do this is to use the "Include stepnr in filename?" option or to use the internal variable `${Internal.Step.Unique.Number}` in the filename. This last option allows you to divert the writing load to a different disk by creating symbolic links to the right disk location, for example `/disk1`, `/disk2`, and so on.

Note that internal variable `${Internal.Step.Unique.Number}` also is valid when used in clustering and that the total number of step copies is available in variable `${Internal.Step.Unique.Count}`.

Changing Disks and Writing Text Files

The same advice that applies to reading a text file goes double for writing one because writing is typically up to twice as slow as reading. If you can't change the location of the file and you see a significant slowdown due to a slow disk, consider compressing the text file. This can reduce the size of the output file by a considerable amount; you may use as little as one-tenth the space you would otherwise. Because of this the write performance of the target disk becomes much less of a problem.

Improving Database Performance

In any typical data warehouse, you read and write from and to all sorts of relational and non-relational databases. In most cases, you use a database driver to make this possible. So again, there are abstraction layers involved in the reading and writing. Let's take a look at what's going on.

If you specify a database connection in a step, you usually do so by specifying the name of the database connection. This allows the step to find the details, the metadata, of the connection. What typically happens then is that the connection is opened during the initialization phase of the steps. If all steps are initialized correctly, they start to process rows. It's at this time that rows of data are read from or written to the database.

It's important to note that, as a general rule, each step copy creates a separate connection to the used database. This means that if you have ten step copies in your transformation that use a database, you have ten connections to the database.

This behavior can be changed by enabling the "Make the transformation database transactional" option. This will open a single connection per defined and used database. It will also run in a single transaction. A commit will be executed at the successful end of the transformation. A rollback will be performed in case there is an error anywhere in the transformation.

Avoiding Dynamic SQL

Once the connection is established, as much of the work that can be done up front is done up front. For most steps, this means the compilation of SQL statements on the databases, also known as the *preparation of statements*. It is a time-consuming process because the database needs to validate whether tables and columns exist, choose indexes, set join conditions, and much more. You want to do this only once if the statements are going to be the same for all rows that are processed.

The notable exceptions where the preparation of SQL doesn't happen are the "Execute SQL Script," "Execute row SQL script," and "Dynamic SQL row" steps. Therefore, these steps will always cause the performance of your transformation to be lower compared to single-purpose steps such as Table Output, Table Input, Database Lookup, and so on.

Handling Roundtrips

Once the statements are prepared, values are set on the parameters, usually based on input parameters that correspond to values in the input rows. These parameter values are transmitted to the database, the statement is executed on the database, and a result is retrieved. This means that for most database operations, a roundtrip to the database is made over the network.

Therefore, performance of most steps that use a database depends heavily on the speed of the roundtrip, which is a combination of the speed of the network, the latency of the network, and the performance of the database.

Reducing Network Latency

It's always useful to be aware of whether a database is running on the same machine, on the local network, or in some remote location accessed over a VPN connection. The difference in latency for these three cases will be noticeable because the network characteristics play a big role in the performance.

You can get an idea of the latency of a network by using the `ping` command. Table 15-1 shows the effects on latency with a maximum throughput calculated under several different conditions.

Table 15-1: Network Condition Effects on Latency

DESCRIPTION	LATENCY	MAXIMUM THROUGHPUT
Same machine	0.035 ms	28,571 rows/second
Local network	0.800 ms	1,250 rows/second
Server in the same country	10.5 ms	95 rows/second
Server in a different country	150 ms	400 rows/minute
Over a satellite in a train	1500 ms	40 rows/minute

There is usually little you can do about the latency of the network itself. You thus have to look at other possible solutions.

A first solution for high database latency is the reduction of the number of roundtrips to the database. This is usually accomplished by loading lookup data in memory (a single roundtrip for all rows) or by using caching (a single roundtrip per unique key you look up). Most steps that perform lookups on a database have options to cache data, and a Stream Lookup step is available to load any set of rows in memory. This step allows you to look up rows very quickly. It is also possible to use the "Merge join" step to join large data sets in a streaming fashion with very little memory consumption and good performance. "Merge join" requires that the input data for the step be sorted. However, databases can usually sort very quickly compared to a streaming engine (see the following paragraph), especially if the `ORDER BY` clause involves an indexes column.

A second solution against high latency is the execution of multiple step copies at the same time. This is usually easy to do with a step. In theory, you can slash the average latency in half and double the performance by doubling the number of step copies you use. However, in practice we see increases in performance on the order of 30 to 50 percent because of limits ranging from the way that databases handle the connection requests to saturations in the networking layers.

Finally, batching requests together usually makes a big difference in performance. For example, the "Use batch updates for inserts" option in the "Table Output" step reduces the number of roundtrips by sending the rows over in large batches. In certain cases where you have a fast network and a high latency the use of multiple step copies

can dramatically increase the performance of your transformation. However, typically you can gain between 20 and 40 percent in throughput.

Network Speed

Obviously it's not just the latency of the network that's important when looking at the performance of a transformation. The network speed, also known as *bandwidth*, is critical, too. This is especially relevant when large amounts of data are being moved over the network. For example, if you are loading data with the Table Output or Insert/Update steps, a lot of information is going over the network to and from the database.

Again, there is usually little you can do about the network performance itself. However, it might be worth trying to move the data as close to the database as possible before loading. In that respect the saying "nothing beats the bandwidth of a truck full of hard disks" comes to mind. In other words, it's sometimes faster to avoid the use of a slow network by unconventional means like by putting the data on an external hard disk. You can then use overnight shipping to the other side of the world.

NOTE An extreme example of moving the data close to the database prior to loading is *bulk loading*. This technique is often used by databases to load large amounts of data directly from text files. Kettle has several bulk loaders available that perform this task for you in a transparent way.

Handling Relational Databases

As explained earlier, the database itself obviously also plays a big role in the performance of a step. Tuning a database is usually highly dependent on the underlying technology. However, there are some general rules of thumb that can be applied to all relational databases.

Commit Size

Depending on the underlying database, it might be a good idea to try to make the transactions bigger in the steps that support it. For example, if you change the `commit size` parameter in the Table Output step, the performance will change because the relational database has to manage the transactions. Typically, you get better performance if you increase the commit size. Too much of a good thing can turn bad so make sure not to exaggerate. You'll notice that there is usually no benefit in increasing the value beyond ten to twenty thousand rows.

Indexes

It's critical whenever lookups or join operations are being performed that appropriate indexes are applied to the database tables. For example, Figure 15-11 shows the performance graph of a "Dimension lookup / update" step when there is no index defined on the natural keys to help the lookup of the dimension row.

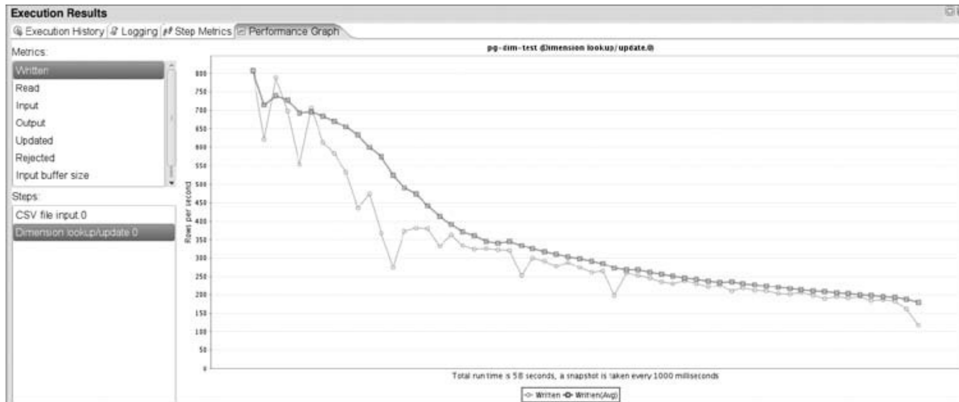


Figure 15-11: Performance degradation caused by a missing index

As you can see, despite the best efforts of the database to cache as much data as possible, the performance keeps going down because the database needs to perform what is called a *full table scan* to determine the location of a row of data. If there are *N* rows, it takes on average *N*/2 compares to find the row. Because rows are continually being added to the table, the *N*/2 figure keeps growing, too. It doesn't really matter if the rows are being looked up in memory or not. Without an index, the number of compares keeps growing. As a direct consequence, lookups get slower and slower. The lookup of a row using an index is much less dependent on the number of rows that are being queried. Figure 15-12 shows a performance graph of the same task with an appropriate index defined on the natural keys of the table.

As you can see, performance is pretty much constant for the step after the creation of the index.

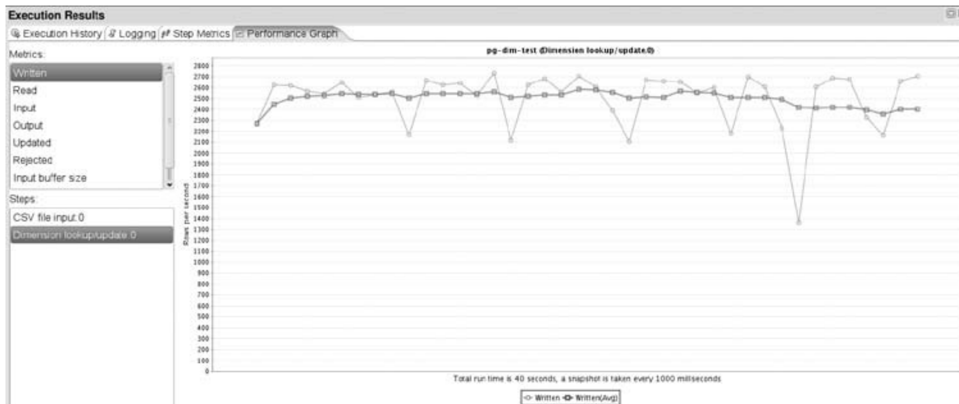


Figure 15-12: The proper index ensures consistent performance

At the same time, you should be aware of the fact that adding an index to a database table slows down any changes that occur to the table, including inserts, updates, and deletes. The more indexes you add to a relational table, the slower these operations become. For this reason, it is often beneficial to drop the index before making large changes to a table such as initial loads or massive amounts of updates. The re-creation of the indexes after the changes is bound to take up less time because all the rows can be considered all at once making that operation more efficient.

Table Partitioning

Table partitioning is usually done by splitting up a large table into smaller parts using a simple partitioning rule such as a range. For example, each table partition can contain the data for a single day, month, or year. Another rule can be a simple hash key. For example, a single partition can contain all the rows for customers who have the letter *A* as the first character of their last name.

Whatever the partitioning logic used, it typically allows the database to quickly find out to which partition a certain row belongs. Before the database even has to use an index, it can eliminate most of the possible locations for a record. This in turn increases the performance of lookups.

Another advantage that partitioning offers is that it can increase parallelism in the loading and lookup of data. Some relational databases allow multiple partitions to be loaded simultaneously, increasing the total throughput. This in turn increases overall performance.

Constraints

The presence of constraints on a database table can considerably slow down updates to a table. That is because for every insert, update, or delete the constraints are validated. Because of this, it is often recommended that you disable constraints prior to loading large amounts of data. In a multi-dimensional (analytical) data warehouse, it's common practice to deploy constraints only during development. That is because all foreign keys are looked up in the ETL process anyway. It's usually not sensible to have the database enforce a rule that is already enforced by the ETL processes.

Triggers

Another source for considerable performance problems is the presence of one or more triggers on a database table. *Triggers* are database procedures that are executed whenever there is an insert, update, or a delete in a database table. Depending on the configuration, triggers can considerably slow down large amounts of updates to a database table. As a best practice, it's recommended that you not have triggers on a table in a data warehouse.

Sorting Data

Sorting large amounts of data can be an interesting performance problem because of the streaming nature of a Kettle transformation. The main issue is that you can never know what the maximum number of rows might be that need sorting. If you assume that all

rows would always fit in memory for sorting, then sorting could be done reasonably fast. Suppose you need to sort a million records and you have enough memory to sort all these rows in memory. In that case, sorting will take only about 10 seconds.

However, in cases where you could store only half the number of rows in memory, you would need to perform what is called an *external sort*. This means that the first half a million rows are sorted in memory and placed in a temporary file on disk. Then the next batch of rows is sorted and placed on disk. When all rows are sorted, the rows are read back from disk one by one in order.

Obviously, the external sort will be slower than the sort in memory because of the file serialization performance penalty. However, because the alternative is running out of memory, an external sort is preferable to having no sorted data at all. That by itself can be considered a considerable performance boost.

The first performance-tuning tip is to give the Kettle transformation engine a lot of memory to work with. To do this, specify a large amount for the `Sort size (rows in memory)` parameter in the "Sort rows" step dialog.

Also avoid having large numbers of small temporary files. Don't set a sort size of 5,000 if you are expecting to sort 2 billion rows because that will produce 400,000 files. The operating system will have a lot of trouble dealing with this. Because the external sort algorithm reads from all files, performance will drop significantly because the operating system will have trouble caching efficiently and this will result in large disk latency penalties.

Sorting on the Database

It's important to note that both the in-memory sort and the external sort need to read and persist the complete input data set. This is in contrast with how a relational database table works. Once data is stored in a relational database, only the sort keys need to be considered, along with a reference to the location of the row of data. Because the memory consumption is also smaller in this case, you can sort more rows in memory without having to use an external sort algorithm. Even better, if an index exists on the columns on which you want to perform a sort, the database doesn't need to perform a sort at all. It can simply traverse the index table in this case and return the rows in the correct order with great ease.

For these reasons, whenever you need to have data sorted *and* if the data already resides in a database table, it's faster to have the database sort the data for you.

Note that it's typically not faster to first load the data and then perform the sort on the database. If you don't have the data in a database, perform the sort in Kettle.

Sorting in Parallel

As described previously, the sorting exercise involves processing power and in most cases also incurs disk I/O. Because of this, it can be interesting to sort a large number of rows in parallel. This can be done by splitting the data set into parts in the ETL. Consider the transformation shown in Figure 15-13.

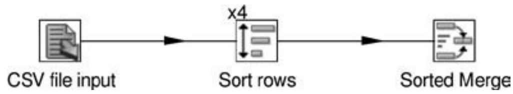


Figure 15-13: Sorting rows in parallel

In this simple case, four copies of the "Sort rows" step are sorting the rows simultaneously. Rows that are read by the "CSV file input" step are simply distributed over the four copies in a round-robin fashion. The output of the four copies consists of four sorted data sets. To keep the data sorted during the merging of the four streams of data, we use a Sorted Merge step that uses the same sort condition.

The same rules apply when we perform a sort in a clustered fashion, as in Figure 15-14.

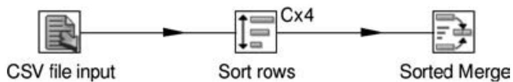


Figure 15-14: Clustered sorting of data

In this case, the rows of data are distributed over four different slave servers on four different hosts. It's easy to imagine that four times the processing power is available as well as four times the disk I/O throughput needed to sort the complete data set. If the overhead of passing the data back and forth over the network is smaller than the gain in sort speed, we have a winner. In tests on big iron and cloud computing settings such as Amazon EC2 we found that there is indeed a huge benefit to sorting data in parallel.

Reducing CPU Usage

The transformation engine streams data from one step to another in a multithreaded fashion. The engine tries to avoid disk I/O at all costs unless it is absolutely needed. While this is obviously a good choice, it does increase the likelihood of a transformation becoming limited in performance by the power of the CPUs of the machine(s) it runs on. This section examines a number of the causes for high CPU consumption.

Optimizing the Use of JavaScript

JavaScript (ECMAScript) is a popular way of solving complex problems in Kettle because the language is popular and well known by a lot of developers. It's also quite easy to develop solutions with it. The drawback, however, is that it typically consumes a lot of CPU power and doesn't perform on par with most of the other steps. The reason for this simply is that JavaScript is a scripting language. Even though the Script Values Step has seen a lot of performance fixes over time and JavaScript itself has advanced a lot

with just-in-time compilation, the performance problem still exists. So here are a few tips when working with JavaScript steps in your transformation:

- **Turn off compatibility mode.** The compatibility mode was introduced to allow easy migration from older versions of Kettle. The cost of having it enabled is high because a compatibility layer must be maintained for every row that passes through the JavaScript engine. For more information on how to migrate your older Kettle 2.*x* scripts to the cleaner format introduced in Kettle version 3.0, see `http://wiki.pentaho.com/display/EAI/Migrating+JavaScript+from+2.5.x+to+3.0.0`.
- **Avoid JavaScript.** A lot of use cases that required the use of JavaScript in the past are no longer needed. Most of these use cases have since led to the creation of special purpose steps that are easier to maintain and function at optimal performance.
- **Data conversion.** Use the Select Values step. The Metadata tab allows you to convert from one data type to another.
- **Create a copy of a field.** Again, the Select Values step will do the trick nicely. Alternatively, use the "Create a copy of field A" function in the Calculator step.
- **Get information from a previous row.** You can use the Analytic Query to solve this issue. This step allows you to get information from previous or next rows in a stream. For more information about this step, see Chapter 10.
- **Split a single field to multiple rows.** The "Split field to rows" step was specifically created to eliminate this complex use case.
- **Number range.** Avoid slow maintenance and error-prone *if-then* constructs in JavaScript and use the Number Range step.
- **Random values and GUID.** The "Generate random value" step allows you to create all sorts of random values at optimal speed. It also allows you to generate Globally Unique identifiers (GUID).
- **Checksums.** The "Add a checksum" step allows you to calculate all sorts of checksums like CRC-32 on rows of data.
- **Write a step plugin.** If you find you need to solve the same problem over and over again in JavaScript it might be useful to write your own step plugin. This will allow you to encapsulate the functionality with a nice user interface while at the same time getting optimal performance. One drawback of this is that you need Java experience. Another is that you need to set up a separate project outside of Kettle. For more information on the development of plugins, see Chapter 23.
- **Use the User Defined Java Class step.** This step allows you to write your own step plugin in the form of Java code that is entered in the dialog of the step itself. This code is compiled at run-time and executed at optimal performance. For more information on this step, see Chapter 23.
- **Combine JavaScript steps.** If you feel like you still have no other option but to use the JavaScript step, make sure you try to avoid having several separate JavaScript steps running when you can have a single one. There is a substantial overhead in the exposure of field values as variables to the JavaScript context.

- **Variable creation.** If you have variables in your JavaScript code that can be declared once at the start of the transformation, make sure to put them in a separate script and to mark that script as a "Startup script" by right-clicking on the script name in the tab in the JavaScript step dialog. JavaScript object creation is very easy to do but will also consume a lot of CPU cycles, so try to avoid allocating a new object for every input row.
- **Add static values.** You can use steps such as "Add constants" or "Get variables" instead. These steps offer better performance and are easier to maintain.

Launching Multiple Copies of a Step

If you have a step that consumes a lot of CPU power, for example a Modified Java Script Values or a Regex Evaluation step, you might consider running the step in multiple copies. Because most computer systems are equipped with multiple CPU cores these days, it makes sense to use them all. Launching a step in multiple copies will spread the load over the available cores, thereby increasing performance. Using multiple step copies for a slow step should be a last resort. The step itself needs to be optimized first. Make sure to optimize your JavaScript code or your regular expressions because those too can be a cause for performance problems.

A specific case of running in parallel is reserved for partitioning. In Kettle, each partition of a step is handled by a copy of that step. That means that you can drive rows with identical keys to the same copy of a step. This in turn allows you to make workloads run simultaneously that are otherwise not easy to parallelize, as with a Memory Group By operation. Consider the example in Figure 15-15.

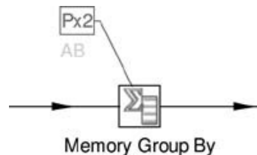


Figure 15-15: Partitioning a Memory Group By step to spread the workload over multiple processors

In this example, we partitioned the data on the grouping key. That guarantees that the result of the grouping is correct for each step copy. That same principle can be applied to caching strategies. If you want to avoid each step copy having a complete copy of the data cache, consider partitioning the data in such a way that the same rows always end up on the same step copy. This in turn will guarantee cache hits and improve performance even more.

Selecting and Removing Values

The Select Values step allows you to conveniently select and remove fields. Doing so, however, is usually an expensive operation with respect to the CPU usage. Both operations force the system to re-create every single input row. Then these new rows need to be populated with the correct values in the correct order.

In older versions of Kettle, you were required to select the output values. Steps such as Table Output didn't support field selection so you had no other choice. In recent versions of Kettle, you no longer need to do this.

The most common reason for selecting only certain fields is when you send rows from multiple steps to the same target steps. In that case, there is an absolute requirement that the layouts of the rows be all the same. When you encounter that situation, consider that adding constant values to rows is faster than creating a whole new row layout.

There are a few cases where selecting as few fields as possible is beneficial to performance and memory consumption. This includes all situations where you keep rows in memory or write rows to disk, for example when you sort rows, cache data in a database lookup step, or pass data from one slave server to another in a clustered execution.

Managing Thread Priorities

Since version 3, Kettle has included a "Manage thread priorities?" option in the Transformation Settings dialog that allows you to enable or disable the management of step threads. This option can be used to address a limitation in the implementation of the Java Virtual Machine on which Kettle runs. This limitation causes excessive locking on row sets with very few rows in them or when the row sets are full. In both these situations, the option will slow down the appropriate step to avoid the disadvantageous locking situation.

If you are migrating transformations from an older version or you disabled this option by accident, it makes sense to try to re-enable this option in recent versions. In most, if not all, cases you will see a performance gain.

Adding Static Data to Rows of Data

In situations where you are producing a single row of data in multiple steps, it makes sense not to perform these operations in the main stream of the transformation. Consider the transformation shown in Figure 15-16.

In this transformation, the first three displayed steps retrieve two variables, concatenate them, and finally convert them into a `Date` value. For all the input values, this date is going to be the same. Thus, it makes sense to calculate this information only once. You can do this by using the "Join Rows (cartesian product)" step. For example, consider the revised transformation shown in Figure 15-17.

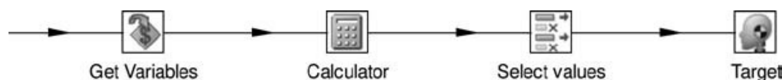


Figure 15-16: Calculating a constant value many times

Figure 15-17: Calculating a constant value only once

In this case, it's important to specify "Main step to read from" in the Join Rows step. This will ensure that the single row coming from the "Select values" step is always kept in memory and not the bulk of data passing through to the target. Once the Join Rows step is configured, it will easily outperform the previous solution by a fair margin.

Limiting the Number of Step Copies

A transformation works by creating a single thread for every step. A thread is a concurrently running task. Since a computer essentially can only process instructions one at a time, concurrency or multi-tasking is simulated by the processor of that computer. The simulation works by switching from task to task and by only spending a tiny amount of time on each task. The only exception to that rule is when threads run on different processors and when no switching is needed.

The direct consequence of multi-tasking or parallel processing, the task switching, causes a performance penalty because the processor needs to keep careful track of what it is doing before it can switch to another task. It also needs to restore these settings when it comes back to a task.

While the task switching overhead is small, it can add up if you are running a lot of threads on a system. Since every step copy in an executing transformation represents a separate thread, you will see performance degradation when you add a lot of steps. Because the efficiency of task switching varies from one computer system to another, it's hard to come up with a specific rule. However, as a rule of thumb, make sure to keep the number of steps lower than three or four times the number of processing cores in your system when performance is important.

Avoiding Excessive Logging

It might make sense to run your transformation with a high logging level like Debug when you are looking for a specific problem. However, keep in mind that when a lot of logging text is generated, a lot of processing power and memory is consumed. Because of this it is usually not advised to run your transformations and jobs at a logging level that is higher than Detailed outside of Spoon. Before you run your nightly production job with a high logging level, make sure that it doesn't include any transformations that could potentially log a lot of information and as a result possibly exceed the agreed batch window. For more information on the topic of logging consult Chapter 14.

Improving Job Performance

The bulk of the performance issues you will encounter are caused by transformations. However, there are a few things to look out for while you write a job. In this section we explain how you can improve the performance of loops in jobs and how you can speed up short-lived transformations that connect to a database.

Loops in Jobs

A lot of beginning Kettle users who need a loop in a job create the loop in a simple and straightforward way, as shown in Figure 15-18.

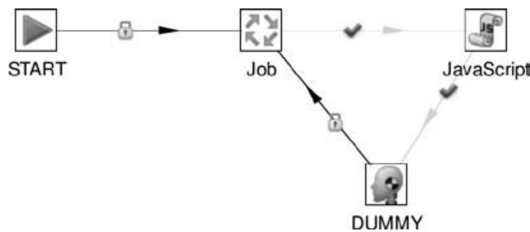


Figure 15-18: A simple but slow job loop

Typically, a single file is handled in each pass of the Job job entry. JavaScript or some other job entry is then used to evaluate if there are more files to process and, if there are, the job is continued.

If all you need to do is handle a few iterations, you will probably never get into trouble. However, if you need to perform a large number of iterations you will notice that the solution is slow. This performance problem is caused by the involvement of two extra steps as well as the loading of the metadata for the job that performs the actual work. All this extra work slows down the job. In addition to being slow, the job also runs the risk of running out of heap space because of the back-tracking algorithm used. See Chapter 2 for more information.

The better performing solution consists of a transformation that retrieves a list of files to process. The transformation, executed in the "Get filename" job entry in Figure 15-19, uses the "Copy rows to result" step to pass the list of files to the Job job entry.

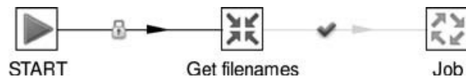


Figure 15-19: The proper way to loop in a job

With this setup, all you need to do is enable the "Execute for every input row?" option to make the Job job entry loop over the result rows. The advantage of this system is that no extra heap space is being used so the performance is a lot better because only the Job entry is being executed and because the Job metadata is loaded only once.

Database Connection Pools

In situations where small amounts of data need to be processed repeatedly in a job, for example as shown earlier in a loop, you will notice that connecting and disconnecting from a relational database can cause performance problems. Oracle and certain clustered databases are particularly slow to connect to. If the amount of data processed is large and it takes a while to process the data, then it might be fine to spend a few seconds to connect to a database. However, if you have tens of thousands of small files that need to be processed individually, the connection delay itself limits the performance of the job.

If you experience slow connection time, you might want to check the Enable Connection Pooling option in the Database Connection dialog as shown in Figure 15-20.

Connection pooling makes use of the Apache DBCP project (`http://commons.apache.org/dbcp/`). It essentially creates a pool of open database connection to make sure that these don't have to be closed and re-opened each time for every tiny transformation or job that is executed.

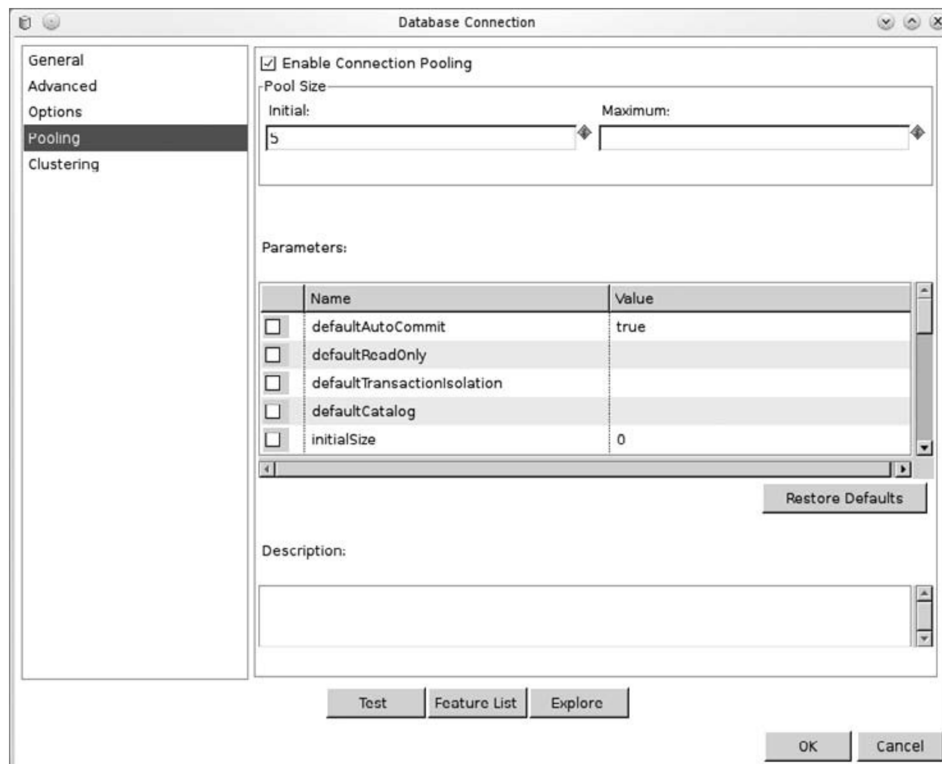


Figure 15-20: Enabling database connection pooling

Summary

In this chapter, you learned how the transformation engine works and how to detect performance bottlenecks in your transformations by simplification and by reading measurements.

You also learned how to improve the performance of your transformations on the following topics:

- Reading and writing text files
- Handling relational databases
- Reducing CPU usage with advice on how to tune your transformations

Finally, you learned how to create proper loops in jobs to increase performance and how to deal with slow connecting databases by turning on database connection pooling.

CHAPTER

16

Parallelization, Clustering, and Partitioning

When you have a lot of data to process it's important to be able to use all the computing resources available to you. Whether you have a single personal computer or hundreds of large servers at your disposal you want to make Kettle use all available resources to get results in an acceptable timeframe.

In this chapter, we unravel the secrets behind making your transformations and jobs scale up and out. *Scaling up* is using the most of a single server with multiple CPU cores. *Scaling out* is using the resources of multiple machines and have them operate in parallel. Both these approaches are part of ETL subsystem #31, the Parallelizing/Pipelining System.

The first part of this chapter deals with the parallelism inside a transformation and the various ways to make use of it to make it scale up. Then we explain how to make your transformations scale out on a cluster of slave servers.

Finally we cover the finer points of Kettle partitioning and how it can help you parallelize your work even further.

Multi-Threading

In Chapter 2, we explained that the basic building block of a transformation is the step. We also explained that each step is executed in parallel. Now we'll go a bit deeper into this subject by explaining how the Kettle multi-threading capabilities allow you to take full advantage of all the processing resources in your machine to scale up a transformation.

By default, each step in a transformation is executed in parallel in a single separate thread. You can, however, increase the number of threads, also known as *copies*, for any single step. As explained in Chapter 15, this can increase the performance of your transformation for those steps that are consuming a lot of CPU time.

Take a look at the simple example in Figure 16-1, where rows of data are processed by a User Defined Java Class step.

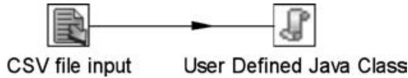


Figure 16-1: A simple transformation

You can right-click on the User Defined Java Class step and select the menu option "Change number of copies to start..." If you then specify **4**, you will see that the graphical representation of the transformation looks like the example shown in Figure 16-2.

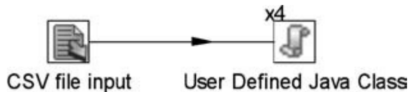


Figure 16-2: Running a step in multiple copies

The "4x" notation indicates that four copies will be started up at runtime.

Note that one copy of the description of the step is ever present or maintained by all step copies. Terminology is important so here are the definitions you need to understand the rest of this chapter:

- **Step:** The definition or metadata that describes the work that needs to be done
- **Step copy:** One parallel worker thread that executes the work defined in a step

In other words, a *step* is just the definition of a task, whereas a *step copy* represents an actual executing task.

Row Distribution

In the example, you have one step copy that sends rows to four copies. So how are the rows being distributed to the target step copies? By default, this is being done in a round-robin fashion. That means that if there are N copies, the first copy gets the first row, the second copy gets the second row, and the Nth copy receives the Nth row. Row N+1 goes to the first copy again, and so on until there are no more rows to distribute.

There is another option for the rare case in which you want to send all rows to all copies; you can enable the "Copy data to all steps" option in the step context menu. This option sends the rows to several target steps, as for example when you write data to a database table as well as to a text file. In this case, you'll get the warning dialog shown in Figure 16-3, asking which option you prefer.

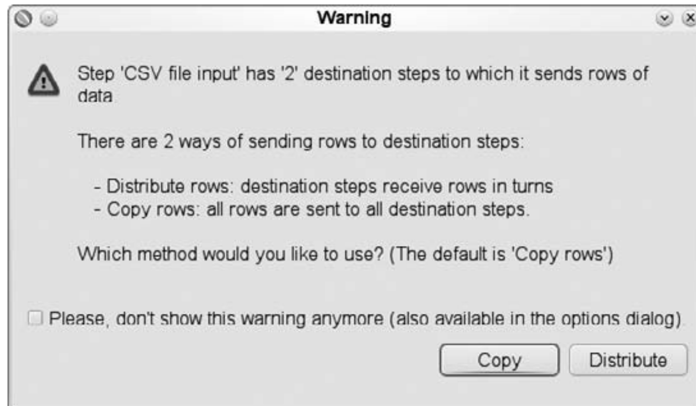


Figure 16-3: A warning dialog

Because you want to copy all the rows of data to both the database and the text file, you select Copy. The resulting transformation will look like the example in Figure 16-4.

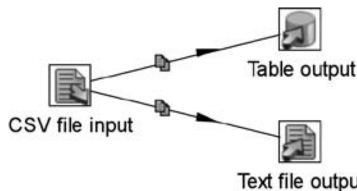


Figure 16-4: Copying data to multiple target steps

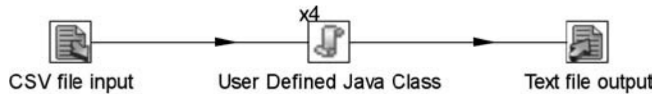
Because this is the exception and you usually want to process each row only once, the examples in rest of this chapter use row distribution, not row copying.

Row Merging

Row merging occurs when several steps or step copies send rows to a single copy. Figure 16-5 shows two examples of this.

From the standpoint of the "Text file output" and the "Add sequence" steps, rows are not read one at a time from each source step copy. That could lead to serious performance problems in situations where one step copy is sending few rows of data at a slow pace and another copy is producing rows at a fast pace. Instead, rows of data are being read in batches from the source step copies.

WARNING **The order in which rows are being read from previous step copies is never guaranteed!**



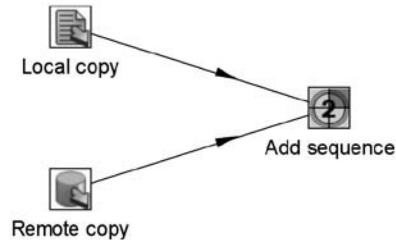


Figure 16-5: Merging rows of data

Row Redistribution

In *row redistribution*, you have *X* step copies that send rows to *Y* target step copies. Consider the example in Figure 16-6.

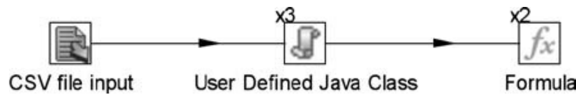


Figure 16-6: Row redistribution

The same rule applies as before with the distribution of rows: in the example, each of the three source step copies of User Defined Java Class distributes the rows over the two target step copies. The result of this is equivalent to the transformation shown in Figure 16-7.

The main advantage of the redistribution algorithm is that rows are equally distributed across the step copies. That prevents a situation in which certain step copies have a lot of work and others have very little to do.

As you can see in Figure 16-7, there are *X* times *Y* row buffers being allocated between the UDJC and the Formula step. In our example, there are six buffers (arrows) being allocated for three source and two target steps. Keep this in mind if you are designing transformations. In particular, if you have slow steps at the end of the transformations, these buffers can fill up to their maximum “Row set size.” That in turn can increase the memory consumption of your transformation. For example, take a look at the example in Figure 16-8.

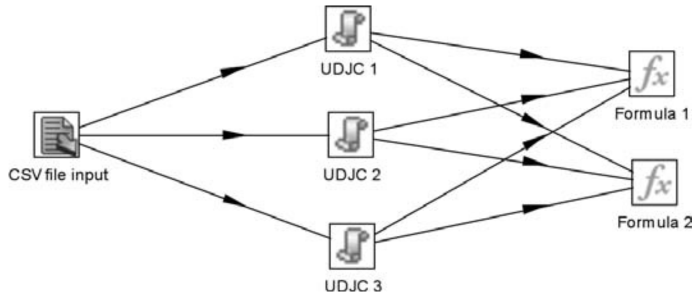


Figure 16-7: Row redistribution expanded

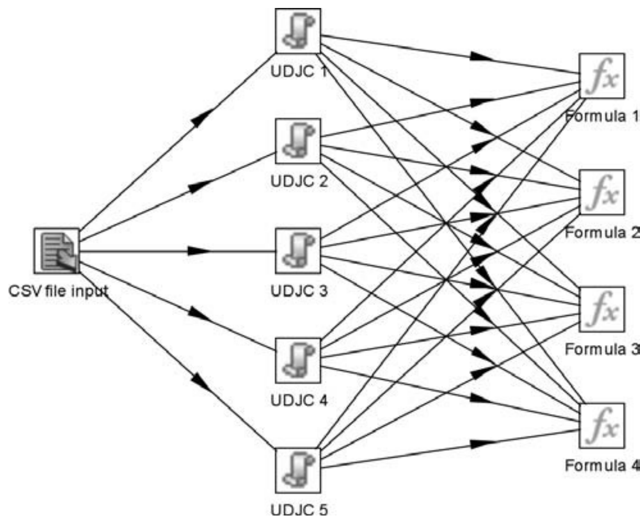


Figure 16-8: Redistribution can allocate a lot of buffers

Here you have five source and four target step copies and 20 buffers are allocated. That is despite the fact you only see a single arrow in the transformation! Because the default maximum row set size is 10,000, the total number of rows that could be kept in memory is 200,000.

Data Pipelining

Data pipelining is a special case of redistribution where the number of source and target step copies is the same (X==Y). In this case, rows are *never* being redistributed over all the step copies. Instead, the rows that are produced by source step copy 1 are being sent to the target step copy with the same number. Figure 16-9 offers an example of such a transformation.

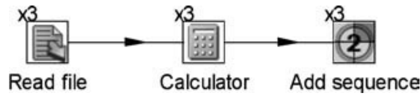


Figure 16-9: Data pipelining

It is technically the equivalent of the transformation shown in Figure 16-10.

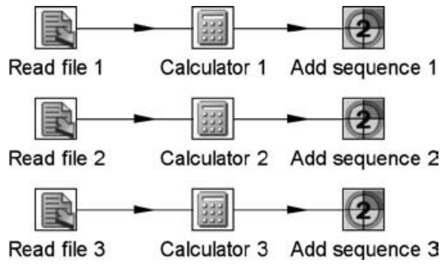


Figure 16-10: Data pipelining expanded

The process of distributing and merging rows has a small but measurable overhead. It is often better to keep the number of copies of consecutive steps the same to prevent this overhead.

The process of reducing the overhead of the data communication between the step copies is also referred to as putting the data into *swim-lanes*.

Consequences of Multi-Threading

In the previous section you learned that a transformation is multi-threaded and that all steps run in parallel. The topics that follow will show you the possible consequences of this execution model and how to deal with those consequences.

Database Connections

The recommended approach to dealing with database connections in multi-threaded software is to create a single connection per thread during the execution of a transformation. As such, each step copy opens its own separate database connection during execution. Each database connection uses a separate transaction or set of transactions.

This has as a potential consequence that race conditions can and often will occur in those situations where you are using the same database resource, such as a table or view, in the same transformation.

A common situation where things go wrong is when you write data to a relational database table and read it back in a subsequent step. Because the two steps run in

different connections with different transaction contexts, you can't be sure that the data being written in the first step will be visible to the other step doing the reading.

One common, straightforward solution to this challenge is to split up the transformation into two different transformations and keep data in a temporary table or file.

Another solution is to force all steps to use a single database connection with a single transaction. This is possible with the help of the "Make the transformation database transactional" option in the transformation settings dialog shown in Figure 16-11.

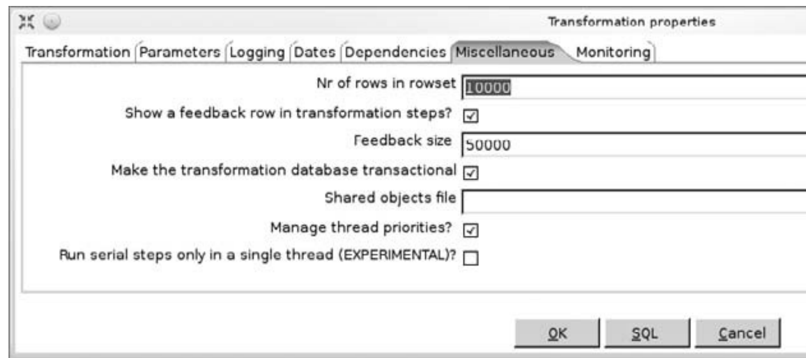


Figure 16-11: Making a transformation transactional

This option means that Kettle will use only a single connection per named database and will refrain from performing a commit or rollback until the transformation has finished. At that point, a commit will be performed if the transformation ran without errors and a rollback if any errors occurred. Note that any error that is being handled by step error handling will not cause a rollback.

The drawback of using this option is that it typically reduces the performance of the transformation. There are numerous reasons for this, ranging from the fact that all database communication now passes over a single synchronized connection to the fact that there is often only a single server-side process handling the requests.

Order of Execution

Because all steps in a transformation are executed in parallel there is no order of execution in a transformation. However, there are still things in data integration that are required to be executed in a certain order. In most situations, the answer to these problems is the creation of a job that will execute tasks in a specific order.

There are also ways of forcing things to be executed in a certain order in a Kettle transformation. A few tips follow.

The Execute SQL Step

If you need to execute SQL before everything else in the transformation, you can use the Execute SQL step. In normal operational mode, this step will execute the specified

SQL during the initialization phase of the steps. That means it is executed before the steps start to run.

You can also make the step operate during the execution of the steps by enabling the "Execute for each row?" option, as shown in Figure 16-12.

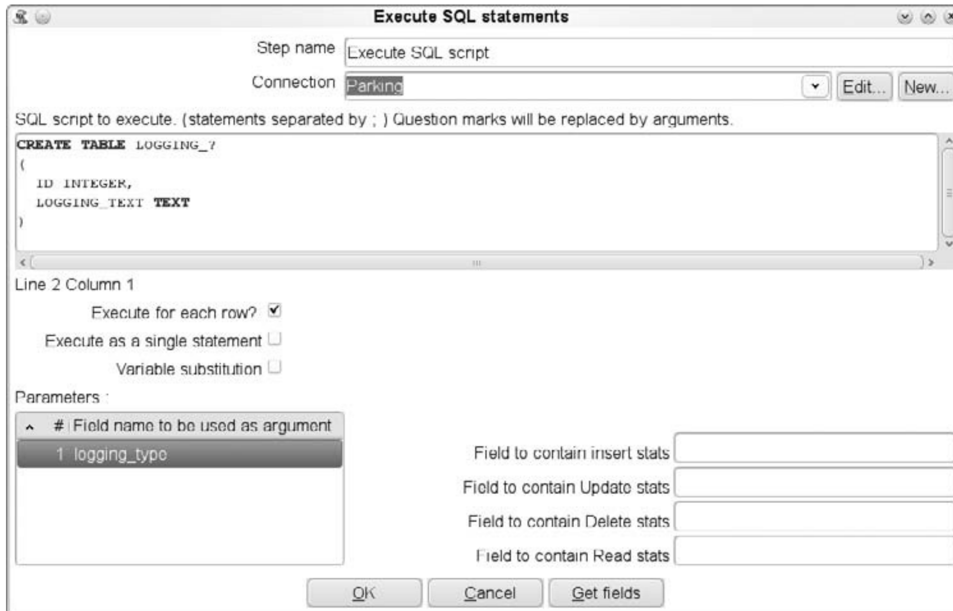


Figure 16-12: Execute SQL statements dialog

The Blocking Step

Another common use case is that you want to perform an operation after all the rows have passed a certain step. To do this, you can use the Blocking Step, as shown in Figure 16-13.

Figure 16-13: The Blocking Step

The Blocking Step simply eats all rows in the default configuration. When all rows have been eaten, it passes the last row to the next steps. This row will then trigger the subsequent steps to perform an operation. In this way, you are certain that all the other rows have been processed.

In the example shown in the figure, a SQL statement is performed after all the rows have been populated in the database table.

Parallel Execution in a Job

Job entries in a job execute one after the other. This is the default behavior because you usually want to wait for the completion of one job entry before starting the other. However, as mentioned in Chapter 2, it is possible to execute job entries in parallel in a job. In the case of parallel execution of job entries, different threads are started for all the job entries that are found after the job entry that executes in parallel.

For example, if you want to update multiple dimension tables in parallel you can do so as shown in Figure 16-14.

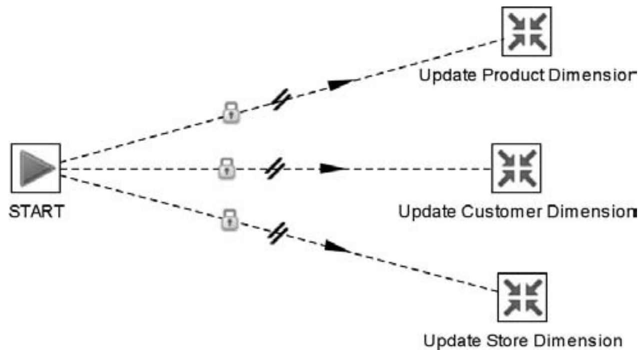


Figure 16-14: Updating dimensions in parallel

Using Carte as a Slave Server

Slave servers are handy building blocks that allow you to execute transformations and jobs on a remote server. Carte, a lightweight server process, allows for remote monitoring and enables the transformation clustering capabilities described in the next section of this chapter.

A *slave server* is the smallest building block of a cluster. It is a small HTTP server that accepts command from remote clients. These commands control the deployment, management, and monitoring of jobs and transformations on the slave server.

As described in Chapter 3, the Carte program is available to perform the function of a slave server. Carte also allows you to perform remote execution of transformation and jobs.

The easiest way to start a slave server is by specifying the hostname or IP address to run on as well as the port the Web server should run on. For example, the following command will start up a slave server on port `8181` on `server1`:

```
sh carte.sh server1 8181
```

The Configuration File

In earlier versions of Kettle, you specified configuration options on the command line. Because the number of options has increased, the latest version of Kettle relies on an

XML format for the configuration of the slave server. If you have a configuration file, you can also execute the slave server like this:

```
sh carte.sh slave-simple.xml
```

The configuration file in our example `slave-simple.xml` is a block of XML that describes all the attributes of a slave server. Here is a simple example:

```
<slave_config>
  <!--
     - A simple slave server configuration
  -->

  <max_log_lines>0</max_log_lines>
  <max_log_timeout_minutes>0</max_log_timeout_minutes>
  <object_timeout_minutes>5</object_timeout_minutes>

  <slaveserver>
    <name>server1</name>
    <hostname>server1</hostname>
    <port>8181</port>
  </slaveserver>

</slave_config>
```

The `<slaveserver>` XML block describes the hostname and port that the slave server should listen to and enables you to configure various aspects of the slave server. In general, these options enable you to fine-tune the memory usage of a long-running server process such as Carte:

- `max_log_lines`: Set this option to configure the maximum number of log lines that the slave server logging system should keep in memory at most. See also Chapter 14 for more information about the logging system.
- `max_log_timeout_minutes`: This parameter describes the maximum time in minutes that a log line should be kept in memory. This is especially useful for long-lived transformations and jobs to prevent the slave server from running out of memory. For more information about this topic see Chapter 18.
- `object_timeout_minutes`: By default, all transformations and jobs stay visible in the slave server status report indefinitely. This parameter allows you to automatically clean out old jobs from the status lists.

Defining Slave Servers

To define a slave server in your transformation or job, simply go to the View section on the left side of Spoon, right-click on the “Slave server” tree item and select New. You can then fill in the details for the slave server, as in the example in Figure 16-15.

Figure 16-15: Defining a slave server

Remote Execution

A transformation or a job can be executed remotely by specifying the slave server to run on in the Spoon "Execute a transformation" dialog. When called from a job, it can be executed remotely by execution in a Job or Transformation job entry by setting a remote slave server value in the Job or Transformation job entry dialog.

Monitoring Slave Servers

A slave server can be monitored remotely in a few different ways:

- **Spoon:** Right-click on a slave server in the View tree of Spoon and select the Monitor option. This will present a monitoring interface in a separate tab in the Spoon user interface that contains a list of all the transformations and jobs that are running on the slave server.
- **A Web browser:** Open a browser window and type in the address of the slave server. In our example, that would be `http://server1:8181/`. The browser will show you a basic but functional slave server menu that provides the ability to control and monitor your slave server.
- **PDI Enterprise Console:** Part of the Pentaho Data Integration Enterprise Edition, the enterprise console is capable of monitoring and controlling slave servers.
- **Your custom application:** Each of the services exposed by a slave server gives back data in the form of XML. These simple web services allow you to communicate with the slave server in a convenient and standard way. If you are using the Kettle Java libraries, you can also take advantage of the fact that the parsing of XML is already handled by Kettle Java classes.

Carte Security

Carte uses simple HTTP authentication by default. The usernames and passwords are defined in the file `pwd/kettle.pwd`. The default username/password that Kettle ships with is `cluster`.

The password in this file can be obfuscated with the Encr tool that ships with Kettle. To generate a password for a Carte password file, use the option `-carte`, as in this example:

```
sh encr.sh -carte Password4Carte
OBF:1ox61v8s1yf41v2p1pyr1lfe1vgt1vg11lc41pvv1v1p1yf21v9u1oyc
```

The returned string can then be placed in the password file after the username, using a text editor of your choice:

```
someuser: OBF:1ox61v8s1yf41v2p1pyr1lfe1vgt1vg11lc41pvv1v1p1yf21v9u1oyc
```

The `OBF:` prefix tells Carte that the string is obfuscated. If you don't want to obfuscate the passwords in this file, you can simply specify the password in clear text like this:

```
someuser: Password4Carte
```

Note that the passwords can be *obfuscated*, not *encrypted*. The algorithms used make it harder to read the passwords, but certainly not impossible. If a piece of software is capable of reading the password, you must assume that someone else could do it, too. For this reason, you should always put appropriate permissions on the password file. If you prevent unauthorized access to the file, you reduce the risk of someone being able to decipher the password in the first place.

It is also possible to use JAAS, short for the Java Authentication and Authorization Service, to configure security of Carte. In that case, you have to define two system properties (for example in the `kettle.properties` file):

- `loginmodulename`: The name of the login module to use.
- `java.security.auth.login.config`: This points to the JAAS configuration file that needs to be used.

The name of the JAAS user realm is Kettle. The details of configuring JAAS are beyond the scope of this book. More information can be found on the JAAS home page at `http://java.sun.com/products/jaas/`.

Services

A slave server provides a bunch of services to the outside world. Table 16-1 lists the defined services that are provided. The services live in the `/kettle/` URI on the embedded web server. In our example server, that would be `http://server:8181/kettle/`. All services accept the `xml=Y` option to make it return XML that can be parsed by a Kettle Java class. The class used, from package `org.pentaho.di.www`, is also mentioned in Table 16-1.

Table 16-1: Slave Server Services

SERVICE NAME	DESCRIPTION	PARAMETERS	JAVA CLASS
`status`	Gives back a summary status containing all transformations and jobs.		`SlaveServerStatus`
`transStatus`	Retrieves the status of a single transformation and lists the status of steps.	`name` (name of the transformation); `from line` (start logging line for incremental logging)	`SlaveServerTransStatus`
`prepareExecution`	Prepares a transformation for execution, performs the initialization of steps.	`name` (name of the transformation)	`WebResult`
`startExec`	Starts the execution of the steps.	`name` (the name of the transformation)	`WebResult`
`startTrans`	Performs initialization and execution of a transformation in one go. While convenient this is not used for clustered execution since initialization needs to be performed simultaneously across the cluster.	`name` (the name of the transformation)	`WebResult`
`pauseTrans`	Pauses or resumes a transformation.	`name` (the name of the transformation)	`WebResult`
`stopTrans`	Terminates the execution of a transformation.	`name` (the name of the transformation)	`WebResult`
`addTrans`	Adds a transformation to the slave server. This requires the client to post the XML of the transformation to Carte.		`TransConfiguration` `WebResult`
`allocateSocket`	Allocates a server socket on the slave server. Please see the "Clustering Transformations" section later in this chapter for more information.		

Continued

Table 16-1 *(continued)*

SERVICE NAME	DESCRIPTION	PARAMETERS	JAVA CLASS
`sniffStep`	Retrieve the rows that are passing through a running transformation step.	`trans` (name of the transformation); step (name of the step); `copy` (copy number of the step); `lines` (number of lines to retrieve); `type` (input or output hop of a step)	`<step-sniff>` XML containing a `RowMeta` object as well as serialized row data.
`startJob`	Start the execution of a job.	`name` (the name of the job)	`WebResult`
`stopJob`	Terminates the execution of a job.	`name` (the name of the job)	`WebResult`
`addJob`	Adds a job to the slave server. This requires the client to post the XML of the job to Carte.		`JobConfiguration` `WebResult`
`jobStatus`	Retrieves the status of a single job and lists the status of job entries.	`name` (name of the job); `from` (start logging line for incremental logging)	`SlaveServerJobStatus`
`registerSlave`	Registers a slave with a master (see the "Clustering Transformations" section). This requires the client to post the XML of the slave server to the slave server.		`SlaveServerDetection` `WebResult` (reply)
`getSlaves`	Gives a list back of all the slave servers that are known to this Master slave server.		`<SlaveServerDetections>` XML block containing `SlaveServerDetection` items
`addExport`	This method allows you to transport an exported job or transformation over to the slave server as a .zip archive. It ends up in a temporary file. The client needs to post the content of the zip file to the Carte server. This method always returns XML because it has no use in any other capacity.		`WebResult` containing the file URL of the created temporary file

Clustering Transformations

Clustering is a technique that can be used to scale out transformations to make them run on multiple servers at once, in parallel. It spreads the transformation workload over different servers. In this section, we cover how you can configure and execute a transformation to run across multiple machines.

A cluster schema consists of one master server that is being used as a controller for the cluster, and a number of non-master slave servers. In short, we refer to the controlling Carte server as the *master* and the other Carte servers as *slaves*.

A cluster schema also contains metadata on how master and slaves pass data back and forth. Data is passed between Carte servers over TCP/IP sockets. TCP/IP was chosen as the data exchange protocol because passing through Web services would be too slow and cause unnecessary overhead.

NOTE The concepts *master* and *slave* are only important when dealing with cluster schemas. To make a slave server a master, simply check the "Is the master?" check-box in the slave server dialog. You do not need to pass any specific option to Carte.

Defining a Cluster Schema

Before you define a cluster schema, you need to define a number of slave servers. (See the previous section in this chapter for instructions on defining a slave server.) Once that is done, you can right-click on the "Kettle cluster schemas" tree item and select the New option, as shown in Figure 16-16.

You can then specify all the details for your cluster schema. Make sure to select at least one master to control the cluster and one or more slaves (see Figure 16-17).

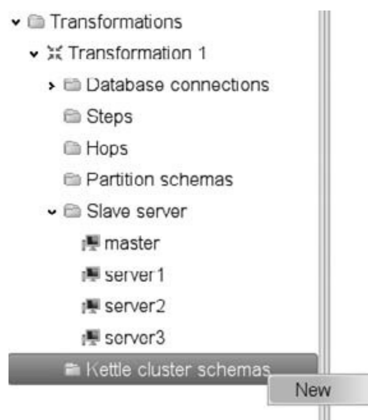


Figure 16-16: Creating a new cluster schema

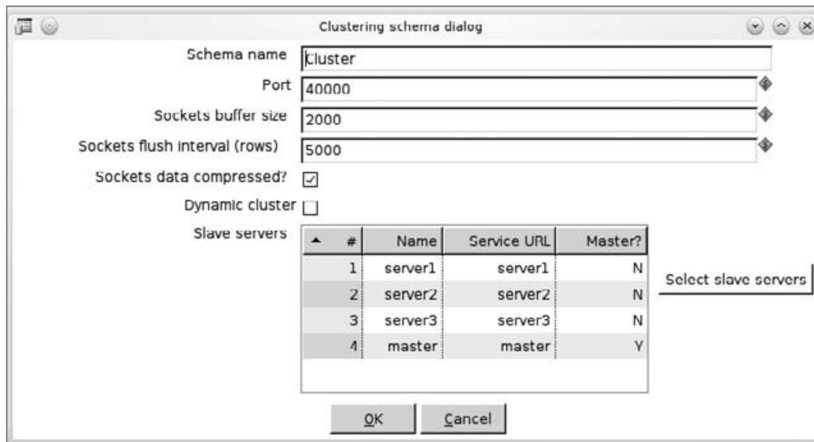


Figure 16-17: The "Clustering schema" dialog

Here are the notable options:

- **Port:** The lowest TCP/IP socket port number that will be used to transport data from one slave to another. It is only a starting point. If your cluster transformation requires 50 ports, it means that all ports between Port and Port+50 will be used.
- **Sockets buffer size:** The buffer size used to smooth communications between slave servers. Make sure that you don't make this value too high because it can cause undesirable oscillations in the data passing process.
- **Sockets flush interval (rows):** This is the number of rows after which the transformation engine will perform a flush on the data sockets to make sure the data is forced to the remote slave server. The performance implications of setting this parameter and the value to select are heavily dependent on the speed and latency of the network between the slave servers.
- **Sockets data compressed?** Determines if the data will be compressed as it is passed between slave servers. While this is great for relatively slow networks (10Mbps for example), setting this to "Yes" causes the clustered transformation to slow down a fair amount because additional CPU time is being utilized for compression and inflation of the data streams. As such, it's usually best to disable this option if you do not find that the performance of the network is a limitation.
- **Dynamic cluster:** When enabled, this option will make Kettle look at the master to determine the list of slaves for the cluster schema. For more information about dynamic clusters, see Chapter 17.

Designing Clustered Transformations

To design a clustered transformation, start by building a regular transformation. To transition to clustering, create a cluster schema as described previously and then select

the steps that you want to execute on slave servers. Right-click on the step to select the cluster you want the step to execute on.

For example, you might want to read data from a large file that is stored on a shared network drive, sort the data, and write the data back to another file. You want to read and sort the data in parallel on your three slaves. Figure 16-18 illustrates how you would start.

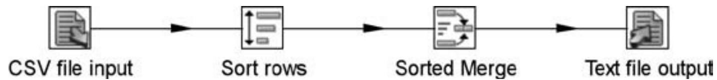


Figure 16-18: A regular transformation

The next step is to then select the steps you want to execute on the slaves, the "CSV file input" and "Sort rows" steps. Select Clustering... from the step's context menu. After selecting the cluster schema this step is to run on, you end up with the transformation shown in Figure 16-19.

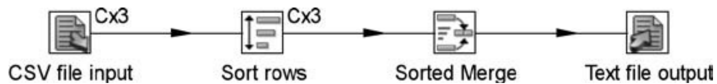


Figure 16-19: A clustered transformation

When you execute this transformation, all steps that are defined to run clustered (those with the Cx3 notation in Figure 16-19) will be run on the slaves. Those steps that don't have the cluster indication will run on the master.

NOTE **In Figure 16-19 rows are sorted in parallel using three "Sort rows" steps on three different slave servers. This results in the same number of groups of sorted rows that are sent back to the master. Because Kettle reads rows in blocks from previous steps you have to take action to keep the rows in a sorted order. This task is performed by the Sorted Merge step that reads rows one by one from all input steps and keeps them in a sorted order. Without this step, the parallel sort would not lead to correct results.**

A transformation is considered to be a *clustered transformation* if at least one step in the transformation is assigned to run on a cluster. Clustered transformations can be executed in a non-clustered method for testing and development using the Execution dialog in Spoon.

NOTE **It's important to remember that only one cluster can be used in any single transformation!**

Execution and Monitoring

You have a couple of choices for running a clustered transformation. One option is to run it in Spoon by selecting the "Execute clustered" option in the execution dialog (see Figure 16-20).

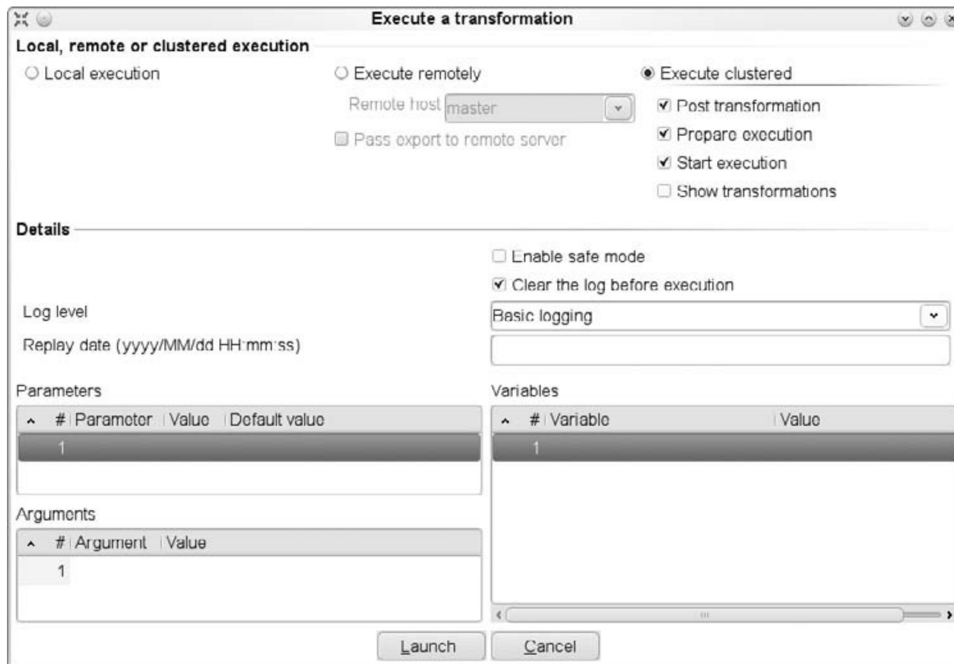


Figure 16-20: Executing a clustered transformation from the "Execute a transformation" dialog

For debugging purposes, you can use the following clustering options:

- **Post transformation:** When selected, this will post the generated transformations to the slaves and master.
- **Prepare execution:** This option will run the initialization of the generated transformations on the slaves and the master.
- **Start execution:** When this option is enabled, the clustered transformation will be started on the master and the slaves.
- **Show transformations:** This option will open the master and slave transformations in Spoon so you can see what kind of transformations are generated. More information on the slave and master transformations is provided in the following section.

Please note that the first three options must be enabled to completely execute a clustered transformation. The fourth option does not require running the transformation but simply enables you to see the generated transformations.

Another option to run a clustered transformation is to run it as part of a job with a Transformation job entry. In that job entry, you can enable the "Run this transformation in a clustered mode?" option to make the transformation run on a cluster (see Figure 16-21).

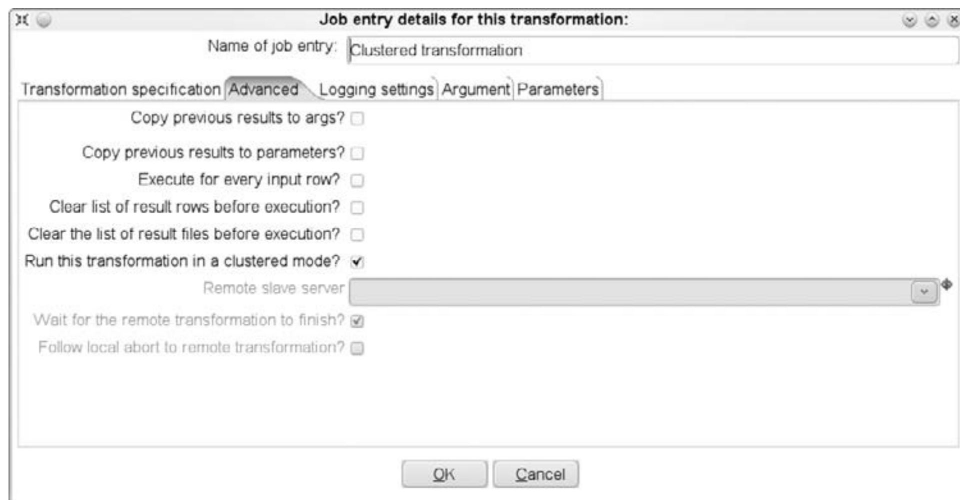


Figure 16-21: Execute clustered transformations with a job entry

Metadata Transformations

It is not sufficient to simply run the same transformation on the master and the slaves. That would not typically lead to a correct implementation for your parallel data processing requirements. The transformations that get executed on the master and the slaves are generated after a translation process called *metadata transformations*. The ETL metadata of the original transformation, the one you designed in Spoon, is chopped up in pieces, reassembled, enriched with extra information and sent over to the target slave.

Following the metadata transformation, you have three types of transformations:

- **Original transformation:** A clustered transformation as designed in Spoon by the user.
- **A slave transformation:** A transformation that was derived from the original transformation to run on a particular slave. There will be one slave transformation for each slave in the cluster.
- **A master transformation:** A transformation that was derived from the original transformation to run on the master.

In the case of the clustering example in Figure 16-19, three slave transformations and one master transformation will be generated. Figure 16-22 shows what the master transformation looks like in our example.

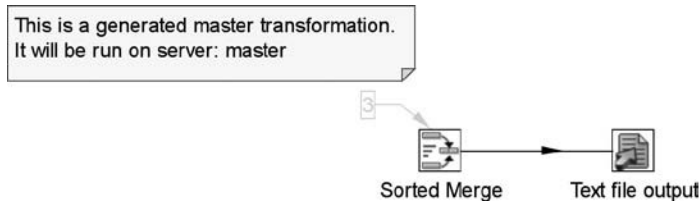


Figure 16-22: A master transformation

Figure 16-23 illustrates what the slave transformations look like.

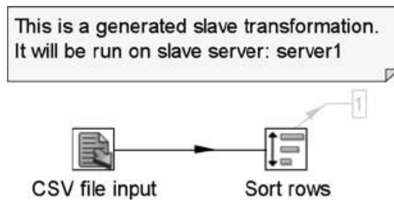


Figure 16-23: A slave transformation

The light-gray numbered areas of the transformations indicate that those steps have remote input or output connections called *Remote Steps*. In our example, there are three slaves. Each slave sends data from the "Sort rows" to the Sorted Merge step. This means that each of the three "Sort rows" steps has one remote output step and that the one Sorted Merge step has three remote input steps. If you hover the mouse over the light-gray rectangle, you will get more information on the remote steps and the port numbers that got allocated, as shown in Figure 16-24.

Figure 16-24: Tooltip information showing remote steps

Rules

As you can imagine, there are a lot of possibilities to consider when doing these metadata transformations. Let's take a look at a few of the general rules that Kettle

uses during generation to ensure correct logical operations that match the original transformation:

- A step is copied to a slave transformation if it is configured to run clustered.
- A step is copied to a master transformation if it is not configured to run clustered.
- Remote output steps (sending data over TCP/IP sockets) are defined for steps that send data to a clustered step.
- Remote input steps (accepting data from TCP/IP sockets) are defined for steps that accept data from a clustered step.

The following rules are more complicated because they deal with some of the more complex aspects of clustering:

- Running steps in multiple copies is supported. In such cases the remote input and output steps will be distributed over the number of copies. It makes no sense to launch more copies as these are remote steps.
- In general, Kettle clustering requires you to keep transformations simple in nature to make the generation of transformations more predictable.
- When a step reads data from specific steps (info-steps), Socket Reader and Socket Writer steps will be introduced to make the transformation work, as is the case in the transformation shown in Figure 16-25.

Figure 16-26 illustrates what the resulting slave transformations look like. Figure 16-27 shows what the master transformation will look like.

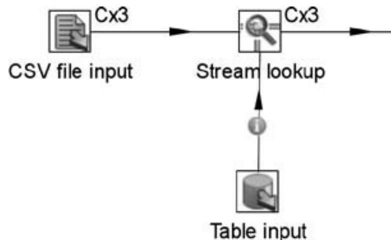


Figure 16-25: Delivering data to a clustered step

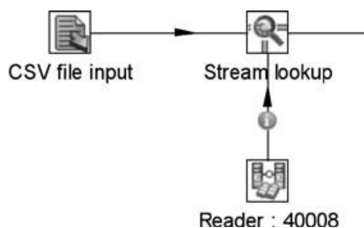


Figure 16-26: A slave transformation with a reader

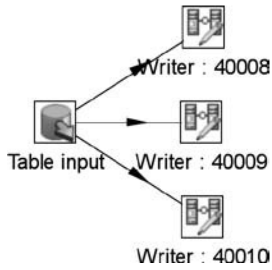


Figure 16-27: A master distributing data to slaves

As a careful eye will notice, the "Table input" step is distributing the rows over the different socket writers that pass the data to the slave servers. This is not really what we intended to happen. Make sure to copy the data to the multiple copies running on the remote slaves in these situations (see Figure 16-28).

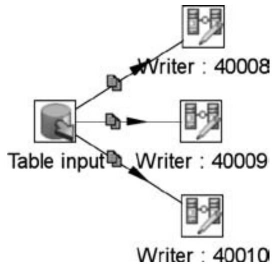


Figure 16-28: Copying data to the slaves

Again, it might be more prudent to simply read the data three times on the slaves, as shown in Figure 16-29.

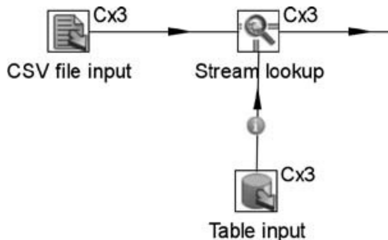


Figure 16-29: Acquiring data on the slaves

Data Pipelining

Recall the observation made earlier in this chapter regarding data pipelining or data in swim-lanes: the more data that is communicated between Carte servers, the slower the transformation will get. Ideally, your data is structured in such a way that you can parallelize everything from source to target. In that respect, it is easier to have 100 small XML files to process compared to having a single large one because the data can be read in parallel with the multiple files.

As a general rule, this is the key to getting good performance out of your clustered transformations: Keep it simple, do as much as possible in swim-lanes on the same slave, and reduce the amount of data passed between servers.

Partitioning

Partitioning is a very general term and in a broad sense simply means splitting up in parts. In terms of data integration and databases, *partitioning* refers to splitting of database tables or entire databases (*sharding*). Tables can be partitioned into *table partitions* and entire databases can be partitioned into *shards*.

In addition, it is quite possible to have text or XML files partitioned, for example per store or region. Because a data integration tool needs to support all sorts of technologies, partitioning in Kettle was designed to be source- and target-agnostic in nature.

Defining a Partitioning Schema

Partitioning is baked into the core of the Kettle transformation engine. Whenever you distribute rows over a number of target steps you are partitioning the data. The partitioning rule in this case is round robin. Because this rule is, in fact, not much better than random distribution, it is not usually referred to as a partitioning method.

When we talk about partitioning in Kettle, we refer to the capability Kettle has to direct rows of data to a certain step copy based on a partitioning rule. In Kettle, a given set of partitions is called a *partitioning schema*. The rule itself is called the *partitioning method*. A partitioning schema either can contain a list of named partitions or can simply contain a number of partitions. The partitioning method is not part of the partitioning schema.

Figure 16-30 offers a simple example where we define a *partitioning schema* with two partitions, *A* and *B*.

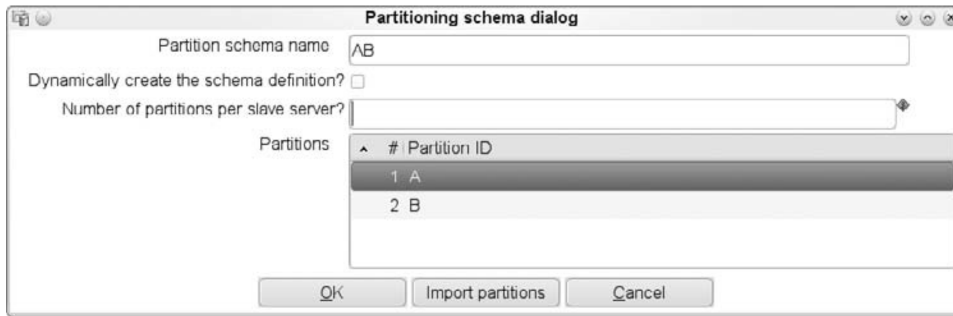


Figure 16-30: The "Partitioning schema" dialog

Once the partitioning schema is defined, you can use it in a transformation by applying it to a step along with a partitioning method. When you select the Partitioning... option from the step context menu, you are presented with a dialog to choose which partitioning method to use (see Figure 16-31). The partitioning method can be one of the following:

- **None:** Does not use partitions. The standard "Distribute rows" or "Copy rows" rule is applied.
- **Mirror to all partitions:** A special case that we describe later in the "Database Partitions" section.
- **Remainder of division:** The standard partitioning method in Kettle. Kettle divides the partitioning field as an integer (or a checksum if another data type) by the number of partitions. The remainder, or modulo, is used to determine which partition the row will be sent to. For example if you have user identification number "73" in a row and there are 3 partitions defined, then the row belongs to partition 1. Number 30 belongs to partition 0 and 14 to partition 2.
- **Partition methods implemented by plugins:** This option is not available from the partitioning method dialog. Refer to Chapter 23 for a partitioning plugin example and more information on how to write plugins.

You then need to specify the partitioning schema to use. In the example, you would select AB, as shown in Figure 16-32.

At that point, a plugin-specific dialog will be displayed to allow you to specify the arguments for the partitioning method. In our case, we need to specify the field to partition on (see Figure 16-33).

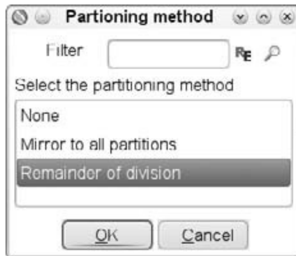


Figure 16-31: Selecting the partitioning method

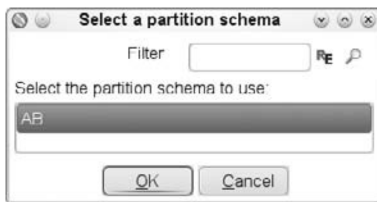


Figure 16-32: Selecting a partitioning schema

Figure 16-33: Specifying the field to partition on

Objectives of Partitioning

The objective of using partitioning is to increase the parallelism in a transformation. In many cases, this simply can't be done because the problem is not parallelizable by just distributing the load across multiple copies, or servers.

Take for example a Group By step; for simplicity, let's use the Memory Group By step. If you were to run multiple copies of this step with the standard transformation row distribution, you would almost certainly not end up with correct results because records belonging to a certain group could end up at any given step copy. The aggregate totals would not be correct because the steps may not have seen all the rows that are part of a group.

Let's consider a simple example. You have a text file containing customer data and you want to calculate the number of distinct ZIP codes per state in the file (see Figure 16-34).

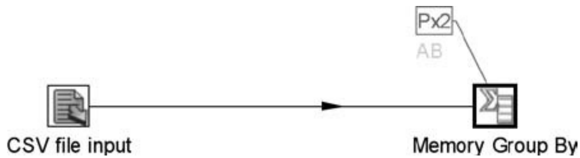


Figure 16-34: A partitioning sample

The file is read and the partitioning method is applied to the state field. Using this partitioning method, you can ensure that you always send rows of data with the same state to the same step copy. This in effect allows you to run the Memory Group By step in multiple copies and produce correct results, something that would not be possible otherwise.

Another reason to partition would be if you would like to run Database Lookup and Dimension Lookup/Update steps in parallel. If you apply partitioning to these steps, you would likely increase the positive cache hits. That is because you guarantee that a row of data with the same key would wind up at the same step copy and increase the likelihood that the value would already be in memory.

Implementing Partitioning

The implementation of partitioning in Kettle is simple: for each partition that is defined, multiple step copies are started for steps with a partitioning method defined. That means that if you define five partitions, you have five step copies to do the work. The step prior to the partitioned step, in Figure 16-34 the "CSV file input" step, is the one that is doing the re-partitioning. Re-partitioning is performed when data is not partitioned and needs to be sent to a partitioned step. It is also performed when data is partitioned using one partition schema and is sent on a hop to a step that is using a different partition schema.

Internal Variables

To facilitate data handling that is already in a partitioned format, Kettle defines a number of internal variables to help you:

- `${Internal.Step.Partition.ID}`: This variable describes the ID or name of the partition to which the step copy belongs. It can be used to read or write data external to Kettle that is in a partitioned format.
- `${Internal.Step.Partition.Number}`: This variable describes the partition number from 0 to the number of partitions minus one.

For example, if you have data that is already partitioned in *N* text files (file-0 to fileN with N being the number of partitions minus one), you could create a "CSV file input" step that reads from filename `file-${Internal. Step.Partition.Number}.csv`. Each step would read data only from the file that contains its partitioned data.

Database Partitions

Database partitions, or shards, can be defined in Kettle in the database connection dialog. You can do so on the Clustering tab when configuring a database connection. Kettle assumes that all database partitions are of the same database and connection type (see Figure 16-35).

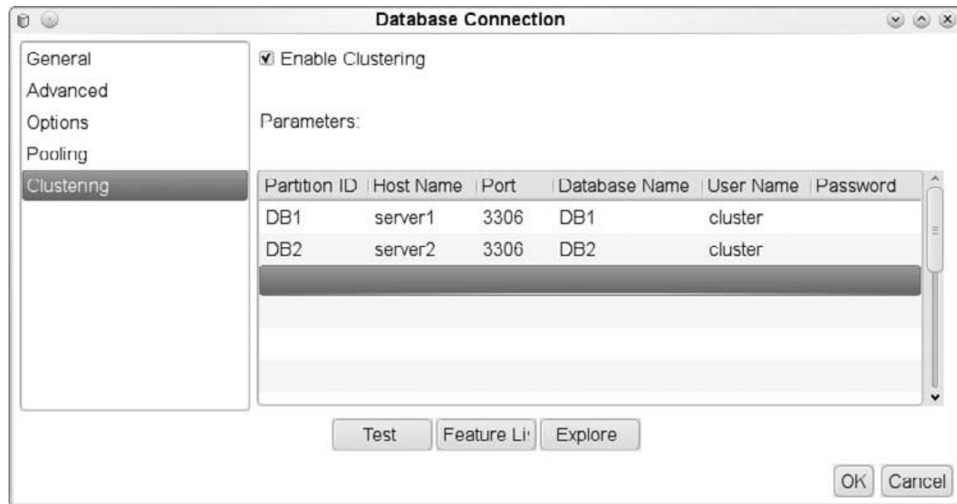


Figure 16-35: Defining database partitions

The goal of defining partitions is to read and write data pertaining to a certain partition to a certain physical database. Once you define the database partitions in the database connection, you create a partitioning schema based on this information. To do this, you use the "Import partitions" button in the "Partitioning schema" dialog (refer back to Figure 16-30).

Now you can apply the partitioning schema to any step that uses this partitioned database connection. One step copy will be launched for each database partition and it will connect to the physical databases defined as database partitions with the same name as the one used to partition the step.

Figure 16-36 shows an example of a query that is being executed in parallel against two different database partitions. The data is pipelined to the next two copies of the step to calculate certain things.

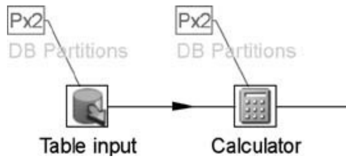


Figure 16-36: Reading database partitions

The same principle applies to all database steps and can be used to maintain a number of databases in parallel. The "Mirror to all partitions" partitioning method was designed specifically to allow you to write the same data to multiple database partitions in parallel. This is useful to populate lookup tables that need to be replicated on multiple database partitions without the need to define multiple connections.

Partitioning in a Clustered Transformation

In a situation in which many partitions are being defined, the number of step copies can grow prohibitively large in a transformation. Solving this involves spreading the partitions over a number of slaves in a clustered transformation.

During transformation execution, the available partitions are allocated equally among the available slaves. If you have defined a partitioning schema with a static list of partitions, those partitions will be divided over the number of slave transformations at run-time. The limitation in Kettle here is that the number of partitions has to be equal to or larger than the number of slaves and is usually a multiple of the number of slaves for even distribution (slaves × 2, slaves × 3). A simple way to solve this is to specify the number of partitions you want to run per slave server dynamically without preconfiguring a fixed set of partitions. This is possible as well, as shown previously in Figure 16-30.

Keep in mind that if you use a partitioned step in a clustered transformation, the data needs to be re-partitioned across the slaves. This can be cause for quite a bit of data communication between slave servers. For example, if you have 10 slaves in a cluster schema with 10 copies of step A and you partition the next step B to run with 3 partitions per slave, 10 × 30 data paths will need to be created, similar to the sample in Figure 16-7. 10 × 30 – 30 = 270 of these data paths will consist of remote steps, causing a lot of network traffic as well as CPU and memory consumption. Please consider this effect when designing your clustered and partitioned transformations.

Summary

In this chapter, you took a look at multi-threading in transformations, clustering, and partitioning. Here are some of the main points covered:

- You learned how a transformation executes steps in parallel and how rows are distributed when steps are executed with multiple step copies. We described

how data is distributed and merged back together, and we covered a few typical problems that can arise from this.

- We showed you how a slave server can be deployed to execute, manage, and monitor transformations and jobs on remote servers.
- You took an in-depth look at how multiple slave servers can be leveraged to form a cluster and how transformations can be made to utilize the resources of these slave servers.
- Finally, you learned how Kettle partitioning can help you parallelize steps that operate on groups of data and how partitioning can improve cache hits. You also saw how this can be applied to text file and database partitioning by using partition variables and partitioned database schemas.

CHAPTER

17

Dynamic Clustering in the Cloud

This chapter continues where Chapter 16 left off and explains how clusters don't have to consist of a fixed number of slaves but rather can be dynamic in nature. Once you've learned all about dynamic clustering, we introduce you to a dynamic set of resources called *cloud computing*. We then move on to a practical implementation of one cloud computing service: the *Amazon Elastic Compute Cloud* (EC2). We finish off the chapter by explaining how you can configure your own set of servers on Amazon EC2 for use as a cluster.

Dynamic Clustering

While at most organizations it is still standard practice for most ETL developers to have only one or two servers to work with, it's becoming more common to have a whole set of machines available as a set of general compute resources. This section describes how Kettle clustering can enable you to take advantage of a dynamic pool of computer resources.

Even before terms such as *cloud computing* and *virtual machines* became popular, initiatives like SETI@Home were already utilizing computer resources dynamically. SETI@Home was one of the very first popular distributed dynamic clusters; people all over the world contributed processing power to help the Search for Extra Terrestrial Intelligence. The SETI@Home cluster is dynamic in configuration because the number of participating nodes is constantly changing. In fact, SETI@Home is implemented as a screensaver so it's impossible to say up-front how many machines participate in the cluster at any given time.

With the advent of cloud computing and virtual machines, computing resources have become available at very low cost. As such, it makes sense to have support for dynamically changing cluster configurations in Kettle.

In a dynamic Kettle cluster schema you will define only a master. Slaves are not registered in the cluster schema as described in the previous chapter. Rather, the slaves are configured to register themselves with the master server. This way the master knows about the configuration of the cluster at all times. The master will also check the availability of slaves and will remove them from the cluster configuration if they are not available.

Creating a dynamic cluster schema is much like creating a normal cluster schema (see the section "Defining a Cluster Schema" in Chapter 16). You can make a cluster schema dynamic simply by checking the "Dynamic cluster" option in the "Clustering Schema dialog," as shown in Figure 17-1.

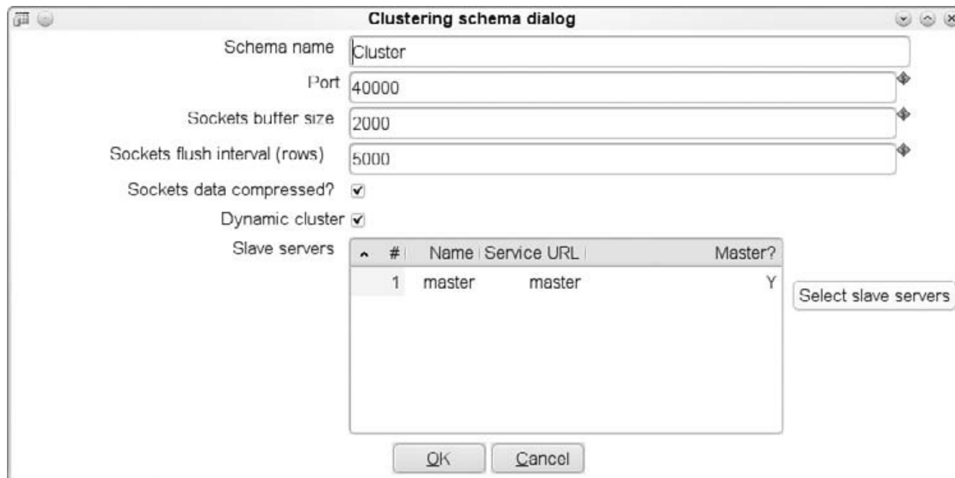


Figure 17-1: Creating a dynamic cluster schema

Setting Up a Dynamic Cluster

To set up a dynamic cluster, start by launching a single instance of Carte that you will use as the master. Ideally, the master should be run on a system that is known in the network. Typically, a machine with a fixed IP address or hostname is used. Because slaves need to be able to contact this server, it is important that the location of the master is known before any slaves are started.

By way of example, a minimal configuration file for the master is available in your Kettle 4.*x* distribution in the `pwd/` folder. The file is called `carte-config-master-8080.xml` and its content is shown following:

```
<slave_config>
  <slaveserver>
```

```
    <name>master1</name>
    <hostname>localhost</hostname>
    <port>8080</port>
    <master>Y</master>
  </slaveserver>
</slave_config>
```

This configuration can be supplemented with additional logging parameters, as described in the previous chapter. From the viewpoint of Carte, this is a slave server like any other and you start it up as one. Execute the following command on a new UNIX terminal in the Kettle distribution folder:

```
sh carte.sh pwd/carte-config-master-8080.xml
```

This is the command on a Windows system:

```
 Carte.bat pwd/carte-config-master-8080.xml
```

The difference in dynamic clustering is specified in the slaves, not the master. Take a look at how you can make a slave register with your master. First, you must specify in the slave configuration file that the slave needs to report to the master, how it should connect to it, and what the username and password are:

```
<slave_config>
  <masters>
    <slaveserver>
      <name>master1</name>
      <hostname>localhost</hostname>
      <port>8080</port>
      <username>cluster</username>
      <password>cluster</password>
      <master>Y</master>
    </slaveserver>
  </masters>

  <report_to_masters>Y</report_to_masters>

  <slaveserver>
    <name>slave1-8081</name>
    <hostname>localhost</hostname>
    <port>8081</port>
    <username>cluster</username>
    <password>cluster</password>
    <master>N</master>
  </slaveserver>
</slave_config>
```

In the `<masters>` section of the configuration file, you can specify one or more Carte servers to which the slave needs to report. While multiple slave servers are allowed, failover and load balancing are not yet supported in version 4.0 of Kettle. Note that you

need to specify the username and password of *both* the master *and* the slave server. That is because the slave server needs to register with the master and the master needs to be able to see if the slave is still functional.

Finally, the `<report_to_masters>` option specifies that the slave should report to the master. Because this configuration file is also available in your Kettle download, you can start it up right away:

```
Sh carte.sh sh carte.sh pwd/carte-config-8081.xml
```

This time around, you will not only see the regular web server messages on the console, but also the following log entry:

```
Registered this slave server to master slave server [master1] on address
[localhost:8080]
```

At the same time, you can now open the following web page in a web browser: `http://carteserverhostname:8080/kettle/getSlaves/`. After logging in, your browser will show you the result of the request in XML:

```
<SlaveServerDetections>
 <SlaveServerDetection>
  <slaveserver>
   <name>Dynamic slave [localhost:8081]</name>
   <hostname>localhost</hostname>
   <port>8081</port>
   <webAppName/>
   <username>cluster</username>
   <password>Encrypted 2be98afc86aa7f2e4cb1aa265cd86aac8</password>
   <proxy_hostname/>
   <proxy_port/>
   <non_proxy_hosts/>
   <master>N</master>
  </slaveserver>
  <active>Y</active>
  <last_active_date>2010/03/02 22:22:47.156</last_active_date>
  <last_inactive_date/>
 </SlaveServerDetection>
</SlaveServerDetections>
```

With the `getSlaves` request, you can get a list of all the active and inactive slaves for a master at any time.

Using the Dynamic Cluster

Once you have started a number of slaves that are registered to a master and you have created a dynamic cluster schema with the master in it, you can design your transformation to use it. Because Kettle doesn't know at design time how many slaves are involved, the step is annotated with *Cx*N with *N* instead of the number of slave servers, as shown in Figure 17-2.

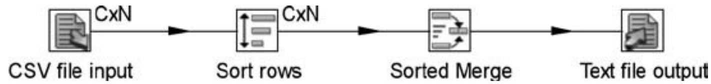


Figure 17-2: Steps to be executed by a dynamic cluster are marked as CxN

It is only at run-time that the list of slaves is retrieved and used. If new slaves are added during the execution of the transformation, they are not added dynamically to the running transformation. If slaves are turned off during the execution of the transformation, the slave transformation stops and the complete transformation will fail.

Cloud Computing

In the last couple of years, we've seen the introduction of a number of cloud computing services on the Internet. It all started with the experimental launch of Salesforce.com in 1999. Salesforce.com was one of the first companies to offer enterprise grade applications in the form of a website. Because of its success, many other companies followed suit. The resulting web applications have been labeled *Software as a Service* or *SaaS*.

A few years later, in 2002, Amazon felt that it could make more money out of its vast array of computing hardware and started its Amazon Web Services. Initially, it offered a number of web services to provide storage (S3), computation (EC2), and even a Mechanical Turk (`https:// www.mturk.com/`). It was, however, the launch of the *Elastic Computing Cloud* (EC2) initiative in 2006 that really helped create a new market, also called *Infrastructure as a Service* (*IaaS*). This made it possible to run and manage your own servers via a web service at a very low hourly rate.

Before EC2 and comparable services, you had all sorts of *grid-based* services, usually from very large companies like Oracle, Sun and IBM, that promised computing resources at a low hourly rate as well. The main difference between these grid computing platforms and cloud computing is that cloud computing uses virtualized resources to deliver computing power whereas grid computing uses actual machines.

The advantages to cloud computing are very important: A virtual server can be managed very tightly and the software you can run on it is without limits. Once you run a machine with a cloud provider like Amazon EC2, you use it exclusively and you can do whatever you want with it. Then you can create an image out of it and launch 20 copies to do more of the same. Because it can all be managed remotely, it can be scripted and managed automatically. The combination of these advantages, together with the low pricing (starting at USD $0.085 per hour) quickly made Amazon EC2 a very popular choice to run all kinds of services.

The fact that you can create more virtual machine instances based on an image on the fly is what allows the creation of dynamic clusters. It is also a challenge, because it means software must be created in such a way that it can take advantage of these instances without knowing typical identifying characteristics such as IP address in advance. This is what necessitates the concept of *dynamic clustering*: the ability to define a cluster without knowing in advance which machines will take part in it.

EC2

EC2 is an array of real, physical servers offered to you by Amazon in the form of virtual machines (VM). The management of these virtual machines is performed by the users. The responsibility of Amazon is to keep the VM running, and the responsibility of the user of EC2 is to use the VM. Management is done using a set of web services that includes not only EC2 but also infrastructure services such as the Simple Storage Service (S3) and the Elastic Block Service (EBS), and middleware such as SimpleDB and the Relational Database Service (RDS). You can find information about these web services at `http://aws.amazon.com/`.

Getting Started with EC2

To get started with EC2, you first need to set up an account with Amazon AWS. Go to `http://aws.amazon.com/` to register and be sure to sign up for the EC2 service.

Next you need to set up the EC2 command-line tools. The examples that follow use a Linux operating system. Note that the commands described here are all written in Java and work on Linux, UNIX, and Mac OS X as well as Windows operating systems. For more information on how to install the command-line tools, refer to the following site:

```
http://docs.amazonwebservices.com/AmazonEC2/gsg/2006-06-26/setting-up-
your-tools.html
```

Costs

Because real hardware is being used to run the virtual machines, you must pay to use it. (For pricing information, see `http://aws.amazon.com/ec2/#pricing`.) At the time of this writing, a small server instance costs $0.085 per hour while on the other end of the spectrum, a *Quadruple Extra Large High Memory* instance costs $2.40 per hour.

WARNING Note that while these prices are very low and quite suitable to use for demo purposes as part of this book, the monthly and yearly costs are not negligible. Thus, it's important to shut down or terminate your virtual machine instances when they have served their purpose. Also note that usage is measured per hour with a minimum of one hour. That means that even if you only started an instance for 5 minutes, you will still be charged for a full hour.

Not only do computing resources cost money, but if you store bundles or large amounts of data on Amazon's S3 you will have to pay between $0.055 and $0.15 per GB depending on the storage usage (see `http://aws.amazon.com/s3/#pricing`). An Amazon EBS volume costs $0.10 per GB per month of provisioned storage.

Finally, data transferred from an EC2 instance to the Internet (outbound volume) costs between $0.15 per GB and $0.08 per GB depending on the total traffic volume. Data transferred to an EC2 instance is charged at $0.10 per GB.

Customizing an AMI

AMI is short for *Amazon Machine Image*. This image contains all the software that a virtual machine needs to run. It can contain a wide range of operating systems in both 32- and 64-bit variations. So the first task for this example is to select an operating system for the virtual machine. This example uses Ubuntu Server Edition—specifically, the official Ubuntu AMI for version 9.10 (Karmic Koala). Ubuntu has an excellent tutorial on how to get started with EC2 at `https://help.ubuntu.com/community/EC2StartersGuide`.

In this example, you are running a small instance (32-bit) in the Amazon data center in northern Virginia. The AMI number to use for this is `ami-bb709dd2`, but a newer version of an Ubuntu AMI should work too. You can find a list of all AMIs that are directly available at the website `http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171`. If you browse by Operating System, you can find other operating systems.

Before you start this image, make sure that you can access the machine remotely once it is started. To achieve this, you will need to generate a new key pair. This key pair will give you secure shell (ssh) access to the instance once it is launched. Run the following commands to generate a new key pair:

```
ec2-add-keypair pentaho-keypair > pentaho-keypair.pem
chmod 600 pentaho-keypair.pem
```

Now start the Ubuntu server instance:

```
ec2-run-instances ami-bb709dd2 -k pentaho-keypair
```

To monitor the state of the machine while it's booting, execute the following command:

```
ec2-describe-instances
```

After a few minutes, you will see a line similar to the following:

```
INSTANCE        i-e7855a8c      ami-bb709dd2
ec2-72-44-56-194.compute-1.amazonaws.com   ip-10-245-30-146.ec2.internal
running pentaho-keypair 0          m1.small   2010-03-08T12:21:25+0000
us-east-1d      aki-5f15f636    ari-d5709dbc
```

The `running` state indicates that the machine is up and running and ready to accept input. Now you can use secure shell to log in to the instance. You'll need to use the generated key pair to authenticate:

```
ssh -i pentaho-keypair.pem ubuntu@ec2-72-44-56-194.compute-1.amazonaws.com
```

Now that you are on the remote machine, you can customize it to your liking. First, add a few software repositories:

```
sudo vi /etc/apt/sources.list
```

Add the following lines to the file:

```
deb http://us.archive.ubuntu.com/ubuntu/ karmic multiverse
deb-src http://us.archive.ubuntu.com/ubuntu/ karmic multiverse
deb http://us.archive.ubuntu.com/ubuntu/ karmic-updates multiverse
deb-src http://us.archive.ubuntu.com/ubuntu/ karmic-updates multiverse
```

Now perform the following command to update the software inventory:

```
sudo apt-get update
```

You need to install the following additional software on this host:

- **Java:** Use the Sun JRE version 6 to run Carte.
- **Unzip:** To unpack a PDI software bundle.
- **ec2-ami-tools:** For AMI bundling purposes later on.

The following command will install these packages:

```
sudo apt-get install sun-java6-jre unzip ec2-ami-tools
```

Once that is done, you can download a recent version of Pentaho Data Integration 4.0 or later. For the community edition, you can find the location of the download archive at the Pentaho project page on `sourceforge.net`. If you're a Pentaho Data Integration Enterprise Edition customer, you can obtain the location via the customer support portal.

```
wget http://sourceforge.net/projects/pentaho/files/Data
 Integration/4.0.0-stable/pdi-ce-4.0.0-stable.zip/download
```

It is also possible to secure copy the file onto the box using `scp`:

```
scp -i pentaho-keypair.pem pdi-ce-version.zip ubuntu@server
...amazonaws.com:/home/ubuntu
```

Then unpack the Kettle software as usual:

```
mkdir pdi
cd pdi
unzip ../pdi-ce-4.x.x.zip
```

This will give you access to the complete Kettle software package on the EC2 instance.

NOTE **If you are using Kettle plugins, make sure to add them in the appropriate subdirectory of the** `plugins` **directory beneath the Kettle home directory. Adding them here for the master will make sure that they are picked up on all slaves later on.**

Next, make sure that the instance is used as a Carte instance. Because you want to pass in a different Carte configuration file for the master and the slaves, you will pass in an XML data file with the `-f` option of the `ec2-run-instances` command. The maximum size of this data file is 16KB, but that's plenty for this example.

On the instance itself, you can retrieve the data file using a web service call:

```
wget http://169.254.169.254/1.0/user-data -O /tmp/carte.xml
```

This command needs to be run as part of the boot sequence of the instance. That means that you need to create a boot file called `/etc/init.d/carte` containing the following lines:

```
#!/bin/sh

su ubuntu -c "/home/ubuntu/runCarte.sh"
```

This executes a shell script to run Carte at boot time. It doesn't execute this script as `root` but as the stock `ubuntu` user. The script that gets executed, `/home/ubuntu/runCarte.sh`, contains the following content:

```
#!/bin/sh

LOGFILE=/tmp/carte.log
exec 2> $LOGFILE

cd /home/ubuntu/pdi

# Get the user data : a carte configuration file
#
wget http://169.254.169.254/1.0/user-data -O /tmp/carte.xml >> $LOGFILE

# Now start carte...
#
sh carte.sh /tmp/carte.xml >> $LOGFILE
```

This script will log to file `/tmp/carte.log`. In case anything goes wrong, you can always ssh into the instance and look at the log to see what happened. If required, you can change the script to your liking.

Finally, make sure that the system launches the Carte service at startup:

```
sudo chmod +x /etc/init.d/carte
sudo chmod +x /home/ubuntu/runCarte.sh
sudo update-rc.d carte defaults
```

The first two commands make the script executable while the third adds the appropriate configuration to the Ubuntu server to start the Carte service at boot time.

Packaging a New AMI

Now that the stock Ubuntu Server AMI is modified to your liking, it's time to create an image of your own for further re-use. This process is called *bundling* and is done with the `ec2-bundle-vol` command. For detailed information on the bundling process, please consult the Amazon EC2 documentation at `http://docs.amazonwebservices.com/AmazonEC2/gsg/2006-06-26/`.

First you need to secure copy your private key and certificate over to the EC2 instance. You do this using the secure copy or `scp` command:

```
scp -I pentaho-keypair.pem ~/.ec2/*.pem
ubuntu@ubuntu@ec2-72-44-56-194.compute-1.amazonaws.com:/tmp/
```

You can then bundle up your own image:

```
sudo bash
ec2-bundle-vol -d /mnt/ -k /tmp/ -k /tmp/pk-your-pkfilename.pem
-u your-accountnr -s 1536 --cert /tmp/cert-your-cert-file.pem
```

Then upload this AMI to Amazon S3 for further re-use:

```
ec2-upload-bundle -b kettle-book -m /mnt/image.manifest.xml
-a your-access-key -s your-secret-key
```

Finally, register it on our local machine (not the EC2 host):

```
ec2-register kettle-book/image.manifest.xml
IMAGE   ami-bbfb14d2
```

The AMI number you get is the one you can use to launch your Carte servers.

Terminating an AMI

You no longer need the instance you customized so you can terminate it now. To terminate an instance, simply run the following command (replace *`your-instance-nr`* with the name and number of your instance):

```
ec2-terminate-instances i-your-instance-nr
```

The instance number can be obtained with the `ec2-describe-instances` command.

Running a Master

To run a master, you first create an XML file to use as a parameter for your master instance. This is the same slave configuration file as described previously. Store it in a file called `carte-master.xml`:

```
<slave_config>
```

```
    <max_log_lines>10000</max_log_lines>
    <max_log_timeout_minutes>600</max_log_timeout_minutes>
    <object_timeout_minutes>60</object_timeout_minutes>

    <slaveserver>
      <name>Master</name>
      <network_interface>eth0</network_interface>
      <port>8080</port>
      <username>cluster</username>
      <password>cluster</password>
      <master>Y</master>
   </slaveserver>

  </slave_config>
```

As you can see, you don't specify a hostname for the Carte web server to listen to. Instead, it looks at the IP address of the specified network interface and uses that. On an Ubuntu server on EC2, `eth0` happens to be the internal network interface.

Now you can run a single instance of your new AMI and pass this as a parameter:

```
ec2-run-instance ami-bbfb14d2 -f carte-master.xml -k pentaho-keypair
```

Again, you can monitor the boot process by running `ec2-describe-instances`. After a few minutes you will notice that the instance has started.

To test if all went well, open a web browser to the EC2 instance on port 8080:

```
http://ec2-ip-address-etc.amazon.com:8080/
```

Please note that you need to authorize (analogous to opening a port in a firewall) this port to protect it from being accessed from the outside world with the `ec2-authorize` command:

```
ec2-authorize default -p 8080
```

Running the Slaves

Now that you have the master running, you can simply start up any number of slaves to report to it. The slave configuration file `carte-slave.xml` looks like this:

```
<slave_config>

  <max_log_lines>10000</max_log_lines>
  <max_log_timeout_minutes>600</max_log_timeout_minutes>
  <object_timeout_minutes>60</object_timeout_minutes>

  <masters>
    <slaveserver>
      <name>master1</name>
```

```
        <hostname>internal-ip-address-of-the-master</hostname>
        <port>8080</port>
        <username>cluster</username>
        <password>cluster</password>
        <master>Y</master>
      </slaveserver>
    </masters>

    <slaveserver>
      <name>Master</name>
      <network_interface>eth0</network_interface>
      <port>8080</port>
      <username>cluster</username>
      <password>cluster</password>
      <master>Y</master>
   </slaveserver>

  </slave_config>
```

The internal IP address of the master is the only thing you'll need to configure. You can obtain it by looking at the result of the `ec2-describe-instances` command. The address starting with `ip-` and ending with `.ec2.internal` is the one you want. The IP address is the part in between. So if you have an internal server name of `ip-10-245-203-207.ec2.internal`, the internal IP address is 10.245.203.207. This is what we put in the `carte-slave.xml` file.

Now you're ready to start a number of slaves. In this example you'll start three:

```
ec2-run-instance ami-bbfb14d2 -f carte-slave.xml -k pentaho-keypair -n 3
```

The option `-n 3` specifies that you want to start three instances of your AMI. After a few minutes, you can browse to your master again using the getSlaves service this time:

```
http://ec2...amazon.com:8080/kettle/getSlaves
```

You'll notice the slaves appear in the list one by one. Note that not all EC2 nodes start up at equal speed.

Using the EC2 Cluster

If you look in the slave server detection list using the `/kettle/getSlaves` service, you'll notice that all the slaves registered using their internal IP address. It's best to use the internal IP address of EC2 because traffic on the internal backbone network is not charged for and is in general a lot faster. This can make a huge difference, not only in performance but also in cost savings. However, this does present a unique problem. If the client starting the clustered transformation can't reach the slave servers, how can you post, run, and monitor anything on it? To tackle this problem and others like it, Kettle contains what is called a *resource exporter*. This feature is capable of exporting all the resources that a certain job or transformation uses to a `.zip` file. As such, to

run the clustered transformation on EC2 you need to execute the transformation from within a job. You can then use the resource exporter to pass a `.zip` file containing the job as well as the clustered transformation over to the master server—the master will distribute the relevant pieces to the slaves since all are reachable from the master. When the clustered transformation is executed on the master, the slaves are visible and the execution will proceed as planned.

Figure 17-3 illustrates how you can enable the resource exporter from within the "Execute a job" dialog in Spoon.

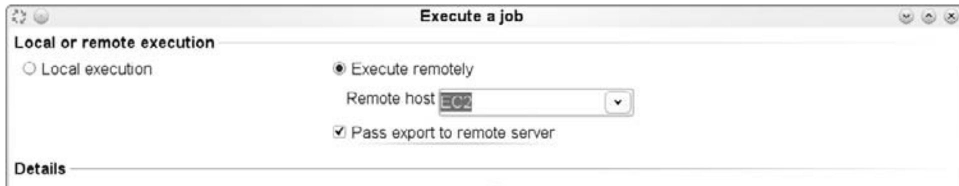


Figure 17-3: Passing a job export to a slave

You can enable the same option in the Job job entry, as shown in Figure 17-4.

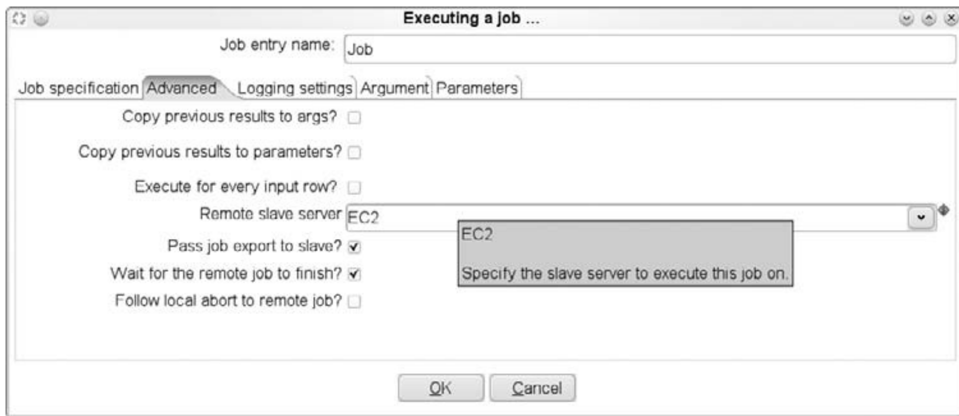


Figure 17-4: Pass the export of a job to a slave.

Monitoring

Even though the slaves report to the master over the internal EC2 backbone network, you can still reach them on the Internet. Simply browse to their public IP address on the specified port (8080 for example) to see how they are doing.

The transformations that are executed as part of a dynamic cluster are renamed with the term *Dynamic Slave* in the new name as well as the slave server URL to which they belong. If you set up the transformation to perform logging, keep this in mind as you will be able to extract valuable information from the logging tables.

If you enter the details of one of the slaves in a slave server definition in Spoon, you should have a representative view of the cluster as a whole. In that case, make sure to try out remote row sniffing. This is shown in Figure 17-5. It's a good way to see what's going on in the various steps on the other server.

For additional information on row sniffing and real-time monitoring, see Chapter 18.

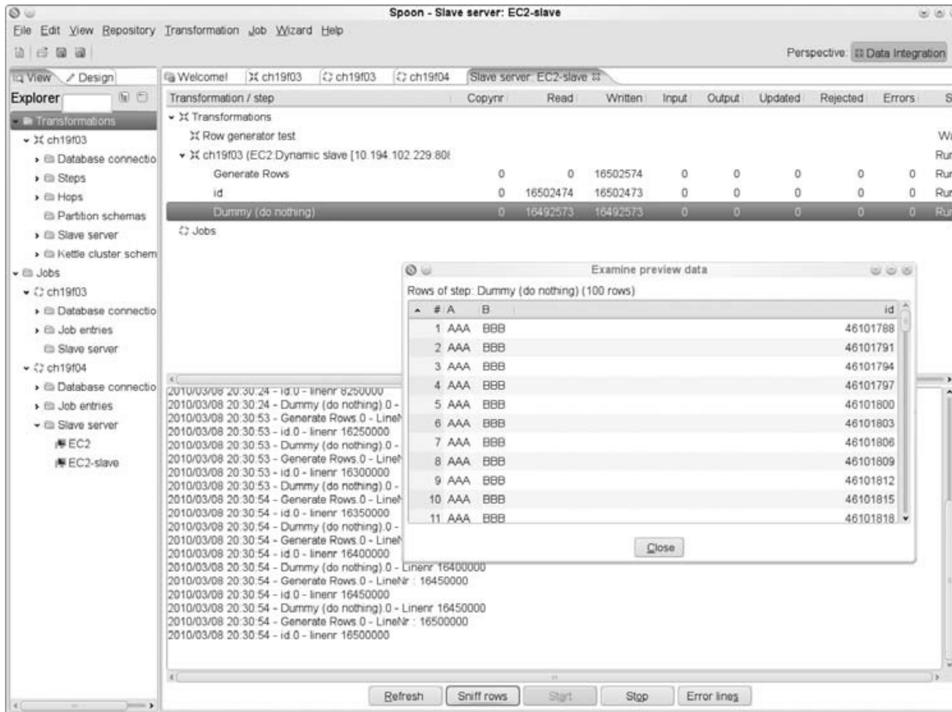


Figure 17-5: Remote step row sniffing

The Lightweight Principle and Persistence Options

In the previous chapter and in this one, we covered a lot of ground with respect to Kettle clusters. However, we didn't cover the persistence options for a slave server. The reason for that is simple: There aren't any! A slave server is designed to be a lightweight piece of software that is totally controlled and monitored from the outside world. It doesn't store transformations or jobs locally. If you want to reference transformations from jobs, or sub-transformations from transformations in a clustered run, you need to make sure that the required objects are available on the master and slaves.

When a slave server is terminated, all the execution results are gone. This sort of behavior was specifically designed for use in grid and cloud environments where you typically start the Carte slave server or a complete node when data integration work needs to be done. When the work is completed, the slave server (or the complete server

in the case of *IaaS*) is shut down. This is typical because every hour of usage is paid for and because the typical grid- or cloud-based workload is occasional in nature.

Like Carte, an EC2 server doesn't persist anything. Governed by the same light-weight principle as a Carte instance, an EC2 instance only keeps information on local disks until it is shut down or terminated. The only thing that is persisted is the data that is captured in the AMI. Fortunately, Amazon has made available the *Elastic Block Service (EBS).* You can create a file system on the EBS that can be shared by the master and slaves alike. Information on the EBS is indeed persisted separately from the EC2 instances and will be kept around until the EBS volume is specifically destroyed. If you want to share data to all slave servers or read large files in parallel, this is currently the best option to use.

There is no doubt in the case of both a Carte server and an EC2 instance that not having a persistence option can pose a few problems here and there. However, the up side is that if you configure things correctly, you can always tear down clusters and complete sets of servers without consequences. At the first sign of trouble, you can simply decide to pull the plug and restart.

Summary

In this chapter, we dug a little bit deeper into the realm of clustering to show you how Kettle clusters can be configured automatically and dynamically. In this chapter you learned:

- How cloud computing is bringing vast computing resources to your doorstep at very low prices
- How to customize your own Kettle slave server AMI on Amazon EC2
- How to run your clustered transformations on EC2

CHAPTER

18

Real-Time Data Integration

In this chapter, we offer a closer look at how real-time data integration can be performed with Kettle. You'll start by exploring the main challenges and requirements to figure out when it will be useful for you to deploy this type of data integration.

After that, we explain why the transformation engine is a good match for streaming real-time BI solutions, and discuss the main pitfalls and considerations you need to keep in mind.

As an example, we include the full code for a (near) real-time Twitter client that continuously updates a database table for further consumption by a dashboard. Finally, we cover third-party software such as database log readers and we provide guidelines on how to implement your own Java Message Service (JMS) solutions in Kettle.

Introduction to Real-Time ETL

In a typical data integration setting, jobs and transformations are run at specific times. For example, it's quite typical to have nightly, weekly, or monthly batch runs in place. The term *batch run* comes from the fact that a whole batch of data or group of work is executed in sequence in one go. It is also referred to as *batch processing*. Usually the batch is scheduled to run at a time when computing resources are readily available. For example, most data warehouses are updated with batch processing during the night when there are few users on the operational systems.

Usually it's sufficient to have nightly jobs in place to satisfy your requirements. In fact, the vast majority of all Kettle jobs and transformations are nightly batch jobs.

However, there are exceptions for those types of jobs that need to get source data in the hands of users quicker. When you make the interval between batches smaller, usually between a minute and an hour, the jobs are referred to as *micro-batches* or *small periodic batches*. If you make the interval between batches even smaller, you can speak of *near real-time* data integration.

Finally, when you need to see changes in your source system reflected in your business intelligence solution as fast as possible or near instantaneously you need to use *real-time*, *continuous*, or *streaming* data integration. The total delay for information to travel from source to target is then typically measured in seconds and sometimes milliseconds. Examples of real-time data integration include alerting systems that report on the state of industrial processes. When the pressure in a tank reaches critical values, the operators need to know this as fast as possible and preferably not a few minutes later. In such cases, the read-out values from a pressure valve need to be sent to the person who is monitoring the values using a real-time data integration process. The pressure values can be represented with a simple dial readout or progress chart called a *real-time business intelligence solution*.

Real-Time Challenges

Using real-time data poses a variety of challenges. Most commonly, the challenges of real-time processing involve working with the data itself; typical examples include comparing data, performing lookups, calculating averages and sums, and performing fraud detection and all sorts of pattern matching. Executed in batches, these operations can be time consuming. In real-time scenarios, you face an extra challenge to make them perform optimally.

In addition to these actual data challenges, whenever you bridge any data gap between a source and a target system, you have to deal with both the sourcing and the delivery of the data. As such, challenges for real-time, near real-time, or continuous data integration can arise in either the sourcing or the delivery.

For the sourcing of the data, it's obviously very important to get your hands on data as soon as possible after data has changed or an event of significance has occurred. You need to know immediately if new data arrives, or if data has been updated or deleted so you can propagate these changes to the target system. Chapter 6 covers the various options for retrieving these changes. The following list shows a few of the typical options that are available for change data capturing (CDC):

- Read timestamps to determine changed records in a source system.
- Compare snapshots of the data.
- Deploy database triggers to figure out changes in a source database.
- Tap directly into the transactional logging information of a relational database to figure out what changes are occurring.

Obviously, you want to pick a method of extraction that is both non-intrusive and low-latency. Usually this means that you want to deal with the internals of a relational database and use one of the last two options in our list. The first two are more

passive methods, and tend to create a heavier load on the source systems in addition to increasing the latency for detecting changes.

As you can imagine, the changes you receive from either a set of database triggers or the transaction log of a relational database are very low level. For example, you might get informed of an update to a certain column in a certain table using for a subset of rows. At that moment, you need to convert that change to a change in the target system. This requires rather complex operations at times because you have only the change of a record, not the complete record.

In the end, what it comes down to is this: The lower the level you get with a database, the higher the complexity. At times, you might even have to re-create some of the application logic to determine the meaning of a certain change and how to handle it in the system. It is also important to remember that there are no standards whatsoever as far as handling the output of the change log of a relational database is concerned. Most relational databases have their own system for this sort of *log data change processing* available in place as part of their product. The methods and output are usually highly proprietary and closed in nature. As such, another main challenge is the high cost of log-reading software.

Combine the expensive software needed to read a transaction log with the high level of customization you need to go through to translate and process the changes, and you end up with a lengthy and costly process just to read the data from the source systems you are interested in.

For the output side of the equation, you need systems that are capable of handling the never-ending flow of changes data. For example, OLAP becomes much more complex if you can't trust the caches you built up to speed up querying. Incremental aggregation can become very complex as well. Real-time charting and trending can also pose problems because you need to feed the changes from the source system directly into the chart.

Requirements

As you can infer from the challenges, the implementation of real-time data integration systems can be prohibitively time consuming and expensive, so it's important to take a look at what the requirements are before the start of the actual implementation.

A very important requirement that drives many implementations of real-time business intelligence systems is the need to get information changes in the hands of end users as quickly as possible so that they can take appropriate action. To continue with the tank pressure example, when the end user is an operator in a chemical plant, a computer might report on the pressure in a certain tank. We want to display the current pressure in the tank and not the pressure of the night before. We also want to have the operator perform an action in case the pressure exceeds a certain threshold.

If we take this example apart we end up with three main components in the real-time system:

- A change in a source system is being captured with a certain delay, represented as S (the refresh rate of the pressure sensor).

- The changes are presented to the end user after another delay, represented as P (refresh rate of the meter).
- An action (release of pressure in the tank) is taken after a certain delay, represented as A, if required. A is as such the time spent between reading the value of the meter and the action taken.

From this simple requirements analysis you can see that it would make little sense to implement a real-time data integration system where changes are captured (S) and presented (P) in real-time, but where action is only taken after a long delay (A). For a successful outcome, a real-time system must be equipped to act quickly based on the real-time data. The most extreme case would be a real-time alerting system where there is no operator (human or machine) in place to take appropriate action. The whole expenditure of time and effort to put the real-time data integration system in place would be a waste; the tank would have exploded!

Because of the high costs associated with real-time data integration and business intelligence, many organizations are settling on the notion of implementing *right-time* business intelligence solutions. In those setups, information is delivered when it is needed, not when it becomes available. The right-time strategy forces an organization to focus on the delivery time requirements of information and away from a costly real-time setup.

Transformation Streaming

As explained in Chapter 2, a Kettle transformation involves streaming data from step to step. This is implemented with the help of buffers (hops) between the steps that have a maximum capacity, which in turn forces rows of data through the steps in a streaming fashion. Not only does it stream data through the steps, but it allows the steps to run in parallel to improve performance on machines with multiple CPU cores.

In a batch transformation, a fixed number of rows are read, say from a database table. The rows are read one at a time and handed over to the next step. That step hands it over to the next step until all rows are processed. When all steps finish processing, the transformation is considered finished as well. While that is the typical case, there is nothing in the architecture of a transformation that prevents a transformation from running non-stop for weeks, months, or even years on end. For example, consider a source system that continuously generates data but only has small buffer. In that situation you either read out the data or lose it forever. Because of this drawback the transformation that reads out the data from the source system optimally always keeps running indefinitely.

A transformation has three phases:

- The *initialization* when memory is allocated, files opened, database connections made, and so on
- The *execution* of all the steps processing rows
- The *cleanup* when memory is cleaned up, files closed, connections severed, and so on

This three-phased approach is typical for most, if not all, ETL tools on the market. For long-running real-time transformations, you need to consider issues such as time-outs, memory consumptions and logging:

- **Time-outs:** Many databases have connection time-outs in place. As such, if you have a transformation running for weeks on end with calm periods over the weekend where no rows of data are being updated, you risk time-out situations. Make sure to plan for this situation up-front by modifying the appropriate settings in both the client database connection in Kettle and the server. See Chapter 2 for more information on how to set database options.
- **Memory Consumption:** There are only a few situations in which you can consume memory indefinitely during the execution of a transformation. One such situation is data caching, which helps speed up steps such as "Database lookup," "Dimension lookup / update," and others by preventing roundtrips to the database table. In situations where you keep receiving new rows indefinitely, make sure to at least set a maximum cache size or confirm that the cache memory usage is going to remain flat. Pay attention also to data structures you allocate in any scripts that you write. Make sure you don't accidentally use a construct that will continue to consume memory. This is usually a Java map in a User Defined Java Class step or an associative array in JavaScript.

  Another situation in which extra memory is consumed is when it is done explicitly. The "Sort rows" step, for example, needs to consider all input rows before the data can be sorted. In a real-time data integration situation you will always continue to receive more rows. Because of this you cannot use the sort step, as you would run out of memory eventually. If you try to use a continuous stream of rows as lookup for a "Stream lookup" step, you would also run out of memory eventually, because that step too will continue to read rows until it has a complete set of data to look up with. Because of the all-consuming nature of these steps, they can only be used in real-time data integration if they do not feed on a never ending stream of data.
- **Logging:** In previous versions of Kettle, it was standard practice to write a log record into a database table when a transformation (or job) started. Kettle then performed an update of the row when the task was finished. When a transformation is never-ending, this is not going to be very useful because you will never be able to see the actual logging data appear in the log table. To address this, the *interval logging* feature, shown in Figure 18-1, was added to Kettle in version 4.0. Interval logging will periodically update the aforementioned log record, allowing you to keep track of what a long-running transformation is doing by looking at the log table, by using the history view in Spoon, or by using Pentaho Enterprise Console.

  Another change in Kettle 4.0 is a limitation on the number of log lines that are being written. This helps to reduce the memory consumption and improves the readability of the log field in the log table. However, that alone is not enough. Beginning with version 4, Kettle uses a central log buffer for all transformations and jobs that run on the same Java Virtual Machine. This could be a Carte, Spoon,

Pan, or Kitchen instance, but it might be the Pentaho BI server as well. In all those cases, you can set the following environment variable (for example, in the `kettle.properties` file in your `KETTLE_HOME` directory):

- `KETTLE_MAX_LOG_SIZE_IN_LINES`: The maximum number of lines that the logging system back-end will keep in memory at any given time.
- `KETTLE_MAX_LOG_TIMEOUT_IN_MINUTES`: The maximum age of any log line (in minutes) before it is discarded.

Setting both options will ensure that you won't run out of memory because of excessive logging, even if this is done by another transformation or job that runs on the same server. Because of the memory consumption of the logging back-end, it is essential that you set these parameters for long-running transformations (or jobs).

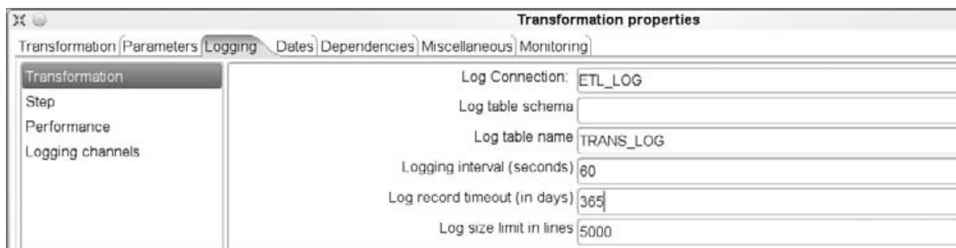


Figure 18-1: Interval logging option

A Practical Example of Transformation Streaming

In this trendy example, we want to list all the recent messages from Twitter on the front page of a business intelligence dashboard. Doing so will alert the users to chatter that concerns the company in question. For simplicity, we want the dashboard to retrieve this data from a local database table, so the data in that local database table always needs to be up-to-date.

The first thing we need is a new step that allows us to search messages on Twitter. Fortunately, a number of Twitter Java libraries can be found on the Internet. One of the libraries called *jtwitter* (`http://www.winterwell.com/software/jtwitter.php`) is very small (160kb) and carries an LGPL license so it can be shipped with Kettle to accommodate this example.

This example can be found in the `samples/transformations` folder of your Pentaho Data Integration distribution and is called `User Defined Java Class - Real-time search on Twitter.ktr`.

Figure 18-2 shows what the sample transformation looks like after we added a step to update the messages in a database table.

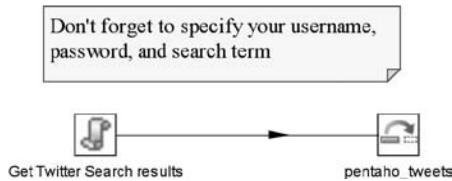


Figure 18-2: Continuously storing tweets in a database

The following example shows how the code is conceived in the User Defined Java Class step called "Get Twitter Search results."

```
import winterwell.jtwitter.*;

private Twitter twitter;
private long lastId;

public boolean processRow(StepMetaInterface smi, StepDataInterface sdi)
  throws KettleException
{
    java.util.List timeLine = twitter.search(getParameter("SEARCH"));
  logBasic("Received "+timeLine.size()+" tweets!");

  // First see if we need to look at any of the status reports
  // Determine the maximum ID and compare it to last times' status ID
  //
  long maxId=-1L;
  for (int i=0;i<timeLine.size();i++) {
    Twitter.Status status = (Twitter.Status)timeLine.get(i);
    if (maxId<status.getId()) {
      maxId=status.getId();
    }
  }

  // Do we have anything to do with this batch of statuses?
  //
  if (maxId>lastId) {
    // Process all the status reports...
    //
    for (int i=0;i<timeLine.size();i++) {
      Twitter.Status status = (Twitter.Status)timeLine.get(i);

      // If the id is recent, process it
      //
      if (status.getId()>lastId) {
        // New things to report...
        //
```

```
            Object[] rowData = RowDataUtil.allocateRowData
              (data.outputRowMeta.size());
            int index = 0;

            rowData[index++] = status.createdAt;
            rowData[index++] = Long.valueOf(status.getId());
            rowData[index++] = status.inReplyToStatusId;
            rowData[index++] = status.getText();
            rowData[index++] = status.getUser().toString();
            rowData[index++] = Boolean.valueOf(status.isFavorite());

            putRow(data.outputRowMeta, rowData);
          }
        }
      } else {
        // Wait for 30 seconds before retrying
        //
        int delay = Integer.parseInt(getParameter("DELAY"));
        for (int s=0;s<1000 && !isStopped();s++) {
          try {
                Thread.sleep(delay);
          } catch(Exception e) {
              // Ignore
          }
        }
      }

      lastId=maxId;

      // Never end, keep running indefinitely, always return true
      //
      return true;
    }

    public boolean init(StepMetaInterface stepMetaInterface,
      StepDataInterface stepDataInterface) {
       if (super.init(stepMetaInterface, stepDataInterface)) {
         twitter = new Twitter(getParameter("USER"),
            getParameter("PASSWD"));
         lastId=-1;
         return true;
       }
       return false;
    }
```

As you can read in the code, this step will only finish when it is forced to do so by a user. This is because the `processRows()` method always returns `true`, which simply means "continue to run this step and continue to call this method." Note that the Twitter

client we use here is simplistic and doesn't do a lot of optimization, for example, by keeping HTTP connections open for performance reasons. However, for this sample and for most use cases it will do just fine. A delay after each search can be specified in the step by using a parameter, as shown in Figure 18-3.

Fields | Parameters | Info steps | Target steps

Parameters:

#	Tag	Value	Description
1	USER	username	The twitter username
2	PASSWD	password	The twitter password
3	SEARCH	pentaho	The search term
4	DELAY	60	Delay between searches in seconds

Figure 18-3: Delay parameter

Earlier in this chapter, we defined this delay to be the sourcing delay. If we use a very low number, we will continuously query the Twitter Web service for updates to our query. If a new message is found, the database table will be updated shortly after that message is received. If, in our example, the dashboard is automatically refreshed every 60 seconds the presentation delay is 60 seconds. If you add up the sourcing delay and the presentation delay, you have the total delay after which a user can take action.

NOTE **We can't recommend continuously querying the Web service because it would put an unnecessarily high load on the free Twitter services. Make sure to use a delay value of at least sixty seconds while testing the Twitter example shown.**

Debugging

While a real-time transformation is running, you need new ways of examining the data. A preview operation would work, but you would only be able to see the first rows. If you want to see after a few days what rows are passing through a transformation you need different tools. To accommodate this, Kettle version 4 sports two ways of viewing rows as they pass through steps, called row sniffing. The first method is available in Spoon when you right-click a running step. From the context menu you can select the "Sniff test during execution" option. You can then further select to view the input or output of the step. Once these choices have been made, you are presented with a Preview Rows dialog where rows will be added while they pass in the step.

This way of viewing rows is also available when you execute a transformation on a remote slave (Carte) server. From the Slave Browser tab in Spoon, you can access the Sniff Test button, as shown in Figure 18-4 to see the rows that pass in a selected step:

Figure 18-4: Sniff rows on a slave server

Third-Party Software and Real-Time Integration

As mentioned earlier, if you want to get your hands on certain real-time information streams, you need specialized third-party software. For example, if you want to be informed of changes in an Oracle database, you could purchase software such as LogMiner or Oracle Streams. In fact, a lot of relational databases have all sorts of transaction log–reading software tailored for various real-time purposes. While a full list is beyond the scope of this chapter, we can group these tools into two main categories:

- **Software that allows you to pick up the changes:** This software exposes the changes over standard or non-standard interfaces like SQL. One example is SQLStream (`http://www.sqlstream.com`), co-founded by Julian Hyde, lead architect of Pentaho Analysis (Mondrian). In this case, you can connect to SQLStream via JDBC and execute a SQL statement in a Table Input step against one of the real-time data streams. This SQL query will never end and will continue to run until either the transformation or the SQLStream server is stopped. Conversely, you can use a Table Output step to write data to the SQLStream server and the data will be picked up by the server as a continuous stream of data.
- **Software that will pass the data over to the third-party tool (Kettle in our case):** In this case, the log reading tool will repeatedly call the transformation every time there is new data. Based on the actual payload you will need to take appropriate action in the transformation. The following section, "Java Message Service," presents an example of this kind of software.

Because these tools are usually tailored for a specific database, you will need a separate transaction log reading tool for every database vendor you encounter in your enterprise.

Java Message Service

One popular example of real-time technology that is often used in data integration and application integration is Java Message Service, or JMS. JMS is a messaging standard that allows Java application components to pass messages around. These messages are passed in a distributed and asynchronous fashion. JMS is usually deployed on a Java 2 Enterprise Edition (J2EE) server to facilitate its continuous operation as a service.

JMS is a complete API that allows a wide range of possibilities for sending and receiving messages and supports various operational models. The *publish and subscribe* model is probably the most interesting because it allows multiple consumers to be registered for the same topic. The model also supports *durable subscriptions* that are capable of retaining unread messages even when a subscriber is momentarily not connected.

The publish and subscribe model includes two main scenarios: producing messages and consuming them. Producing messages means that Kettle will be sending messages to another service, while consuming messages means that Kettle will be receiving messages from another service.

In the following JMS Java code examples we first explain how you can define a JMS connection and create sessions, and then show how to consume and produce messages.

For those less familiar with Java, note that the Pentaho Data Integration Enterprise Edition (PDI EE) version 4.0 has both a consumer (JMS Input) and a producer (JMS Output) step available and does not require Java expertise.

Creating a JMS Connection and Session

The first thing you need to do is to create the connection, a session object, and a messages queue. This is usually dependent on the JMS implementation you are using. For example, the Apache ActiveMQ library allows you to do this:

```
ConnectionFactory connFactory = new ActiveMQConnectionFactory(url);
Connection connection = connFactory.createConnection(username, password);
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createQueue(queueName);
```

For other libraries, it's usually only the `ConnectionFactory` that changes. For example, with OpenMQ you will use the following very similar code; the differences are indicated in bold.

```
ConnectionFactory connFactory = new com.sun.messaging.
    QueueConnectionFactory(url);
Connection connection = connFactory.createConnection(username, password);
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createQueue(queueName);
```

With appropriate exception handling, you can place this code straight into the `init()` method of the User Defined Java class example shown earlier.

Consuming Messages

To consume a message, you need to create a message consumer object like this:

```
MessageConsumer consumer = session.createConsumer(queueName);
```

Again you can add this code to the `init()` method of your plugin or User Defined Java Class step. Once that is done, you want to start reading messages from the `queueName` queue. This is done using the following line of code:

```
Message message = consumer.receive();
```

You can put this method at the start of the `processRows()` method in the earlier sample. This gives you a message object to work with. If the method returns `null`, you know that the queue is closed and you have no choice but to abort; return `false` in that case. In all other cases, as in the Twitter example, return `true`. Once you have the message, you can treat it as simple text or as XML (two popular formats). You can pass the message along to the next steps or parse it on the spot in your code. That choice is entirely up to you. Suffice it to say that at this point the hard part is over.

Producing Messages

To produce a message, you need to modify the consumer code only slightly:

```
MessageProducer producer = session.createProducer(destination);
```

Then you can create a text or XML message by creating a string `messageContent` that contains the correct text :

```
TextMessage message = session.createTextMessage();
message.setText(messageContent);
```

Note that you can also set a series of key-value pairs on the message this way:

```
message.setStringProperty(keyString, valueString);
```

To send the message to the consumers, you can use the following line of code:

```
producer.send(message);
```

Closing Shop

If the transformation or the message queue is stopped for some reason, you need to properly close and stop all the message consumers and producers that you used in your plugin as well as the session and connection. This is done in the `dispose` block of your plugin:

```
public void dispose(StepMetaInterface smi, StepDataInterface sdi) {
```

```
    try {
      messageProducer.close();
      session.close();
      connection.stop();
      connection.close();
    } catch (JMSException e) {
      logError("Unable to close JMS objects", e);
    }

    super.dispose(smi, sdi);
  }
```

While some Java knowledge is involved in the creation of JMS steps in PDI, it is far from an insurmountable task to create consumer and producer steps for various purposes. Anyone familiar with JMS code should be able to integrate their code into Kettle in either a plugin or with the User Defined Java Class step.

Summary

This chapter offered a look at real-time data integration and explained the challenges and requirements for real-time ETL. The chapter discussed:

- How a transformation is ideally suited for handling streams of data, and the attention points with respect to database time-outs, memory consumption, and logging
- A practical example in the form of a step that continuously reads Twitter messages
- How to debug a long-running transformation
- Integration possibilities with third-party real-time integration software
- How to send and receive messages using Java Message Services

Part

V

Advanced Topics

In This Part

Chapter 19: Data Vault Management
Chapter 20: Handling Complex Data Formats
Chapter 21: Web Services
Chapter 22: Kettle Integration
Chapter 23: Extending Kettle

CHAPTER 19

Data Vault Management

This chapter was written by Kasper de Graaf of DIKW-Academy, a well-known expert in the field of Data Vault modeling.

Data warehousing started somewhere in the 1990s when Bill Inmon and Ralph Kimball started publishing their data warehousing ideas. Both approaches can be used to create an environment that supports analysis and reporting. However, Inmon and Kimball have some differences of opinion, sometimes referred to as "The Big Debate."

Before we dive into the differences, let us start with the similarities. Inmon and Kimball do not disagree about the usage of data marts. A data mart is a database that is aimed at end user usage. It is usually modeled using star schemas (Chapter 4 contains an example of a star schema—The Rental Star Schema) and optimized for analysis and reporting.

The biggest difference between the two architectures is about the need for an enterprise data warehouse (EDW). Inmon says you need one, Kimball says you don't. An EDW is basically a large database that contains integrated, historical data from several other databases. The EDW is not used for querying by end users. It is used solely for complete, transparent, and auditable storage of all data that is considered relevant for reporting. In the vision of Bill Inmon, an EDW sits between the source databases and the data marts and thus acts as the single source for the data marts.

NOTE **For a more elaborate explanation about data warehousing please see Chapter 6 of the book *Pentaho Solutions*, Wiley, 2009.**

The chapter begins with an introduction to Data Vault modeling, followed by an example implementation using the sakila database.

Introduction to Data Vault Modeling

Data Vault (DV) modeling is a methodology specifically designed for the enterprise data warehouse. It was developed by Dan Linstedt (`http://www.danlinstedt.com`) in the late 90s. During the last couple of years, Data Vault modeling has gained a lot of attention and an increasing number of followers in the BI community.

Dan Linstedt defines a Data Vault as follows:

> *The Data Vault is a detail oriented, historical tracking and uniquely linked set of normalized tables that support one or more functional areas of business. It is a hybrid approach encompassing the best of breed between 3rd normal form (3NF) and star schema. The design is flexible, scalable, consistent and adaptable to the needs of the enterprise. It is a data model that is architected specifically to meet the needs of today's enterprise data warehouses.*
>
> `http://www.danlinstedt.com/about`

From this definition, you can conclude that the Data Vault is both a data modeling methodology and an approach or architecture to enterprise data warehousing. Data Vault modeling is based on three basic building blocks: hubs, links, and satellites. The *modeling methodology* defines the components a Data Vault is made of and also defines how these components interact with one another. The Data Vault *approach* consists of good practices one should follow when building an enterprise data warehouse. It states, for example, that business rules should be implemented downstream. This means that the Data Vault stores the data that come from the source systems *as is*, without any interpretation, filtering, cleansing, or conforming. Even when the data from the different sources are contradictory (as with different addresses for the same customer), the Data Vault will not apply a business rule such as "always use the address info from source system A." It will simply store *both versions of the truth*; the interpretation of the data is postponed to a later stage in the architecture (the data marts).

This chapter discusses the Data Vault modeling methodology. The Data Vault architecture is beyond the scope of this book.

Do You Need a Data Vault?

Although we cannot answer the question of whether you need a Data Vault solution, we can try to help you choose. We think the decision comes down to the following question: Do you need (or can you benefit from) an enterprise data warehouse? In the next paragraphs, we try to explain some of the advantages of having an enterprise data warehouse and why adding an additional component and a large amount of code to your environment might be a good idea.

An enterprise data warehouse enables a clear separation of responsibilities: The data warehouse is responsible for storing all the data, including history, as source independent as possible and without making any changes (application of business rules or quality improvement) to the data. This gives you a solid foundation to build your reporting environment. The sole responsibility of the reporting environment is

to present the data to the end user in a form suitable to assist in decision-making. This means that the data is subjected to today's set of business rules, cleansing routines, and interpretations. Thus, the data marts or star schemas in this layer interpret the *facts* that are stored in the warehouse and create today's *truth*.

Because the enterprise data warehouse contains all data without interpretation and with a complete history trace, it renders the reporting environment disposable and therefore allows the truth to be redefined (because of new insights or a changed set of business rules for instance).

Last but not least, we want to mention the traceability of data. Traceability means that every single piece of data can be traced back to the source(s). This implies that every report can be explained to the business, and every interpretation can be explained, argued, and undone if needed.

Data Vault Building Blocks

The following sections describe the role of the major constructs of Data Vault modeling: hubs, links, and satellites, and discusses the interactions among them. It outlines some of the general characteristics and processes of Data Vault modeling.

Hubs

A *hub* is a table that contains business keys for an identifiable entity in your organization. Examples of potential hubs are customer, employee, order, product, building, resource, and vacation.

One highly important aspect of Data Vault modeling is the aforementioned business key. A *business key* uniquely identifies a single instance of an entity to the organization. This means that given a business key for a certain entity, everybody in your organization knows which one is meant. Think of it as the key to your house; it will open the door to *your* house, *only* your house, and *nothing but* your house. An important aspect of a business key is that it has business meaning and is used in everyday operation, for example, a customer number, an order number, or a product code. In other words, business keys are chosen from a business perspective (meaning), not from a technical perspective (like technical IDs and primary keys).

The hub tables (each separate entity gets its own hub table) thus house the unique list of business keys for each entity in an organization. Except for some metadata, the business key is all that a hub contains. One of the essential aspects of Data Vault modeling that is particularly appealing is that hub tables are source independent. When the same business key is used in more than one system, it is recorded only once. All other Data Vault building blocks are connected to this one business key. This automatically implies that the data is integrated across the enterprise.

Each hub contains the fields listed in Table 19-1 (all required, no other attributes allowed).

Table 19-1: Hub Attributes

ATTRIBUTE	DESCRIPTION
Primary key	Surrogate key, system generated, for internal use
Business key	Uniquely identifiable business element, used in the source systems, known to the business
Load DTS	The timestamp the data is first loaded in the EDW, system generated
Record source	Defines the origin of the data (for example, source system or table)

Table 19-2 shows an example hub table, `hub_customer`.

Table 19-2: Example hub_customer

ID	BUSINESSKEY	LOAD_DTS	RECORD_SOURCE
1	cst_24125670	12/17/2009-03:05:04	sales.cust
2	cst_24125894	12/17/2009-03:05:04	sales.cust
3	cst_67904567	12/17/2009-03:00:00	mkt.customer
4	cst_67904568	12/17/2009-03:00:00	mkt.customer

Note that the sample data in this table apparently resided in two different source systems: `sales` and `mkt`. The attribute `Record_Source` defines that the customers `cst_24125670` and `cst_24125894` were first encountered in the source system `sales`, and the customers `cst_67904567` and `cst_67904568` were first encountered in the source system `mkt`. It is possible that these customers reside in the other system as well; since only unique business keys are loaded into the hub we cannot tell this from the example.

Links

A second Data Vault construct is the link structure. A *link* is the intersection of business keys (hubs). This means that a link indicates that two (or more!) hubs have a relationship. A link is based on an identifiable business relationship; this is usually a foreign key, a business event, or a transaction between business keys.

Each link contains the fields listed in Table 19-3 (all required, no other attributes allowed). Table 19-4 shows an example link table.

Table 19-3: Link Attributes

ATTRIBUTE	DESCRIPTION
Primary key	Surrogate key, system generated, for internal use
Hub Surrogate Keys {1..n}	The surrogate keys (foreign keys) of the hubs whose relationship is defined by this link table
Load DTS	The timestamp the data is first loaded in the EDW, system generated
Record source	Defines the origin of the data (for example, source system or table)

Table 19-4: Example Link Table lnk_customer_store

ID	HUB_ CUSTOMER_ID	HUB_ STORE_ID	LOAD_DTS	RECORD_ SOURCE
1	1	1	12/17/2009-03:05:04	sales.cust
2	2	1	12/17/2009-03:05:04	sales.cust
3	3	2	12/17/2009-03:00:00	sales.cust
4	4	3	12/17/2009-03:00:00	sales.cust

Unlike a data model in third normal form, a Data Vault model completely ignores the cardinality of relations (1:N; N:M; . . .). Every relationship in a Data Vault model is stored as if the cardinality were N:M (many-to-many). This gives the model the most flexibility; no matter what the data looks like in the different sources it can always be stored in the Data Vault.

The composite of the hub surrogate keys forms an alternate key, which is sometimes referred to as the business key of the link table, though technically this is not 100 percent correct.

Satellites

So far we have covered hubs and links. The hubs correspond to the different entities that exist in an organization: customers, stores, products, employees, orders, and so on. The links add the dynamics, the transactions, and the relationships within the data. With hubs and links, we created a skeleton model; all we need to do now is add the "flesh" to it in the form of *satellites*.

Data Vault uses satellite tables to store the attributes of hubs and links, including all historical changes. A satellite table always has one (and only one) foreign key that refers to the single hub or link it belongs to.

Each satellite contains the fields listed in Table 19-5 (all required, no other attributes allowed). Table 19-6 shows an example satellite table.

Table 19-5: Satellite Attributes

ATTRIBUTE	DESCRIPTION
`Primary key`	Surrogate key, system generated, for internal use
`Foreign Key`	The foreign key of the hub or link the satellite row describes
`Load DTS`	The timestamp the data is loaded in the EDW, system generated
`Load End DTS`	The timestamp the data is no longer valid because it was changed
`Record source`	Defines the origin of the data (for example, source system, table)
`Attributes {1..N}`	The attributes themselves

NOTE **The Data Vault standard states that the primary key of a satellite should consist of the foreign key to the hub or the link table together with the field** `Load DTS`**. Although that is correct, we added a surrogate primary key. This diversion from the standard is based on technical reasons. A SQL** `UPDATE` **statement is easier (and probably faster) when a single surrogate primary key is present. However, the creation of a unique index on the foreign key and the** `Load_DTS` **is advisable.**

The data in Table 19-6 contains two attributes from a customer: `City` and `Birthdate`. If one of these changes in the source system (`sales.cust`), a new row is inserted in the satellite table and the old row is end-dated. The rows with ID 4 and 56 both describe the same customer with `cust_id` 41 for a different period in time.

Table 19-6: Example Satellite Table sat_customer

ID	HUB_CUST_ID	LOAD_DTS	LOAD END DTS	RECORD_SOURCE	CITY	BIRTHDATE
1	14	12/17/2009-03:05:04	NULL	sales.cust	Paris	05/09/2006
2	26	12/17/2009-03:05:04	NULL	sales.cust	New York	06/24/1998
3	37	12/18/2009-03:00:00	NULL	sales.cust	London	07/20/1963
4	**41**	12/18/2009-03:00:00	**02/30/2010-03:00:00**	sales.cust	**Amsterdam**	04/03/1941
...	...	...	...	...	...	...
56	**41**	**02/30/2010-03:00:00**	NULL	sales.cust	**Utrecht**	04/03/1941

Any hub or link can have more than one satellite; in fact, this is recommended. Imagine two source systems contain customer data: `sales` and `marketing`. The same business key is present in both systems (if you're lucky). It is good practice to create at least two satellites, one for each source system. Other reasons (besides source system) for splitting up satellites are rate of change (slowly and rapidly changing attributes) or type of data.

Note that a satellite table, like a link table, does not contain a real business key, but the alternate key, consisting of the composite of the Foreign Key and Load DTS, is often referred to as such.

NOTE **A pure DV model will use NULL for the Load End DTS, as displayed in Table 19-6, but this might not be an optimal solution for querying the model. Sometimes a value such as `12/31/2999` or another future date is used instead. Purists might argue that this is in fact a lie; if the date is unknown, it shouldn't have a value at all. When building DV solutions that need to be portable and adhere to the standard, stick to NULL. Otherwise, be pragmatic but consistent. In all cases, you should always have any default values used signed off by the end users.**

Data Vault Characteristics

The following list is not meant to be a complete and/or formal specification of the Data Vault standard. It is used to show some of the unique qualities of a Data Vault model and to increase the general knowledge and understanding of both the architecture and the modeling technique.

A well designed Data Vault model has the following characteristics:

- Temporal storage of all relevant data, even data that is of "low" quality; you do not want disruptions during the ETL process.
- As few dependencies as possible.
- As source-independent as possible.
- Designed for change.
  - Agile to changes in the source tables
  - Extensible without changing current model (incremental approach)
- ETL jobs are completely (and always) restartable.
- Full traceability of data.

Building a Data Vault

The general steps for building a Data Vault are as follows:

1. Model the hubs, strongly focused on business keys.

2. Model the links; look for transactions and relationships (hint: foreign keys). Hubs and links together should give you a good understanding of the way the organization operates today.
3. Model the satellites; provide the context. This completes the Data Vault.

NOTE There can be a fourth step, "Model the Point-In-Time tables." Point-In-Time (PIT) tables are redundant helper tables, based on satellites that can be created to make querying the Data Vault easier. We will skip this step because PIT tables are beyond the scope of this book.

Transforming Sakila to the Data Vault Model

In Chapter 4, the sakila database was directly transformed to a dimensional model. In this chapter, the model is first transformed to a data vault and then to the same star schema as in Chapter 4.

Sakila Hubs

For this example, the following entities will be converted to hubs: `actor`, `category`, `customer`, `film`, `staff`, and `store`. These are the easy ones; some others are less obvious. The `inventory`, `payment`, and `rental` tables might also be links (because of their transactional nature). However, when we convert these tables to links, we will end up with link tables that link to other link tables, so called `link-to-link` tables. The architecture does not recommend using such tables, so we'll make them hubs also.

The final interesting set of tables to consider when looking for hubs is the geographical collection: `address`, `city`, and `country`, and the `language` table. The developers of Sakila decided to model these tables as separate entities. For your purposes, these will be reference data. For your business (DVD rentals) these tables currently do not qualify as hub entities (which means the data will end up in satellite tables as we move on).

The final list of hubs is presented in Table 19-7.

NOTE The selection of hubs, links, and satellites is not as definitive as you might think. If you decide after a while that, for instance, address, city, and country should be hubs, just convert them. You may find a Data Vault model to be surprisingly forgiving.

Table 19-7: Sakila Hubs

ENTITY	BUSINESS KEY
hub_actor	actor_id
hub_category	category_name

ENTITY	BUSINESS KEY
hub_customer	customer_id
hub_film	film_id
hub_inventory	inventory_id
hub_payment	payment_id
hub_rental	rental_id
hub_staff	staff_id
hub_store	store_id

Sakila Links

After the selection of hub tables, it is time to focus on the links. This means you have to look for transactions and relationships. Foreign keys in the source model(s) can be very helpful with this step.

The transactional tables are pretty easy to locate: `payment` and `rental`. These are the transactions that keep Sakila in business, after all. So even though `payment` and `rental` were selected as hubs in the previous section, they must be links as well, given their transactional nature.

The other link tables are relationships. A customer usually visits only one store (probably the one closest to his or her home). This is modeled in sakila with a foreign key in the `customer` table, `store_id`. This is a typical link that shows the difference between a Data Vault model and a normalized data model. The source data model (sakila) defines the cardinality of this relationship as one-to-many (a customer can have one and only one store). Data Vault models define every relationship as many-to-many, therefore a separate table is required: `link_customer_store`.

The complete list of link tables can be found in Table 19-8.

Table 19-8: Sakila Links

LINK	HUBS THAT ARE LINKED
link_film_actor	hub_film, hub_actor
link_film_category	hub_film, hub_category
link_customer_store	hub_customer, hub_store
link_inventory	hub_inventory, hub_film, hub_store
link_payment	hub_payment, hub_customer, hub_staff
link_payment_rental	hub_payment, hub_rental
link_rental	hub_customer, hub_inventory, hub_staff
link_staff_worksin_store	hub_staff, hub_store
link_store_manager	hub_staff, hub_store

NOTE **In the sakila database, the `store` table contains a foreign key to the `staff` table (`manager_staff_id`), which defines the store manager. The `staff` table also contains a foreign key to the `store` table (`store_id`), which probably defines in which store a member of the staff works. These foreign keys are each converted to a link table between `hub_staff` and `hub_store`; they're both converted to a many-to-many relationship and therefore look very much alike.**

Sakila Satellites

The hubs and links together form the data skeleton, which is shown in Figure 19-1.

The satellite tables complete the Data Vault model and provide the attributes of both hubs and link tables. Every attribute in the source tables that is used in the Data Vault model should eventually be housed in the Data Vault structure.

The definition of the satellite tables is fairly straightforward, so take a look at Table 19-9 for the list of satellite tables.

Table 19-9: Sakila Satellites

SATELLITE	DESCRIBES
sat_actor	hub_actor
sat_film	hub_film
sat_customer	hub_customer
sat_payment	hub_payment
sat_rental	hub_rental
sat_staff	hub_staff
sat_store	hub_store

NOTE **When selecting the hub tables, we chose to create separate hubs for rental, payment, and inventory. This is why all our satellites link to hub tables. One might question whether these hubs are really necessary. A rental, for example, is a transaction, which can be modeled using just a link table (`link_rental`). There is no *real* business key. The most important reason to add the `hub_rental` table as well is to prevent link-to-link relationships. A link-to-link is allowed in Data Vault modeling but it is not recommended, mainly because of issues when querying the Data Vault.**

Figure 19-2 shows the complete Data Vault for Sakila.

NOTE **For your convenience, the diagram is available as a Power*Architect file (`Sakila-data-vault.architect`) on the book's companion website at `www.wiley.com/go/kettlesolutions`.**

hub_actor
hub_actor_id: INTEGER [PK]
actor_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_film_actor
link_film_actor_id: INTEGER [PK]
hub_film_id: INTEGER [FAK]
hub_actor_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_film
hub_film_id: INTEGER [PK]
film_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_category
hub_category_id: INTEGER [PK]
category_name: VARCHAR(25) [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_film_category
link_film_category_id: INTEGER [PK]
hub_category_id: INTEGER [FAK]
hub_film_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_store
hub_store_id: INTEGER [PK]
store_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_inventory
link_inventory_id: INTEGER [PK]
hub_inventory_id: INTEGER [FK]
hub_film_id: INTEGER [FK]
hub_store_id: INTEGER [FK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_inventory
hub_inventory_id: INTEGER [PK]
inventory_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_rental
link_rental_id: INTEGER [PK]
hub_rental_id: INTEGER [FAK]
hub_customer_id: INTEGER [FAK]
hub_staff_id: INTEGER [FAK]
hub_inventory_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_rental
hub_rental_id: INTEGER [PK]
rental_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_payment_rental
link_payment_rental_id: INTEGER [PK]
hub_rental_id: INTEGER [FAK]
hub_payment_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_payment
hub_payment_id: INTEGER [PK]
payment_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_customer_store
link_customer_id: INTEGER [PK]
hub_customer_id: INTEGER [FAK]
hub_store_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_customer
hub_customer_id: INTEGER [PK]
customer_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_store_manager
link_store_manager_id: INTEGER [PK]
hub_staff_id: INTEGER [FAK]
hub_store_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_staff_worksin_store
link_staff_worksin_id: INTEGER [PK]
hub_staff_id: INTEGER [FAK]
hub_store_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

hub_staff
hub_staff_id: INTEGER [PK]
staff_id: INTEGER [AK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

link_payment
link_payment_id: INTEGER [PK]
hub_payment_id: INTEGER [FAK]
hub_customer_id: INTEGER [FAK]
hub_staff_id: INTEGER [FAK]
load_dts: TIMESTAMP
record_source: VARCHAR(100)

Figure 19-1: The Sakila data skeleton

Figure 19-2: The complete Sakila data vault

Loading the Data Vault: A Sample ETL Solution

We are about to dive into the details of our sample ETL solution, and you are encouraged to follow along and examine its nuts and bolts directly—"live" so to speak on your own computer, running your own copy of Kettle. So before we actually look at the individual transformations and jobs of the sample ETL solution, it is a good idea to first obtain the files that make up the sample solution, and verify that you can open them.

Installing the Sakila Data Vault

You can download the SQL script files for the Sakila Data Vault from the book's companion website. As with the Sakila sample database, the script file is archived and available as a `.zip` and a `.tar.gz` archive.

So, the installation procedure for the Data Vault schema is the same as for the original Sakila sample database: Simply unpack the archive and use the `SOURCE` command in the MySQL command-line utility to execute the script. Unlike the Sakila sample database, we do not provide the data in a script; we want you to load the data in the Data Vault using the ETL packages. The script is named `Sakila_data_vault_schema.sql`.

Setting Up the ETL Solution

You can obtain all these files from this book's website in the folder for Chapter 19. All files are available in `.zip` and `.tar.gz` archives called `ch19_ktr_and_kjb_files`. Simply download the archive, and extract it to some location on your hard disk.

Creating a Database Account

The transformations for the ETL solution use two specific database accounts to access the sakila and Data Vault schema: a sakila account, which is used to read from the sakila sample database (which you probably already created in Chapter 4), and a `sakila_dv` account (`sakila_data_vault` exceeds the maximum length for a user name in MySQL), which is used to read from and write to the Data Vault schema. To create the account, use the `mysql` command-line client to log into MySQL as a user with `SUPER` privileges, such as the built-in `root` account. Then, enter the following commands:

```
CREATE USER sakila_dv IDENTIFIED BY 'sakila_dv';
GRANT ALL PRIVILEGES ON sakila_data_vault.* TO sakila_dv;
```

For your convenience, these commands are also available as a SQL script file, named `create_sakila_dv_account.sql`, which is included in the `ch19_ktr_and_kjb_files` archive along with the Kettle transformation and job files.

The Sample ETL Data Vault Solution

Now that we have described the database schemas, we can examine how a Data Vault model can be loaded from a source system. Please keep in mind that this is a very simple example with just a single source system.

NOTE **In a real world implementation you would definitely use a staging area. For reasons of brevity and clarity we've decided to leave this out in the following examples.**

One of the nice aspects of Data Vault modeling is the repeatability of the design and load processes. Because of the rather strict modeling rules, every hub has similar attributes. The same can be said about links and satellites. This means that loading routines will be very similar as well and that automatic generation of loading process or even modeling the Data Vault might be worth exploring.

Because of these similarities we will describe only three transformation files: a sample hub, a sample link, and a sample satellite.

Sample Hub: hub_actor

The transformation file for loading the `hub_actor` table is called `hub_actor.ktr`. The transformation is shown in Figure 19-3.

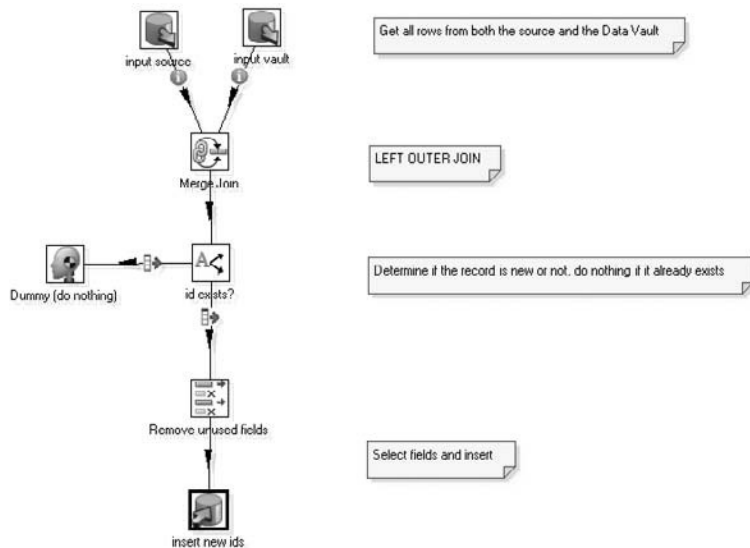


Figure 19-3: The hub_actor transformation

The transformation `hub_actor` is responsible for inserting all actor entities (business keys) from the `actor` table in `sakila.actor` that are not yet present in the Data Vault table `hub_actor`.

The following list provides a brief description of the steps shown in Figure 19-3:

- **input source:** The "input source" step executes the following SQL code against the Sakila source database:

  ```
  SELECT actor_id AS id
  ,      'sakila-db.actor' AS record_source
  FROMactor
  ORDER BY 1
  ```

  Basically it selects every `actor_id` (business key) from Sakila. For your convenience, it also defines the record source as `sakila-db.actor`. The `ORDER BY` clause sorts the data by the first field, `actor_id`.

- **input vault:** The "input vault" step selects the `actor_ids` from your target schema, the Data Vault, using the following SQL statement:

  ```
  SELECT   actor_id AS id_existing
  FROM     hub_actor
  ORDER BY 1
  ```

  Again, this results in a sorted list of `actor_ids`, similar to the one from the previous step.

- **Merge Join:** The Merge Join step that follows performs a `LEFT OUTER JOIN` on the two sets of `actor_ids`. The result is a list that might look like the following:

  ```
  id  record source     id_existing
  1   sakila-db.actor   1
  2   sakila-db.actor   2
  3   sakila-db.actor   3
  4   sakila-db.actor   4
  5   sakila-db.actor   NULL
  6   sakila-db.actor   NULL
  ```

  This list tells you that your source system, sakila, contains six `actor_ids` and that the Data Vault table `hub.actor` contains only `actor_ids` 1–4. Apparently `actor_ids` 5 and 6 have been inserted since the last time the Data Vault was loaded and hence need to be added in this load.

NOTE **You probably noticed the two icons with the letter *i* in blue circles on the hops between the "input source," "input vault," and Merge Join steps. These are warnings that the Merge Join step expects the incoming streams to be sorted. This can be achieved with the Sort Rows step. In our case, we let the database perform this task (using the `ORDER BY` clauses in the SQL queries).**

- **id exists?:** The next step is a Switch / Case type step. All rows with a NULL value in the id_existing column are sent to the "Remove unused fields" step; the other rows are sent to the "Dummy (do nothing)" step where they will be ignored.
- **Remove unused fields:** This step strips the data stream from the fields that are no longer needed, leaving you with two fields: actor_id (previous id) and record_source.
- **Dummy (do nothing):** This step does exactly what you would expect based on its name.
- **insert new ids:** This final step is a "Table output" type step and inserts the remaining rows into the Data Vault table hub_actor.

NOTE **In addition to** actor_id **and** record_source**, the table** hub_actor **contains two additional fields:** hub_actor_id **and** load_dts**. These fields are automatically filled by the database.** Hub_actor_id **is of type** auto_increment**, and** load_dts **has a default value of** CURRENT_TIMESTAMP**, which ensures that the field always contains the timestamp of insertion into the Data Vault.**

If your database is running and you set up the Sakila_data_vault schema and corresponding user account, you should be able to simply run the transformation and load the hub_actor table. The transformation should complete in a matter of seconds on any modern laptop or desktop computer.

Sample Link: link_customer_store

The transformation file for loading the link_customer_store table is called link_customer_store.ktr. The transformation is shown in Figure 19-4.

The transformation link_customer_store is responsible for inserting all relationships between the customer and store entities from the customer table in sakila.actor that are not yet present in the Data Vault link_customer_store table. This relationship describes the store a customer is registered for.

A brief description of the steps shown in Figure 19-4 follows:

- **input:** The "input" step executes the following SQL code against the sakila source database:

```
SELECT   customer_id
,        store_id
,        'sakila-db.customer' AS record_source
FROM     customer
ORDER BY 1, 2
```

Basically it selects every combination of customer_id and store_id (business keys) from sakila.customer.

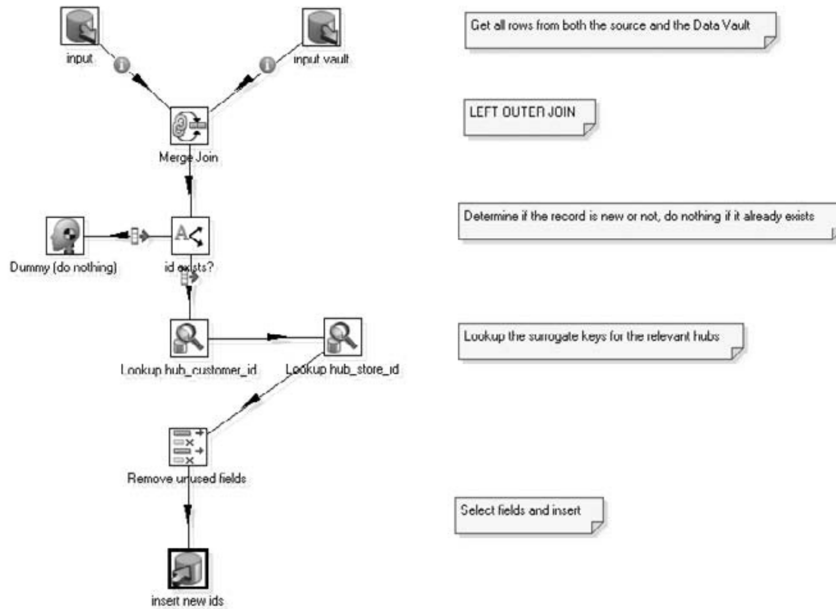


Figure 19-4: The link_customer_store transformation

- **input vault:** Similar to the "input vault" step from the sample hub transformation, another list is created from the Data Vault, containing the same information. Because we have to join on the business keys in the next step, we need to join the table `link_customer_store` with `hub_customer` and `hub_store`:

```
SELECT     customer_id AS customer_id_vault
,          store_id    AS store_id_vault
FROM       hub_customer
INNER JOIN link_customer_store
USING      (hub_customer_id)
INNER JOIN hub_store
USING      (hub_store_id)
ORDER BY   1, 2
```

  The columns `customer_id` and `store_id` are renamed in the query (`_vault` is added to the name) to distinguish them from the columns of the "input" step.

- **Merge Join:** Again, similar to the hub example, the Merge Join step that follows performs a `LEFT OUTER JOIN` on the two sets of business keys.
- **id exists?:** The "'id exists?" step filters the rows that are new and redirects the other rows to the dummy step.
- **Lookup hub_customer_id / Lookup hub_store_id:** At this point in the transformation, you have created a list of new `customer_ids` and `store_ids` that need to be inserted in your link table. However, a link table does not contain business keys, only surrogate keys from hubs or links.

The "Lookup hub_customer_id" step looks up the business key (customer_id) in the hub and converts it to the corresponding surrogate key (hub_customer_id). Naturally, the next step, "Lookup hub_store_id," performs a similar action for store_id.

- **Remove unused fields:** This step leaves you with the fields you need for insertion: hub_customer_id, hub_store_id, and record_source.
- **insert new ids:** Finally, the data is inserted in the table link_customer_store.

If your database is running and you set up the sakila_data_vault schema and corresponding user account, you may have tried to run the transformation, only to find out that it failed on execution. This is because it needs at least the tables hub_customer and hub_store to contain all their data.

For your convenience we added a job to load all hub tables, named load hubs.kjb. You can find the jobs load links.kjb and load satellites.kjb as well. Figure 19-5 shows the job to load all hubs.

As you can see in Figure 19-5, all hub tables can be loaded in parallel. There is absolutely no relationship between hubs (and there shouldn't be one, hence the rule that hubs may not contain foreign keys).

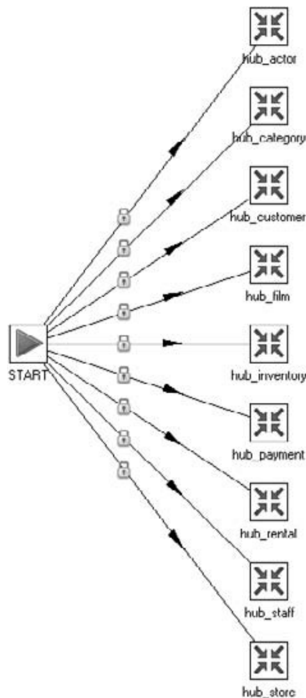


Figure 19-5: Load all hubs in parallel

After loading the hubs, you can start the link_customer_store transformation. This transformation should complete in a matter of seconds on any modern laptop or desktop computer.

Sample Satellite: sat_actor

The transformation file for loading the `sat_actor` table is called `sat_actor.ktr`. The transformation is shown in Figure 19-6.

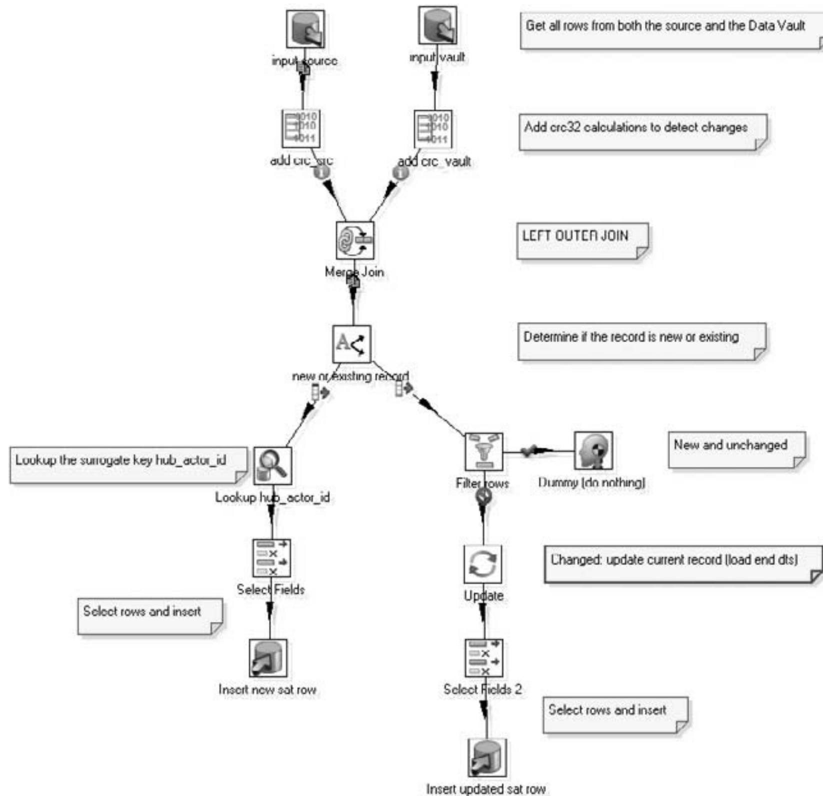


Figure 19-6: The sat_actor transformation

The transformation `sat_actor` is responsible for capturing all actor attributes *and the historical changes* from the `actor` table in `sakila.actor` that are not yet present in the Data Vault table `sat_actor`.

A brief description of the (relevant) steps shown in Figure 19-6 follows:

- **input source:** The "input source" step executes the following SQL code against the Sakila source database:

```
SELECT   actor_id
,        IFNULL(first_name, '')  AS first_name
,        IFNULL(last_name, '')   AS last_name
,        IFNULL(last_update, '') AS last_update
,        'sakila-db.actor'       AS record_source
,        NOW()                   AS load_end_dts
FROM     actor
ORDER BY 1
```

The SQL code selects the attributes from the source database that need to be inserted in the satellite table. In the next step, you create a Cyclic Redundancy Check (CRC) from these attributes to detect possible changes. CRC functions usually do not like NULL values; therefore, we used the ISNULL function from MySQL. Every common database management system contains a similar function.

The load_end_dts field is used later in this transformation. Right now it stores the current datetime-stamp using the MySQL function NOW().

- **input vault:** Again, similar to the sample hub and the sample link, a similar list is built based on the data in the Data Vault, ready for joining in the Merge Join step.

```
SELECT      actor_id
,           sat_actor_id
,           hub_actor_id
,           IFNULL(sa.first_name, '') AS first_name_vault
,           IFNULL(sa.last_name, '')  AS last_name_vault
,           IFNULL(last_update, '')   AS last_update_vault
FROM       sat_actor sa
INNER JOIN hub_actor
USING      (hub_actor_id)
WHERE      load_end_DTS IS NULL
ORDER BY   1
```

Because you need the business key (actor_id), you have to join the hub_actor table in this query.

The WHERE statement ensures that you're selecting only the latest version of the satellite data. The historical attributes are not relevant for this situation.

NOTE **When you use a CRC technique (or similar method—Kettle also supports ADLER 32, MD5, and SHA-1) to detect changes, make sure that you use exactly the same functions (such as ISNULL()) on both sides of the calculation.**

- **add crc_src and add crc_vault:** The two steps "add crc_src" and "add crc_vault" each add an additional field to the stream. These are calculated fields, based on a standardized calculation that is guaranteed to generate the same result when the same data is fed to the algorithm.

In this case, you feed three fields to the CRC32 calculations: first_name, last_name, and last_update. All NULL values are replaced with empty string values.

As long as the data in the source system has not changed, the CRC calculations will generate the same results. When the data in the source system is changed, you will detect this change because the results of the calculations will no longer be the same.

NOTE **Theoretically, CRC calculations with a different input can generate the same output. So while there is a very small chance that a change in the source system remains undetected, we decided to take this chance. To completely avoid this chance, you have to do a field-by-field comparison which will slow down the process considerably.**

- **Merge Join:** Again, similar to the hub example, the Merge Join step that follows performs a `LEFT OUTER JOIN` on the two sets of business keys.
- **new or existing record:** The next step determines whether you are dealing with a new or an existing record. When `sat_actor_id` contains the value `NULL`, it must be a new record; otherwise it is existing. New records are sent to "Lookup hub_actor_id" while existing records follow "Filter rows."
- **Lookup hub_actor_id:** Similar to the earlier link example, you have to convert a business key (`actor_id`) to the corresponding surrogate key (`hub_actor_id`). This is what happens in the step "Lookup hub_actor_id" before the row gets stripped of unnecessary fields in the Select Fields step.
- **Insert new sat row:** A new row is inserted using the following fields: `first_name`, `last_name`, `hub_actor_id`, `record_source`, and `last_update`.

  There is another path you need to examine, however: The "Filter rows" step is the start for every existing row.
- **Filter rows:** The "Filter rows" step compares the two CRC32 calculations you added a couple of steps earlier. When these fields are the same, the row has not been altered and can be ignored. When the two stream fields are different, you need to do two things:
  - End-date the current satellite row.
  - Insert a new satellite row (that will be current from now on).

  This is exactly what the next two steps will do.
- **Update:** The Update step sets the field `load_end_date` in the satellite table to the value of the stream field `load_end_date` (which is calculated in the step "Input vault").
- **Insert updated sat row:** Finally, a new satellite row is inserted, based on the stream fields `first_name`, `last_name`, `hub_actor_id`, `record_source` and `last_update`.

Loading the Data Vault Tables

Now that we have described the three different transformations for the Data Vault model, it's time to load the actual tables. For this purpose, we added three Kettle jobs that load the hubs, the links, and the satellites: `Load Hubs.kjb`, `Load Links.kjb`, and `Load Satellites.kjb`. Refer back to Figure 19-5 to see `Load Hubs.kjb`.

The jobs need to be executed in this order: hubs ➪ links ➪ satellites. However, note that all hubs can be loaded in parallel, all links can be loaded in parallel, and all satellites can be loaded in parallel.

Updating a Data Mart from a Data Vault

By now you realize that a Data Vault model is very well equipped for integrating, storing, and safeguarding your valuable data. However, it is less suitable for intensive querying or reporting.

This is why a data warehousing architecture that uses Data Vault modeling for the enterprise data warehouse usually contains a data mart layer where one or more star schemas exist for end-user access to the data. In this section, we show you how you can populate a star schema based on a Data Vault enterprise data warehouse.

We will use the exact same star schema that was first introduced in Chapter 4. For a brief explanation and installation instructions please see the section "The Rental Star Schema" in Chapter 4.

The Sample ETL Solution

This sample ETL solution is very similar to the one described in Chapter 4 (some of the code is even identical). However, we wanted to provide you with a complete solution that enables you to gain some experience with Data Vault models and their behavior. The next subsections describe the transformations we created to load the star schema from the Data Vault. Most of the transformations are described individually, except for `dim_staff` and `dim_store` because of their resemblance to `dim_customer`.

NOTE `Dim_date` **and** `dim_time` **are static dimensions that are initially loaded and do not need to be reloaded or refreshed periodically. The transformation files for loading** `dim_date` **and** `dim_time` **are called** `dim_date.ktr` **and** `dim_time.ktr` **and are identical to the transformations used in Chapter 4. They are added to this chapter for your convenience only and will not be discussed in detail here.**

The dim_actor Transformation

The `dim_actor` table is populated by the transformation `dim_actor.ktr` and is displayed in Figure 19-7.

Because `dim_actor` does not use slowly changing dimension logic (only the most recent attributes of actors are stored in the star schema; history is ignored), the transformation that loads the `dim_actor` table is very straightforward.

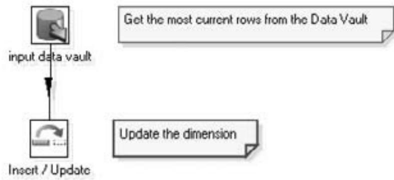


Figure 19-7: The dim_actor transformation

A "Table input" type step is followed by a step that inserts new records and updates existing records where appropriate.

The steps from the `dim_actor` transformation are:

- **Input Data Vault:** In this step, the following SQL query loads the relevant data from the Data Vault:

```
SELECT     sa.last_update AS actor_last_update
,          ha.actor_id    AS actor_id
,          sa.first_name  AS actor_first_name
,          sa.last_name   AS actor_last_name
FROM       sat_actor sa
INNER JOIN hub_actor ha
USING      (hub_actor_id)
WHERE      sa.load_end_dts IS NULL
```

  Basically this step reads the attributes from `sat_actor`. The table `hub_actor` is joined because you need the attribute `actor_id`. Because you are currently not interested in the history of actors, the statement `WHERE sa.load_end_dts IS NULL` ensures that you receive only the most recent attributes.

NOTE **Although right now, you may not be interested in the historical attributes of actors, this may very well change in the future. If this happens, you can rely on the Data Vault model as it still contains a full history trace of actors.**

- **Insert / Update:** Thanks to the functionality of the Insert / Update step, this transformation is easy. There is no need to track changes yourself.

  The properties of the Insert / Update step are shown in Figure 19-8. The correct record is matched using `actor_id` and the attributes will be updated if they are changed.

Figure 19-8: The Insert / Update step

The dim_customer Transformation

The `dim_customer` transformation looks rather complex (see Figure 19-9) compared to `dim_actor`. When you look back to Chapter 4, it is also more complex than the `load_dim_customer` transformation that is discussed there.

The reason for this complexity lies in the type 2 slowly changing dimension logic. As discussed before, the Data Vault model contains full history; this means that it can contain numerous versions of the same customer and that your SQL query can result in multiple versions of the same customer. These versions need to be stored in the type 2 slowly changing dimension table `dim_customer` accordingly. The component "Dimension lookup/update" currently does not support this behavior and therefore cannot be used. This means you have to do it yourself.

The transformation consists of two streams (one for the Data Vault data and one with existing data in the star schema). The two streams are joined together to detect new customers and changed customers. New customers are simply inserted; changed customers need to be updated (field `customer_valid_through`).

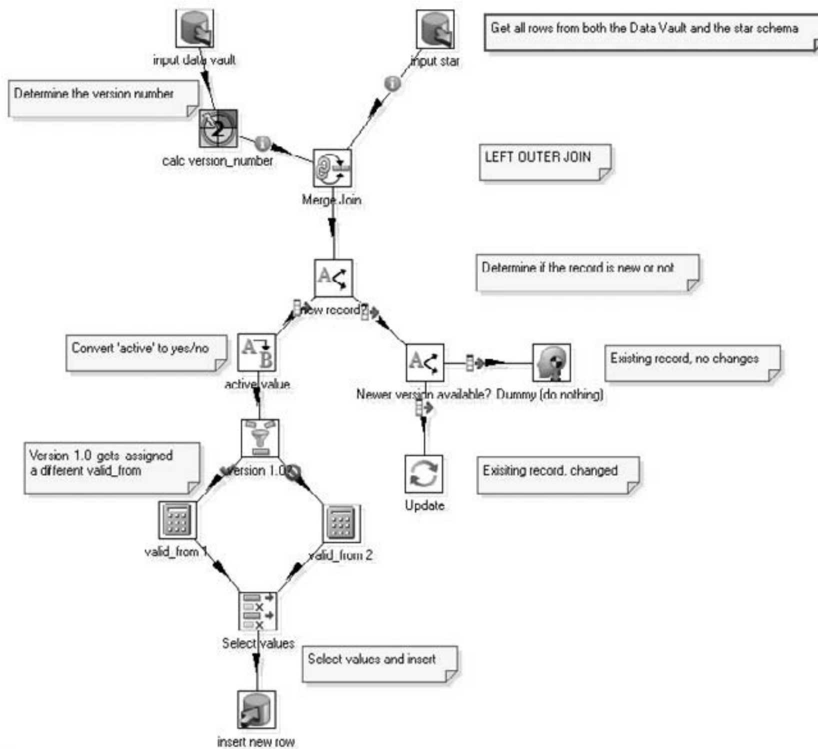


Figure 19-9: The dim_customer transformation

The steps from the dim_customer transformation are:

- **input data vault:** The first step fires the following SQL query to the Data Vault:

```
SELECT
  DATE(sc.load_dts)             AS customer_valid_from_tmp
, IFNULL(DATE(sc.load_end_dts),
  DATE('2199/12/31'))           AS customer_valid_through
, sc.last_update                AS customer_last_update
, DATE(sc.create_date)          AS customer_created
, hc.customer_id                AS customer_id
, sc.first_name                 AS customer_first_name
, sc.last_name                  AS customer_last_name
, sc.email                      AS customer_email
, sc.active                     AS customer_active
, sc.address                    AS customer_address
, sc.district                   AS customer_district
```

```
, sc.postal_code                AS customer_postal_code
, sc.phone                      AS customer_phone_number
, sc.city                       AS customer_city
, sc.country                    AS customer_country
FROM
  sat_customer sc
INNER JOIN
  hub_customer hc
USING
  (hub_customer_id)
ORDER BY
  hc.customer_id, sc.load_dts
```

This is not a very complex query. It selects the relevant fields from `sat_customer`, joined with `hub_customer` because you need the `customer_id`. The two fields, `customer_valid_from` and `customer_valid_through`, are based on the Data Vault fields `load_dts` and `load_end_dts`, but this needs some additional attention.

The Data Vault does not know when the first version of a customer started its validity. It is fair to say that is was probably created before it was loaded in the Data Vault, so you need to trust the source system and use the field `create_date` from Sakila. If this field is not available (which is the case for `dim_store` and `dim_staff`) you will use the value `1970/01/01`.

In Chapter 4, the convention is introduced to set the `customer_valid_through` to `2199/12/31` when a customer is still valid. You will use the same convention; the preceding `IFNULL` statement implements this convention.

- **input Star:** The next step, "input star," executes the following SQL query:

```
SELECT   customer_key
,        customer_id         AS star_customer_id
,        customer_valid_from AS star_customer_valid_from
FROM     dim_customer
ORDER BY customer_id, customer_valid_from
```

It simply selects the existing customers in `dim_customer` and needs no further explanation.

- **calc version_number:** Before the two streams are joined, the Data Vault stream uses a very smart step known as "Add fields changing sequence." This step generates a sequence that resets whenever a certain field changes (see Figure 19-10). This is exactly what you need to create a version number for customers. When the stream is sorted by `customer_id` and `load_dts`, the version number increases until `customer_id` changes, which renders the version number back to 1.

Figure 19-10: Calculate the version number

The transformation continues with a Merge Join step that joins the two streams. Based on the customer_key field, you can determine whether a row is new or not. Existing rows may need to be updated.

- **Newer version available?:** When a customer changes (for example, a new address), a new row is added to the satellite table. This means that a new row will be added to the dim_customer table in the star schema. However, the current row needs to be updated (the field customer_valid_through). The step "Newer version available?" detects this. The next step, Update, changes the value of customer_valid_through.
- **Version 1.0?:** As discussed previously, the first version of a customer needs a different value for the field customer_valid_from. It must be set to the value of create_date instead of load_dts. The step "Version 1.0?" uses the stream field customer_version_number to redirect the row to the correct calculation. Figure 19-11 shows the details of this step.

Figure 19-11: customer_version_number

The dim_film Transformation

The transformation for dim_film would have been very easy, similar to dim_actor if the designers hadn't created the sets of [film_has_. . .] and [field_in_category_. . .] fields. The set [film_has_. . .] is a flattened version of the field special_features and the set [field_in_category_. . .] is based on the many-to-many relationship between film and category. Figure 19-12 shows the transformation.

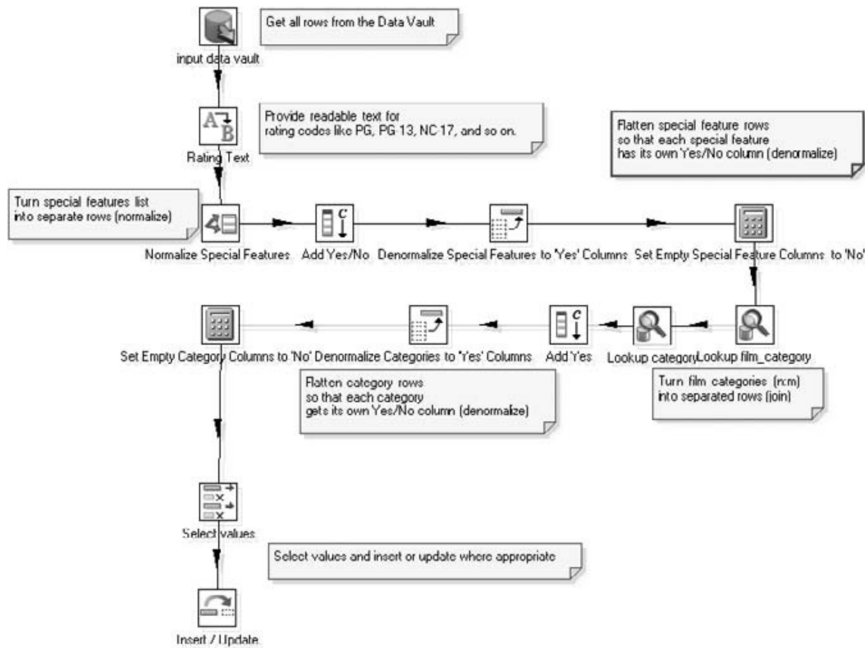


Figure 19-12: The dim_film transformation

Although the transformation dim_film looks rather complex we will not discuss it here. The transformation is very similar to load_dim_film, which is described in Chapter 4.

The dim_film_actor_bridge Transformation

The relationship between film and actor is many-to-many (one actor can play in many films and in one film many actors can play). Each rental (the granularity of the fact table) captures the rental of a single film. This means that the dimension table dim_actor cannot be related to the fact table fact_rental.

Because you want to be able to report about films and actors, a bridge table is created. This bridge table models the many-to-many relationship between dim_film and dim_actor.

The transformation load_dim_film, as described in Chapter 4, contains the logic to load dim_film_actor_bridge. In this chapter, you make this a separate transformation, as shown in Figure 19-13.

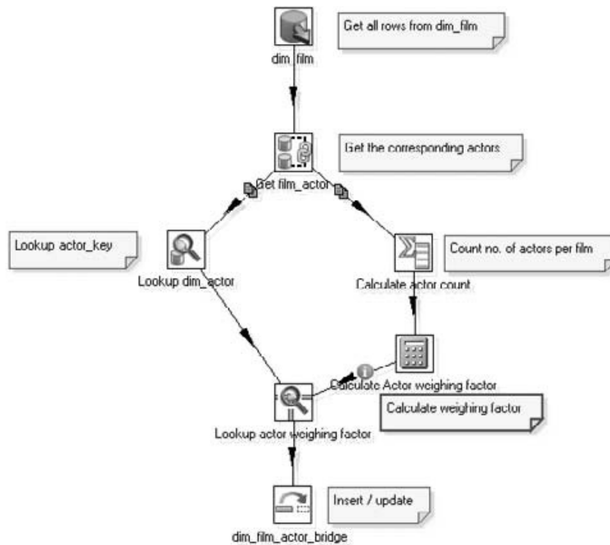


Figure 19-13: The dim_film_actor_bridge transformation

The transformation starts with an "Input table" type step, which loads the key values from dim_film. The next step is a Database Join type step that selects the actor_ids for each film_id.

The resulting stream is split into two separate streams. The left stream uses a "Database lookup" type step to match actor_id to its surrogate key actor_key. The stream on the right uses a "Group by" type step to count the number of actor per film. This number is required to calculate the weighting factor in the following step. Finally, the two streams come together again in the step "Lookup actor weighting factor" and the table dim_film_actor_bridge is updated.

The fact_rental Transformation

The final transformation that we will describe in this chapter loads the fact table fact_rental. The transformation is called fact_rental.ktr. Again, this transformation is very similar to the corresponding load_fact_rentals transformation described in Chapter 4, except for the first two table input steps, which are described following. The transformation is displayed in Figure 19-14.

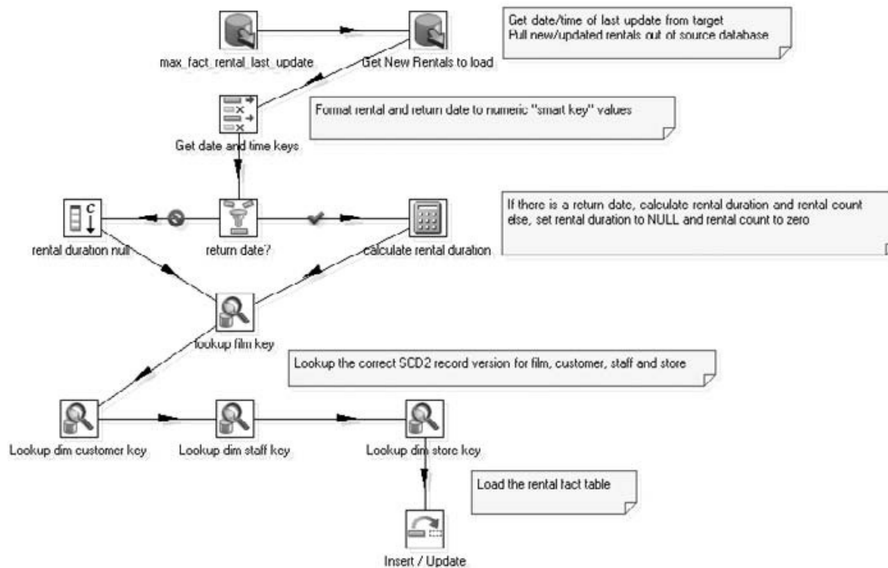


Figure 19-14: The fact_rental transformation

`Fact_rental` starts with two "Table input" type steps that capture changed data. The first step selects the latest value for the field `rental_last_update` and the second step, "Get New Rentals to load," selects the required data from the Data Vault using the following SQL query:

```
SELECT     hr.rental_id
,          sr.rental_date
,          sr.last_update
,          sr.return_date
,          hc.customer_id
,          hs.staff_id
,          hstore.store_id
,          hf.film_id
FROM       link_rental lr
INNER JOIN hub_rental hr
USING      (hub_rental_id)
INNER JOIN sat_rental sr
USING      (hub_rental_id)
INNER JOIN hub_inventory hi
USING      (hub_inventory_id)
INNER JOIN hub_customer hc
USING      (hub_customer_id)
INNER JOIN hub_staff hs
USING      (hub_staff_id)
INNER JOIN link_inventory li
USING      (hub_inventory_id)
INNER JOIN hub_store hstore
```

```
USING      (hub_store_id)
INNER JOIN hub_film hf
USING      (hub_film_id)
WHERE      sr.last_update > ?
```

Although this looks complex, it really is not. The data you need to insert into `fact_rental` consists of data from several tables in the Data Vault. You need data about the rental, the customer, the store, the staff, and the film. This query shows the essence of a Data Vault model; every distinct piece of data gets assigned its own table, thus creating optimal flexibility when data is added or changed. This query simply joins the tables that contain the required data.

Loading the Star Schema Tables

To load the star schema tables, we added two Kettle jobs, one that loads the date and time dimension, `Load date time.kjb`, and one that loads the dimensions and fact tables, `Load dimensions facts.kjb`.

The job that populates the `dim_date` and `dim_time` tables needs to be executed only once; it is not really part of the ETL solution.

Summary

In this chapter, we introduced the concept of Data Vault modeling as a technique to build an enterprise data warehouse. The Data Vault has many advantages over other data modeling techniques, with the following as the most notable ones:

- **Auditability/traceability:** Every change to the data in the source system is captured without alteration, making a DV well suited for environments where compliance is an issue.
- **Repeatability/simplicity:** The DV model has only a limited set of entity types (hubs, links, satellites), with easy, repeatable structures, making a DV very suitable for automatic generation of models and ETL code.
- **Adaptability:** Changes in the structure of the source systems can easily be *added* to the DV without having to change the existing tables.

As an example, we created a Data Vault database based on sakila, the same sample database used throughout the book. The steps and transformations needed to load the Data Vault from the source system and the data marts from the Data Vault were explained in detail. This chapter also showed that Kettle is a general-purpose ETL tool that can be used in any data warehouse scenario.

CHAPTER

20

Handling Complex Data Formats

In typical data warehousing scenarios, the ETL process extracts data from operational systems to load it into the data warehouse. In most cases, both the operational system and the data warehouse use an RDBMS to store the data. For the ETL process, this means that the data is at least available in a comprehensible relational format.

As should be clear from Chapters 6 through 9, a relational format does not suddenly mean that the transformations required to get the data from source to target database are trivial or even simple. But at least data is clearly separated from metadata, and there is no question with regard to fundamental relationships between the individual pieces of data: Rows tie attribute values together in a uniform format, and foreign keys establish relationships between rows in a clear and unambiguous manner.

Perhaps a more important feature is that the consistent use of relational data simplifies transformations. No matter how drastically data is transformed, input and output still share the fundamental characteristics of a table, and can still be seen as a collection of rows that are segmented into columns.

So what if the source data is *not* in a tabular format? In Chapter 21, you will read about XML and JSON, which are non-tabular because they allow arbitrary nesting of data structures. But still, although non-tabular, these data formats have a clear grammar and thus a clear separation of data and metadata.

In this chapter, we take a look at some non-relational or even non-tabular data formats that, in contrast to XML and JSON, do not comply to a clear grammar.

Non-Relational and Non-Tabular Data Formats

We are not always so lucky as to have relational data at our disposal. While the following is not an exhaustive list of data formats, we identify a number of categories of non-tabular data:

- **Structured non-tabular data formats:** Here, the data is highly structured, and there is a relatively small set of metacharacters and rules that define the formal characteristics of the data format. XML and JSON are formats that match this category. These formats are discussed in detail in Chapter 21.
- **Non-relational but tabular:** In this case, the data consists of a collection of rows, and each row is clearly segmented in fields. However, the table may still contain multi-valued attributes and/or repeating groups. In this case, the data is not relational.
- **Semi-structured or unstructured data formats:** In these cases, it may not be clear what the rules are that control the shape and form of the data, or the set of rules may be very complex and hard to formulate.
- **Key/value pairs:** Often, data is line-oriented: one record is kept on one line of text, which is then segmented into fields. But sometimes, records appear as a sequence of lines, where each line contains a key and a value part.

In the remainder of this chapter, we show examples of the last three of these data formats and briefly highlight which Kettle features you can use to handle them. We also discuss some sample transformations to explain these Kettle features in detail.

Non-Relational Tabular Formats

Tabular data is non-relational when it contains either or both:

- Multi-valued attributes
- Repeating groups

If the source data contains either or both of these structures, then it will usually be desirable to use the ETL process to restructure the data to a regular relational format. This does not mean that the data must be stored in a relational format in the data warehouse. But at the least, one would want to transform the data to a relational format to do efficient cleansing and conforming.

Handling Multi-Valued Attributes

A multi-valued attribute is an attribute that may have a collection of values rather than a single scalar value. You encountered an example of a multi-valued attribute in Chapter 4: The `special_features` column of the `film` table of the Sakila sample database has the MySQL-specific `SET` data type (see Figure 4-1). Columns with this data type contain values that appear as a comma-separated list of string values, which is called a *set*.

Using the Split Field to Rows Step

In the `load_dim_film` transformation shown in Figure 4-16, we used a "Split field to rows" step to split the list into individual rows, each having a single value for the attribute. In the `load_dim_film` transformation, the "Split field to rows" step is labeled "Normalize special features." Chapter 4 does not discuss a separate transformation to illustrate this step so we discuss it in detail here. Figure 20-1 shows the configuration of that particular step.

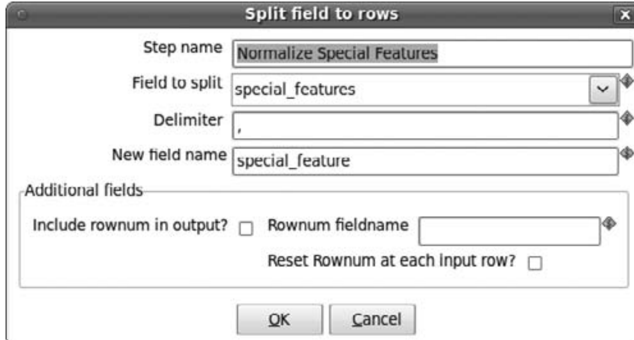


Figure 20-1: The "Split field to rows" step

The top section of the dialog in Figure 20-1 is used to define a number of properties that are essential for transforming the multi-valued field to a list of rows.

In the "Field to split" property, you can use the drop-down list to pick the field from the incoming stream that contains the multi-valued attribute. In this case, it is the field that originates from the `film.special_features` column. In the definition of the `film` table, the field is declared as:

```
special_features SET('Trailers','Commentaries',
                     'Deleted Scenes','Behind the Scenes')
```

This definition will allow the field to contain a comma-separated list composed from any combination of any of the values named in the comma-separated list within the parentheses following the `SET` keyword.

The Delimiter field is used to specify a string that separates the individual values in the multi-valued field. Because values for MySQL `SET` columns are comma-separated lists, we specify a comma (`,`) here. This causes the step to generate a new row for each distinct value in the comma-separated list. For example, the film titled `ACADEMY DINOSAUR`, which has `'Deleted Scenes,Behind the Scenes'` as special features will yield two output rows because the comma separates two distinct values.

Once the multi-valued attribute is split, it becomes a normalized scalar attribute. The values for this attribute are added to the output stream in a new field, and you can use the "New field name" property to specify the name of the new field.

The "Additional fields" section of the dialog shown in Figure 20-1 can be used to add row numbering to the newly generated rows. To generate row numbers, select

the "Include rownum in output?" checkbox. You can specify the name for the row number field in the "Rownum fieldname" property. Finally, you can select the "Reset Rownum at each input row?" checkbox to restart numbering from 1 for each row in the incoming stream.

Handling Repeating Groups

A repeating group occurs whenever a particular type of column (or group of columns) occurs multiple times in a table. For example, the `film` table in the sakila database has two columns that refer to the `language` table: `language_id` and `original_language_id`, and this may be construed as a repeating group.

NOTE **Although the sakila example where the `film` table has two foreign keys to the `language` table can be explained as a repeating group, many information analysts will be reluctant to do so. The mere fact that in both cases, the column serves to identify a language is not enough to deem it a repeating group of language because the two references to the language table each have their own well-defined semantics. So although this particular example is far from perfect, we stick with it because it's the only one that's readily available in the sakila database.**

Using the Row Normaliser Step

With Kettle, you can normalize repeating groups of columns in tables using the Row Normaliser step. You can find this step beneath the Transform folder in the left side pane tree view. The `sakila-normalize-repeating-groups` transformation provides an illustration of the Row Normaliser step. The transformation and the configuration dialog of the Row Normaliser step are shown in Figure 20-2.

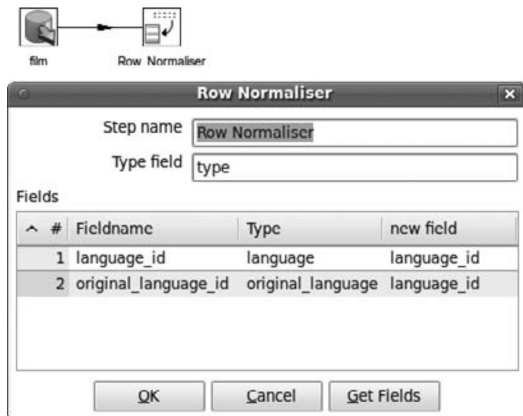


Figure 20-2: Using the Row Normaliser step to normalize repeating groups

For each repeating group, a distinct value is entered in the Type column. In this case, the group occurs twice, and so there are two distinct values for Type. For all columns in a group defined by a distinct Type value, a row is entered in the grid. In this case, the repeating group consists of just one column. The Type values are added to a new field in the outgoing stream, and the "Type field" property is used to specify the name for the field that receives the Type values.

For each incoming row, the Row Normaliser step generates just as many rows as there are distinct values in the Type column of the grid. The generated rows have a new field for the Type value, but the repeating instances of the columns in the repeating group are removed, leaving only one instance of the columns in the repeating group.

Semi- and Unstructured Data

In many cases, the data is available in files that are much less rigorously structured. Whereas humans are often capable of identifying interesting bits of data and understanding how the pieces of data are related, automated data extraction requires the definition of a *pattern* that describes the format of the data. Identifying and defining such a pattern is a prerequisite to data extraction: Without it, you have no way of automating the process.

Regular expressions are an old and proven method to deal with searching and matching text patterns. Regular expressions are widely supported in many languages, including Perl, JavaScript, and Java.

NOTE A detailed discussion of regular expressions is outside the scope of this book. However, there are many good books and online tutorials on regular expressions, and you shouldn't have much trouble learning more about them. See, for example, the Wikipedia article at `http://en.wikipedia.org/wiki/Regular_expression` **or the site** `http://www.regular-expressions.info/`**.**

As it turns out, you can identify a pattern and write a regular expression for it in most cases, although it may be hard, and sometimes *very* hard, to do so. It may even be possible that developing a catch-all pattern is not worth the cost. In these cases, you can only aim for a "good enough" solution, and have to accept that you are currently not capable of accessing all the data you would like.

In any case, you should always put something in place to register any data that does not match your regular expression, and inspect the unmatched data regularly in order to improve your regular expression. Sometimes you may find it easier to develop a separate processing path to handle these cases.

NOTE As an example of a good-enough solution, consider e-mail addresses. An extensive discussion of the considerations and trade-offs for this particular problem is included in the regular-expressions info site at `http://www.regular-expressions.info/email.html`**. Defining a regular expression**

that matches the large majority (99 percent) of the valid e-mail addresses and that correctly detects most invalid e-mail addresses is a simple task and can be done in just 40 characters:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$
```

The actual syntax for e-mail address is defined in RFC 2822, at `http://tools.ietf.org/html/rfc2822#section-3.4.1`**. The regular expression pattern that corresponds to this specification is quite convoluted, and is 426 characters long, more than 10 times the length of the naïve approach.**

For an example of seemingly unstructured data, consider the following snippet from the `movies.list` file provided by the Internet movie database, which hosts a large collection of information about movies and television series at `http://www.imdb.com/`:

```
#1 (2005)                                    2005
#1 Fan: A Darkomentary (2005) (V)            2005
$100,000 Pyramid DVD Game, The (2006) (VG)   2006
$50,000 Challenge, The (1989) (TV)           1989      (unreleased)
'Columbia' Winning the Cup (1901/I)          1901
'Columbia' Winning the Cup (1901/II)         1901
1 Second Film, The (2008) {{SUSPENDED}}                2008
21 Days (1940)                               1940    (shot 1937)
Andrey Rublyov (1966)                        1966    (shot 1964-1965)
"#1 College Sports Show, The" (2004)         2004-????
"#1 Single" (2006) {Cats and Dogs (#1.4)}    ????
"$1,000,000 Chance of a Lifetime" (1986)     1986-1987
"$10,000 Pyramid, The" (1973)                1973-1988,1991-1992
"$10,000 Pyramid, The" (1973) {(1973-03-26)} 1973
"10 Years Younger" (2004/I)                  2004
"10 Years Younger" (2004/I) {(#2.8)}         2005
```

You can download this and many other lists as a gzipped archive from one of the FTP servers listed at the imdb website at `http://www.imdb.com/interfaces#plain`.

Although there definitely is a pattern in the film list, it is not straightforward. At the very least, it is not regular enough to expect anything else from the standard text file input steps than reading individual lines. You can't use these steps for parsing individual fields. Even if you could, the construction of the initial field is so complex, that you would still need another pass to parse it. To give you an idea of the complexity, here are a few facts that are known about the format:

- Each line represents one "movie," which may be a cinematic film, TV movie, TV series, video release, mini series, or videogame.
- The line starts with a complex key that identifies the movie within the file.
- The key is followed by a variable number of tabs, an optional parentheses-enclosed code indicating the movie type (`TV` for TV film, `V` for video release, `VG` for video game, `mini` for mini-series), another sequence of one or more tab characters, and

the four-digit release year. Optionally, the release year is followed by yet another collection of tabs and a parenthesized bit of extra information, such as `(shot 1937)` to indicate it was filmed in a particular year, `(unreleased)` to indicate the work was never officially released.

- If there is no explicit type indicator, the item is either a cinematic movie or a TV series. TV series titles are enclosed in double quotes but cinematic movie titles are not.

Kettle Regular Expression Example

In order to extract data from the IMDB movies list, we prepared the `imdb-movies` transformation. You can download this transformation from this book's website in the download folder for Chapter 20. To run the transformation, you need to download the `movies.list.gz` file from one of the IMDB FTP sites and place it in the same directory as the `imdb-movies.ktr` transformation file. The transformation works directly on the gzipped archive, so you shouldn't uncompress it. The transformation is shown in Figure 20-3.

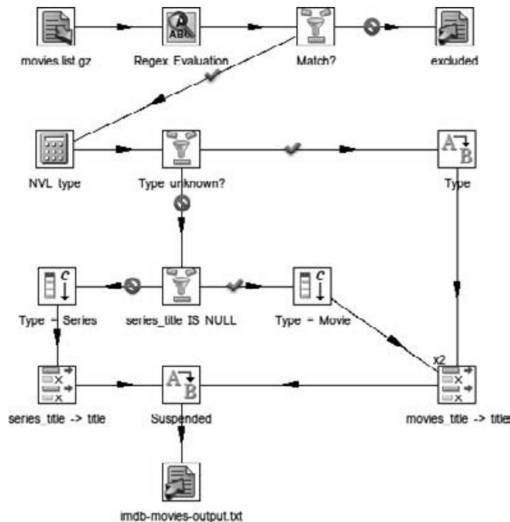


Figure 20-3: Loading data from the IMDB movies list using regular expressions

The top stream of the transformation is the focus for this example. The remainder of the transformation is mostly about determining the type of the movie. In the final step of the transformation, the entries are written to the text file `imdb-movies-output.txt`. By then, the information hidden in the movie keys is nicely separated into all kinds of

fields, such as the title of the entry, type, any part or episode number (for series), and so on. For example, the entry with the key

```
"10 Years Younger" (2004/I) {(#2.8)}
```

is processed to a record such as:

```
line_number: 566296
year: 2004
part: I
secondary_title: (#2.8)
real_type: Series
title: 10 Years Younger
```

Note that the preceding snippet is just an enumeration of some of the extracted fields and their values. The actual `imdb-movies-output.txt` file uses a separated values format having each record on one line, and has far more fields than the ones shown.

Configuring the Regex Evaluation Step

In Kettle, you can use the Regex Evaluation step for working with Java regular expressions. This step resides beneath the Scripting category in the left side pane tree view.

For the syntax supported by the Kettle Regex Evaluation step, see the API documentation of the `Pattern` class in the `java.util.regex` package. You can find the relevant documentation at `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`.

The configuration dialog for the Regex Evaluation step contains two tab pages and a field grid for the output fields. We describe each of them in this subsection.

Settings Tag Page

In the top of Figure 20-4, you can see the Settings tab page in the configuration dialog of the Regex Evaluation step from the `imdb-movies` transformation.

In the "Regular expression" text area shown in the middle of Figure 20-4, you must enter the regular expression pattern. This pattern will be matched against the entire field value; it will not search for an occurrence of the pattern inside the field value. The regular expression text may contain Kettle variable references, and if you rely on those, you need to select the "Use variable substitution" checkbox to allow Kettle to replace them with their value.

For the `imdb-movies` transformation, the regular expression is quite complex, which is why it is spread over multiple lines. Almost each line is commented using a single line comment initiated by the hash sign (`#`). Multiple lines, white space, and comments are entirely optional and do not influence how the text will be matched, but it is advisable to use comments anyway for complex regular expressions like this.

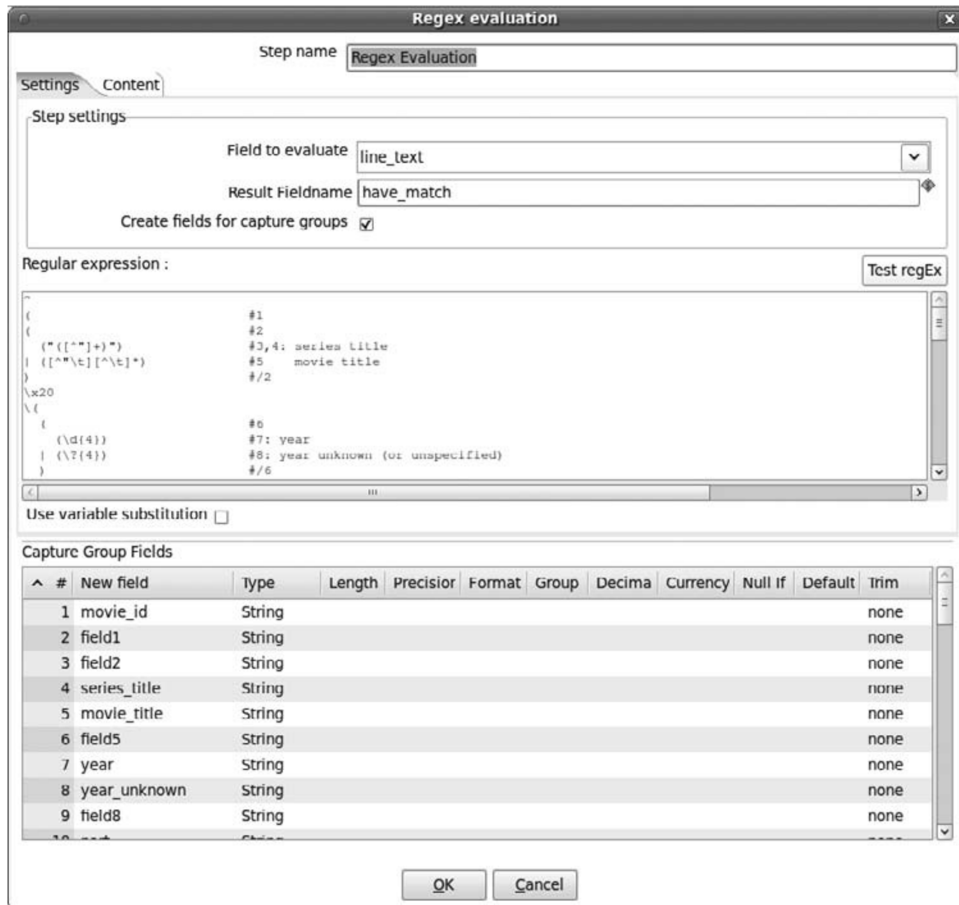


Figure 20-4: The Settings tab page and Fields grid of the Regex Evaluation step

In the Regex Evaluation step of the `imdb-movies` transformation, some comments explain what kind of text it is supposed to match; others have only an integer to identify the capturing group. This is invaluable for keeping track of capturing groups that are spread out over many lines.

For those readers who are not familiar with regular expressions or with the regular expression comments, let's consider this snippet from the top part of the regular expression shown in Figure 20-4.

```
(                   #2
  ("([^"]+)")       #3,4: series title
| ([^"\t][^\t]*)    #5    movie title
)                   #/2
```

On the first line of the snippet, the opening parenthesis starts a capturing group. Capturing groups are the mechanism used by Kettle to extract data from a matched pattern and transfer it to a field. The snippet begins at the second capturing group in

the entire expression, so that's why the comment reads `#2`. The final line of the snippet closes the capturing group `#2`, which is why we use the comment `#/2`.

On the next line is a pattern that is supposed to match a series title. We explained that series titles are enclosed in double quotes, and this is what the pattern attempts to match: a double quote followed by `[^"]+`, which denotes one or more characters that are not a double quote, again followed by a double quote. The pattern to match inside the quotes constitutes a new capturing group, which is why the comments mention `#3,4`.

On the next line is a pipe `|`, which denotes an alternative pattern. The alternative pattern is again a capturing group (`#5`), which tries to first match `[^"\t]`. This expression matches any character that is not (hence the initial `^` inside the square brackets) a double quote or a horizontal tab character (which is denoted with a so-called escape sequence, `\t`). After the initial character, `[^\t]` matches any character that is not a tab. The rationale for this pattern is that a movie title cannot start with a double quote (because it would be a series title in that case) or a tab (because the movie key is separated by a variable number of tabs from whatever more data appears on the line).

In the "Step settings" fieldset, you must specify the field from the incoming stream that will be matched against the regular expression in the "Field to evaluate" property. The Regex Evaluation step will always return the result of matching the value of that field against the regular expression in a new field in the outgoing stream. This field is a Boolean and its value indicates whether the value of the "Field to evaluate" matches the entire regular expression. The name for the result field can be configured in the "Result Fieldname" property.

We just mentioned that Kettle can transfer the pieces of text matched by the capturing groups in fields of the output stream. To enable this behavior, you must check the "Create fields for capture groups" checkbox.

Fields Grid

The Fields grid is shown in the bottom half of Figure 20-4. If the "Create fields for capture groups" checkbox is checked, you must enter a field definition for each capturing group defined in the regular expression.

Content Tab Page

The Content tab page is shown in Figure 20-5.

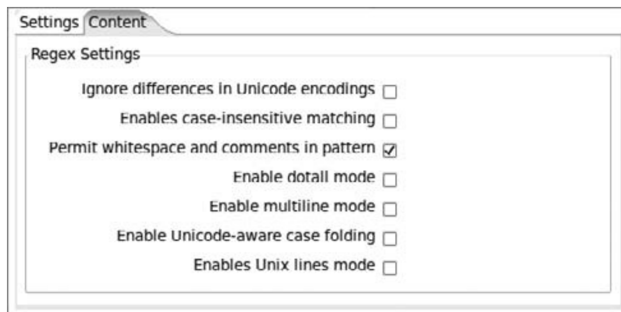


Figure 20-5: The Content tab page of the "Regex evaluation" step

On the Content tab, you can check or uncheck a number of flags that influence the behavior of the matching process. The options are listed here:

- **Ignore differences in Unicode encodings:** Select this option when you are matching ASCII text and you want to optimize performance. Typically, you should not select this option unless you have a very good reason to.
- **Enables case-insensitive matching:** By default, selecting this option enables case-insensitive matching for ASCII text. To enable case-insensitive matching also for Unicode text, be sure to also select the "Enable Unicode-aware case folding" option.
- **Permit whitespace and comments in pattern:** Use this option if you want to spread out the regular expression over multiple lines and document it with comments. In this mode, the whitespace character is not considered part of the regular expression, so to match whitespace, you must use an escape like `\s` or `\x20` to encode whitespace characters. If the option is disabled, then whitespace characters in the regular expression are considered literal, and denote a required match with a whitespace character in the text. Because it can be tedious and cause errors if you switch this option on or off afterward, you better decide in advance if you need it, and then stick by your choice.
- **Enable dotall mode:** Typically, the dot character denotes a match with any character *except* the line terminator. If this option is checked, the dot matches any character, *including* the line terminator.
- **Enable multiline mode:** Typically, the anchors `^` and `$` match the start and end of input respectively. If this option is checked, the `^` anchor also matches each line start, and the `$` also matches each line end.
- **Enable Unicode-aware case folding:** See the previously described "Enable case-insensitive matching" option.
- **Enable Unix lines mode:** Typically, both the control-return/linefeed sequence (`\x13` followed by `\x10`) and the linefeed character are recognized as line terminators. If this option is enabled, only the `\x10` character counts as linefeed, which influences the matching behavior of the dot (.), caret (^), and dollar sign ($) meta characters.

By default, all these options are disabled.

Verifying the Match

The Regex Evaluation step always passes on the input rows to the output, regardless of whether the regular expression matches the input. Typically you should check whether it, in fact, matches your input.

In the `imdb-movies` transformation, the check is performed by the Filter Rows step labeled "Match?" which appears immediately after the Regex Evaluation step. This step simply checks the value of the field that conveys the result of the Regex Evaluation step. The name of this field is configured by the "Result Fieldname" property in the

Settings tab page of the configuration dialog of the Regex Evaluation step shown in Figure 20-4.

If the match is `true`, the row flows to the remainder of the transformation, but otherwise, it is led to the "Text output" step labeled "excluded," where it is logged in a file called `imdb-movies.excluded.txt`. In a production environment, these files should be monitored at least on a daily basis, and a decision should be taken as to what to do with this data. A possible outcome is that you discover that the regular expression isn't quite good enough yet and needs some extra logic to accommodate the hitherto excluded records.

Key/Value Pairs

You can find many examples of key/value structured data on the Internet. For example, an online product catalog may offer a page with detailed information per product, and these pages may enumerate all relevant properties of the product, such as size, color, and price in a list.

For an example of data that uses key/value pairs, see the following code:

```
LASERDISC LIST
==============
----------------------------------------
OT:

LN: 3
LB: Philips
CN: 21317

LT: Stephen King's Nightmare Collection
PC: USA
CF: 16
CA: Movie
GR: Horror
LA: German
SU: -

RD: 1992
ST: Available
PR: DM 69.00

VS: PAL
CO: Color
SE: Digital
AL: -
AR: -
MF: Film
SZ: 12
SI: 2
DF: CLV
```

```
CC: -
QP: -
---------------------------------------
OT: "Absolutely Fabulous" (1992)

... more data here...
```

The listing is a snippet taken from the `laserdisc.list` file provided by the Internet movie database. (See the earlier `movies.list` example for instructions to obtain this file.)

The laserdisc list is a list of records, each describing a particular DVD or Blu-ray edition of a movie, documentary, television series, and the like. Records are separated from one another by a separator line consisting of dashes:

```
---------------------------------------
```

The record separator is followed by a variable number of lines, each consisting of two segments:

- A code consisting of two capital letters, indicating some property that applies to this particular laserdisc. This is the *key*.
- A text that describes the property for the relevant laserdisc record. This is the *value*.

In the laserdisc example, the key and value segments are separated by a colon, immediately followed by a space character. In addition to the key/value pairs, the laser disc records may also contain empty lines, which can be ignored.

Kettle Key/Value Pairs Example

To illustrate how to deal with key/value pairs in Kettle, we created the `imdb-laserdisc` transformation. You can download this transformation from the book's website in the download folder for Chapter 20. The transformation reads the gzipped laserdisc list file from the IMDB archives, and transforms the key/value pairs to proper columns. So, in order to run it, you must also download the `laserdisc.list.gz` file from one of the IMDB FTP sites and place it in the same directory as the `imdb-laserdisc.ktr` transformation file. The transformation is shown in Figure 20-6.

The following sections walk you through the main steps of this transformation.

Text File Input

The `imdb-laserdisc` transformation initially uses the same strategy as the `imdb-movies` example. The content of the list is simply read on a line-by-line basis, and the "Text input" step does not attempt to divide the file into fields. Instead, it outputs the entire line in a single field called `line_text`.

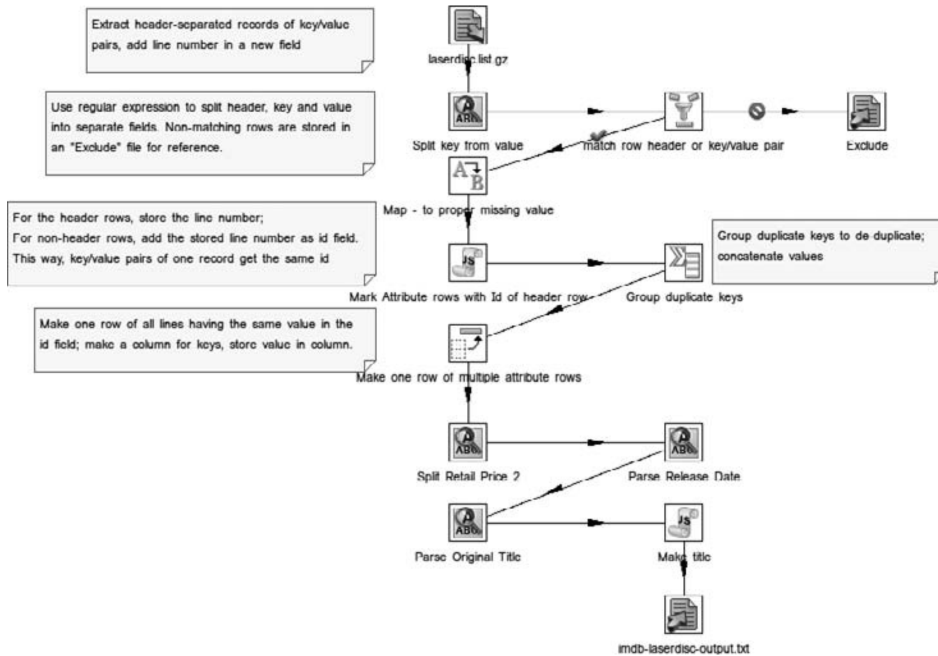


Figure 20-6: Processing key/value pairs from IMDB's laserdisc list

The step also adds an extra `line_number` field to the output stream. As the name suggests, this field contains the line number, which we use later on in the transformation as a key to tie all lines belonging to one laserdisc record together. To add a line number to the output stream of the "Text input" step, configure it and select the "Rownum in output" checkbox and specify the name for the field in the "Rownum fieldname" property. Both of these options are located on the Content tab page in the configuration dialog.

Regex Evaluation

Then, the lines are fed into a Regex Evaluation step labeled "Split key from value." The snippet that follows shows a fragment of the regular expression:

```
^
    (-+)        # 1: record header
|   (           # 2: key/value pair
        (       # 3: key
          LN    # <LaserDisc Number>
        | LB    # <Label>

... many, many more lines of regex pattern code here...

        | AQ    # <Audio Quality>
        )       # /3
```

```
            :\x20?
            (.+)?   #4: value
        ) #/2
    $
```

At the top level, the regular expression matches either the record header (capturing group #1—a line of dashes, which is stored in a field called `header`), or it matches a key/value pair (capturing group #2). The key/value pair itself is divided in a key (capturing group #3, which is stored in a field called `key`) and a value (capturing group #4, stored in the field `value`). After this initial regular expression matching, we again use a "Filter rows" step to confirm the match.

Grouping Lines into Records

The next step of interest is the "Modified Java Script Value" step labeled "Mark Attribute rows with Id of header row." The JavaScript code for this step is shown in Figure 20-7.

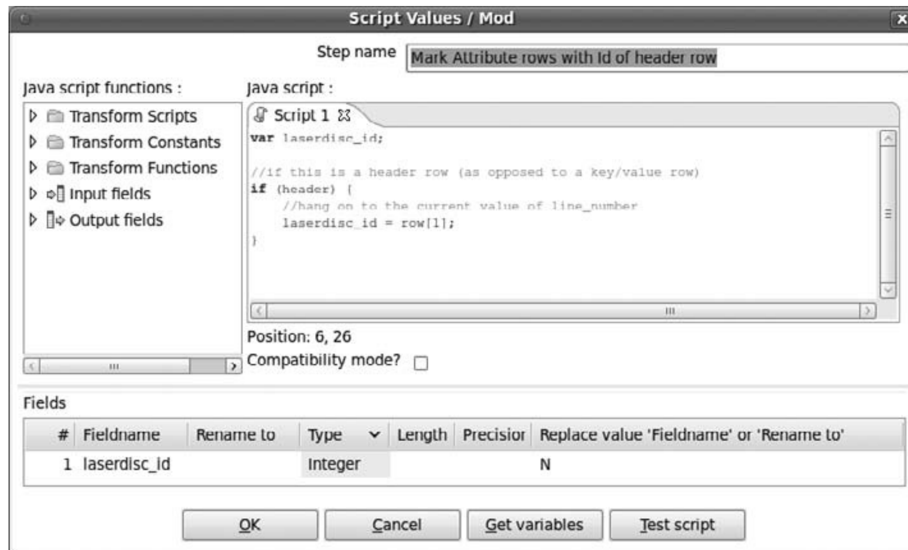


Figure 20-7: The "Mark Attribute rows with Id of header row" step

The JavaScript snippet detects whether the current row is a header row or a key/value row. It does this with a simple `if` statement that checks the value of the `header` field of the current row from the incoming stream. For the "Modified Java Script Value" step, the fields of the current row are automatically available as variables, so we can simply write:

```
if (header) {
    ...
}
```

If it is a header row, the header field will contain a string of dashes, as per the pattern for capturing group #1 in our "Regex evaluation" step. When used as a condition for an `if` statement, JavaScript will treat the non-empty string as `true`, and the code between the curly braces will be executed. If this is a key/value row and not a header row, the header field will be `null`, which is not considered `true`, and the code between the curly braces will be skipped.

So if we have a header row, the code between the curly braces is executed:

```
laserdisc_id = row[1];
```

The `row` variable on the right side of the assignment operator is automatically available and represents the current row from the incoming stream as an array of field values. The set of square braces in the code is the JavaScript syntax for array access so the expression `row[1]` actually gets the field value at index 1. Because the fields are numbered starting at `0`, `row[1]` refers to the second field of the current row, which is the `line_number` field added by the "Text input" step.

In the Fields grid in Figure 20-7, you can see how the `laserdisc_id` variable is added as a field to the outgoing stream. If you run the transformation in preview mode to see the data coming out of the "Modified JavaScript Value" step, you can see how the `laserdisc_id` field changes value at each header, giving all the subsequent key/value rows that belong to one record the same value. As you will see in the remainder of this section, this is essential for making proper columns out of the key/value rows.

Denormaliser: Turning Rows into Columns

The step labeled "Make one row of multiple attribute rows" is of the "Row denormaliser" type. You can find this beneath the Transform category in the left side pane tree view. This step does the actual job of turning the rows with key/value pairs to proper columns. The configuration for the step is shown in Figure 20-8.

The "Row denormaliser" step groups the rows from the incoming stream according to the fields listed in the grid labeled "The fields that make up the grouping" at the top of the configuration dialog. This is where we use the `laserdisc_id` field that we created with the "Modified JavaScript Value" step. By grouping, all the rows from the incoming stream having the same value for the `laserdisc_id` field end up in one row that is emitted to the outgoing stream.

NOTE The "Row denormaliser" step groups only consecutive rows from the incoming stream that have the same value in the `laserdisc_id` field. In the case of the `imdb-laserdisc` transformation, the key/value rows that belong to the same record are consecutive, which guarantees the grouping works out. If the rows that need to be grouped are not consecutive, you have to explicitly sort the rows, for example by using a "Sort rows" step.

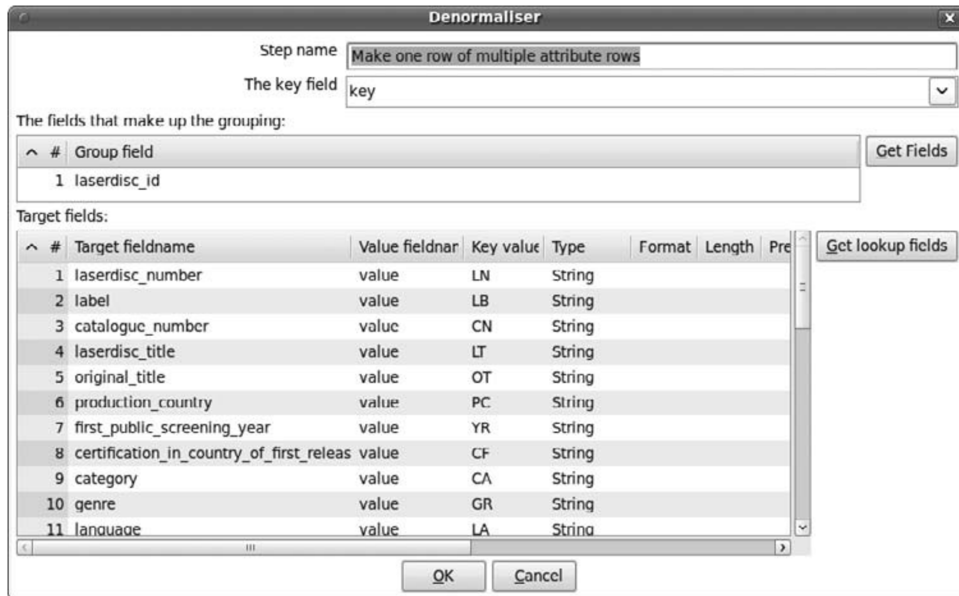


Figure 20-8: The "Row denormaliser" step

The "Target fields" grid in the lower half of the dialog shown in Figure 20-8 defines the fields that are to be added to the outgoing stream. The name for the field is specified in the "Target fieldname" column of the "Target fields" grid. To fill these fields for the outgoing rows, Kettle uses the following procedure:

- For each row in the group of rows defined by the grouping fields, Kettle takes the value of the key field. The key field is configured by choosing a field name for the property labeled "The key field" in the top of the configuration dialog.
- Kettle finds a row in the "Target fields" grid for which the value in the "Key value" grid column equals the value of the key field from the incoming stream.
- Once the row in the grid is determined, Kettle gets the field name from its "Value fieldname" column. This fieldname is used to get a value from the incoming row.
- The value is stored in a field of the outgoing stream using the name set in the "Target fieldname" column.

Summary

In this chapter, we looked at some examples of data that does not fit a relational format, and how to work with such data in Kettle. The chapter covered:

- A categorization of several types of non-relational data: non-relational tabular data is tabular data but with multi-valued attributes and/or repeating groups;

semi-structured and unstructured formats that adhere to complex patterns; and key/value pairs

- Using the Split Field to Rows step to normalize multi-valued attributes
- Using the Row Normaliser step to normalize repeating groups
- Using the Regex Evaluation step to match data against a regular expression pattern and to extract subpatterns of interest using capturing groups
- How to keep key/value pairs of the same record together
- Using the Row Denormaliser step to collapse a collection of key/value pairs into a tabular row

CHAPTER 21

Web Services

In Chapter 6, we discussed a number of Kettle features for Web-based extraction. In this chapter, we take a closer look at the Kettle features for working with websites and web services, and how to deal with the typical data formats they use to exchange data.

Before we dive into the details, we first need to explain how web services fit in the context of ETL and data integration, and discuss which concepts and techniques are involved. We also provide an overview of a few data formats that are commonly used to exchange data over the Web. In the remainder of this chapter, we illustrate a few typical scenarios for using Kettle with web services.

Web Pages and Web Services

The majority of what we typically refer to as "The Web" consists of web pages. Web pages are essentially documents, primarily intended for a human audience, that can be retrieved using Hypertext Transfer Protocol (HTTP) and are typically coded in HTML. Via a web browser application, users are connected to a server that is part of a network (the Internet), allowing them to retrieve web pages from different computers in the network (sites). The web pages themselves are human-readable documents that are coded in some form of hypertext, which simply means it contains conveniently navigable links to access other, related web pages.

HTTP defines how a request from a client (such as a web browser) is transferred over the Internet to finally reach a host that is able to send a response containing a resource. Despite its name, HTTP is not confined to the transfer of hypertext. As long as data

is properly encoded, HTTP can carry any kind of data. One might think of resources carried over HTTP as static documents or files. This needn't be the case, and often isn't: data can be dynamically generated in response to the HTTP request.

Similarly, Uniform Resource Locators (URLs) are used an addressing scheme to identify particular resources, as well as their location on the Internet. Although URLs can be thought of as addresses of static resources, their format is flexible enough to carry information in their own right. As such, they can be used to send a command to a computer over a network to perform a certain task (rather than just retrieving a particular resource). In these cases, the resource returned typically indicates status information about the execution of the command.

Web services differ from web pages in that they deliver a resource that is intended for machine processing: They deliver data that is not necessarily suitable for direct consumption by humans. Instead, web services are used as a mechanism to support data exchange between applications. Thus, a web service resembles an application programming interface (API). For examples of such web services, take a look at the Netflix web services at `http://developer.netflix.com/docs`, or the web services listed at `http://www.webservicex.net/WS/wscatlist.aspx`. Another very comprehensive overview that lists many web services can be found at `http://www.programmableweb.com/`.

Web services typically do not use HTML, but a format that is better suited for data exchange, such as XML or JSON. We will discuss the popular data formats for the Web in more detail later in this chapter.

Kettle Web Features

Kettle offers a number of features that enable you to work with the Web. We briefly discuss these features in the remainder of this section. We describe some of them in detail later on in this chapter as we discuss a few concrete examples.

General HTTP Steps

Kettle offers two general-purpose HTTP steps: "HTTP client" and HTTP Post. Both of these steps reside beneath the Lookup category in the left pane tree view. Both these steps perform an HTTP request and add the retrieved resource in a field of the outgoing stream. Both steps can either specify a URL directly, or accept one from the incoming stream, and for both steps, you can map the fields from the incoming stream to query parameters that are added to the URL.

The difference between these steps is that the "HTTP client" step performs an `HTTP GET` request, whereas the HTTP Post step performs an `HTTP POST` request. The HTTP Post step also allows you to control the message body that is being sent with the request. For example, in addition to sending field values along in the query part of the URL, you can also send them in the message body with the HTTP Post step, or send an entire file as the message body.

You will see the details of these steps later in this chapter in various examples.

Simple Object Access Protocol

Kettle offers support for Simple Object Access Protocol (SOAP) web services through the "Web services lookup" step. Like the general HTTP steps, this step also resides in the Lookup category in the left pane tree view. This step is used to invoke SOAP-based web services.

The "Web services lookup" step can use the WDSL document exposed by a SOAP service to discover which operations are supported by the service, and it uses that information to facilitate the mapping of fields from the incoming stream to parameters, and fields from the outgoing stream to the return value.

We discuss the "Web services lookup" step in more detail later in this chapter in a dedicated SOAP example.

Really Simple Syndication

Kettle offers support for Really Simple Syndication (RSS) input as well as RSS output. RSS is increasingly used to deliver periodically updated data streams. It is very popular for news and blogs, but is also used for publishing stock quotes and currency rates. RSS is discussed in a later section of this chapter.

Apache Virtual File System Integration

Kettle offers integration with the Apache Virtual File System (VFS). This allows you to use URLs in almost any place where you would typically enter a file name. This includes the file name property you enter for the various input steps, but also the name of a transformation file in a Transformation job entry or job file in a Job job entry. In Spoon, you can also open a job or transformation from a URL (main menu ➪ File ➪ Open from URL) or save to VFS (main menu ➪ File ➪ Save As (VFS)).

Because URLs contain a scheme part that identifies the protocol to use, this feature allows you to work not only with local files and resources available on the Web via HTTP, but also with files on a remote FTP server, files stored inside a `.tar`, `.jar`, `.zip`, or `.gzip` archive, and more.

NOTE **For more information on Apache VFS, see** `http://commons.apache.org/vfs/`**. For specific information on the supported file formats and the syntax for corresponding URIs, see** `http://commons.apache.org/vfs/filesystems.html`**.**

Data Formats

Although the HTTP protocol requires data encoding, it does not impose any particular type or format for any data transferred by it. However, some formats are certainly more popular than others. The formats we focus on in particular are XML, HTML, and JSON.

XML

XML is by far the most ubiquitous data format used by web services, both over the Internet and as exchange format for applications. We assume the reader is already familiar with XML and related technologies such as XPath and XML Schema.

NOTE For a good resource on XML, try *Beginning XML* by David Hunter et al., Wrox Press, 2007.

Kettle offers a number of steps and job entries for working with XML data in general. We demonstrate a few of these in the sample transformations later in this section, but for the benefit of those readers who are not yet familiar with XML and related technologies, we provide a quick tour that will give you an idea of the level of XML support offered by Kettle.

NOTE In addition to features for working with XML in general, Kettle also has a number of features for working with specific XML applications, such as RSS (Really Simple Syndication) and SOAP (Simple Object Access Protocol).

Kettle Steps for Working with XML

Kettle supports the following generic transformation steps for working with XML:

- The "Get data from XML" step resides beneath the Input category in the left side pane tree view. This step can read XML from a resource such as a file, a URL, or from a field in the incoming stream. Using XPath expressions, it extracts collections of elements and attributes from XML documents, and then turns them into fields and records, which are then pushed into the outgoing record stream. We discuss this step in more detail later in this chapter when we show a transformation for extracting data from XML.
- The XML Output step resides beneath the Output category in the left side pane tree view. It accepts an incoming record stream, and turns it into one or more XML documents, which are then stored as a file.
- The Add XML step resides beneath the Transform category and can be used to generate XML fragments from the incoming record stream. Typically, this is used to build complex XML documents with multiple levels of nested XML element structures. This step is discussed in detail in the transformation for generating XML documents later in this chapter.
- The XSL Transformation step also resides beneath the Transform category. This step applies an XSLT stylesheet on XML documents conveyed by the incoming stream, adding the transformation result to a field in the outgoing stream. A

detailed description of this step, as well as the XSLT language, is outside the scope of this book. Basically, XSLT is a very powerful method for processing XML data, which can be used both for extraction as well as for generating complex XML documents. It should certainly be considered to meet any requirements that are not easily achieved with Kettle's other XML features.

- The XML Join step resides beneath the Joins category. This step has the ability to merge an XML document with a stream of XML fragments, and can thus be used to generate complex, nested XML structures. This step is described in detail in the example transformation for generating XML data later in this chapter.
- The XSD Validator" step resides beneath the Validation category. This step can be used to check the validity of an XML document against an XML Schema document. XML Schema is a popular way for defining data types and structures in XML documents. It is often used for defining input and output requirements of web service protocols. Although a discussion of the XML Schema language itself is beyond the scope of this book, we describe this step in more detail later in this chapter when we discuss the sample transformation to extract data from XML documents.

NOTE **In addition to the transformation steps described in the preceding list, there are a couple of deprecated XML input steps: the Streaming XML Input step and the XML Input step, which are currently both located in the deprecated folder in the left side pane tree view. These steps are likely to be removed from a future Kettle version, and should not be used when building new transformations.**

Kettle Job Entries for XML

Kettle also offers a number of job entries for working with XML: these all reside beneath the XML category:

- The "Check if XML file is well formed" job entry can be used to bulk-check a folder to see if XML files are well-formed.
- The DTD Validator step can be used to validate an XML document against a document type definition (DTD)
- The XSD Validator step is used for validation against an XML schema, and is the job-entry analogue of the transformation step with the same name. We describe this step in more detail later in this chapter when we present a sample transformation to generate XML documents.
- The XSL Transformation job entry is analogous to the transformation step of the same name.

ECMASCRIPT FOR XML SUPPORT

In addition to the specialized steps and job entries for working with XML, Kettle's Modified Java Script Value step offers an extension to the plain JavaScript language called *ECMAScript for XML*, more commonly referred to as *E4X*.

The E4X standard is maintained by the ECMA and can be downloaded from `http://www.ecma-international.org/publications/standards/Ecma-357.htm`**.**

HTML

HTML is an acronym for Hypertext Markup Language. Like XML, HTML has its roots in SGML and, as such, it uses the same basic syntax constructs as XML: elements, attributes, and text. However, the similarities stop there.

Whereas XML is a general-purpose data exchange format, HTML was designed with the express purpose of creating documents that can be rendered with a web browser and read by a human audience. For that purpose, HTML defines a limited, fixed set of elements and attributes that serve either to structure text documents (using items such as headings, paragraphs, and sections) or style text content (by specifying characteristics such as font and colors).

Even though HTML was not designed for general data exchange, a lot of interesting data is available in HTML format. Although more and more organizations are offering proper web services to make some of their data accessible, many still don't or have no intention of doing so. In those cases, scraping pages from the website may be the only option.

Unfortunately, Kettle does not provide any specific features for extracting data from HTML. However, you can use Kettle to download web pages and use tools such as JavaScript, Formulas, and Java user-defined expressions and classes, to extract data using general string manipulation features.

JavaScript Object Notation

JavaScript Object Notation (JSON, pronounced Jason) is an emerging standard for data exchange that is increasingly favored over XML for communicating with web services. The JSON format was originally specified by Douglas Crockford in `http://www.ietf.org/rfc/rfc4627.txt`. Alternatively, see `http://www.json.org/` for a more accessible formulation of the format.

A detailed comparison between XML and JSON and their respective benefits and disadvantages is outside the scope of this book. However, a useful distinction can be made by characterizing XML as a document language and JSON as an object-serialization format. Although both are easily machine-readable, JSON is more easily mapped to data type systems of popular programming languages.

Syntax

Syntactically, JSON is a proper subset of the JavaScript language. This means that JSON data structures can be accessed and traversed natively from within JavaScript. The following is an example JSON representation of a collection of books:

```
[
    {
        "title": "Pentaho Solutions",
        "publisher": "Wiley"
        "isbn": "978-0-470-48432-6",
        "price": "50.00 USD",
        "pages": 658,
        "inPrint": true,
        "authors": [
            {
                "firstName": "Roland",
                "lastName": "Bouman"
            },
            {
                "firstName": "Jos",
                "lastName": "van Dongen"
            }
        ]
    },
    {
        "title": "Pentaho Kettle Solutions",
        "publisher": "Wiley"
        "isbn": "978-0-470-63517-9",
        "price": "50.00 USD",
        "pages": 744,
        "inPrint": false,
        "authors": [
            {
                "firstName": "Matt",
                "lastName": "Casters"
            },
            {
                "firstName": "Roland",
                "lastName": "Bouman"
            },
            {
                "firstName": "Jos",
                "lastName": "van Dongen"
            }
        ]
    }
]
```

The example illustrates the key features of JSON:

- The outermost structure is an *array* of multiple books. Arrays are denoted using a matching pair of square brackets: `[` and `]`. Array entries are separated using a comma.
- Books are denoted as *object literals*. Object literals are denoted using a matching pair of curly braces: `{` and `}`.
- *Object members* (properties) appear between the curly braces of object literals. A member is denoted as a key/value pair, which are separated from one another by a colon. Multiple key/value pairs are separated from one another using a comma.
- In an object member, the key appears before the colon and must be enclosed in double quotes. The value appears after the colon.
- The member value can be any valid JSON value. There are only five types of JSON values: string, number, object, array, or Boolean.
- A string literal is denoted between double quotes. In the example listing, the value of the `title` member of the first book is the literal string value `Pentaho Solutions`.
- A number can be denoted as a simple integer or floating point number, or using exponent notation. In the example listing, the `pages` member of the second book is the integer number `744`.
- A Boolean value has one of the literals `true` or `false` as a value. In the example listing, the `inPrint` member of the book objects has a Boolean type.

NOTE Despite its ties with the JavaScript programming language, JSON enjoys a wide uptake in other programming environments as well, and free/ open source JSON implementations are available for many languages, including C/C++, C#.NET, Java, and many more. You can find links to JSON libraries for many programming languages on the `json.org` **site.**

It is often claimed that JSON is much simpler than XML. If the size of the specification is any indication, this is certainly true. Whereas the XML syntax is defined in 89 grammar rules, JSON is defined in only 15. The essence of JSON can be summarized in just 6 grammar rules. The remaining 9 rules have to do with the lexical definition of strings (3 rules) and numbers (6 rules).

JSON, Kettle, and ETL/DI

Currently, Kettle does not offer any features for consuming or generating JSON. However, because JSON is JavaScript, you can at least use Kettle's Modified Java Script Value step to access JSON data programmatically. We discuss this method in detail in the "JSON Example" section later in this chapter, where we extracts data from the freebase database. Another possibility to work with JSON is to write a plugin yourself.

Apart from the initial method for extracting and accessing data, transformations for working with JSON face many of the problems encountered when working with XML. Both formats are often used for marshalling objects, and a good deal of the transformation logic has to do with disentangling the nested data structures that are used to represent these objects and the relationships between them.

XML Examples

In this section, we take a closer look at Kettle features that enable you to work with XML. It is impossible to provide a generic example because XML is an extremely flexible format and can be used to express a large variety of high-level data structures. However, at a lower level, XML consists of just a few fundamental constructs, and all higher level structures are built in one way or another based on these fundamental types.

For this section, we will focus on one particular XML document containing video and actor data. In one example, we describe how to extract data from this XML document, and store it in the sakila sample database introduced in Chapter 4. In addition, we describe a transformation that works the other way around, which exports data from the sakila sample database to an XML document.

Example XML Document

The `videos.xml` document is provided as a free sample with Stylus Studio, an XML development environment. You can download the file from `http://www.stylusstudio.com/examples/videos.xml`. In addition to the XML document itself, there's also an XML Schema file that describes the structure of the XML document. This can be downloaded from `http://www.stylusstudio.com/examples/videos.xsd`.

XML Document Structure

The following snippet illustrates the structure of the `videos.xml` document:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>

  <actors>
    <actor id="00000015">Anderson, Jeff</actor>
    ...more actor elements here...
    <actor id="916503210">Sharif, Omar</actor>
  </actors>

  <videos>
    <video id="id1235AA0">
        <title>The Fugitive</title>
        <genre>action</genre>
        <rating>PG-13</rating>
        <summary>...text..</summary>
```

```
        <details>...more detailed text...</details>
        <year>1997</year>
        <director>Andrew Davis</director>
        <studio>Warner</studio>
        <user_rating>4</user_rating>
        <runtime>110</runtime>
        <actorRef>00000003</actorRef>
        <actorRef>00000006</actorRef>
        <vhs>13.99</vhs>
        <vhs_stock>206</vhs_stock>
        <dvd>14.99</dvd>
        <dvd_stock>125</dvd_stock>
        <beta>1.03</beta>
        <beta_stock>12</beta_stock>
        <LaserDisk>12.00</LaserDisk>
        <LaserDisk_stock>10</LaserDisk_stock>
    </video>
    ...more video elements here...
    <video>
        ...
    </video>
  </videos>

</result>
```

The `<result>` document element contains an `<actors>` and a `<videos>` element containing a collection of `<actor>` and `<video>` elements, respectively.

The `<actor>` elements have an `id` attribute that uniquely identifies actor elements throughout the document and contain a piece of text consisting of the actor's last name, followed by a comma and then the actor's first name.

Each `<video>` element contains a collection of elements that describe some property or characteristic of the containing `<video>` element, such as `<title>`, `<genre>`, and `<rating>`.

A `<video>` element also contains one or more `<actorRef>` elements, each of which contains the value of the `id` attribute of one of the `<actor>` elements, thereby conveying the fact that the corresponding actor appears in that particular video.

Mapping to the Sakila Sample Database

For this example, we assume a fairly straightforward mapping between the database tables and the elements in the `videos.xml` file:

- The `<actors>` element corresponds to the `actor` table, and each `<actor>` element corresponds to a row in the `actor` table.
- The `<videos>` element corresponds to the `film` table, and each `<video>` element corresponds to a row in the `film` table.

- The `<genre>` element inside a `<video>` element corresponds to a row in the `film_category` table.
- An `<actorRef>` element inside a `<video>` element corresponds to a row in the `film_actor` table.

The mapping between the XML structure and the sakila database is illustrated in Figure 21-1.

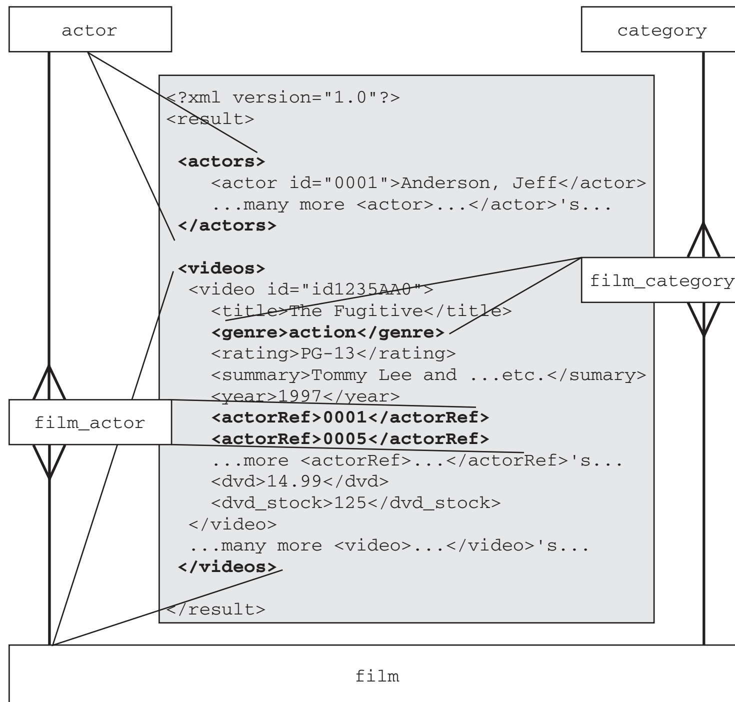


Figure 21-1: The mapping between the videos.xml file and the sakila sample database

Extracting Data from XML

To load the data from the `videos.xml` file and store it in the sakila sample database, we prepared the `import_xml_into_db` transformation. The transformation is shown in Figure 21-2.

NOTE **The transformation file** `import_xml_into_db.ktr` **is available on the book's website (**`www.wiley.com/go/kettlesolutions`**) in the folder for this chapter. Successfully running the transformation assumes an existing setup of the sakila sample database and sakila user account. You can find instructions for this setup in Chapter 4.**

The transformation also relies on a local copy of the `videos.xsd` XML Schema, and this is expected to be in the same directory as the transformation. You can download this file from the Stylus Studio website mentioned earlier.

Finally, the transformation will attempt to download the `videos.xml` XML documents from the Stylus studio website, so a working connection to the Internet is required.

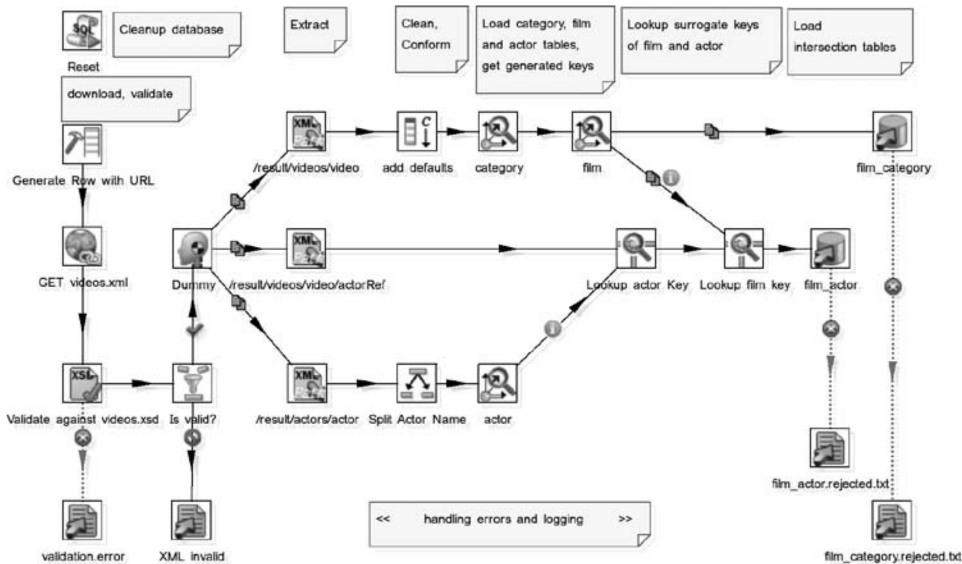


Figure 21-2: The import_xml_into_db transformation

Overall Design: The import_xml_into_db Transformation

Before we dig into the gritty details of the `import_xml_into_db` transformation, it is useful to consider the high-level data flow. At the top of Figure 21-2, you can see the notes that explain what is going on in the steps immediately beneath the notes.

The first step to execute will be the "Execute SQL script" step labeled Reset. In Figure 21-2, this step is located at the top-left corner. This step runs in the initialization phase and does not take part in the actual transformation. Its purpose is to remove any data added to the Sakila database on top of the standard data set. This is just a precaution to start with a clean slate before adding new data with this transformation.

After the initialization phase, the transformation proper starts to run by generating a row with the Generate Rows step labeled Generate Row with URLs, which appears at the top-left side of the transformation shown in Figure 21-2. The generated row defines a constant string field that specifies the URL to retrieve the `videos.xml` file from the Stylus Studio website.

This generated row is first fed into the "HTTP client" step labeled "GET videos.xml" to actually retrieve the `videos.xml` file. The content of the `videos.xml` file is then

added to stream in its own field. Configuration of the "HTTP client" step is discussed in detail later in this chapter.

After downloading the XML document, it is validated against an XML schema by the "Validate against videos.xsd" step to ensure it meets the expectations of the transformation. This validation step is the first real XML-related step, and is covered later in this chapter.

After validation, the data is extracted with three "Get data from XML" steps running in parallel. The configuration of this type of step is discussed in detail later in this chapter, but for now, let's focus only on the data flow. The data in the XML document will lead to inserts into all tables shown in Figure 21-2. The transformation needs to maintain data integrity, and therefore it must load the tables in the right order: rows in the `film_category` and `film_actor` tables reference the rows in the `category`, `film`, and `actor` tables, so we must ensure the referenced rows are present before loading the referencing rows.

An additional complication is that the sakila database uses surrogate keys, which are automatically generated in the database. The XML document also uses surrogate keys to identify videos and actors that are used to establish the relationship between videos and actors via the `actorRef` elements that appear inside the `video` elements. You can treat the key from the XML document as a business key (see Chapter 8 for a definition of the concept of a business key).

In Chapter 8, we mentioned that in data warehouses, business keys can be stored in dimension tables to allow a database lookup of the dimension table's surrogate key based on the business key. However, the sakila database was not designed to be loaded from XML documents like this, and does not have columns for storing the business keys from the XML document. This means that you cannot simply first load the `film`, `actor`, and `category` tables in their entirety before loading the `film_actor` and `film_category` tables. If you do that, you have no way of looking up the surrogate keys of the referenced tables when loading the referencing tables.

To correctly handle these issues, the transformation uses "Combination lookup / update" steps for loading the `actor`, `category`, and `film` tables. In addition to adding the rows to the appropriate tables, these steps also add the database-generated keys to the stream. Because the `video` elements contain a single `genre` element, `film` and `category` are loaded in the same flow, and this means the stream right after these steps already contains both a `film_id` and a `category_id`, which can then be added to the `film_category` table using a simple "Table output" step. (This is in the top flow in the `import_xml_into_db` transformation.)

For `film_actor`, the situation is less straightforward. (This is the middle flow of the transformation.) The `film_actor` table is loaded from the extracted `actorRef` elements, and the extract contains the business keys of the videos and its actors. The corresponding surrogate keys for the `film_id` and `actor_id` keys in the database are looked up from the film stream (top flow) and actor stream (bottom flow) using two "Stream lookup" steps, after which the `film_actor` table is loaded using a simple "Table output" step.

Using the XSD Validator Step

If you have access to the XML Schema definition, it is usually a good idea to use it to check XML documents before extracting data. The risk of omitting validation is that the remainder of the transformation could extract bits and pieces of data from it, and even process that data, store it in a database, and so on, only to find out later on that the XML document actually didn't have the expected structure and/or content. If you have access to the XML Schema, you have the fortunate opportunity to build your transformation with exact prior knowledge of what to expect from the XML document, and in this case a simple check for validity allows you to prevent a lot of potential problems up front.

In Kettle transformations, XML Schema validation is done using the XSD Validator step (labeled "Validate against videos.xsd" in Figure 21-3).

NOTE **You can also use XSD validation in Kettle jobs using a similar step—there is no generic way to determine which one is more appropriate.**

The XSD Validator step resides beneath the Validation folder in the left side pane tree view. The configuration for the "Validate against videos.xsd" step is shown in Figure 21-3.

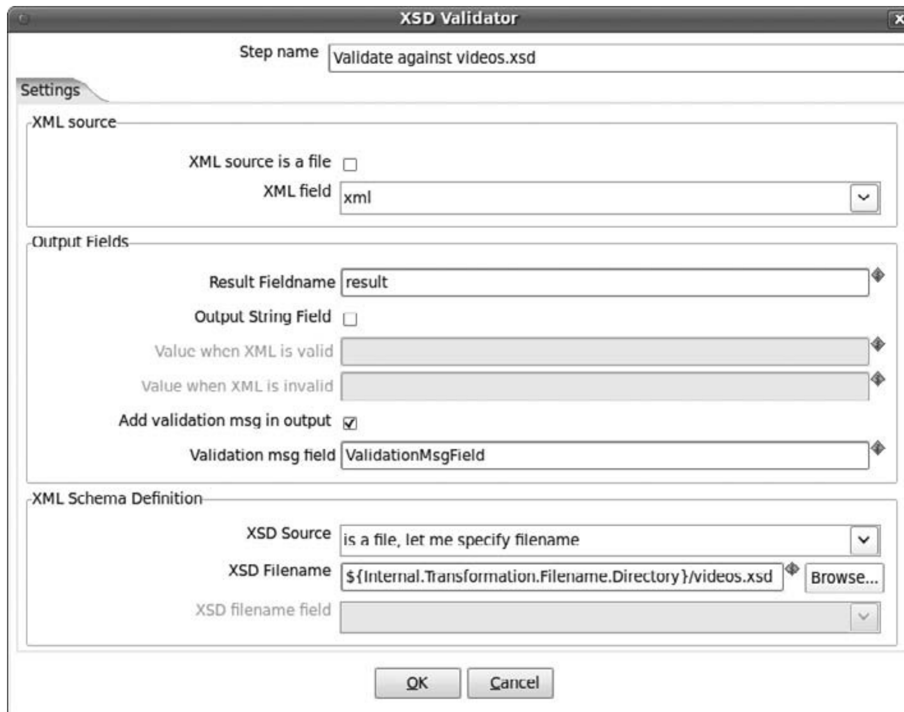


Figure 21-3: Validating the videos.xml document against the videos.xsd XML Schema

The configuration dialog contains three sections, which we discuss in the paragraphs that follow.

XML Source

The first section, “XML source,” is used to specify the properties of the XML document that is to be validated. The XML document is specified through a field in the incoming stream, which can be specified by choosing the appropriate field with the “XML field” combo box. If the “XML source is a file” checkbox is checked, the value of the specified field will be interpreted as a VFS file name.

In Figure 21-3, the checkbox is unchecked because the text that makes up the XML document is present as a string, which was added to the stream by the “Get videos .xml” step.

Output Fields

In the second section, Output Fields, you can control features of the validation result. The result of validating the XML document is always added as a field to the outgoing stream, and you can use the Result Fieldname property to specify the name of that field. By default, the field is called `result` and has a Boolean data type, which will take on one of the Boolean truth values (in Kettle, `Y` and `N`) to indicate whether the XML document is valid or invalid, respectively.

If you like, you can return the validation result as a custom string value. In this case, you should check the Output String Field checkbox and provide a custom string value in the “Value when XML is valid” and “Value when XML is invalid” properties.

In addition to the field for the result, you can also obtain any messages describing the validation result. (This is especially useful when validation fails because it allows you to troubleshoot the problem with the XML document.) Check the “Add validation msg in output” checkbox if you want to obtain the messages and use the “Validate msg field” property to specify the name of the field that should be added to the outgoing stream to convey the validation messages.

XML Schema Definition

The final section, XML Schema Definition, is used to specify the document that contains the XML Schema definition to validate against. Use the XSD Source combo box to specify the origin of the XML Schema. You can choose either one of the following options:

- **Is a file, let me specify filename:** This option means that the XML Schema is contained in a file specified by the XSD Filename property.
- **Is a file, filename is defined in a field:** This allows you to specify the file name through the value of a field in the incoming stream. If you use this option, you should pick the field using the “XSD Filename field” combo box.
- **Is defined inside XML:** In this case, the XSD Validator step expects that the document element of the XML document itself has an `xsi:schemaLocation` attribute that contains the URI for the XML Schema definition document.

In Figure 21-3, we used the first option, and used a reference to the built-in `${Internal.Transformation.Filename.Directory}` variable to point to the `videos.xsd` file located in the same directory as the transformation file.

NOTE **Although it would have been possible to dynamically retrieve the `videos.xsd` file from the Stylus Studio website, just like the `videos.xml` document, that would not be such a good idea.**

We used the schema as a guideline to construct our transformation. If, for some reason, someone decided to change the XML schema (and the `videos.xml` document), then the validation could still check out positive, even though the contents of the `videos.xml` document could be entirely different from what our transformation expects.

If you're going to use XML Schema validation, be sure to use it in such a way that validation actually fails if the XML document has different contents from what you expect.

Error Handling

The XSD Validator step supports error handling. In Figure 21-2, you can see that any errors that occur when validating the document are logged to the "validation.error" step, which is of the "Text output file" type and located beneath the XSD Validator step. Error handling would take effect if there is a problem in validating the document—for example, the XML Schema definition itself could be malformed.

Don't confuse error handling with failure to validate the XML document. It is not an error if the document cannot be validated because it contains content or data structures that are not compatible with the XML Schema definition. Rather, the validation procedure is successful; it's just that the result is negative because the document is invalid. We discuss this situation in the next subsection.

Checking the Validation Result

Whenever you use the XSD Validator step, it is essential to check its result. The XSD Validator step passes on both valid and invalid documents in its outgoing stream, along with the validation result, and, if you configured the step to do so, any validation messages. It is up to the ETL designer to act upon the validation result and do something meaningful with it.

In the `import_xml_into_db` transformation, the validation result is checked using the "Filter rows" step labeled "Is valid?" This step simply checks the value of the `result` field added by the "Validate against videos.xsd" step, passing on those rows for which the result is true, and storing the invalid rows to file using the "Text output" step labeled "XML invalid."

Using the "Get Data from XML" Step

After validating the incoming XML document, you can finally start with the actual data extraction. In Kettle this is done using the "Get data from XML" step, which resides

beneath the Input category in the left side pane tree view. The `import_xml_into_db` transformation contains three steps of the "Get data from XML" type:

- `/result/videos/video`: This extracts the actual elements that represent a video from the XML document, and each of these will eventually end up as a row in the `film` table in the Sakila database.
- `/result/videos/video/actorRef`: Each `video` element contains a variable number of `actorRef` elements indicating which actors play a part in that particular movie. These will eventually be stored in the `film_actor` table.
- `/result/actors/actor`: This extracts the `actor` elements, which will result in rows for the `actor` table.

These steps all get handed the same XML document retrieved by the "HTTP client" step, reading it in parallel. We could discuss any one of these instances of the "Get data from XML" step, but both for simplicity and completeness (as will become apparent later on in this section) we settle for the middle one labeled `/result/videos/video/actorRef`.

The configuration dialog for the "Get data from XML" step has three tabs. We will discuss these in detail in the remainder of this section.

The File Tab

In the File tab (see Figure 21-4), you can define the source of your XML document.

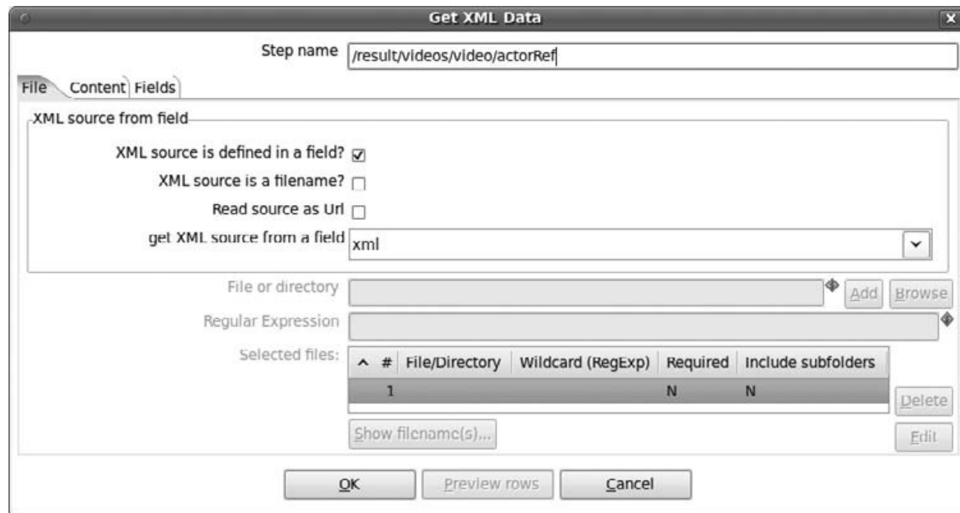


Figure 21-4: The File tab of the "Get data from XML" step

The source of the XML document can be specified either using a field in the incoming stream or using the standard file and directory interface similar to that used by the various file input steps. To use a field, check the "XML source is defined in a field"

checkbox in the top of the “XML source from field” fieldset located in the top half of the File tab of the configuration dialog. Clear the checkbox to specify one or more directories and regular expressions for finding file names in the lower half of the configuration dialog.

NOTE **When developing your transformation, it is usually a good idea to clear the “XML source is defined in a field” checkbox and use the file and directory interface to work with a local copy of a representative instance of the XML documents you plan to work with. Using a file greatly simplifies configuring other properties of this step, because in this case, Kettle can always locate the XML document, parse it, and offer suggestions for XPath selectors. When a field is specified as source, the contents of the field are known only at runtime when the transformation is being executed, not at design time when you’re configuring the step.**

You can always switch back and forth between using a field and a file.

If the “XML source is defined in a field?” checkbox is checked in order to use a field, you must specify the field name using the “get XML source from a field” combo box, which is the last item in the “XML source from field” fieldset. Finally, you need to specify how Kettle should interpret the contents of the field:

- Checking the “XML source is a filename?” checkbox specifies that the value of the field specified by the “get XML source from a field” property is a file name.
- Checking the “Read source as Url” checkbox specifies that the value of the field specified by the “get XML source from a field” property is a URL rather than a filename.
- If neither the “XML source is a filename” nor the “Read source as Url” checkbox is checked, Kettle expects the field to contain the XML document as text.

In the `import_xml_into_db` transformation, we used the last option because this allows us to perform only a single HTTP request (which is slow) to retrieve the XML document, passing it on to be read in parallel by the three “Get data from XML” steps. Any of the other configuration options for the XML source would have required another HTTP request.

The Content Tab

In the Content tab, you can specify how to extract rows from the XML document. The Content tab of the /result/videos/video/actorRef step from the `import_xml_into_db` transformation is shown in Figure 21-5.

The most important property on this tab is Loop XPath. Here, you can specify an XPath expression. This XPath expression will be applied to the XML document and evaluates to a node set, which is typically a collection of XML elements. The “Get data from XML” step will loop over the nodes in this set, and generate one row in the outgoing stream for each iteration. For this step, the Loop XPath property is configured to be `/result/videos/video/actorRef[text()]`.

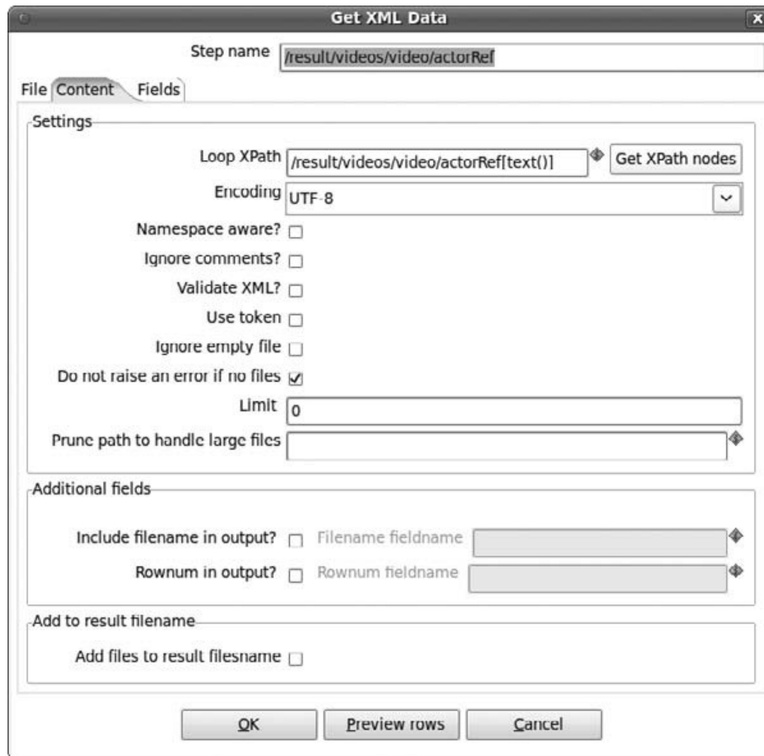


Figure 21-5: The Content tab of the "Get data from XML" step

If you specified a file as source in the File tab, you can use the "Get XPath nodes" button to help fill in the Look XPath property. This will scan the XML document and generate a list of XPath expressions for all levels of element node sets (see Figure 21-6).

Other options on the Content tab include:

- **The Encoding list box:** Can be used to explicitly set the character encoding used for the XML document. This option is useful in case the XML document does not specify its own encoding. Typically, XML documents start with an XML declaration that specifies the encoding. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- **Namespace aware?:** Check this if the document uses namespaces.
- **Ignore comments?:** Typically, XML comments count as nodes. If you want to ignore comments, check this option.
- **Validate XML?:** Check this if you want this step to use DTD validation prior to extracting data.
- **Use tokens:** This option applies to settings configured on the Fields tab, and we discuss it in the next subsection.

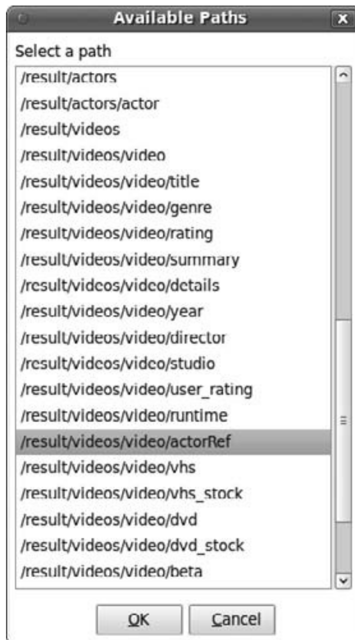


Figure 21-6: The picklist invoked by the Get XPath Nodes button

- **Ignore empty file:** Check this if you do not consider it an error if the file specified as XML source is empty.
- **Do not raise an error if no files:** Check this if you're using files as source and it is not an error if no files were found.
- **Limit:** Use this to limit the number of rows that are to be generated. The default is zero, which means one row should be generated for each node in the node set retrieved by the Loop XPath expression.
- **Prune Path to handle large files:** Typically, the XML document is read in one go, and the Look XPath expression is applied to the entire document. But if the XML document is very large, the resulting node set may be too large to fit into memory. In these cases, you can specify an XPath expression for this property that allows Kettle to apply the Loop XPath expression to chunks of the XML document. This property does not support the full XPath syntax: You can only specify element names separated by slashes. Predicates and namespaces are not allowed. In addition, the Prune Path must select chunks that are at the same or a higher level as the node set defined by the Look XPath expression.
- **Include filename in result and Filename fieldname:** If using a file as source for the XML document, you can check the checkbox to include the filename as a field in the outgoing stream and specify the name of the new field.

- **Rownum in output and Rownum fieldname:** Check the checkbox in case you want to add a field that contains a sequence number for the generated rows, and specify the name of the new field.
- **Add to result filename:** If using files as source, then the files will be added to the list of file results, which allows you to process these files in the parent job (if any).

The Fields Tab

The Fields tab (see Figure 21-7) is used to specify how Kettle should extract fields from the node set retrieved by the Loop XPath expression that you specified in the Content tab.

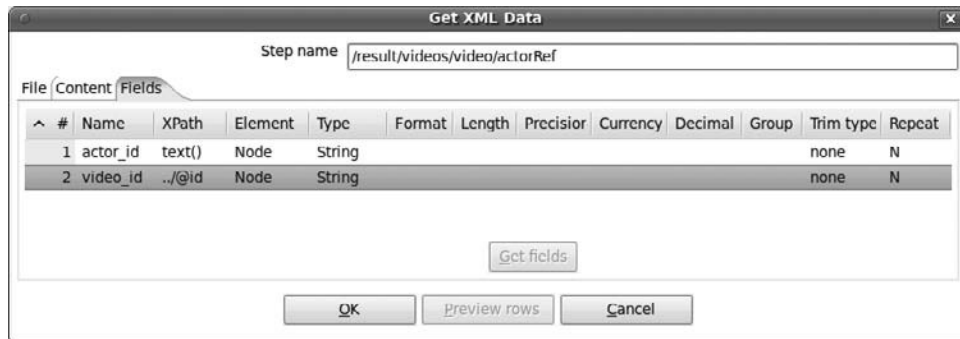


Figure 21-7: The Fields tab of the "Get data from XML" step

As you can see in Figure 21-7, fields are also defined using the XPath syntax. In the Name column of the grid, you specify the name of the field that is to be added to the outgoing stream. In the XPath column, you specify where the field gets its value from using an XPath expression. This XPath expression is executed relative to the current node from the node set retrieved by the Loop XPath expression.

For regular evaluation of the XPath expression, the `Element` column should be set to `Node` (as shown in Figure 21-7). You can set the `Element` column to `Attribute` instead, in which case the expression in the XPath column is evaluated differently to match only attributes. For instance, the second line in the grid shown in Figure 21-7 uses the XPath expression `../@id`, which means: get the value of the `id` attribute of the parent node of the current node. The same result could have been achieved by setting the `Element` column to `Attribute`, and writing `../id` instead. Note that this would be evaluated as regular XPath; it would mean: get the value of the `id` element beneath the parent node of the current node, which is something entirely different.

In Figure 21-7, the `actor_id` field gets its value from the text contents of the `actorRef` elements. The `video_id` element uses the expression `../@id` to look up from the current `actorRef` element to the `video` element that contains the current `actorRef` element, from which it extracts the value of the `id` attribute.

The outgoing stream now has two fields containing the values that are used internally in the XML document to identify the `video` and `actor` elements. These are used

downstream in the transformation to look up the corresponding database key values so they can be used to add new rows to the `film_actor` table.

Using Tokens

The XPath expressions in the Fields tab support a non-standard extension called *tokens*. Tokens are a way to bind values of fields to the XPath expression in order to parameterize it. This is best explained with an example.

Let's continue with our /result/videos/video/actorRef step from the `import_xml_into_db` transformation, and suppose that in addition to the `actor_id` and `video_id`, you would also like to retrieve the name of the actor. Once you extract the text contents of the `actorRef` element, you have the value of the `id` attribute of the `actor` element, so it would seem that you should be able to select the element to extract the name. Or could you?

Provided you have a specific value for the actor's `id`, you can certainly write an XPath expression that does the job. For example, if the actor `id` equals `1234`, then something like `/result/actors/actor[@id = `**`1234`**`]/text()` does the job. But the problem is that you don't have literally `1234`. Instead, all you have is the field `actor_id`, which gets its value from the `text()` expression that is evaluated in context of the current `actorRef` element.

You can't just substitute the `1234` in the previous expression with `text()` and expect it to work. The expression `/result/actors/actor[@id = `**`text()`**`]/text()` simply means: Get the text content inside the actor nodes for which the value of the `id` attribute equals the value of its text content. It doesn't work because the expression `text()` would be evaluated in the context of the `actor` element, not in the context of the `actorRef` element from which you extracted the value for the `actor_id` field. What you need is some mechanism to plug in the value of the `actor_id` field itself. This is exactly what tokens are meant for.

To use the `actor_id` field in the XPath expression for looking up the author's name, you have to write the expression like this: `/result/actors/actor[@id = `**`@_actor_id-`**`]/text()`. The expression `@_actor_id-` is the token, and it consists of an at sign (`@`), followed by an underscore (_), followed by the field name `actor_id`, and then followed by a dash (-). In addition, the "Use tokens" checkbox on the Content tab must be checked to use a token like this.

There are a number of limitations to using tokens:

- The field upon which the token is based (`actor_id` in the example) must appear before the field that uses it as token.
- In Kettle versions prior to 4.0, you can use only one token in a field. This limitation was lifted for Kettle 4.0.
- Token syntax is available only for XPath expressions in the Fields tab—not in the XPath expression in the Content tab.

NOTE The transformation file `export_xml_from_db.ktr` is available on the book's website in the folder for this chapter. Successfully running the transformation assumes an existing setup of the sakila sample database and sakila user account. You can find instructions for this setup in Chapter 4.

Generating XML Documents

To export data from the sakila database into a format like that of the `videos.xml` file discussed earlier, we prepared the transformation `export_xml_from_db` shown in Figure 21-8.

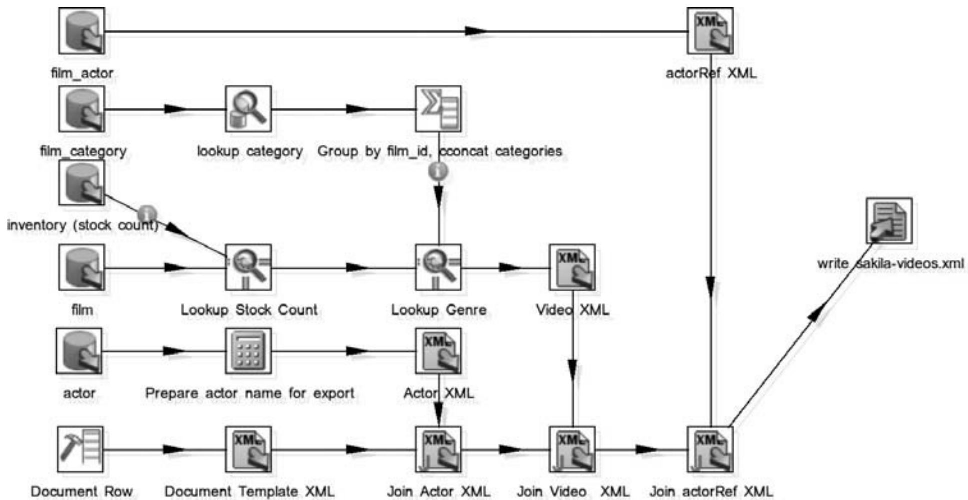


Figure 21-8: Generating XML

The transformation uses multiple instances of two particular types of Kettle steps that are especially designed for generating XML: the "Add XML" step and the "XML join" step. Before you look into the details of configuring these steps, let's first consider the main data flow of the `export_xml_from_db` transformation.

Overall Design: The export_xml_from_db Transformation

The best way to understand the `export_xml_from_db` transformation is to focus on the bottom stream in Figure 21-8. This is the main stream of the transformation that is responsible for XML document construction.

The bottom stream starts with a Generate Rows step, which generates one row having three empty string fields called `actors`, `videos`, and `video_template`. This row is sent to the step labeled "Document template XML," which is of the Add XML type. We discuss the exact configuration and operation of this step later in this chapter, but for now it is enough to know that this step adds a new string field to the stream containing an empty but complete XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <video_template/>
  <actors/>
  <videos/>
</result>
```

If you compare this listing to the one of the `videos.xml` provided in the beginning of this section, you will notice that this document already has the correct top-level structure; it is missing only the contents of the `video_template`, `actors`, and `videos` elements. The `export_xml_from_db` transformation actually does nothing to fill in the `video_template` element. Even in the original `videos.xml` file, this is an essentially static element that is not particularly interesting. Instead, we will focus on generating the XML content of the `actors` and `videos` elements.

All but the bottom streams of the transformation start with a "Table output" step, resulting in parallel extraction of data from the Sakila database. You will notice all tables mentioned in the mapping shown previously in Figure 21-1 are present, and in addition, an extra step to extract the stock count of each film.

For real-world transformations, one might not find it particularly desirable to have too many of these extractions going on in parallel, but we'll gloss over this detail for this particular sample. For now, it is enough to mention that the parallel extraction is of no consequence for XML document construction: Although the actual XML *generation* may occur in a parallel fashion, XML *document construction* is always a serialized operation because there is only one document at all times.

The streams coming from the "Table output" steps labeled film_category and "inventory (stock count)" are merged into the stream coming from the `film` table via some denormalizing operations such as look up (see the steps "Lookup stock count" and "Lookup genre") and aggregation (see the step "Group by film_id, concat categories"). The result is a stream that still has the granularity of the original film stream, but with additional fields from detail tables, which are then flattened to become attributes of individual films.

The original streams originating from the "Table output" steps eventually result in three main data streams, which are then led into the Add XML steps labeled Actor XML, Video XML, and "actorRef XML." This takes care of the actual XML generation, and results in a collection of the appropriate XML fragments. These streams are then led down to the bottom stream to participate in document construction. This is achieved using several "XML join" steps.

We discuss the configuration and operation of the "XML join" step in detail later in this chapter, but for now it is enough to note that this step merges the collection of generated fragments into the XML document coming in from the main stream. So, after the Join Actor XML step, the `actors` element from the skeleton document will be filled in and contain `actor` elements, and after the Join Video XML step, `video` elements will be added into the document's `videos` element. The final XML join step, "Join actorRef XML," concludes document construction by merging `actorRef` XML elements into their respective `video` element.

The last step of the transformation simply writes the resulting XML document to file using an ordinary File Output step. The result is one XML document containing all film-related data from the Sakila database.

Generating XML with the Add XML Step

The Add XML step is used for the actual generation of XML elements. It resides beneath the Transform category in the left side pane tree view. For each row in the incoming stream, the Add XML step emits one row to the outgoing stream, adding a new string

field that contains an XML element. Through the configuration dialog, you can control the name of the generated element, its attributes, and its content. The configuration dialog has two tabs: Content and Fields.

The Content Tab

In Figure 21-9, you can see the Content tab of the "Video XML" step from the `export_xml_from_db` transformation shown in Figure 21-8.

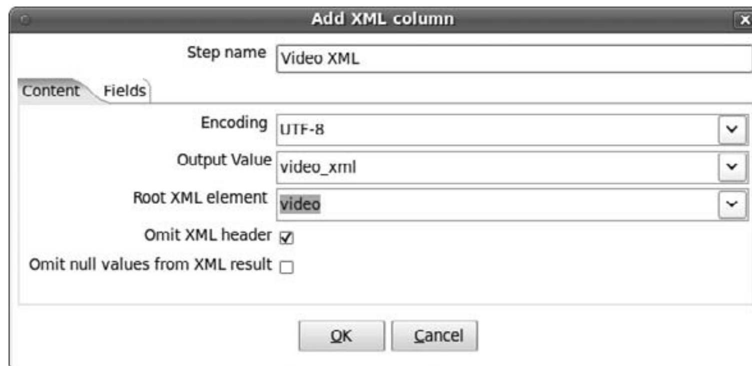


Figure 21-9: The Content tab of the Video XML step

The name "content page" is somewhat confusing, as you use it to control the properties of the generated XML element, not its contents.

You can use the Output Value property to configure the name for the field that contains the element. The "Root XML element" is where you specify the element name. Note that currently, the element name is a string constant—you cannot specify a field to set the element name dynamically. You can use the Encoding list box to choose a character encoding (default is UTF-8).

The "Omit XML declaration" checkbox can be checked to generate only the XML for the element. This is typically what you need for generating pieces of XML that are later merged into a document. For the outermost top-level element, you typically want to clear the checkbox to generate an XML document with the XML declaration. For example, in the `export_xml_from_db` transformation, this checkbox is unchecked in all steps of the Add XML type, except for Document Template XML, because this step generates the document element.

The "Omit null values from XML result" checkbox can be used to control the representation of `NULL` data. It is a bit confusing in the context of this tab because it exerts control over the content of the generated element, not the generated element itself. Therefore, this option is explained in more detail in the following section.

The Fields Tab

You can configure the Fields tab to control how the fields from the incoming stream are used to create the content and/or attributes of the generated elements. The Fields tab for the Video XML step is shown in Figure 21-10.

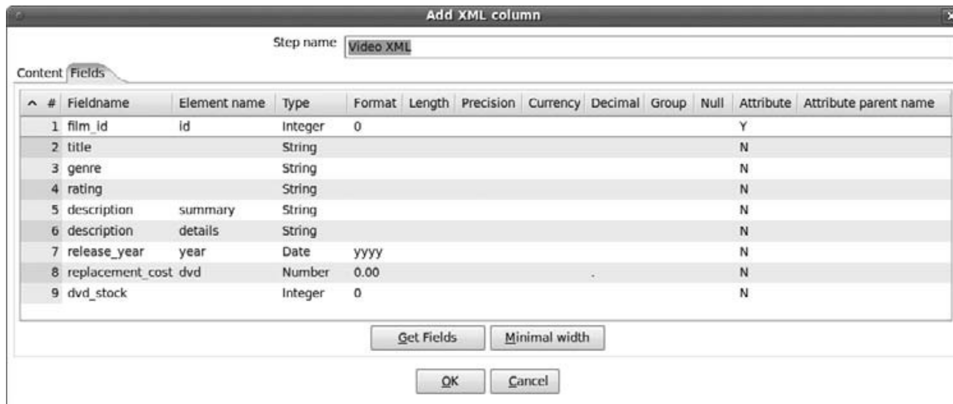


Figure 21-10: The Fields tab of the Video XML step

Each row entered into the grid represents a field from the incoming stream that will be used to generate XML. Use the "Element name" property to specify which fields you want to use. There are four different ways for the fields from the incoming stream to contribute to the generated XML:

- **Generate child elements of the "Root XML element" using the field value as text content.** You can use the "Element name" property to control the name of the generated elements. For example, in Figure 21-10, the description field is listed twice and will generate a child element called `summary` and another child element called `details`. If you omit the "Element name," then the field name will be used as element name. For example, the `title`, `genre`, and `rating` fields all end up as child elements with the name equal to the field name.
- **Generate attributes of the "Root XML element" using the field value as the attribute value.** For this setting, you must set the Attribute property to `Y`, and the Attribute Parent Name must be left blank. In this case, the "Element name" property will be used as the attribute name. If no "Element name" is specified, the "Field name" will be used as attribute name instead. For example, in Figure 21-10, the `film_id` field is used to generate an `id` attribute of the "Root XML element."
- **Use the field value as text content of the "Root XML element."** As far as the configuration is concerned, this is almost like the first option, except that you must use whatever name you specified as "Root XML element" for the "Element name" property in the field grid. Although the configuration change is small, the difference in effect is quite large: No child element is generated, and the value of the field is dumped as text inside the root element instead. To see an example of this configuration, take a look at the "actorRef XML" step. This lists only the `actor_id` field in the Fields tab, using `actorRef` as "Element name." The identifier `actorRef` is also used as the "Root XML element" in the content page, resulting in generation of elements such as `<actorRef>1234</actorRef>`, where `1234` is the value of the `actor_id` field.

- **Generate attributes of the child elements of the "Root XML element."** Configuring this option is almost like the second option (generating attributes of the "Root XML element"). The only difference is that in this case, you should enter the name of the element that is to receive the attribute in the "Attribute Parent name" property in the grid.

If the field has a `NULL` value, the default behavior is to generate an empty element or attribute value. You can also choose to omit such elements and attributes by checking the "Omit null values from XML result" checkbox on the Content tab.

The following is an example of an XML fragment generated by the Add Video XML step for one particular input row:

```
<video id="2">
  <title>ACE GOLDFINGER</title>
  <genre>Horror</genre>
  <rating>G</rating>
  <summary>
    An Astounding Epistle of a Database Administrator
    And a Explorer who must Find a Car in Ancient China
  </summary>
  <details>
    An Astounding Epistle of a Database Administrator
    And a Explorer who must Find a Car in Ancient China
  </details>
  <year>2006</year>
  <dvd>12.99</dvd>
  <dvd_stock>3</dvd_stock>
  <actorRef>19</actorRef>
  <actorRef>85</actorRef>
  <actorRef>90</actorRef>
  <actorRef>160</actorRef>
</video>
```

Using the XML Join Step

Although the Add XML step can be used to generate XML elements with element content, the level of nesting is limited to only one level. Another limitation is that you can only generate a fixed set of child elements, which must be known in advance. The XML Join step provides an answer to these limitations.

The XML Join step can be used to merge a collection of XML elements in a stream (the source) into another single XML fragment (the target). So although this step in fact joins two streams of XML in the sense of uniting two distinct things, one should not compare this to a database join operation. The best way to explain this is to look at an example.

Let's consider the Join Actor XML step from the `export_xml_from_db` transformation. The configuration for this step is shown in Figure 21-11.

Figure 21-11: The configuration of the Join Actor XML step

Target

In this case, the target is the empty template document, which enters the Join Actor XML step via the Document Template XML step in the bottom flow of the transformation. The target is configured by setting the "Target XML step" property to the name of the step from where the target stream originates, and the "Target XML field" is the field in the outgoing stream of that step that contains the XML fragment that will receive the XML elements from the source step.

Source

The source is the Actor XML step, which delivers a collection of `actor` XML elements (namely, one for each row in the incoming stream of the Actor XML step). This is configured by setting the "Source XML step." The "Source XML field" has to be set to the field in the source step's outgoing stream that contains the XML fragments that are to be merged into the target.

Join Condition

In the "Join condition properties" fieldset, you can control where in the target XML fragment the source is to be inserted. This is done by specifying an XPath expression in the "XPath statement" property. In Figure 21-11, the XPath expression is `/result/actors`, which means that the actor elements coming in from the source stream are to end up as content of the `actors` element beneath the `result` element in the target fragment.

Result Stream

In the "Result stream properties" fieldset, you can set the name of the field in the outgoing stream that is to contain the resulting target fragment. In addition, you can set things such as the Encoding, whether or not to omit the XML declaration, and how to handle NULL values. These properties are similar to the properties of the same name we discussed for the Add XML step.

Here's an example of what the target document looks like after the "Join actor XML" step:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <video_template/>
  <actors>
    <actor id="1">Guiness, Penelope</actor>
    ...many more actor elements...
    <actor id="884">Sharif, Omar</actor>
  </actors>
  <videos/>
</result>
```

Complex Join Conditions

For both the Join Actor XML and Join Video XML steps, the task of merging the source stream with the target document is quite simple: The XML elements coming in from the source stream are simply dumped in a single container element. For the "Join actorRef XML," it's not so simple.

The task of the "Join actorRef XML" step is to merge the `actorRef` elements coming in from the "Add actorRef XML" step into the target document. As we mentioned several times before, `actorRef` elements are child elements of the `video` elements to indicate which actors appear in which video. So the difference as compared to the other XML Join steps is that, in this case, there are multiple `video` elements in the target that are to receive `actorRef` elements, and we must be sure to add a particular `actorRef` element only to those `video` elements if the respective actor does, in fact, appear in that video. In other words, we need some way to correlate the source and target elements.

You can correlate source and target elements by using a so-called *complex join condition*. The "Join actorRef XML" step uses such a condition (see Figure 21-12).

To enable complex join conditions, the "Complex join?" checkbox has to be checked. The XPath expression used for the "Join actorRef XML" step uses a predicate with a `?` placeholder to seek out a specific `video` target element. In addition, the "Join comparison field" is set to the name of the `film_id` field from the source stream. For each row in the source stream, the placeholder is bound to the value of the "Join comparison field," and the XPath expression is evaluated to find the appropriate target element. This way, the `actorRef` elements are merged one at a time into whatever `video` element is indicated by the value of the `film_id` field in the source stream. Like the tokens supported by the "Get data from XML step," the `?` placeholder is not standard XPath.

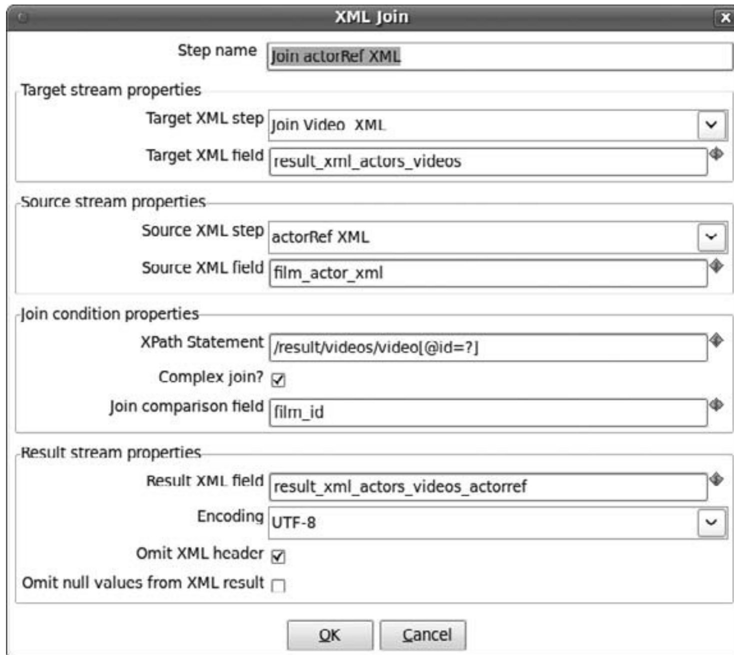


Figure 21-12: Correlating source and target elements using complex joins in the "Join actorRef XML" step

SOAP Examples

In this section, we take a closer look at using the "Web services lookup" step to call SOAP-based web services.

Using the "Web services lookup" Step

The step within Kettle that does all the SOAP magic is "Web services lookup." This step resides beneath the Lookup category in the left side pane tree view.

Unlike the "HTTP client" step, the "Web services lookup" step doesn't have to be kick-started by feeding it a row with the Generate Rows step (or any other step that can output at least one row to the next step). The "Web services lookup" step uses the Web Services Description Language (WSDL) to retrieve web services. See `http://en.wikipedia.org/wiki/Web_Services_Description_Language` for more information on WSDL.

Configuring the "Web services lookup" Step

Kettle ships with one working example of the "Web services lookup" step. This sample resides in the `samples/transformations` directory beneath the Kettle installation

directory and is called "Web services lookup - convert degrees Celsius to Fahrenheit." We will now discuss the configuration options of the "Web services lookup" step using this sample.

The Web Services Tab

The configuration dialog of the "Web services lookup" step always has one tab labeled "Web Services" (see Figure 21-13).

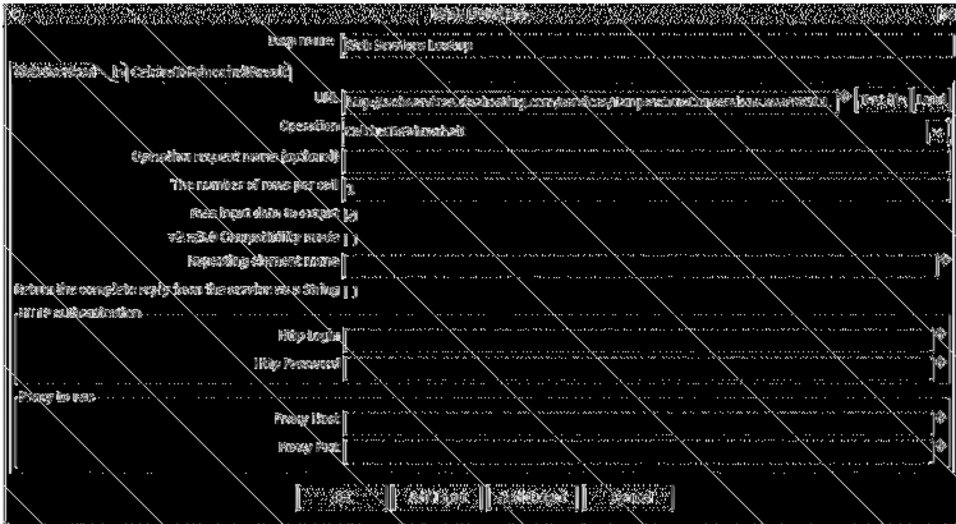


Figure 21-13: The Web Services configuration dialog

The most important item in this tab is the URL. Here, you must enter the URL that returns the WDSL document that describes this web service. If the web service requires authentication, you also need to enter your credentials in the Http Login and Http Password fields in the "Http Authentication" fieldset in the middle section of the configuration dialog. If you do not have direct access to the Internet and use a proxy instead, you should enter appropriate values in the Proxy Host and Proxy Port properties in the "Proxy to use" section at the bottom of the configuration dialog.

After entering the URL, press the Load button. This will perform a request for a WDSL description, and automatically populate the Operation combo box. In our example, the box will contain four entries: `CelciusToFahrenheit`, `FahrenheitToCelcius`, `WindChillInCelcius`, and `WindChillInFahrenheit`. These are the methods implemented by this web service, and you must choose which one you want to use.

After choosing a value or the operation, Kettle creates two more tabs: one tab to specify the parameters for the web service, and one tab page to control how to process the result returned by the web service.

The "in" Tab

The "in" tab is automatically added after you choose an operation that requires input parameters. It features a grid for entering the input parameters for the specified

operation of the web service. The "in" page of the temperature conversion example transformation is shown in Figure 21-14.

Figure 21-14: The "in" tab

The WS Name and WS Type columns of the grid are automatically populated with parameter names and types respectively after pressing the Load button. You still need to map the fields from the incoming stream to these parameters in the Name column.

The Result Tab

The results tab can be used to map the result of the web service call to the output stream. The configuration for the temperature conversion example is shown in Figure 21-15.

Figure 21-15: The results tab

Initially, the grid in the result tab is empty, but you can use the Fields button to populate it. You can use the Name column to map the fields from the response to the output stream.

Accessing SOAP Services Directly

WSDL leaves room for different implementations and dialects, of which only a part is supported by Kettle at the time of this writing. If the "Web services lookup" step doesn't work as expected, returns an empty set, or returns an error message, you can

check the "Return the complete reply from the service as a String" checkbox to receive the raw XML response from the service, and parse that using a "Get data from XML" step. Alternatively, you can try to access the service using an "HTTP client" or HTTP Post step and parse the response using a "Get data from XML" step.

A common use case for web services is to retrieve master data such as county codes and names. There are many publicly available web services that let you retrieve this data. One example is the collection of country info web services available at `http://www.oorsprong.org/websamples.countryinfo/CountryInfoService.wso`. The website lists a large collection of different operations that can be called, including `ListOfCountryNamesByName`, which we'll use in the following example.

TIP **An excellent way to test and play with web services outside Kettle is to use the open source tool soapUI. You can find it at** `http://www.soapui.org`**.**

The challenge with using web services in Kettle is that you need to find out what the output looks like in order to parse the resulting XML correctly. This is where a tool like soapUI might come in handy. It's also possible to use the preview options in Kettle for this. Just create a new transformation and add a Generate Rows step with the complete URL, as shown in Figure 21-16.

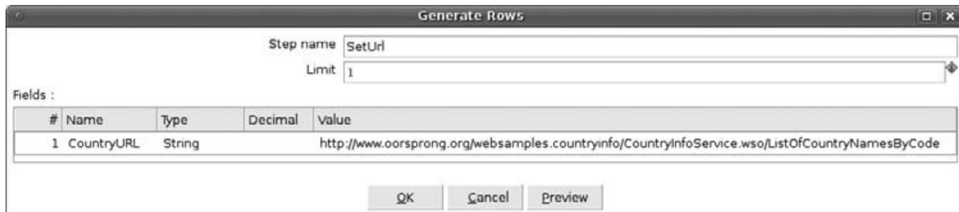


Figure 21-16: Generate Rows with URL

Then, add an "HTTP client" step to call the web service, get the data from this URL, and pass it on as an XML file, as shown in Figure 21-17.

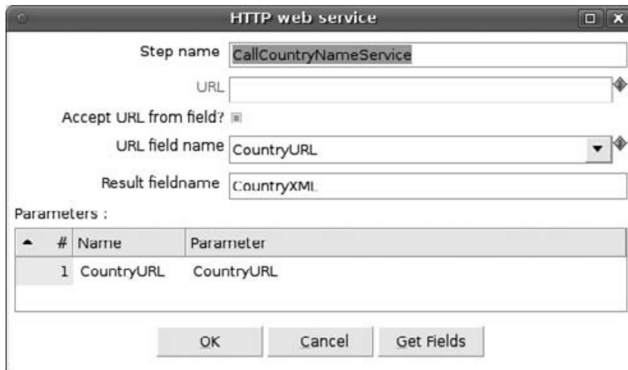


Figure 21-17: Call web service

NOTE The “HTTP client” step issues an HTTP GET request. Some SOAP web services may not be designed to respond to GET requests and require a POST request instead. In those cases, try using the HTTP Post step instead of the “HTTP client” step. The HTTP Post step also resides beneath the Lookup category in the left side bar tree view.

In order to enter the correct values in the next “Get data from XML” step, you first need to find out what the actual result string looks like. You can do this by issuing a preview on the “HTTP client” step. Figure 21-18 shows that the result is a single XML row with the root element name `ArrayOftCountryCodeAndName` and repeating element `tCountryCodeAndName`. The elements themselves are called `sISOCode` and `sName`.

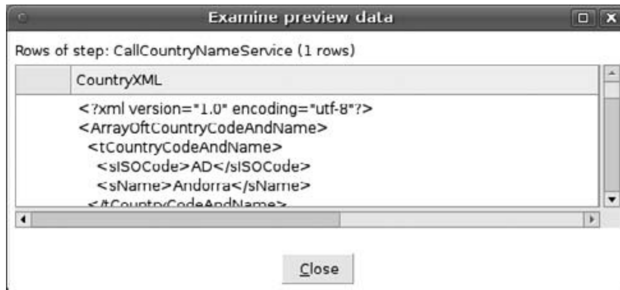


Figure 21-18: Examine preview data

This is all the information you need to correctly parse the XML. First you mark the first checkbox in the “Get data from XML” step (“XML source is defined in a field?”) and specify `CountryXML` as the source field. Then, in the Content tab of the step, the value of Loop XPath is set to the value `ArrayOftCountryCodeAndName/tCountryCodeAndName`. The final step is to list the XPath values of the elements we want to retrieve and the output column names we want to use for them, as shown in Figure 21-19.

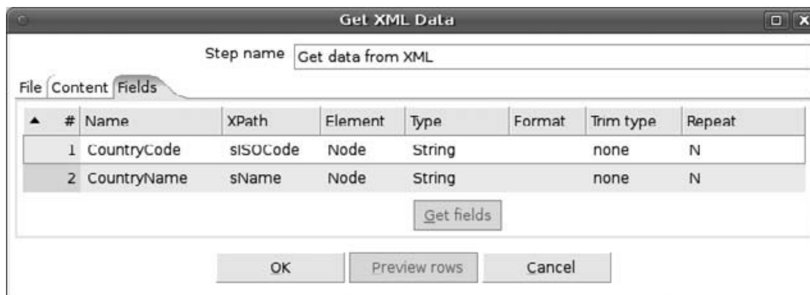


Figure 21-19: Get XML data

The final step in this example is “Select values” from the Transform category, which gets rid of the XML data, passing on only the ISO code and country names as stream

fields. The preview of this step, as displayed in Figure 21-20, shows that we've accomplished our goal of getting a list with country codes and names.

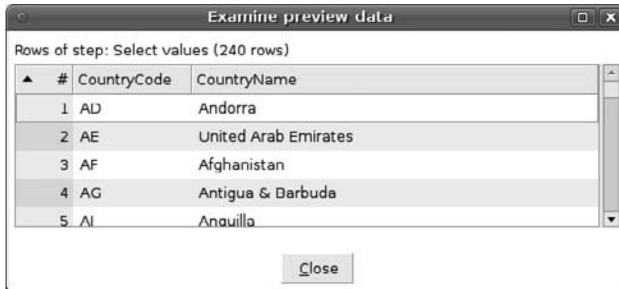


Figure 21-20: Country code and name preview

JSON Example

We already mentioned that Kettle does not have any special features for dealing with JSON but that you can use the Modified Java Script Value step to evaluate the JSON to a JavaScript data structure. What we didn't discuss yet is how to access the data within a JavaScript data structure and turn that data into rows and fields for the outgoing stream.

We will discuss in detail how to do this using an example to extract data from the Freebase open database project. In order to understand this example, it is necessary to provide a little background information about the Freebase project, its web services, and the way these web services work with JSON.

The Freebase Project

We will base our example on data and services offered by the Freebase project. The Freebase project is a large database containing information on all kinds of topics.

Freebase is sponsored by a company called Metaweb, which owns the (proprietary) database software on which freebase runs. The Freebase data, however, is accessible to everybody through a Creative Commons license.

NOTE **For more information about freebase, read the Freebase FAQ at:** `http://wiki.freebase.com/wiki/FAQ`.

For more information about Metaweb, see the Metaweb FAQ at `http://www.metaweb.com/faq`.

Freebase Versus Wikipedia

Freebase resembles Wikipedia in that it is an open and collaboratively collected and maintained encyclopedia. What distinguishes it from Wikipedia is that its data is quite

rigorously structured in entities and relationships and that these entities and relationships can be accessed and manipulated through a query language that is available through web services. Whereas Wikipedia is intended primarily as a human-readable knowledge base application, Freebase is essentially a knowledge base infrastructure, allowing developers to build all kinds of applications, including but not limited to Wikipedia-like encyclopedia applications.

To source its data, Freebase uses an open collaboration model with a moderation system similar to that of Wikipedia. In addition, many topics contained in Freebase are directly sourced from Wikipedia itself.

Freebase Web Services

The principal means to access and manipulate data in Freebase is to use web services. The web services offered by Freebase fall into the category of REST APIs. Although libraries are available for various programming languages, the API exposed by these libraries simply wraps around the web services, hiding the details of doing a HTTP request, receiving the response, and processing the resource.

NOTE **For an overview of the web services offered by Freebase, see** `http://www.freebase.com/docs/web_services`**.**

Freebase offers a number of web services for a number of operations, including authentication, reading and writing data, search and more. All services operate through HTTP GET and POST requests, and use JSON exclusively as a data format, both for specifying parameters in the request as well as encoding the data in the response.

To make it easy to integrate freebase into modern web applications, all web services support a `callback` parameter. Specifying the `callback` parameter causes the JSON response to be returned as an argument to a JavaScript function call, where the `callback` parameter specifies the name of the function. This technique is known as JSONP and is used predominantly in web browser applications to get an automatic notification once the service request returns a response. When evaluating the response as JavaScript, the callback function is called and passed the actual JSON response so it can be processed.

The Freebase Read Service

For our example, we will focus on the Freebase read web service, which is used to retrieve data from freebase. The read web service can be called with an HTTP GET or POST request using the following URL: `http://api.freebase.com/api/service/mqlread`. You can enter this URL directly in the browser address window, and you will receive the following response:

```
{
  "code": "/api/status/error",
  "messages": [
    {
      "code": "/api/status/error/input/invalid",
```

```
      "info": {
        "value": null
      },
      "message": "one of query=, or queries= must be provided"
    }
  ],
  "status": "400 Bad Request",
  "transaction_id":
    "cache;cache01.p01.sjc1:8101;2010-04-06T17:36:16Z;0050"
}
```

The response is a human-readable (well sort of, anyway) JSON object, and most people will recognize that the service is trying to explain that the request is not considered valid. This is because the read service requires a query that specifies exactly which data you want to retrieve.

The Metaweb Query Language

Freebase queries must be specified in the Metaweb Query Language (MQL) and passed to the service via a parameter called `query`. So the URL would become:

```
http://api.freebase.com/api/service/mqlread?query=...mql query here...
```

The MQL query itself is a small piece of JSON that provides a kind of query by example to describe what kind of data you want to retrieve. A detailed explanation of MQL is outside the scope of this book, but we can illustrate its essence with a simple query example. The following snippet is a valid MQL query to retrieve all film directors stored in freebase:

```
[{
  "type": "/film/director",
  "name": null
}]
```

As you can see, the query is a JSON array containing one object literal with two members, `type` and `name`. This object acts like an example or template for the object instances that you want to find.

To execute this query, you need to wrap it inside another JSON structure called a *query envelope* and append it to the freebase service URL like this:

```
http://api.freebase.com/api/service/mqlread?query=
{"query":[{"type":"/film/director","name":null}]}
```

In the snippet, the envelope and MQL query appear on a separate line. In reality, this should be entered as a single line. When you execute this in the browser, the result looks like this:

```
{
  "code": "/api/status/ok",
  "result": [
```

```
      {
        "name": "Blake Edwards",
        "type": "/film/director"
      },
      ...many more directors go here...
      {
        "name": "Andrew Stanton",
        "type": "/film/director"
      }
    ],
    "status": "200 OK",
    "transaction_id":
      "cache;cache03.p01.sjc1:8101;2010-04-06T18:51:11Z;0023"
}
```

NOTE By default, Freebase limits the number of returned results to 100 items for reasons of performance and scalability. It is possible to retrieve more results by issuing multiple requests. More information on this topic can be found in the section about retrieving large resultsets in the MQL Reference Guide at `http://www.freebase.com/docs/mql/ch04.html#cursors`**.**

As you can see, the response is also a JSON object, which is called the *result envelope*. The result envelope contains a few properties such as `code`, `status`, and `transaction_id`, which convey information about executing the query. You already encountered these properties when you first invoked the service.

Just as the MQL query was wrapped in the query envelope as the `query` member, the actual query result is contained in a `result` member of the result envelope. If you compare this result with the original MQL query, you should notice that the JSON structure of the query is mirrored in the result: Both the MQL query and its result have the form of a JavaScript array, and the result array entries have the same properties as the single object entry contained in the array specified by the query. This is shown in Figure 21-21.

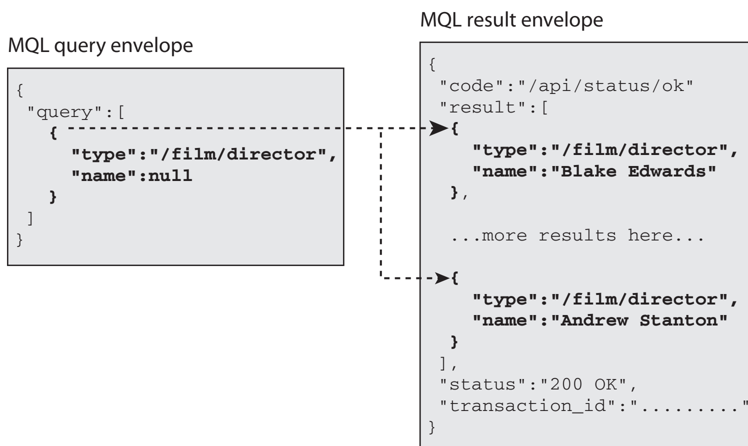


Figure 21-21: The MQL query and result envelopes

In the query, the `type` property denotes which kinds of entities to retrieve, and as such it is analogous to the `FROM` clause in an SQL query. This property is repeated as-is in the result entries. The query specified a `null` for the `name` property, and in MQL this means that the values for the `name` property should be filled in in the query result. This can be thought of as an expression in the `SELECT` list of an SQL query.

NOTE **Although it is possible to some extent to point out analogies between MQL and SQL queries, it's important to realize that Freebase is not a relational database. The analogies are very superficial and serve only to give you a quick idea how to get things done with MQL.**

Now that we mentioned a few analogies to the SQL language, you might wonder what MQL construct is analogous to the SQL `WHERE` clause. MQL does not offer condition-based filtering in the same way as SQL, but instead supports a kind of querying by example. To find instances having a particular property, you simply specify that property in the query. Executing the query then automatically filters for only those objects that resemble the example provided by the query while filling in the blanks (that is, the properties with `null` values).

The MQL basics we just discussed should be just enough to ensure that you can follow the actual Kettle example.

NOTE **There is much more to the MQL query language than the simple constructs we just discussed: MQL is a full-fledged database query language with features resembling joins and subqueries, comparison operators, aggregation, built-in history, and much more.**

For more information, visit the Freebase developer zone, in particular the MQL reference guide at `http://www.freebase.com/docs/mql`**. The developer zone also features useful applications for working with Freebase and MQL, such as the schema explorer, which is useful for discovering what types of entities are stored in freebase, which properties they have, and which relationships they have with other entities:** `http://schemas.freebaseapps.com/`**. For writing MQL queries, we highly recommend the MQL query editor (**`http://www.freebase.com/app/queryeditor/`**), which offers a much more user-friendly interface to writing MQL queries than the browser address bar.**

Extracting Freebase Data with Kettle

The transformation we prepared extracts films stored in the Freebase database and is simply called `freebase-films.ktr`. It can be obtained from the download area of this book's website, in the folder for Chapter 21. Figure 21-22 shows what the transformation looks like.

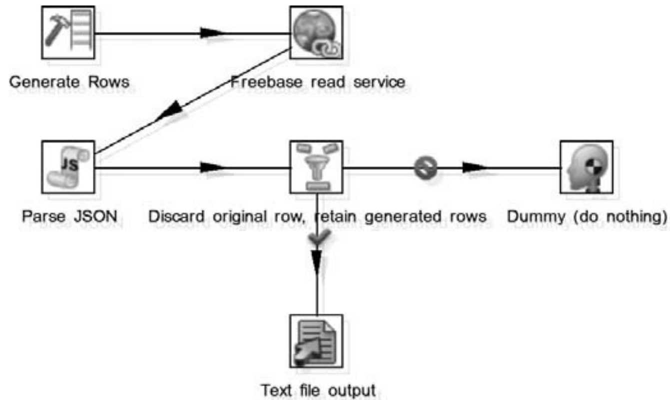


Figure 21-22: The freebase-films transformation

Generate Rows

The transformation works by first generating one row with the Generate Rows step. The configuration for the step is shown in Figure 21-23.

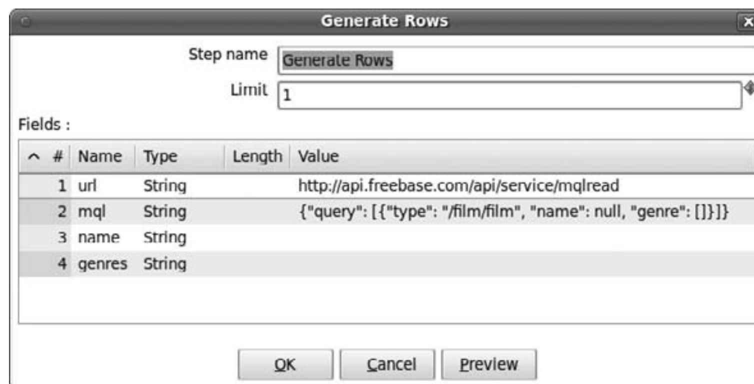


Figure 21-23: Generate Rows

This row contains a few constant string fields:

- `url`: The URL of the freebase read service:

```
http://api.freebase.com/api/service/mqlread
```

- `mql`: The MQL query envelope with a query to retrieve the freebase films, along with any genres:

```
{"query":[{"type":"/film/film","name":null,"genre":[]}]}
```

- `name` and `genre`: Both are empty but will be filled later on in the transformation after we have processed the result envelope.

Issuing a Freebase Read Request

The generated row is fed into the "HTTP client" step labeled "Freebase read service." The configuration for this step is shown in Figure 21-24.

Figure 21-24: Invoke the Freebase read service

The "HTTP client" step is configured to accept the URL from the `url` field from the incoming stream. The resource will be placed in the `result` field of the outgoing stream. To pass the MQL query to the Freebase read web service, the parameters grid maps the `mql` field from the incoming stream to a `query` parameter. This will result in responses like the following:

```
{
  "code": "/api/status/ok",
  "result": [
    {
      "genre": [
        "Black comedy",
        "Thriller",
        "Psychological thriller"
      ],
      "name": ".45",
      "type": "/film/film"
    },
...many more films and genres...
    {
      "genre": [
        "Disaster",
        "Thriller",
        "Drama",
        "Natural disaster"
      ],
      "name": "10.5",
      "type": "/film/film"
```

```
      }
    ],
    "status": "200 OK",
    "transaction_id":
      "cache;cache04.p01.sjc1:8101;2010-04-06T22:23:15Z;0017"
  }
```


Processing the Freebase Result Envelope

The response is then processed using a Modified Java Script Value step labeled “Parse JSON.” This step does not have any particular configuration, except for the script itself. The JavaScript code is as follows:

```
var o, r, i;

//Turn response envelope text into a JavaScript
//object and assign it to the local variable o.
eval("o = " + result);

//look through the array entries of the result
for (i=0, rows = o.result; i < rows.length; i++){

    //Get current result row
    r = rows[i];

    //Use putRow() method of the builtin _step_
    //object to push a row to the output stream
    _step_.putRow(
        rowMeta,   //use builtin row metadata
        [
            null,              //field: url
            null,              //field: mql
            r.name,            //field: name
            r.genre.join(","), //field: genre
            null               //field: result
        ]
    );
}
```

First, the script uses a call to the JavaScript built-in function `eval()`. This function takes its string argument and attempts to dynamically evaluate the text as JavaScript. The expression passed into the `eval()` function effectively assigns the JSON response contained as text in the `result` field of the incoming stream to the local variable `o`. So after executing the `eval()` function, you can access the result envelope as a JavaScript object through the `o` variable.

The second part of the script initializes a loop to iterate over the outermost array stored in the `result` member of the result envelope. Each entry in this array corresponds to one film retrieved from the Freebase read service.

Inside the loop, the `putRow()` method of the built-in `_step_` object is used to generate new rows for the outgoing stream. This `_step_` object is actually the instance of the Modified Java Script Value step itself, exposed as a JavaScript object. The `putRow()` method of this object maps to the underlying Java method that actually emits rows to the output stream.

The `putRow()` method takes two arguments: `rowMeta`, which describes the layout (metadata) of the generated row, and an array that contains the actual field values for the outgoing row. The `rowMeta` object is built-in, just like the `_step_` object, and automatically reflects the layout of the incoming rows. To save the hassle of changing the row layout, we added fields for the film `name` and `genre` instead up front in the Generate Rows step, which is why we had to define two empty fields there.

NOTE Always make sure the row metadata matches the metadata of the values that you assign to the row; otherwise the script won't work, or may lead to unexpected results.

As you can see, the new row sets all fields that were filled in the incoming stream to `null`. The only fields that are assigned a value are `name` and `genre` field. The value for the `name` field is taken immediately from the `name` property of the current entry of the query result.

For the `genre` field, things are slightly more complicated because the corresponding `genre` property in the query result can be an array of genre values. For this reason, the script uses the JavaScript built-in `join()` method of the array object. This method simply serializes the array to string by concatenating its entries, separating them with a comma. So in the outgoing stream, the genre field will contain a comma-separated list of genre values, which can be further processed downstream. This can, for example, be done using a "Split field to rows" step. This step is discussed in more detail in Chapter 20, in the section on multi-valued attributes.

Filtering Out the Original Row

Apart from outputting the rows you generate by calling the `putRow()` method, the Modified Java Script Value step also emits the original row coming in from the input stream. Because this row does not contain any data from the query result, it is useless to you, and you have to discard it. This is done using a simple Filter Rows step.

It is easy to spot the original row: Because the JavaScript code fills the `name` and `genre` fields, which you know are empty upstream of the Modified Java Script Value step, you can simply check if the name field `IS NOT NULL`. If it's not, then you're apparently dealing with a row having a film name, so you have to retain that. If, on the other hand, the `name` field is `NULL`, then this must be the original row, which is then discarded by leading it to the Dummy step.

Storing to File

The final step in the transformation is storing the data to file. Here's an example of the output:

```
mql;name;genres;result
;.45;Black comedy,Thriller,Psychological thriller;
...many more rows...
;13 Ghosts;Horror;
;10.5;Disaster,Thriller,Drama,Natural disaster;
```

As you can see, our transformation got rid of the nested JSON completely, transforming the data neatly into columns.

RSS

RSS stands for Really Simple Syndication and refers to a collection of XML-based data formats designed to exchange information about updates in some information source. It is in widespread use in blogs and news websites to allow visitors to subscribe to updates to the site—a usage commonly referred to as a *feed* (or XML-feed, RSS-feed or webfeed).

RSS Structure

RSS is a relatively simple XML vocabulary consisting of two fundamental elements: channel, and items. We discuss both in the remainder of this section.

Channel

At the top level, an RSS document is an `rss` element, with a mandatory `version` attribute that specifies the version of RSS that the document conforms to. In the following document, the version attribute is 2.0.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version = "2.0">
...
</rss>
```

The `rss` element has a single `channel` element, which contains information about the source that provides the feed and its contents. The channel represents the metdata of the feed.:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version = "2.0">
  <channel>
    <title>Press Releases</title>
    <link>
      http://www.pentaho.com/news/index.php?tab=news
```

```
      </link>
      <description>
        The latest Pentaho News
        and Press Releases from
        Pentaho.com.
      </description>
    </channel>
  </rss>
```

The following elements are required for `channel`:

- `title`: The name of the channel
- `link`: The URL of the website corresponding to the channel
- `description`: A phrase describing the channel

A channel can have optional elements (for example the language the channel is written in, copyright notice for content in channel, and so on) and some optional sub-elements.

Item

A channel may contain many `item` elements. Each `item` element represents a particular update or change to the site and typically contains a `description` element that serves as a summary of the update, and a `link` element that contains a URL pointing to the actual changed part of the site. For example, in the RSS feed of a news site, the `items` typically represent new articles, the `description` would provide a summary or first paragraph of the article, and the `link` would point to the actual article itself. All elements of an `item` element are optional; however, a `title` or `description` element should be present.

An `item` can have the following elements:

- `title`: The title of the item
- `description`: The item synopsis
- `link`: The URL of the item
- `pubDate`: Indicates when the item was published
- `comments`: The URL of a page for comments relating to the item
- `guid` (Globally Unique Identifier): A string that uniquely identifies the item

The following document is an example of channel with two items:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version = "2.0">
  <channel>
    <title>Press Releases</title>
    <link>
      http://www.pentaho.com/news/index.php?tab=news
    </link>
```

```
    <description>
      The latest Pentaho News
      and Press Releases from
      Pentaho.com.
    </description>
    <item>
      <title>
        Pentaho Releases Analyzer as First
        Deliverable in Agile BI Initiative
      </title>
      <link>
        http://www.pentaho.com/news/releases/20091104.php
     </link>
     <pubDate>Wed, 04 Nov 2009 9:00:00 EST</pubDate>
    </item>
    <item>
      <title>
        Pentaho Announces Strategic
        Technology Acquisition
      </title>
      <link>
        http://www.pentaho.com/news/releases/20091005.php
      </link>
      <pubDate>Mon, 05 Oct 2009 9:00:00 EST</pubDate>
    </item>
  </channel>
</rss>
```

You can extend feeds with geographic information. At this time, Pentaho Data Integration only supports points (longitude and latitude) on basic geometries (GeoRSS-Simple) and GeoRSS GML, which supports a great range of features. The following illustrates a Simple geographic item:

```
<georss:point>45.256 -71.92</georss:point>
```

And here's a GML item:

```
<georss:where>
  <gml:Point>
    <gml:pos>45.256 -71.92</gml:pos>
  </gml:Point>
</georss:where>
```

RSS Support in Kettle

Now that you know how RSS works, let's see how Kettle deals with syndication. Kettle can both read and write RSS feeds using the RSS Input and RSS Output steps. We examine these steps in more detail in the following sections.

RSS Input

The RSS Input step is available in the Input category from the left side pane tree view. It allows you to extract data from most RSS and Atom syndication formats, including:

- RSS 0.90
- RSS 0.91 Netscape
- RSS 0.91 Userland
- RSS 0.92
- RSS 0.93
- RSS 0.94
- RSS 1.0
- RSS 2.0
- Atom 0.3
- Atom 1.0

Here we present this step through an example that consists of reading all articles from Pentaho.com published since the first of January 2010. First, you need to specify the URL from which you read feeds in a URL list as shown in Figure 21-25.

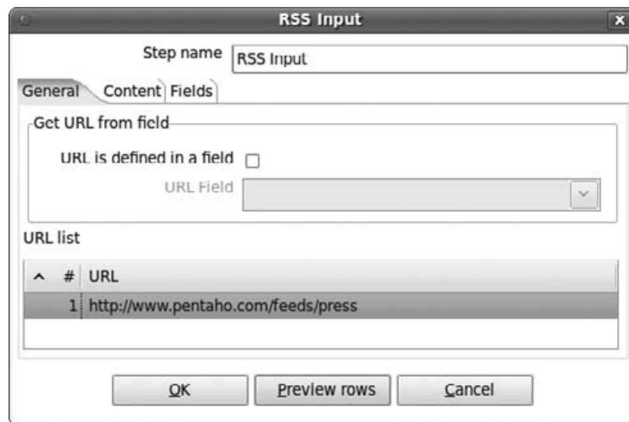


Figure 21-25: General tab of RSS Input

As you can see, the General tab specifies the URL list that you can enter in a static way or dynamically by using an environment variable or the field of an incoming stream (check "URL is defined in a field" and select the field in the "URL Field" box). Once you have the source URL, you can specify on the Content tab that you need only feeds published since the first of January 2010. This is shown in Figure 21-26.

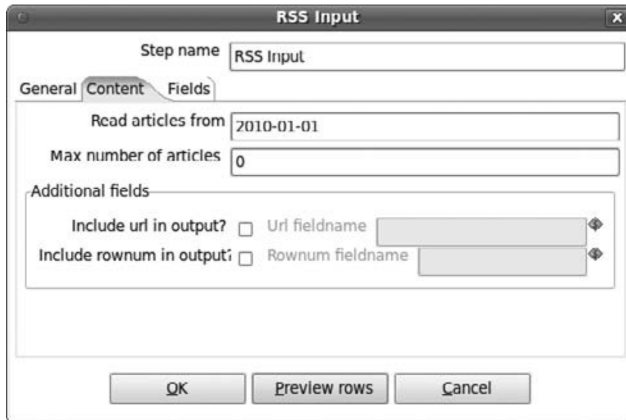


Figure 21-26: Content tab of RSS Input

The content tab has the following options:

- **Read articles from :** Enter the date from which you will retrieve articles. You must enter the date with the following format: `yyyy-MM-dd HH:mm:ss`.
- **Max number of articles:** Limit the number of articles to the top N (0 means extract all articles).
- **Include url in output:** If you have specified the URL in a static way, you can add it in the outgoing stream by checking this option. A field (string) named for the value you entered in "URL fieldname" will be added to outgoing stream.
- **Include rownum in output:** This option, available in many input steps, will add a field that contains a row number (first row will have number 1, etc.) in the outgoing stream. Simply select this value and enter a name for the field in the "Rownum fieldname" box.

The final step is to get the fields. On the Fields tab, shown in Figure 21-27, use the "Get fields" button to return all available fields. Simply remove and/or rename outgoing fields at your convenience.

You can check feeds using the "Preview rows" button on the General tab. As you can see, it is very easy to retrieve feeds from RSS thanks to Kettle, and you can then process your data using other steps.

RSS Output

The RSS Output step is the functional complement of the RSS Input step: it takes data from an input stream and writes it in RSS format to a file. These files can then be served through a web server to implement a feed for a website or web application. The RSS Output step lets you output a standard RSS feed (with elements explained previously), but you can also configure it to deliver a customized RSS feed.

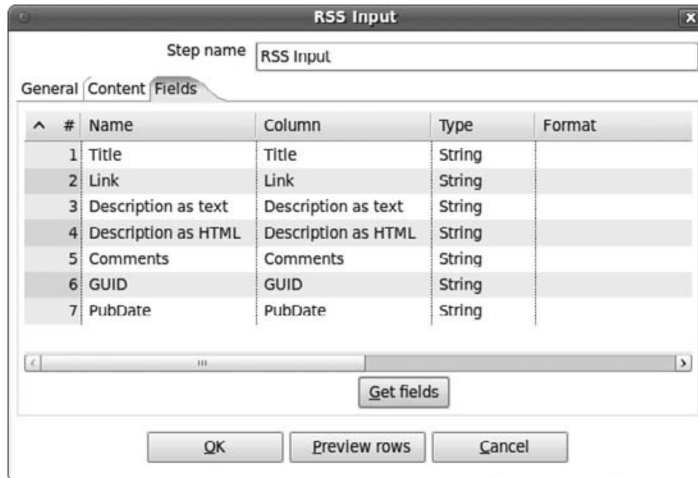


Figure 21-27: Select fields

The following example demonstrates a standard RSS feed with one item. The output RSS feed of this example is shown following:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>My Channel</title>
    <link>http://www.mychannel.com</link>
    <description>Example of RSS feed</description>
    <item>
      <title>Test item</title>
      <link>http://www.mychannel.com/feed0</link>
      <description>An item in the channel</description>
      <guid>http://www.mychannel.com/feed0</guid>
    </item>
  </channel>
</rss>
```

The transformation that generates this output is shown in Figure 21-28.

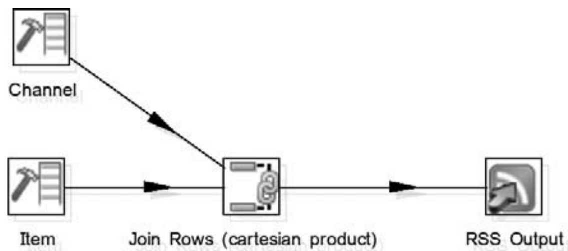


Figure 21-28: Transformation demonstrating RSS output

Source Data

For this example, you use two "Generate rows" steps:

1. For the child elements of the `channel` element (`channelTitle`, `channel-Description`, and `channelLink`), simply drop a "Generate rows" step to the workbench and enter the values as shown in Figure 21-29.

Generate Rows

Step name: Channel

Limit: 1

Fields :

#	Name	Type	Format	Length	Precision	Currency	Decimal	Group	Value
1	channelTitle	String							My Channel
2	channelDescription	String							Example of RSS feed
3	channelLink	String							http://www.mychannel.com

OK Cancel Preview

Figure 21-29: RSS Output source channel

2. For the child elements of the `item` element, (`itemTitle`, `itemDescription`, and `itemLink`), enter the values shown in Figure 21-30.

Generate Rows

Step name: Item

Limit: 1

Fields :

#	Name	Type	Format	Length	Precision	Currency	Decimal	Group	Value
1	itemTitle	String							Test item
2	itemDescription	String							An item in the channel
3	itemLink	String							http://www.mychannel.com/item1

OK Cancel Preview

Figure 21-30: RSS Output source item

3. Next, link these two steps with a "Cartesian join" step.

Now that you have your source data, the next sections walk you through creating a standard feed and a custom feed.

Standard Feed

A standard feed has predefined mandatory or optional elements in both channel and item, as mentioned earlier. To produce standard feed, make sure that “Create custom RSS” is unchecked as in Figure 21-31, and remove the channel and item elements from incoming fields.

Figure 21-31: RSS Output Channel tab

The Channel tab (shown in Figure 21-31) is dedicated to channel items; mandatory fields are marked by an asterisk (*), all others fields are optional and PDI will not complain if you leave them blank. At the bottom of the page, you can set the character encoding and specify the RSS version. Typically, you should use UTF-8 here (which is also the default encoding for XML in general).

One you have mapped the channel elements with incoming fields, you need now to do the same work for items. There is no mandatory item element, as you can see in Figure 21-32. Set only those elements that you need and leave all others blank.

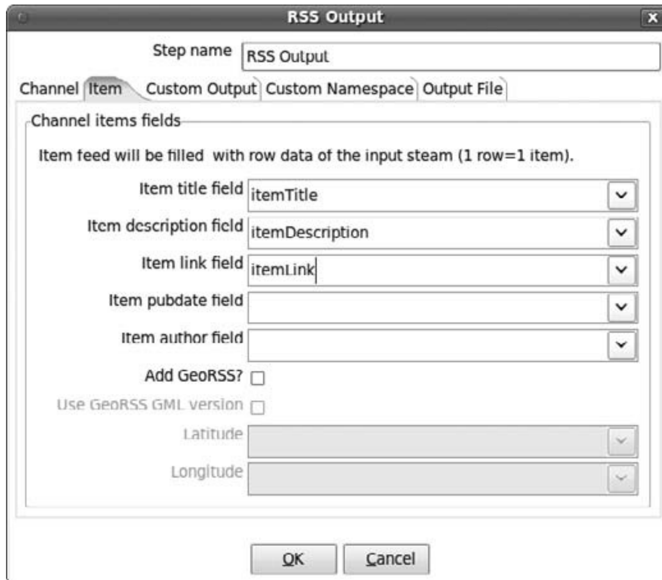


Figure 21-32: RSS Output Item tab

The final step is to configure the output file that will contain the feed. The Output File tab is shown in Figure 21-33.

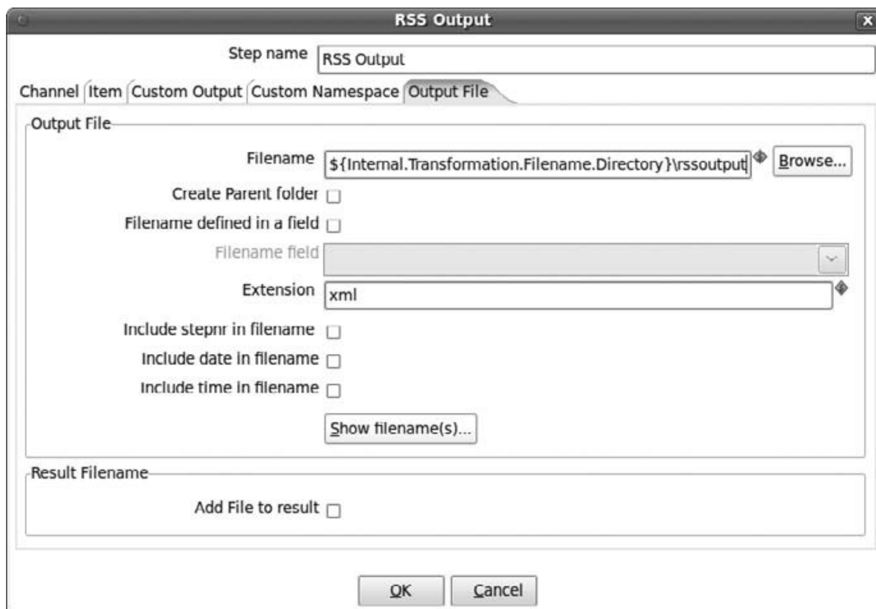


Figure 21-33: RSS Output File tab

The top section of the tab offers the following settings:

- **Filename:** Specify the full filename that will contain the feed. Use the "Browse…" button if necessary.
- **Create Parent folder:** Select this option if you need to create a parent folder at run time; otherwise, if you select an output filename with a nonexistent parent folder, PDI will fail.
- **Filename defined in a field:** You can send a filename dynamically. PDI will read the value from the "Filename field."
- **Filename field:** Name of the field that contains filename (see previous bullet). The value will be extracted in the first received row.
- **Extension:** Output filename extension that will be added to filename.
- **Include stepnr in filename:** Adds copy number of the step to filename (the copy number that begins with 0, …).
- **Include date in filename:** Adds date to filename.
- **Include time in filename:** Adds time to filename.
- **Show filename(s):** Click this button to see what the filename looks like after setting all the previous settings.

The lower section, Result Filename, includes one option:

- **Add File to result:** Add filename to result filenames.

That's it for a standard feed. The sample transformation `RSSOutputStandard.ktr` will create a file containing the feed.

Custom Feed

You saw that in a standard feed, the channel, and item elements tags are prefixed. This is not the case for a custom feed. In fact, you can define any element tag that you need by checking the "Create custom RSS" box on the Channel tab. After checking this, the standard fields for channel and item you just configured are not used and are in fact disabled in the configuration dialog. To control the output of a custom feed, you need to activate the custom output tab. There, you'll find two grids: one for the channel and one for the items. In the grids, you can define element names and assign the field from the incoming stream that is to provide the content for these elements.

Summary

In this chapter, we took a closer look at the Kettle features that allow you to work with the Web and web services. We covered a lot of ground:

- A review of Kettle features for web access, such as general HTTP steps, SOAP, RSS and Apache virtual file system.

- Data formats that are widely in use on the Web, such as XML, HTML, and JSON.
- A detailed example for importing XML data into a database, demonstrating the use of the XSD Validator step to validate an XML document against an XML Schema, and the “Get data from XML” step to extract data from an XML document using XPath syntax.
- A detailed example for exporting data from a database to an XML format, demonstrating the Add XML step for generating simple XML structures such as elements and attributes and the XML Join step for generating nested XML elements.
- How to access SOAP web services using the “Web services lookup” step.
- How to digest SOAP web services directly using either an “HTTP client” or an HTTP Post step, and then parsing the XML result.
- A detailed example for extracting data from Freebase using the JSON-based MQL language, and how to use the Modified Java Script Value step to extract data form a JSON result.
- An example that shows how to extract data from a RSS web feed.
- How to generate a standard RSS feed from source data.

CHAPTER 22

Kettle Integration

Kettle contains a rich set of data integration functionality that is exposed in a set of data integration tools. However, you can also use Kettle as a library in your own software and solutions. The re-use of other software is typical for open source software. It allows you to stop re-inventing the same wheel time and again. In this chapter we explain the license that makes this possible. Then we introduce you to the finer points of integrating Kettle in your own Java software. We start at a high level with a few examples from Kettle usage in the rest of the Pentaho software stack. For each example, we explain how you can perform a similar integration with the Kettle Java API. We also explain how you can parameterize and customize the Spoon user interface. We finish off with a few words about forking.

The Kettle API

In this section, we take a look at what makes Kettle popular as an API. We explore the consequences of the LGPL license that comes with the Kettle software and provide a number of examples of the Kettle API.

The LGPL License

When the development team agreed to make Kettle an open source product, they decided to use the Lesser GNU Public License, or LGPL, (`http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`). This license fell somewhat out of the

good graces of the GNU and Free Software Foundation (FSF) folks for the same reason that it was selected in the first place: The LGPL license allows you to link proprietary (closed source) code without the need to open source that code. While the LGPL license requires you to make changes to the Kettle source code public if you redistribute these changes, it does not require you to release code that simply uses the Kettle libraries. This is in contrast to the GPL license. Software that carries this license can be used or embedded, too. However, said software needs to release that source code to remain compliant with the license.

As you can imagine, the capability to not only re-use the rich Kettle API but also to be able to sell or distribute your new software is valuable to software vendors specifically and corporations in general. It is because of this that we can say that the LGPL is a business-friendly license.

The LGPL also offers another advantage for Kettle. Kettle has had the ability to integrate plugins since version 2.0. The LGPL license makes this particularly interesting because the source code of a plugin does not have to be released. It allows any corporation to extend Kettle for its own needs with minimal effort. For more information on this topic, see Chapter 23. Over the years, many step and job-entry plugins have been created at corporations all over the world (see `http://wiki.pentaho.com/display/EAI/List+of+Available+Pentaho+Data+Integration+Plug-Ins`). For some of these, it was absolutely not possible or it made no sense to release the source code. However, over the years a number of plugins have been released as source code and some of them even made it into the main Kettle source tree as standard steps and job entries. That makes LGPL a dependable and popular choice for the Kettle project.

The Kettle Java API

In this part of the chapter we explain how you can obtain and build Kettle from source code. We also detail how you can compile the Java documentation. Finally, we explain what's in the four kettle libraries and list the libraries you need to include in the class path of your Java software.

Source Code

If you want access the source code, you can either download it from Sourceforge.net at `http://sourceforge.net/projects/pentaho/` or get it directly from the Subversion source code repository. The Kettle source code repository is open to everyone in read-only mode at `http://source.pentaho.org/svnkettleroot/Kettle`.

The stable and patch releases are located under the `branches/` subfolder while the latest new developments are provided under the `trunk/` folder. As such, if you want to get the source code for the very latest (possibly unstable!) source code base, you can do a Subversion checkout of the trunk folder with the following command(note that the UNIX command-line `svn` command needs to be installed for this to run. Users of a Windows compatible operating system can install Tortoise SVN. See `http://tortoisesvn.tigris.org/` for more information.):

```
svn co http://source.pentaho.org/svnkettleroot/Kettle/trunk/
```

Building Kettle

Once you have the source code extracted, you can build Kettle from source. You do this with the Apache Ant build tool (`http://ant.apache.org/`). To build a distribution of Kettle in the `distrib/` subfolder, simply type **ant** on the command line in the directory that contains the Kettle source code.

Building javadoc

If you want to browse the Java documentation of the source code, you can invoke the following command:

```
ant javadoc
```

This will generate the javadoc in the `docs/api` folder, which will allow you to see the comments written to document the various classes and methods in the Kettle codebase. You can use an Internet browser to start reading the file `docs/api/index.html`.

Libraries and the Class Path

Pentaho Data Integration (Kettle) is not just a collection of data integration and business intelligence tools; it can also be used as a Java API. The API is split up into four main parts:

- **Core:** This contains the core classes for Kettle, stored in `.jar` file `kettle-core.jar`.
- **Database:** Containing the database-related classes of Kettle, stored in `.jar` file `kettle-db.jar`.
- **Engine:** The Kettle runtime classes from `kettle-engine.jar`.
- **GUI:** All classes related to graphical user interfaces such as Spoon and that depend on Eclipse SWT classes are stored in `.jar` file `kettle-ui-swt.jar`.

For the samples that follow, you need to put the first three `.jar` files in your class path along with the `.jar` files you find in the `libext/` folder of your Kettle distribution. You might not need all of them but in general it's better to be safe than sorry because you usually have no idea up-front what sort of transformation you'll be executing.

Executing Existing Transformations and Jobs

The following section explains how you can run your existing Kettle transformations and jobs with the Java API.

Executing a Transformation

Examples of software that executes a Kettle transformation are, for obvious reasons, mainly found in the Pentaho software stack. For example, Spoon, Pan, Carte, the Pentaho BI server, and the Pentaho Data Integration server all are capable of executing a transformation. Executing a transformation with the aforementioned tools is easy. However, it is also fairly easy to do this with the Kettle 4.0 Java API. The following code, `ExecuteTrans.java`, demonstrates this in only a few lines.

```
package example.ch22;

import org.pentaho.di.core.KettleEnvironment;
import org.pentaho.di.trans.Trans;
import org.pentaho.di.trans.TransMeta;

public class ExecuteTrans {
  public static void main(String[] args) throws Exception {
    String filename = args[0];

    KettleEnvironment.init();

    TransMeta transMeta = new TransMeta(filename);
    Trans trans = new Trans(transMeta);
    trans.prepareExecution(null);
    trans.startThreads();
    trans.waitUntilFinished();

    if (trans.getErrors()!=0) {
      System.out.println("Error encountered!");
    }
  }
}
```

As you can see, the listing is quite small and easy to understand. Let's start with this line:

```
KettleEnvironment.init();
```

This line of code initializes the complete Kettle environment. It loads all the plugins it can find, initializes the logging environment, sets up and reads the variables system, and even creates a Kettle home directory if it's not already present. If anything goes seriously wrong in there, an exception will be thrown for your convenience.

The following line creates a new `TransMeta` (transformation metadata) object by reading it from a file:

```
    TransMeta transMeta = new TransMeta(filename);
```

It is also possible to read this metadata from a repository, which is easy to do with the `Repository.loadTransformation()` method. Getting your hands on a Repository interface is a bit more complex. You need to have the name (ID) of the repository to

reference. Then you have to connect to the repository with a username and password. The metadata for the repository is stored in a `repositories.xml` file. This can be read by the `RepositoriesMeta.readData()` method. That allows you to look up the `RepositoryMeta` object that describes the repository. The Repository interface is then obtained by asking the `PluginRegistry` to load it for you:

```
RepositoriesMeta repositoriesMeta = new RepositoriesMeta();
repositoriesMeta.readData();
RepositoryMeta repositoryMeta = findRepository( repositoryName );
PluginRegistry registry = PluginRegistry.getInstance();
Repository repository = registry.loadClass(
RepositoryPluginType.class,
repositoryMeta,
Repository.class
  );
repository.connect(username, password);
```

Once you have loaded the transformation metadata, you can execute it as follows:

```
Trans trans = new Trans(transMeta);
trans.prepareExecution(null);
trans.startThreads();
trans.waitUntilFinished();
```

This block creates a new transformation engine object, prepares (initializes) the execution, and starts the transformation threads. Finally, because the whole transformation runs multi-threaded, it waits until all processing has completed.

Last but not least, you see if there were errors during the execution:

```
if (trans.getErrors()!=0) {
  System.out.println("Error encountered!");
}
```

Any errors encountered during the execution of the transformation are printed to the console.

Executing a Job

As with transformations, examples of software that executes a Kettle job are also mainly found in the Pentaho software stack. Spoon, Kitchen, Carte, the Pentaho BI server, and the Pentaho Data Integration server are all capable of executing a job. The execution of a job is similar to the execution of a transformation, as shown in the following code (`ExecuteJob.java`):

```
package example.ch22;

import org.pentaho.di.core.KettleEnvironment;
import org.pentaho.di.job.Job;
import org.pentaho.di.job.JobMeta;
```

```
public class ExecuteJob {
  public static void main(String[] args) throws Exception {
    String filename = args[0];

    KettleEnvironment.init();

    JobMeta jobMeta = new JobMeta(filename, null);
    Job job = new Job(null, jobMeta);
    job.start();
    job.waitUntilFinished();

    if (job.getErrors()!=0) {
      System.out.println("Error encountered!");
    }
  }
}
```

As you can see, loading the job metadata from a file is very similar to what you saw in the previous example. The only difference is that you pass a null repository reference to the `JobMeta` constructor (because there isn't one one in our example).

Embedding Kettle

The Pentaho software stack is a collection of over a hundred small to large components. Many components are small core libraries while others are complete software collections, like the BI Server, hundreds of mega-bytes in size. The reason Pentaho has core libraries is to maximize code re-use to minimize development efforts. Kettle is somewhere in the middle of the pack: it uses Pentaho core libraries but is used itself in other components. As a result, integrating Kettle for various purposes has become easier over time. In the following sections we'll show you usage examples and the source code to demonstrate how you can perform the same integration yourself.

Pentaho Reporting

Pentaho Reporting has the capability to obtain data directly from any step in a transformation. It can do this because it embeds the Kettle libraries. Pentaho Report Designer (PRD) allows you to create a new data source under the Data menu of the type Pentaho Data Integration. That in turn presents a dialog where you can select the transformation to execute and the step to read from. Figure 22-1 shows the Pentaho Report Designer.

NOTE **Because Kettle and Pentaho Reporting are usually on different release schedules and because software changes over time, make sure that you use the appropriate version of Pentaho Reporting to read data from a transformation. Use version 3.5 of Pentaho Reporting to access transformations created with Pentaho Data Integration (PDI) 3.2 and version 3.6 (or later) to access transformations created with PDI 4.0.**

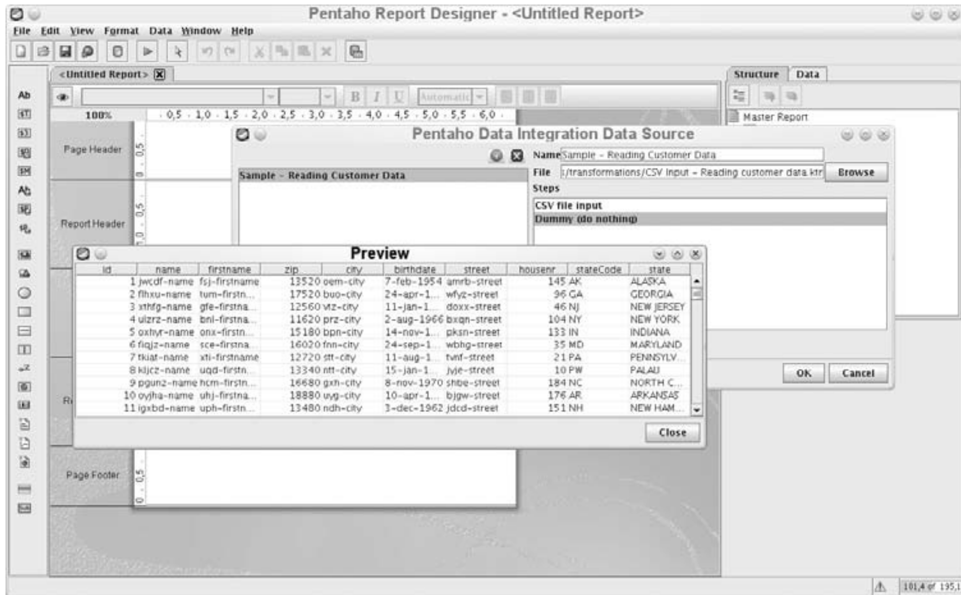


Figure 22-1: Pentaho Reporting using Kettle

Reading information from a step in a transformation is quite easy to do. Now that you know how to execute a transformation, we can extend our example that executes a transformation by reading the data that is produced by a certain step, as shown in the following code (`ReadFromStep.java`):

```
package example.ch22;

import java.util.ArrayList;
import java.util.List;

import org.pentaho.di.core.KettleEnvironment;
import org.pentaho.di.core.RowMetaAndData;
import org.pentaho.di.core.exception.KettleStepException;
import org.pentaho.di.core.row.RowMetaInterface;
import org.pentaho.di.trans.Trans;
import org.pentaho.di.trans.TransMeta;
import org.pentaho.di.trans.step.RowAdapter;
import org.pentaho.di.trans.step.RowListener;
import org.pentaho.di.trans.step.StepInterface;

public class ReadFromStep {
  public static void main(String[] args) throws Exception {
    String filename = args[0];
    String stepname = args[1];

    KettleEnvironment.init();
```

```
      TransMeta transMeta = new TransMeta(filename);
      Trans trans = new Trans(transMeta);
      trans.prepareExecution(null);

      final List<RowMetaAndData> rows = new ArrayList<RowMetaAndData>();
      RowListener rowListener = new RowAdapter() {
       public void rowWrittenEvent(RowMetaInterface rowMeta, Object[] row)
  throws KettleStepException {
          rows.add(new RowMetaAndData(rowMeta, row));
        }
      };
      StepInterface stepInterface = trans.findRunThread(stepname);
      stepInterface.addRowListener(rowListener);

      trans.startThreads();
      trans.waitUntilFinished();

      if (trans.getErrors()!=0) {
        System.out.println("Error");
      } else {
        System.out.println("We read "+rows.size()+" rows from step "+
        stepname);
      }
    }
  }
```

The only thing that changed from the `ExecuteTrans.java` example is that you now attach a `RowListener` object to a step copy. For this example, you assume that there is only one copy. If you want to obtain data from a specific copy, use the `getRunThread(stepname, copy)` method.

When you attach a `RowListener`, you get notified when a row is read or written, or when a row is written to an error handling step. This notification happens in sync with the execution of the transformation. This allows you to handle the records in a streaming fashion, as is the case in the transformation engine. In the sample, we simply add all the output rows to a java list for processing after the transformation finishes executing.

Putting Data into a Transformation

So now that you've learned how to read data from a step in a streaming fashion using the `RowListener` interface, the next task is to put data into a transformation. That is obviously possible, too.

Passing Data with a Result Object

You can use the "Copy rows to result" and the "Get rows from result" steps to pass data from one transformation to the next in a job. You do this with a simple in-memory buffer and it doesn't stream data at all. This means that using a `Result` object is not suited for

large amounts of data. In this case, you can simply code the "Copy rows to result" step using the Java API. In this example, you change the previous example to include the following three lines of code as well as a method to create a list of rows (see `PassDataToTransfer.java` in the download files):

```
. . .
    Result result = new Result();
    result.setRows(createRows());
    transMeta.setPreviousResult(result);
. . .

  private static List<RowMetaAndData> createRows() {
    List<RowMetaAndData> list = new ArrayList<RowMetaAndData>();

    RowMetaAndData one = new RowMetaAndData();
    one.addValue("string", ValueMetaInterface.TYPE_STRING,
      "A sample String");
    one.addValue("date", ValueMetaInterface.TYPE_DATE, new Date());
    one.addValue("number", ValueMetaInterface.TYPE_NUMBER,
      Double.valueOf(123.456));
    one.addValue("integer", ValueMetaInterface.TYPE_INTEGER,
      Long.valueOf(123456L));
    one.addValue("big_number", ValueMetaInterface.TYPE_BIGNUMBER,
      new BigDecimal("1234593943942.39430243953243239434"));
    one.addValue("boolean", ValueMetaInterface.TYPE_BOOLEAN,
      Boolean.TRUE);
    one.addValue("binary", ValueMetaInterface.TYPE_BINARY,
      new byte[] { 0x44, 0x50, 0x49, } );

    list.add(one);

    return list;
  }
```

As you can see, you simply pass a list of rows to a `Result` object and set that as the result of a previous transformation on the `TransMeta` object. The sample shows you a value example for every data type that is used in Kettle. For more information on Kettle rows and values, see Chapter 23.

All you need to do now is include a "Get rows from result" step in your transformation to pass the data when the transformation is executed.

Passing Data in a Streaming Fashion

If you have a lot of data to pass and you want to keep memory requirements as low as possible, you would want to stream the data into a transformation. That is possible, too, using the Injector step. This step is simply a placeholder for data that will arrive at runtime. To pass data to this step you add a `RowProducer` to the `Trans` class object. Consider the example transformation shown in Figure 22-2.

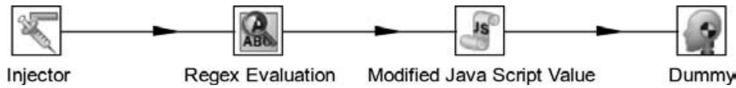


Figure 22-2: Using the Injector step

In this transformation, you don't read from any particular data source or write to any target. The transformation receives information from an outside source through the Injector step, performs some complex regular expression evaluation, and executes a JavaScript program to then return it to use over the Dummy step, as we demonstrated in the previous example. The following code (`InjectDataIntoTransformation.java`) shows how you would pass your rows of data to the Injector step:

```
package example.ch22;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import org.pentaho.di.core.KettleEnvironment;
import org.pentaho.di.core.Result;
import org.pentaho.di.core.RowMetaAndData;
import org.pentaho.di.core.exception.KettleStepException;
import org.pentaho.di.core.row.RowMetaInterface;
import org.pentaho.di.core.row.ValueMetaInterface;
import org.pentaho.di.trans.RowProducer;
import org.pentaho.di.trans.Trans;
import org.pentaho.di.trans.TransMeta;
import org.pentaho.di.trans.step.RowAdapter;
import org.pentaho.di.trans.step.RowListener;
import org.pentaho.di.trans.step.StepInterface;

public class InjectDataIntoTransformation {
  public static void main(String[] args) throws Exception {
    String filename = args[0];

    KettleEnvironment.init();

    TransMeta transMeta = new TransMeta(filename);

    Result result = new Result();
    result.setRows(createRows());
    transMeta.setPreviousResult(result);

    Trans trans = new Trans(transMeta);
    trans.prepareExecution(null);

    final List<RowMetaAndData> rows = new ArrayList<RowMetaAndData>();
```

```
    RowListener rowListener = new RowAdapter() {
      public void rowWrittenEvent(RowMetaInterface rowMeta, Object[] row
        ) throws KettleStepException {
          rows.add(new RowMetaAndData(rowMeta, row));
      }
    };
    StepInterface stepInterface = trans.findRunThread("Dummy");
    stepInterface.addRowListener(rowListener);

    RowProducer rowProducer = trans.addRowProducer("Injector", 0);

    trans.startThreads();

    for (RowMetaAndData row : createRows() ) {
      rowProducer.putRow(row.getRowMeta(), row.getData());
    }
    rowProducer.finished();

    trans.waitUntilFinished();

    if (trans.getErrors()!=0) {
      System.out.println("Error");
    } else {
      System.out.println("We got back "+rows.size()+" rows");
    }
  }

  private static List<RowMetaAndData> createRows() {
    //see the previous example
  }
}
```

It's important to let the transformation create a `RowProducer` for you *before* the threads are started. Once the threads are started, the Injector step will not start to run until you pass rows into it with the `putRow()` method. Please note that this method will pause execution of a step when the associated `RowSet` buffer is full. The buffer size defaults to 10,000 rows. (See Chapter 15 for more information.) A step will momentarily pause execution naturally when there is a lot of data to process and when the rows are injected faster than they can be processed. When you are done, you simply call the `finished()` method to indicate that no more data is to be expected from the Injector step. Then you simply have to wait until the other steps have completed.

Setting Parameters and Variables

One way to pass information to a transformation or a job is parameterization. This can be done in a transformation in the form of variables or parameters.

Setting a variable is easy with the Java API; simply set the variable on the `transMeta` object before you create the `trans` object:

```
transMeta.setVariable("VARIABLE_NAME", "value");
```

Setting a parameter value of a transformation happens in a similar fashion:

```
transMeta.setParameterValue("PARAMETER_NAME", "value");
```

Please note that this method will throw an exception if the parameter name is not known. To obtain a list of all the defined parameters in a transformation, you can use the `transMeta.listParameters()` method. To find out what the default value is for a parameter, you can use the `transMeta.getParameterDefault()` method.

The exact same methods for setting variables and handling parameters are available for the `JobMeta` class as well.

Dynamic Transformations

Spoon itself creates a transformation on-the-fly whenever you hit a Preview button in one of the step dialogs. That transformation consists of the step being previewed and a Dummy step. It is executed and the results are displayed in a dialog. This temporary transformation is never shown afterward. You already know how to execute the transformation. Creating it dynamically is not that hard either. Let's start with a simple use case. You know the name and layout of a CSV file and you simply want Kettle to read it. The separator is always a comma and the enclosure is a double quote character. Then you catch the output from a subsequent Dummy step. Figure 22-3 shows the transformation you create dynamically using the Kettle API.

Figure 22-3: A dynamically generated preview transformation

The name and the layout of the CSV file is the dynamic part of the problem. This information comes from somewhere else. Usually this so-called metadata is stored in another format or entered by the user in a third-party application. We will not focus on this aspect of the problem because it's different every time. Instead, we'll just allow this information to be passed into the `CsvFileReader` class that follows (see `CsvFileReader.java` in the Chapter 22 download files):

```
public class CsvFileReader {

  private static String STEP_READ_A_FILE = "Read a file";
  private static String STEP_DUMMY = "Dummy";

  private String filename;
  private TextFileInputField[] inputFields;
  private List<RowMetaAndData> rows;

  public CsvFileReader(String filename,
                       TextFileInputField[] inputFields) {
```

```
    this.filename = filename;
    this.inputFields = inputFields;
  }

  public void read() throws Exception {

    KettleEnvironment.init();

    // Create a new transformation...
    //
    TransMeta transMeta = new TransMeta();
    transMeta.setName("sample04");

    // Create the step to read the file
    //
    CsvInputMeta inputMeta = new CsvInputMeta();
    inputMeta.setDefault(); // comma separated, " enclosed with header.
    inputMeta.setFilename(filename);
    inputMeta.setInputFields(intFields);

    StepMeta inputStep = new StepMeta(STEP_READ_A_FILE, inputMeta);
    inputStep.setLocation(50, 50);
    inputStep.setDraw(true);
    transMeta.addStep(inputStep);

    // Create the dummy place-holder step
    //
    DummyTransMeta dummyMeta = new DummyTransMeta();
    StepMeta dummyStep = new StepMeta(STEP_DUMMY, dummyMeta);
    dummyStep.setLocation(150, 50);
    dummyStep.setDraw(true);
    transMeta.addStep(dummyStep);

    // Create a hop between the 2 steps
    //
    TransHopMeta hop = new TransHopMeta(inputStep, dummyStep);
    transMeta.addTransHop(hop);

    // Now we can execute the transformation
    //
    Trans trans = new Trans(transMeta);
    trans.prepareExecution(null);

    rows = new ArrayList<RowMetaAndData>();
    RowListener rowListener = new RowAdapter() {
      public void rowWrittenEvent(RowMetaInterface rowMeta,
                                  Object[] row
        ) throws KettleStepException {
        rows.add(new RowMetaAndData(rowMeta, row));
      }
    };
```

```
    StepInterface stepInterface = trans.findRunThread(STEP_DUMMY);
    stepInterface.addRowListener(rowListener);

    trans.startThreads();
    trans.waitUntilFinished();

    if (trans.getErrors()!=0) {
      System.out.println("Error");
    } else {
      System.out.println("We read "+rows.size()+" rows from step "
        +STEP_DUMMY);
    }
  }

  public List<RowMetaAndData> getRows() {
    return rows;
  }
```

As you can see in the preceding code, you're no longer loading a transformation from a file or from a repository; you're creating it from scratch. The `TransMeta` class allows you to add all sorts of metadata-like steps, hops, and database connections to it. The resolution of that metadata happens at runtime so you don't have to worry about that. In the example, we're adding two steps and a hop:

```
TransMeta transMeta = new TransMeta();
. . .
transMeta.addStep(inputStep);
. . .
transMeta.addStep(dummyStep);
. . .
transMeta.addTransHop(hop);
```

This is equivalent to dragging the two steps on the canvas and then creating a hop between them.

Here is a block of code that uses the `CsvFileReader` class:

```
TextFileInputField id = new TextFileInputField("id", -1, 8);
id.setTrimType(ValueMetaInterface.TRIM_TYPE_BOTH);
id.setFormat("#");
id.setType(ValueMetaInterface.TYPE_INTEGER);

TextFileInputField firstname = new TextFileInputField("firstname",
  -1, 50);
firstname.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField name = new TextFileInputField("name", -1, 50);
name.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField zip = new TextFileInputField("zip", -1, 15);
zip.setTrimType(ValueMetaInterface.TRIM_TYPE_LEFT);
```

```
zip.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField city = new TextFileInputField("city", -1, 50);
city.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField birthdate = new TextFileInputField("birthdate",
  -1, -1);
birthdate.setFormat("yyyy/MM/dd");
birthdate.setType(ValueMetaInterface.TYPE_DATE);

TextFileInputField street = new TextFileInputField("street", -1,
  50);
street.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField housenr = new TextFileInputField("housenr", -1,
  15);
housenr.setTrimType(ValueMetaInterface.TRIM_TYPE_LEFT);
housenr.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField statecode = new TextFileInputField("statecode",
  -1, 2);
statecode.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField state = new TextFileInputField("state", -1, 50);
state.setType(ValueMetaInterface.TYPE_STRING);

TextFileInputField[] inputFields = new TextFileInputField[] {
    id, firstname, name, zip, city, birthdate, street, housenr,
    statecode, state,
};

String filename = "samples/transformations/files/customers-100.txt";
CsvFileReader reader = new CsvFileReader(filename, inputFields);
reader.read();
```

Dynamic Template

By setting the user interface location and the drawn attribute you make sure that the transformation you generated is also perfectly valid and editable in Spoon after storing it in a file. The XML representation of any object in the Kettle API can be easily obtained simply by calling the `getXML()` method. It's thus easy to save the transformation metadata to a file:

```
String xml = XMLHandler.getXMLHeader() + transMeta.getXML();
DataOutputStream dos = new DataOutputStream(
  KettleVFS.getOutputStream("csv-reader.ktr", false)
 );
dos.write(xml.getBytes(Const.XML_ENCODING));
dos.close();
```

Suppose you have hundreds of different types of files to read and you need to create a transformation for each type with a "CSV file input" step, a Table Output step, and a few error-handling steps. It would be perfectly valid to create a program that does this if you already have the metadata somewhere. The `CsvFileReader` class then becomes a dynamic template for your work. In a matter of seconds, you can then generate hundreds of transformations all with exactly the right properties and tuned the way you want them.

As in most (if not all) ETL tools, the output of a step needs to be known in advance and predictable. As such, in the traditional sense, the only option you have is to create a few hundred transformations in Spoon and manually configure all steps in each one. That problem can be solved by using a dynamic template, as shown in the `CsvFileReader` example. Because you can already execute these transformations on-the-fly, it's totally up to you whether or not you should store them on disk for integration into your existing ETL framework. By wrapping up the dynamic generation and execution in a step or job entry plugin, it can even be a combination of both. For more information on that topic, see Chapter 23. The settings in such a plugin would allow you to configure the details of the transformation generation itself.

Dynamic Jobs

One example of jobs that get created on-the-fly is the "Copy tables" wizard in Spoon. That wizard allows you to generate a job that copies a selection of database tables from one database to another. The resulting job contains an "Execute SQL script" job entry for every selected table as well as a Transformation job entry to populate the target table.

Because the question about dynamically generating and executing SQL comes back regularly on the Kettle forum, the following code (`DynamicJob.java`) will take a transformation filename as a parameter to generate a job that will execute the SQL required to execute the transformation. Because zero, one, or more databases can be involved, you can have zero, one, or more SQL Script job entries in the resulting job.

```
package example.ch22;

import java.util.HashSet;
import java.util.List;

import org.pentaho.di.core.ObjectLocationSpecificationMethod;
import org.pentaho.di.core.SQLStatement;
import org.pentaho.di.core.database.DatabaseMeta;
import org.pentaho.di.job.JobHopMeta;
import org.pentaho.di.job.JobMeta;
import org.pentaho.di.job.entries.sql.JobEntrySQL;
import org.pentaho.di.job.entries.trans.JobEntryTrans;
import org.pentaho.di.job.entry.JobEntryCopy;
import org.pentaho.di.trans.TransMeta;

public class DynamicJob {

  public static JobMeta generateJobMeta(String transFilename
    ) throws Exception {
    JobMeta jobMeta = new JobMeta();
```

```
jobMeta.setName("sample05");

int x = 50;
int y = 50;

// Add the start entry...
//
JobEntryCopy startCopy = JobMeta.createStartEntry();
startCopy.setLocation(x, y);
startCopy.setDrawn();
jobMeta.addJobEntry(startCopy);
JobEntryCopy lastCopy = startCopy;

// Determine the SQL and databases needed to
// execute the transformation
//
TransMeta transMeta = new TransMeta(transFilename);
HashSet<DatabaseMeta> databases = new HashSet<DatabaseMeta>();
List<SQLStatement> sqlStatements = transMeta.getSQLStatements();
for (SQLStatement stat : sqlStatements) {
  databases.add(stat.getDatabase());
}


// Add "Execute SQL script" for every used database...
//
for (DatabaseMeta databaseMeta : databases) {
  JobEntrySQL jobEntrySql = new JobEntrySQL();
  jobEntrySql.setDatabase(databaseMeta);

  String sql = "";
  for (SQLStatement sqlStatement : sqlStatements) {
    if (sqlStatement.getDatabase().equals(databaseMeta)) {
      if (!sqlStatement.hasError() && sqlStatement.hasSQL())
      {
          sql += sqlStatement.getSQL();
      }
    }
  }
  jobEntrySql.setSQL(sql);

  JobEntryCopy sqlCopy = new JobEntryCopy(jobEntrySql);
  sqlCopy.setName("SQL for "+databaseMeta.getName());
  x+=100;
  sqlCopy.setLocation(x, y);
  sqlCopy.setDrawn();
  jobMeta.addJobEntry(sqlCopy);

  JobHopMeta sqlHop = new JobHopMeta(lastCopy, sqlCopy);
  jobMeta.addJobHop(sqlHop);
  lastCopy = sqlCopy;
```

```
    }

    // Now execute the transformation as well...
    //
    JobEntryTrans jobEntryTrans = new JobEntryTrans();
    jobEntryTrans.setSpecificationMethod(
      ObjectLocationSpecificationMethod.FILENAME);
    jobEntryTrans.setFileName(transFilename);

    JobEntryCopy transCopy = new JobEntryCopy(jobEntryTrans);
    transCopy.setName("Execute "+transFilename);
    x+=100;
    transCopy.setLocation(x, y);
    transCopy.setDrawn();
    jobMeta.addJobEntry(transCopy);

    JobHopMeta transHop = new JobHopMeta(lastCopy, transCopy);
    jobMeta.addJobHop(transHop);
    lastCopy = transCopy;

    return jobMeta;
  }
}
```

As you can see from the `TransMeta.getSQLStatements()` method, it is quite easy to have a transformation generate all the SQL statements it needs to function properly. Note that this is only the case if the transformation itself is not too dynamic. If you, for example, define a table name in a Table Output step with a variable that is dynamically defined, it is not possible to know what SQL needs to be generated.

The `JobMeta` object that is the result of the `generateJobMeta()` method shown here can be executed or saved to disk as described earlier. Again, it's up to you to save this metadata to an XML file (or a repository), to execute this right away, or to make this part of a step or job entry plugin.

Executing Dynamic ETL in Kettle

Of course, it's also possible to execute the code you just saw in a User Defined Java Class or Modified Java Script Value step. For this example, you simply extend the `DynamicJob` example with one method:

```
public static Result executeTransformation(String transFilename
  ) throws Exception {
  JobMeta jobMeta = generateJobMeta(transFilename);
  Job job = new Job(null, jobMeta);
  job.start();
  job.waitUntilFinished();
  return job.getResult();
}
```

This method will execute the specified transformation. Now all you need to do is put the class in a .jar file (see the jar command in the Java Development Kit) and put the .jar file somewhere in the libext/ folder of your Kettle distribution. This makes the class available to Kettle.

Next you can use a bit of JavaScript to execute a list of transformations (.ktr files). You can obtain the list of files from a Get File Names step. If you bring these filenames to a Modified Java Script Value step, the script is then simply:

```
var result = Packages.example.ch22.DynamicJob
  .executeTransformation(filename);
var errors = result.getNrErrors();
```

With a single line of JavaScript, the specified transformation is dynamically executed in a job after all the required SQL is executed.

Result

In the previous example, a Result class is returned and this object contains not only the number of errors but also a lot of other useful information you might want to use to pass data between transformations or jobs. Table 22-1 shows the various types of information you can retrieve from a Result object. The method to use is listed alongside with the data type and description of the information returned.

Table 22-1: The Result Object

METHOD	DATA TYPE	DESCRIPTION
getResult	boolean	Contains true if the object (transformation or job) was executed successfully, false if there was some error.
getExitStatus	int	Exit status of the Shell Script job entry.
getEntryNr	int	The entry number is increased every time a job entry is executed in a job.
getNrErrors	long	The number of errors.
getNrLinesInput	long	The number of rows read from a file or database.[a]
getNrLinesOutput	long	The number of rows written to a file or database.[a]
getNrLinesRead	long	The number of rows read from previous steps.[a]

Continued

Table 22-1 *(continued)*

METHOD	DATA TYPE	DESCRIPTION
getNrLinesUpdated	long	The number of rows updated in a file or database.[a]
getNrLinesWritten	long	The number of rows written to next step.[a]
getNrLinesDeleted	long	The number of deleted rows.[a]
getNrLinesRejected	long	The number of rows rejected and passed to another step via error handling.[a]
getRows	List <RowMetaAndData>	The result rows.
isStopped	boolean	Flag to indicate if the execution was manually stopped or not.
getResultFilesList	List <ResultFile>	The list of all the files used in the executed object(s).
getNrFilesRetrieved	int	The number of files retrieved from FTP, SFTP, and so on.
getLogText	String	The log text of the execution of the executed object and its children.
getLogChannelId	String	The ID of the log channel of the executed object. You can use this to look up information on the execution lineage in the Log Channel log table.

http://wiki.pentaho.com/display/EAI/Evaluating+conditions+in+The+JavaScript+job+entry

a. You need to select a step in the Logging tab of the Transformation Settings dialog to make this metric work properly. You need to select which step is representative for each metric.

Replacing Metadata

On occasion, you might find yourself in a situation in which you would like to load an existing transformation or job only to replace a database or step at runtime. While a lot of parameterization is possible through the clever use of variables and named parameters (as you've already seen), you might need to insert completely new objects.

The use case discussed here is the replacement of a database connection in an existing transformation. You want to do this because you'll not only change the hostname and database name in the connection, but also the database type itself. This database type

can't be specified using a variable or parameter. In the example, the database connection in the transformation is called DB and is used in various steps.

The first step is to create a new `DatabaseMeta` object for the sample. Please remember that this task is quite similar for steps, job entries, slave servers, partition schemas, and cluster schemas:

```
DatabaseMeta databaseMeta = new DatabaseMeta(
"DB", "MySQL", "JDBC", "localhost",
"test", "3306", "user", "password"
  );
```

Direct Changes with the API

To replace the database connection called DB in the transformation or job, use the `addOrReplaceDatabase()` method:

```
transMeta.addOrReplaceDatabase(databaseMeta);
```

This method will make sure that the existing database connection is modified because references to this object are being held by various steps in the transformation.

Using a Shared Objects File

The shared objects file contains databases, steps, and other objects that are considered reusable by the user. You can make an object reusable simply by right-clicking it in the left-hand tree in Spoon and selecting Share. The object in question will then end up in the shared object file of the loaded transformation or job. By default this is `$KETTLE_HOME/.kettle/shared.xml`

The objects in the shared file will take precedence over any object with the same name that is defined in your transformation or job. As such, you can use a shared objects file to dynamically modify the settings of your database connection so you need to first generate the shared objects file:

```
SharedObjects sharedObjects = new SharedObjects();
sharedObjects.storeObject(databaseMeta);
sharedObjects.setFilename("/tmp/shared.xml");
sharedObjects.saveToFile();
```

Now you need to make TransMeta load the shared objects when the transformation is executed:

```
transMeta.setSharedObjectsFile("/tmp/shared.xml");
transMeta.readSharedObjects();
```

If you need to set a default shared objects file location for a complete job, you can also do this by setting a system variable called `KETTLE_SHARED_OBJECTS`:

```
System.setProperty(Const.KETTLE_SHARED_OBJECTS, "/tmp/shared.xml");
```

In order for this to work, you need to make sure you don't have any other shared object file location specified in your transformation or job.

OEM Versions and Forks

In this section you will learn how to customize Kettle and the Spoon user interface to make them look and behave exactly like you want. We will also explain what forking Kettle entails, which forks exist, and what the consequences are.

Creating an OEM Version of PDI

Kettle has made the extension of functionality easy with all sorts of plugin systems and parameterizations. However, it might be interesting for a company or organization to create a rebranded or OEM version for the purpose of being deployed as part of a larger suite. As long as you don't change anything in the original Kettle source code, it's totally fine according to the LGPL license to create such versions. This is why we have made it easy for people to change the look and feel of Kettle, and Spoon in particular.

Most of the Kettle code base supports internationalization, also known as i18n (i-*18 characters*-n). Each piece of text in the Spoon user interface, for example, is translated in a number of different languages. Each piece of text is stored separately in set of text files, each with a different unique key. For example, the name of the Spoon application is stored in key `Spoon.Application.Name` in this file:

```
src-ui/org/pentaho/di/ui/spoon/messages/messages_en_US.properties
```

It would be possible to change the name of Spoon in that file for the English translation. However, that would, in effect, create a fork of Kettle because the source code is changed. This was not deemed sufficient for OEM purposes because it would require OEM customers of Pentaho to maintain their own code base of Kettle. Thus, the Look and Feel (LAF) manager was created, which allows you to define i18n keys, images, and property files, separate from the main Kettle source tree. You do this by creating an alternative properties file such as this one:

```
com/example/book/ui/spoon/messages/messages_en_US.properties
```

In that file, you can then put all the Spoon keys you want to change—for example, the name:

```
Spoon.Application.Name=Data Integration Designer
```

You then put the file together with the complete path in a `.jar` file with the following command:

```
jar -cvf customizations.jar com/
```

The file `customizations.jar` is then placed somewhere in the `libext/` folder to put it in the Kettle class path. Now all you need to do is explain to Kettle where to find these customizations. You do this by setting the `LAFpackage` property in the `ui/laf.properties` file. In this case, we set it as follows:

```
LAFpackage=com.example.book
```

If you then start Spoon, you'll notice that the title of the tool changed and that the term "Spoon" is replaced in various places.

The `ui/laf.properties` file also contains the paths to many properties that will allow you to change many aspects of the way Spoon looks and behaves. These include, for example, the icons used and the name of the Kettle home directory.

Forking Kettle

For customized versions of Kettle or OEM versions, nothing is actually changed in the Kettle source code. For most people and organizations the offered customization and parameterization options are more than sufficient. However, you might encounter a scenario where extensive changes need to be made to the standard functionality of Kettle. In that case you make be tempted to create a fork.

According to the terms of the LGPL as described earlier, a *fork* is a legal copy of the Kettle source code base to start independent development on. It becomes a distinct piece of software. While most companies, groups or organizations usually try to avoid the overhead of maintaining a complete fork of a code base as large as Kettle's, it does happen.

Perhaps the most well-known active fork is GeoKettle. GeoKettle is an open source project of the Spatialytics.org community (`http://www.spatialytics.com/`). The founders of that project, Dr. Thierry Badard and Luc Vaillencourt, have a long history in geospatial software development. The GeoKettle project adds GIS (Graphical Information Systems) capabilities to Kettle such as the capability to read from Oracle Spatial, PostGIS (`http://postgis.refractions.net/`), and MySQL with ESRI shape files. The project also added spatial reference systems management and coordinate transformations. Because it was not possible to add these features through the standard plug-in system, the developers forked the Kettle codebase. GeoKettle is also LGPL-licensed. Traditionally, it keeps up with the release schedule of mainstream Kettle with only a few weeks delay and in general keeps the software compatible with Kettle. Because the needs for spatial data warehousing are very specific, this can be considered a friendly and non-competing fork.

There have been a few isolated occasions in the past where an individual or company felt the need to fork Kettle. On some occasions, the source code was released alongside the binaries; on other occasions, this was not the case. A lot of companies use a forked, self-maintained version of Pentaho Data Integration in-house. This is perfectly fine according to the LGPL license. However, it's important to know that if you do come across a forked version of Kettle you have the right to ask for the source code.

It's also important to know that it takes considerable effort to maintain a successful fork of Pentaho Data Integration given the size of the development and translation

teams. Thousands of changes, large and small, are performed every year and keeping up with that pace is no easy task. That is why most of the known forks have become stale or extinct. We suspect that because of this, the ones that are still around at this time are little more than rebranded versions of the mainstream Kettle distribution, possibly with a few extra plugins. Because these are unknown, unsupported forks of Kettle, it's impossible to say which changes or which patches were installed or not.

Summary

This chapter explained the various ways you can integrate Kettle in your own Java software. We did this with a few easy to understand examples. Here are the most important things that you learned:

- The basics of the LGPL license and how LGPL applies to source code development with the Kettle API
- How to obtain and compile the Kettle codebase
- How to execute transformations and jobs with the Java API, including more advanced features like extraction of data from a step and injection of data into a transformation
- How to create your own OEM version of Kettle
- About GeoKettle and the other forks of Kettle

CHAPTER

23

Extending Kettle

The final chapter of this book will teach you how to get more out of your ETL solution and extend Kettle by developing your own plugins. None of the 34 ETL subsystems covers this, as it doesn't directly belong to the ETL solution. However, any plugin you develop will belong to one of the subsystems covered in this book, whether it is an extraction component for a proprietary data source or a plugin that generates documentation

As you must know by now, Kettle contains a rich set of building blocks like steps and job entries to help you solve complex problems. However, even with all the available functionality at your fingertips, at times you may find yourself in a situation that requires you to extend Kettle. Usually such extension is required to integrate Kettle with third-party or newly emerging technology.

We start by looking at the plugin architecture, the various types of plugins, and what makes it possible for Kettle to load plugins. Next, you learn how to set up your own development environment before we explain how you can develop new steps, job entries, partitioning methods, and so on. You get detailed explanations about the important classes and methods that are required in the various plugin types.

Plugin Architecture Overview

We start with an overview of the plugin capabilities of Kettle. You will learn what kind of plugins can be created and how they are loaded at run-time.

If you had to name the defining characteristics common and essential to all software projects and products that are categorized as open source and/or free software, regardless of license, programming language, or application domain, then these characteristics have to be:

- The unrestricted availability of its source code
- The right to modify that source code to build an alternative version of the executable program
- The right to distribute the software or its source code, or a modified version

You could argue over what all this means exactly and you can browse the mailing lists and user forums of popular open source products to discover that many people do, in fact, argue about this and many other things. However, in practice it means the following:

- The program's source code can be freely downloaded from the Internet. Usually this means you can obtain the code free of charge. However, a nominal fee may be charged to account for any expenses resulting from making the source code available.
- The code can be edited using a common text editor or integrated development environment. You are not required to buy any specific tools to change the source code.
- Assuming proper knowledge and skills, the modified source can be used to construct a working version of the program, and both modified source and/or any programs built from the (modified or original) source code may be passed on to other people.

As described in Chapter 22, all open source and free software, including Kettle, is extensible by default. As per Kettle's LGPL license, everybody can obtain the Kettle source code and change it to their liking to build another working version of the Kettle program and distribute it. This was described in Chapter 22 as *forking*.

However, in addition to the de facto extensibility common to all open source and free software projects, Kettle features a *plugin architecture*. This eliminates most needs to fork the project. It allows extensibility by design; it makes it possible to write essentially isolated pieces of software called *plugins*. Those can be loaded dynamically into an otherwise unmodified version of Kettle.

Plugin Types

The following Kettle plugin types affect run-time behavior:

- **Transformation step plugins:** Implement steps that can be used to process rows flowing through a Kettle transformation.
- **Job entry plugins:** Implement a task that can be executed as part of a Kettle job.
- **Partitioning method plugins:** Allow you to specify your own partitioning rules with the input of field values.

- **Database type plugins:** Database type plugins implement a database connection type.
- **Repository type plugins:** A repository type plugin allows you to handle Kettle metadata persisted in alternate locations or in alternate formats.

NOTE **In addition to these types, it is also possible to inject user interface elements into the Spoon application in the form of Spoon plugins. However, those are not covered in this book.**

Architecture

From a functional viewpoint, there is no difference between an internal Kettle object and a plugin. In other words, there is nothing more you can do with an internal step or job entry because the API is the same for both. The only difference is the way that these objects are loaded at run-time.

Since version 4, Kettle has used a common plugin system called the *Plug-in Registry* that is responsible for the loading of both internal classes and plugins from various locations. There are two things that uniquely identify a plugin:

- **The plugin type:** Represented by interface `PluginTypeInterface`. Examples include `StepPluginType`, `JobEntryPluginType`, `PartitionerPluginType`, and `RepositoryPluginType`.
- **The IDs of the plugin:** This is an array of strings that uniquely identifies a plugin. Because an old plugin can be replaced by a new plugin, a plugin can have multiple IDs. In most cases, however, a plugin has a single ID string. Examples include `TableInput` for the Table Input step and `MYSQL` for the MySQL database type.

When the Kettle environment is initialized, the Plug-in Registry first loads all the internal objects. It reads these from XML files that are located in the Kettle `.jar` files:

- `kettle-steps.xml`: The internal transformation steps
- `kettle-job-entries.xml`: The internal job entries
- `kettle-partition-plugins.xml`: The internal partitioning types
- `kettle-database-types.xml`: The internal database types
- `kettle-database-types.xml`: The internal database types
- `kettle-repositories.xml`: The internal repository types

Once the Plug-in Registry loads the internal objects, it goes on to search for possible plugins. It does this by scanning the `.jar` files that are present in the various subfolders of the `plugins/` directory. It will look for specific Kettle annotations to determine whether or not a class is a plugin. Specific details about the loading process are covered later in this chapter.

An important consequence of plugins being loaded after internal objects and the use of IDs is that you can replace internal functionality with plugins. For example, if you create and deploy a Step plugin with ID `TableInput`, then you'll be replacing the

standard Table Input step. This can be interesting since it allows you to extend standard functionality with a plugin. Extending in turn is easy since you can use Java sub-classing to only enhance those parts that you want.

Prerequisites

In this section, we take a quick look at all the things you need to start developing a Kettle plugin.

Kettle API Documentation

Each plugin implements an application programming interface (API) that is specific for its plugin type. The API comprises a collection of Java interfaces that must be implemented and a number of base classes providing a basic implementation of these interfaces from which you can derive your own classes. In addition, the methods of these interfaces and classes exchange data with the Kettle program proper via a number of helper classes that appear in method parameter lists and/or return types.

The classes and interfaces that make up the plugin API are documented using javadoc source code documentation. The compiled, human-readable javadoc documentation is a great resource when writing plugins. It can be derived from the Kettle source as described in the previous chapter or at `http://javadoc.pentaho.com/kettle`.

Libraries

Besides the basic set of required libraries that was described in the previous chapter, you also need a few libraries from the Eclipse SWT (Standard Widget Toolkit) project. Because users need to interact with the plugins you create, you will need to develop a user interface for them, and thus you also require these third-party libraries. These are:

- `commands.jar`
- `common.jar`
- `jface.jar`
- `runtime.jar`
- `swt.jar`

You can find these libraries in the `libswt/` subdirectory of the Kettle home directory. Note that there is a specific `swt.jar` file for each different target operating system; for each target platform, you will find a respective subdirectory in the `libswt/` directory (such as `linux`, `osx`, and `win32`). While developing, you'll need the `swt.jar` file that corresponds to your development platform.

Integrated Development Environment

Programming tasks are greatly simplified by using an integrated development environment (IDE). They help you organize source code into projects, navigate to related

source code, offer wizards for generating code, and allow you to run and debug, all from within the same environment. There are many different IDEs, but for this chapter we assume the usage of the popular Eclipse project. Eclipse is an open source IDE that is especially suitable for Java development.

You can download a copy of Eclipse at `http://www.eclipse.org/downloads/`. The Eclipse download site offers a number of different versions, optimized for particular programming languages. For Kettle plugin development, we recommend that you use the download labeled Eclipse IDE for Java Developers.

If you follow the instructions on the Eclipse download page, you will end up downloading a 90–100MB zip file. Installation is just a matter of copying it to the desired location and unzipping it.

Linux users may also be able to install Eclipse using their package management system. Please refer to the documentation of your Linux distribution to find out how to install Eclipse. If you don't succeed with a pre-packaged Eclipse for your distro, you can always try to download Eclipse from the eclipse downloads site and install it manually.

Eclipse Project Setup

After creation of a new project in Eclipse, we recommend that you create a package under the default source folder (usually `src/`). To create a new package simply right-click on the source folder and select New ➪ Package. For example, the package could be `org.kettlesolutions.plugin`, as in Figure 23-1.

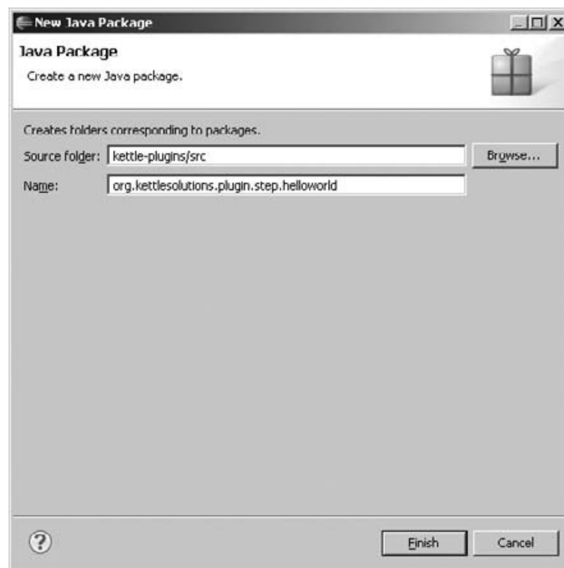


Figure 23-1: Creating a new Java package

Then you can create sub-packages that reflect the type and name of the plugin—for example: `org.kettlesolutions.plugin.step.helloworld`.

To make the setup of your project quick and easy, create folders `libext/` and `libswt/` in your project's root directory. Copy the four kettle libraries from the `lib/` folder in your Kettle distribution into the `libext/` project folder. Also copy the required libraries from the distribution's `libswt/` folder. Take only the `swt.jar` file that corresponds with your system. You can then configure the project's build path to include all libraries under the `libext/` folder and the needed libraries from the `libswt/` folder. Figure 23-2 shows the configured setup.

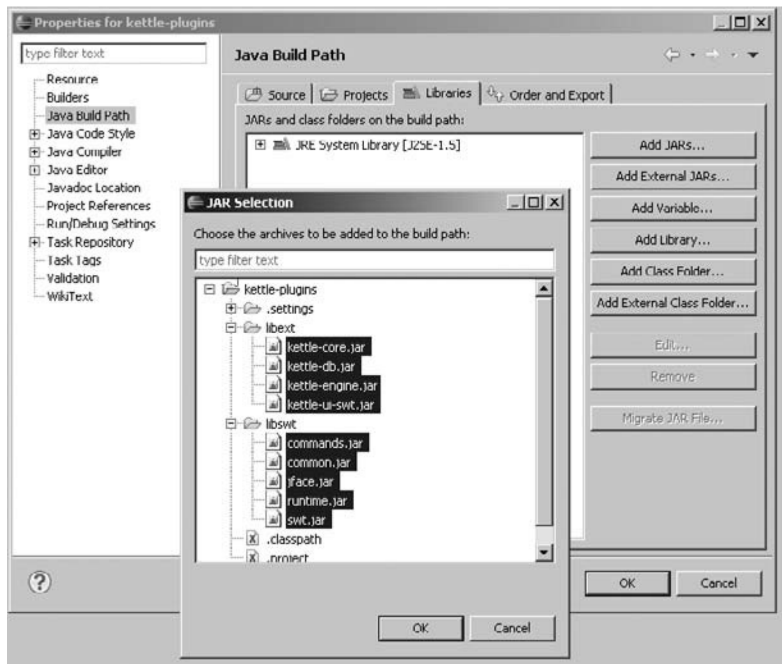


Figure 23-2: Selecting the .jar files for the project

Once that is done, you can start to develop your plugins. Please note that looking up classes and interfaces is easy in Eclipse with the Ctrl+Shift+T keyboard shortcut (or use menu Navigate ➪ Open Type).

Examples

Please remember that Kettle is open source and that essentially all existing predefined steps, job entries, partitioning, repositories, and types can serve as examples. That is because there is structurally no difference between an internal Kettle object (such as a step) and a plugin. The only recognizable difference is the way that they are loaded. Internal objects are loaded from the Kettle `.jar` files, whereas external plugins are loaded from the `plugins/` folder.

You can find the source code for all the steps in the Kettle codebase in the `org.pentaho.di.trans.steps` package. The Kettle job entries can be found in the `org.pentaho.di.job.entries` package.

Transformation Step Plugins

A transformation step plugin is made up of at least four Java classes that implement four interfaces:

- `StepMetaInterface`: This interface describes the step to the outside world and handles serialization.
- `StepInterface`: This interface handles the actual execution of the functionality described in the metadata.
- `StepDataInterface`: A convenient interface that gives you a place to store temporary data, file handles, and so on.
- `StepDialogInterface`: The interface that allows you to edit the metadata using a dialog in the Spoon user interface.

In this section, we cover the basics in these interfaces. For each, we'll show you the corresponding class of a simple "Hello World" example. This example will add the string `"Hello, world!"` in a field with a user-defined name to any input it receives. To finish, we'll explain how you can deploy your newly created step plugin.

StepMetaInterface

Interface `org.pentaho.di.trans.step.StepMetaInterface` has the responsibility to take care of all tasks that are related to the metadata of a step. Those responsibilities include the following tasks (shown with the associated methods):

- Serialize and de-serialize metadata to XML or a repository
  - `getXML()` and `loadXML()`
  - `saveRep()` and `readRep()`
- Describe the output fields to the outside world
  - `getFields()`
- Verify the metadata for correctness
  - `check()`
- Calculate the required SQL to make a step work correctly
  - `getSQLStatements()`
- Set default values on the steps metadata
  - `setDefault()`
- Perform a database impact analysis
  - `analyseImpact()`

- Describe any possible input or output streams
  - `getStepIOMeta()`
  - `searchInfoAndTargetSteps()`
  - `handleStreamSelection()`
  - `getOptionalStreams()`
  - `resetStepIoMeta()`
- Export metadata resources
  - `exportResources()`
  - `getResourceDependencies()`
- Describe used libraries
  - `getUsedLibraries()`
- Describe used database connections
  - `getUsedDatabaseConnections()`
- Describe required fields to make this step function properly (usually targeting a database table)
  - `getRequiredFields()`
- Express capabilities to the user interface and transformation engine:
  - `supportsErrorHandling()`
  - `excludeFromRowLayoutVerification()`
  - `excludeFromCopyDistributeVerification()`

There are also methods that describe how the four interfaces in the step are glued together, such as the following:

- `String getDialogClassName()`: Describes the name of the dialog class that implements the `StepDialogInterface`. If this method returns `null`, the dialog class is automatically determined based on the name and package of the class implementing the `StepMetaInterface`. More information about the step dialog class can be found later in this chapter.
- `StepInterface getStep()`: Creates a new class that implements the `StepInterface` class.
- `StepDataInterface getStepData()`: Creates a new class that implements the `StepDataInterface` class.

Let's now take a look at what the "Hello World" sample step looks like with respect to the metadata interface (this Java code can also be found in download file `HelloworldStepMeta.java`):

```
package org.kettlesolutions.plugin.step.helloworld;

… /* imports removed for brevity */
```

```
@Step(
  id="Helloworld",
  name="name",
  description="description",
  categoryDescription="categoryDescription",
  image="org/kettlesolutions/plugin/step/helloworld/HelloWorld.png",
  i18nPackageName="org.kettlesolutions.plugin.step.helloworld"
)
```

The `@Step` annotation indicates to the Plug-in Registry loading code that this class needs to be considered as a step plugin. The annotation allows you to specify an ID for the step, an icon, a localized name, description, and Spoon step category. The values of the last three are determined by looking up the specified key (`name`, `description`, `categoryDescription`) in an accompanying properties file called a *message bundle*. The `i18nPackageName` option describes the location of the internationalization (i18n) bundle. In the example, the bundle is located in the `org/kettlesolutions/plugin/step/helloworld/messages` folder. For the `en_US` (English, United States) locale, the file would be called `messages_en_US.properties`. In our example, the values in that properties file would include the following items:

```
name=Hello world
description=A very simple step that adds "Hello world" to the incoming
    stream
categoryDescription=Plugin samples
```

Note that by specifying a non-standard category, one that is not yet used by the internal steps, you are creating a separate category that will show up at the top of the list in the Spoon step category listing.

Finally, the `image` tag of the annotation specifies the icon for the plugin. You can use a non-interlaced PNG file of 32 pixels wide by 32 pixels high. Transparency can be used in the icon file.

As the following code line shows, this class implements the `StepMetaInterface`. All the defaults are conveniently implemented for you in the `BaseStepMeta` base class. Subclassing this class will allow you to focus on those methods that you want to implement and ignore all others. For example SQL generation is not relevant for this step.

```
public class HelloworldStepMeta extends BaseStepMeta
  implements StepMetaInterface {
```

The PKG variable marks the location of the `messages` package in which the internationalization (i18n) bundles are located. The `BaseMessages.getString()` occurrences that you'll notice in the code examples in the rest of this chapter look up translations in those bundles depending on the locale that is currently selected by the user. Also see the `$KETTLE_HOME/.kettle/.languageChoice` file for the actual language setting on your system or use the options dialog in Spoon. The PKG variable is typically located at the top of the class for use with the Translator developer GUI tool, which

allows translators to enter i18n strings for their locale. That is the reason why you'll see this construct come back in a lot of classes of the Kettle source code base.

```
private static Class<?> PKG = HelloworldStep.class; //for i18n
```

The field `fieldName` simply keeps track of the single parameter that the user can define: the name of the field that will contain the "Hello, world!" string.

```
  private String fieldName;

  /**
   * @return the fieldName
   */
  public String getFieldName() {
    return fieldName;
  }

  /**
   * @param fieldName the fieldName to set
   */
  public void setFieldName(String fieldName) {
    this.fieldName = fieldName;
  }
```

Next you verify that the user indeed specified the name of a field to use. Any needed remarks are added to the remarks list. (Note that it is common practice to give feedback on all the things that are verified and not just the things that go wrong. That way, the user has a better view of what was verified and what was ignored.)

```
 /**
   * checks parameters, adds result to List<CheckResultInterface>
   * used in Action > Verify transformation
   */
  public void check(List<CheckResultInterface> remarks,
      TransMeta transMeta, StepMeta stepMeta,
      RowMetaInterface prev, String input[], String output[],
      RowMetaInterface info) {

    if (Const.isEmpty(fieldName)) {
      CheckResultInterface error = new CheckResult(
        CheckResult.TYPE_RESULT_ERROR,
        BaseMessages.getString(PKG,"HelloworldMeta.CHECK_ERR_NO_FIELD"),
        stepMeta
      );
      remarks.add(error);
    } else {
      CheckResultInterface ok = new CheckResult(
        CheckResult.TYPE_RESULT_OK,
        BaseMessages.getString(PKG, "HelloworldMeta.CHECK_OK_FIELD"),
        stepMeta
```

```
        );
        remarks.add(ok);
      }
    }
```

As described previously, the `getStep()`, `getStepData()`, and `getDialogClassName()` methods simply provide the bridge to the other three interface classes of the step:

```
    /**
     *  creates a new instance of the step (factory)
     */
    public StepInterface getStep(StepMeta stepMeta,
        StepDataInterface stepDataInterface,
        int copyNr, TransMeta transMeta, Trans trans) {
      return new HelloworldStep(stepMeta, stepDataInterface,
           copyNr, transMeta, trans);
    }

    /**
     * creates new instance of the step data (factory)
     */
    public StepDataInterface getStepData() {
      return new HelloworldStepData();
    }

    public String getDialogClassName() {
      return HelloworldStepDialog.class.getName();
    }
```

The next four methods, `loadXML()`, `getXML()`, `readRep()`, and `saveRep()` simply allow the step to serialize the metadata as XML or as a set of attributes in a Kettle repository. It's perfectly fine to use whatever XML serialization technology, such as XStream (`http://xstream.codehaus.org/`), makes things easier for you.

```
public enum Tag {
    field_name,
  };

  /**
   * deserialize from xml
   * databases = list of available connections
   * counters = list of sequence steps
   */
  public void loadXML(Node stepDomNode, List<DatabaseMeta> databases,
        Map<String, Counter> sequenceCounters
      ) throws KettleXMLException {
    fieldName = XMLHandler.getTagValue(stepDomNode, Tag.field_name.name());
  }
```

```
public String getXML() throws KettleException {
  StringBuilder xml = new StringBuilder();
  xml.append(XMLHandler.addTagValue(Tag.field_name.name(), fieldName));
  return xml.toString();
}

/**
 * De-serialize from repository (see loadXML)
 */
public void readRep(Repository repository, ObjectId stepIdInRepository,
      List<DatabaseMeta> databases,
      Map<String, Counter> sequenceCounters
    ) throws KettleException {

  fieldName = repository.getStepAttributeString(
     stepIdInRepository,
     Tag.field_name.name()
     );
}

/**
 * serialize to repository
 */
public void saveRep(Repository repository, ObjectId idOfTransformation,
    ObjectId idOfStep) throws KettleException {
  repository.saveStepAttribute(idOfTransformation, idOfStep,
        Tag.field_name.name(), fieldName);
}
```

The `setDefault()` method simply sets default values on the various step parameters:

```
/**
 * initiailize parameters to default
 */
public void setDefault() {
  fieldName = "helloField";
}
```

The `getFields()` method is very important because it describes exactly what each output row of the step is going to look like. It needs to modify the `inputRowMeta` object to make it match the desired output. This description of the output row is also used to explain to Spoon and subsequent steps what values are to be expected from this step. In the most common scenario, you'll be adding `ValueMetaInterface` objects to the output `RowMetaInterface`. In general, it's important to add as much information about these values as possible, including type, length and precision, format mask, and so on. The more metadata you add, the more accurate things such as SQL generation will be in the user interface.

```
public void getFields( RowMetaInterface inputRowMeta, String name,
    RowMetaInterface[] info, StepMeta nextStep,
```

```
        VariableSpace space) throws KettleStepException {
      String realFieldName = fieldName;
      ValueMetaInterface field =
            new ValueMeta(realFieldName, ValueMetaInterface.TYPE_STRING);
      field.setOrigin(name);
      inputRowMeta.addValueMeta(field);
    }
  }
```

Value Metadata

Value metadata describes a single field value in a row of data that is handled by a step in a transformation. This description takes the form of the `ValueMetaInterface` interface. This interface can explain to you what the name is of a value, what data type it has, what the length and precision is, and so on. The following example creates a date value:

```
ValueMetaInterface dateMeta = new ValueMeta(
   "birthdate",
   ValueMetaInterface.TYPE_DATE
   );
dateMeta.setConversionMask("yyyy/MM/dd");
```

The interface is also capable of data conversion to another value metadata description. In fact, we recommend that all your data conversion be done using the `ValueMetaInterface` interface. For example, if you have a birth date and you want to convert it to a string using the format specified in the `dateValue` object from the example, you can use the following code:

```
// java.util.Date birthDate

String birthDateString = dateMeta.getString(birthDate);
```

The `ValueMeta` class will take care of the conversion. As such, the safest thing you can do is not make any decisions yourself about the type of data you're getting from previous steps because data conversion will take place if needed through the value metadata interface.

One specific example of the use of the `ValueMetaInterface` interface is the consideration if a value is null or not. In a previous step, you received an object and a `ValueMetaInterface` that describes this object. While it is inviting to simply check whether or not the object is `null`, it would likely be wrong in a lot of cases. For example:

- The object is a String that has ten spaces in it, and the value metadata requires the trimming of the string.
- The value metadata describes the object as being lazily loaded from a text file. As such, the object consists of a byte array (raw data) that first needs to be converted to a character sequence and then to the appropriate data type before it can be evaluated. More information on the details behind lazy loading and lazy conversion can be found in Chapter 15.

To check for null, you should use the following expression:

```
boolean n = valueMeta.isNull(valueData);
```

Important: Make sure that the data you pass into the `ValueMetaInterface` corresponds to the data type described in the metadata. Table 23-1 lists metadata types and their corresponding Java data types.

Table 23-1: Java Primitives to Correspond with Metadata Types

VALUE META TYPE	JAVA CLASS
ValueMetaInterface.TYPE_STRING	java.lang.String
ValueMetaInterface.TYPE_DATE	java.util.Date
ValueMetaInterface.TYPE_BOOLEAN	java.lang.Boolean
ValueMetaInterface.TYPE_NUMBER	java.lang.Double
ValueMetaInterface.TYPE_INTEGER	java.lang.Long
ValueMetaInterface.TYPE_BIGNUMBER	java.math.BigDecimal
ValueMetaInterface.TYPE_BINARY	byte[]

Row Metadata

Row metadata describing not just a value but complete rows of values are represented by the interface `RowMetaInterface`. In essence, a `RowMetaInterface` class simply contains a list of `ValueMetaInterface` classes. It contains a series of methods that allow you to manipulate the row metadata, look up values, check for existence of values, replace value metadata entries, and so on.

The only rule is that the name of a value needs to be unique in a row. Contrary to what was the case in version 2 of Kettle, this rule is enforced in versions 3 and 4. When you add a value with the same name twice to a row, the second occurrence will be automatically renamed. The name will have the text `_2` appended to it. Adding it once more will add `_3` to the name of the value, and so on.

Because we're usually working with rows of data in steps, it's more convenient to not deal with individual values. You can use a number of methods such as `getNumber()` and `getString()` to reach values directly in a row. If the sales figure is stored as the fourth value in a row in a step, you could, for example, write the following expression to obtain it:

```
Double sales = getInputRowMeta().getNumber(rowData, 3);
```

Obtaining a value by index is always the fastest way of getting values from rows. You can use the `indexOfValue()` method to look up the index of a field value in a row. This method scans the list of values one by one and is not very fast. Because of this, we recommend that you only do the lookup once for all rows that you need to process in

your step plugin. As described earlier in the step plugin section, you can handle this in the "first" block right after you received the first row from a previous step.

StepDataInterface

The class that implements `org.pentaho.di.trans.step.StepDataInterface` takes care of maintaining the execution status of the step. You can also use this class to store your temporary objects. Typically, you store the output row metadata in it as a public member as well as any open database connections, input or output streams, and so on.

StepDialogInterface

The class that implements `org.pentaho.di.trans.step.StepDialogInterface` makes it possible for users to enter the metadata (parameters) of a step with a graphical user interface. The form of the user interface element is a dialog. The dialog is presented when the user edits a step in Spoon. The interface is not that complex because it contains only the `open()` method and `setRepository()` for convenience.

Eclipse SWT

Because the Eclipse SWT was selected to develop UIs within Kettle, you have to use that to program your dialogs. SWT provides an abstraction layer around the native operating system toolkits for the various Windows, OS X, Linux, and UNIX variants. It's because of this that SWT applications usually have a look that blends in nicely with the rest of the operating system.

To get started with SWT programming, we recommend that you take a look around the SWT home page at `http://www.eclipse.org/swt/` to get acquainted. Here are a few interesting pages to give you a good idea of what can be done with SWT programming:

- The SWT widgets page, `http://www.eclipse.org/swt/widgets/`, gives a nice overview of all the available widgets you can use.
- The SWT snippets page, `http://www.eclipse.org/swt/snippets/`, provides common samples with Java code.

The best source for information, however, is the source code of the more than 150 internal step dialogs in the Kettle source code.

Form Layout

If you take a look at dialog classes in Kettle, note that they usually contain a lot of verbose code. This is to ensure that all the dialogs present themselves as nicely as possible and render correctly on all possible operating systems with all possible screen and font sizes. It's because of this that most of the code you'll find will have to do with the layout and position of the widgets in the dialogs.

The layout system you'll encounter most often in the Kettle codebase is the FormLayout system. FormLayout works by allowing the programmer to specify relative positions of widgets and work with percentages and offsets. Take a look at a simple example (you can also find this Java code in download file `HelloworldStepDialog.java`):

```
Label label =new Label(shell, SWT.RIGHT);
label.setText("Hello");
props.setLook(wlStepname);
FormData fdLabel=new FormData();
fdLabel.left = new FormAttachment(0, 0);
fdLabel.right= new FormAttachment(50, -10);
fdLabel.top = new FormAttachment(0, 25);
label.setLayoutData(fdLabel);
```

This is code you'll see often so we'll explain it in detail. The first line creates a new Label widget inside of a shell (dialog). The alignment of the text on the widget is right:

```
Label label =new Label(shell, SWT.RIGHT);
```

Next we put the string `"Hello"` in the label:

```
label.setText("Hello");
```

The next line puts the user-selected background color and font on the widget:

```
props.setLook(wlStepname);
```

The next few lines position the left side of the label at the far left of the dialog and the right side of the label at 10 pixels to the left of the middle of the dialog (50%). The top of the label is at positioned 25 pixels below the top of the shell.

```
FormData fdLabel=new FormData();
fdLabel.left = new FormAttachment(0, 0);
fdLabel.right= new FormAttachment(50, -10);
fdLabel.top = new FormAttachment(0, 25);
label.setLayoutData(fdLabel);
```

Instead of providing a percentage as the first argument in the `FormAttachment` constructor, you can also specify another widget (called a control in SWT). This will allow you to specify the location of one widget relative to another.

If you don't specify the bottom of the widget, SWT will pick the natural height of the widget, depending on the selected font. That in turn means that if you want to place a Text widget right below the label with a 10 pixel gap, you can do it like this:

```
fdLabel2.top = new FormAttachment(label, 10);
```

In short: *don't panic*; while good user interface code is usually long and verbose, it tends to be rather simplistic in nature.

Kettle UI Elements

Beyond the standard SWT widgets, you can also use one of the specific Kettle widgets that were created to make the life of the ETL developer a bit easier. Here is a short overview of the possibilities:

- `TableView`: This convenient data grid widget has support for sorting, selecting, keyboard shortcuts, and undo/redo, and has a right-click menu on top of it.
- `TextVar`: This textbox supports variables and is decorated with the $ symbol in the upper-right corner. The user can enter variables with Ctrl+Spacebar. If these keys are used inside the textbox you'll be able to pick a variable from a pop-up window. This variable will then be inserted at the cursor location. Other than that, it's compatible with the standard Text widget.
- `ComboVar`: A standard combo box with support for variables.
- `ConditionEditor`: A widget to enter conditions as found in the Filter Rows step.

There is also a series of additional dialogs at your disposal to help you with various tasks:

- `EnterListDialog`: Allows you to select one or more items from a list of strings. It shows the list on the left and the selected items on the right. Buttons will then allow you to move items in or out of the selection.
- `EnterNumberDialog`: Allows the user to enter a number.
- `EnterPasswordDialog`: Asks the user for a password.
- `EnterSelectionDialog`: Selects items in a list by highlighting them.
- `EnterMappingDialog`: Enters the mapping between two sets of strings.
- `PreviewRowsDialog`: Previews a list of rows in a dialog.
- `SQLEditor`: Shows a simple SQL editor that allows you to enter queries and DLL.
- `ErrorDialog`: The constructor displays an exception with a message and can list the stack trace details.

Hello World Example Dialog

Now that you've had a general look at SWT and how it handles the layout of a dialog, it's time to take a closer look at a practical example. The following Java code can also be found in download file `HelloworldStepDialog.java`.

The first part of the code merely makes the metadata class available to the rest of the dialog class and initializes the base step dialog:

```
public class HelloworldStepDialog extends BaseStepDialog implements
    StepDialogInterface {

  private static Class<?> PKG = HelloworldStepMeta.class;
```

```
private HelloworldStepMeta input;

private TextVar wFieldname;

public HelloworldStepDialog(Shell parent, Object baseStepMeta,
    TransMeta transMeta, String stepname) {
  super(parent, (BaseStepMeta)baseStepMeta, transMeta, stepname);
  input = (HelloworldStepMeta)baseStepMeta;
}
```

The next `open()` method is where all the widgets for the dialog are created and assembled. SWT uses the listener design pattern a lot. You can add all kinds of listeners to widgets to see if the content has changed or if the user performed an action. In this situation, you want to know if the user changed anything in one of the widgets so that you can mark the step and transformation as changed and in need of saving.

```
public String open() {
  Shell parent = getParent();
  Display display = parent.getDisplay();

  shell = new Shell(parent, SWT.DIALOG_TRIM | SWT.RESIZE | SWT.MIN |
      SWT.MAX);
   props.setLook(shell);
   setShellImage(shell, input);

  ModifyListener lsMod = new ModifyListener()
  {
    public void modifyText(ModifyEvent e)
    {
      input.setChanged();
    }
  };
  changed = input.hasChanged();
```

The next part of the code indicates that you want to position the widgets in the dialog shell with the form layout method:

```
  FormLayout formLayout = new FormLayout ();
  formLayout.marginWidth = Const.FORM_MARGIN;
  formLayout.marginHeight = Const.FORM_MARGIN;

  shell.setLayout(formLayout);
```

The right side of all the labels is available as a user-defined percentage: `props.getMiddlePct()`. The margin between widgets was made a constant and is 4 pixels at the time of this writing.

```
  shell.setText(
    BaseMessages.getString(PKG,"HelloworldDialog.Shell.Title"));
```

```
int middle = props.getMiddlePct();
int margin = Const.MARGIN;
```

The next block of code simply adds a label and a text input field on one line at the top of the dialog shell:

```
// Stepname line
wlStepname=new Label(shell, SWT.RIGHT);
wlStepname.setText(
  BaseMessages.getString(PKG, "HelloworldDialog.Stepname.Label"));
props.setLook(wlStepname);
fdlStepname=new FormData();
fdlStepname.left = new FormAttachment(0, 0);
fdlStepname.right= new FormAttachment(middle, -margin);
fdlStepname.top = new FormAttachment(0, margin);
wlStepname.setLayoutData(fdlStepname);
wStepname=new Text(shell, SWT.SINGLE | SWT.LEFT | SWT.BORDER);
wStepname.setText(stepname);
props.setLook(wStepname);
wStepname.addModifyListener(lsMod);
fdStepname=new FormData();
fdStepname.left = new FormAttachment(middle, 0);
fdStepname.top = new FormAttachment(0, margin);
fdStepname.right= new FormAttachment(100, 0);
wStepname.setLayoutData(fdStepname);
Control lastControl = wStepname;
```

Next you add the widgets that will allow you to enter a name for the output field of the “Hello World” step:

```
// Fieldname line
Label wlFieldname = new Label(shell, SWT.RIGHT);
wlFieldname.setText(
  BaseMessages.getString(PKG, "HelloworldDialog.Fieldname.Label"));
props.setLook(wlFieldname);
FormData fdlFieldname = new FormData();
fdlFieldname.left = new FormAttachment(0, 0);
fdlFieldname.right= new FormAttachment(middle, -margin);
fdlFieldname.top = new FormAttachment(lastControl, margin);
wlFieldname.setLayoutData(fdlFieldname);
wFieldname =
  new TextVar(transMeta, shell, SWT.SINGLE | SWT.LEFT | SWT.BORDER);
props.setLook(wFieldname);
wFieldname.addModifyListener(lsMod);
FormData fdFieldname = new FormData();
fdFieldname.left = new FormAttachment(middle, 0);
fdFieldname.top = new FormAttachment(lastControl, margin);
fdFieldname.right= new FormAttachment(100, 0);
wFieldname.setLayoutData(fdFieldname);
lastControl = wFieldname;
```

Next, you create two push buttons for OK and Cancel, add listeners to them, and position them at the bottom of the dialog after the last input field:

```
// Some buttons
wOK=new Button(shell, SWT.PUSH);
wOK.setText(BaseMessages.getString(PKG, "System.Button.OK"));
wCancel=new Button(shell, SWT.PUSH);
wCancel.setText(
  BaseMessages.getString(PKG, "System.Button.Cancel"));

setButtonPositions(
  new Button[] { wOK, wCancel }, margin, lastControl);

// Add listeners
lsCancel = new Listener() { public void handleEvent(Event e) {
 cancel(); } };
lsOK = new Listener() { public void handleEvent(Event e) {
  ok();  } };

wCancel.addListener(SWT.Selection, lsCancel);
wOK.addListener (SWT.Selection, lsOK );
```

The next lines of code make sure that special cases are handled. First you make sure that the dialog is closed affirmatively when the user presses Enter in one of the text input fields. Then you make sure to cancel the dialog input when the shell is closed without using either the Cancel or OK buttons:

```
lsDef=new SelectionAdapter() {
public void widgetDefaultSelected(SelectionEvent e) { ok(); }};

wStepname.addSelectionListener( lsDef );
wFieldname.addSelectionListener( lsDef );

// Detect X or ALT-F4 or something that kills this window...
shell.addShellListener(new ShellAdapter() {
  public void shellClosed(ShellEvent e) {
    cancel();
} } );
```

You copy the data from the input metadata to the widgets in the dialog:

```
// Populate the data of the controls
//
getData();
```

The size and position of the dialog are automatically determined based on the natural size of the dialog (minimum), the previous location and size of the dialog (where the user left it), and the size of the display(s):

```
      // Set the shell size, based upon previous time...
      setSize();

      input.setChanged(changed);

      shell.open();
      while (!shell.isDisposed())
      {
          if (!display.readAndDispatch()) display.sleep();
      }
      return stepname;
    }
```

When you put data into the widgets, make sure to never put null values in them because they don't support it. The static method `Const.NVL()` will help you to take care of that situation:

```
    /**
     * Copy information from the meta-data input to the dialog fields.
     */
    public void getData()
    {
      wFieldname.setText(Const.NVL(input.getFieldName(), ""));
      wStepname.selectAll();
    }
```

Finally, when the user selects OK, you copy the information from the dialog back into the input metadata class:

```
    private void cancel()
    {
      stepname=null;
      input.setChanged(changed);
      dispose();
    }

    private void ok()
    {
      if (Const.isEmpty(wStepname.getText())) return;

      stepname = wStepname.getText(); // return value

      input.setFieldName(wFieldname.getText());

      dispose();
    }
  }
```

StepInterface

The class that implements the interface `org.pentaho.di.trans.step.StepInterface` is responsible for processing rows of data in any shape or form while using the parameters of metadata described in the associated `StepMetaInterface` described previously. It will read data from zero, one, or more input steps and pass data to zero, one, or more output steps. A number of the methods that are described in the interface are used by the transformation engine to do housekeeping. However, there are only a few methods you would likely override from the `BaseStep` class where all methods are implemented:

- `init()`: To initialize a step, you can use an `init()` method. The result of the initialization is a Boolean true or false. If everything you wanted to do during the initialization went okay, you return `true`, otherwise `false`. If you don't have anything to initialize, you can opt not to override this method.
- `dispose()`: If there are things that need to be cleaned up, you can do it in the `dispose()` method. For example, in this method you can close database connections and files, clear out memory buffers, and so on. This method will always be called at the end of the transformation. If you don't have anything to dispose of, you can opt not to override this method.
- `processRow()`: This method is where the actual work is taking place. The transformation engine will call this method repeatedly. As long as the method returns `true` it will continue to be called in a loop.

Here is how the Hello World step uses the `StepInterface` (you can also find this Java code in the download file `HelloworldStep.java`):

```
package org.kettlesolutions.plugin.step.helloworld;

… /* imports removed for brevity */

public class HelloworldStep extends BaseStep implements StepInterface {
```

As explained earlier, you use the `BaseStep` class to allow you to focus on the important methods. It contains a lot of useful methods that can be used inside of the step.

The constructor usually simply passes its argument to the `BaseStep` class. Arguments will be handled appropriately in that constructor. Because of this you can conveniently use objects like `transMeta` in all steps.

```
  public HelloworldStep(StepMeta stepMeta,
      StepDataInterface stepDataInterface,
      int copyNr, TransMeta transMeta, Trans trans) {
    super(stepMeta, stepDataInterface, copyNr, transMeta, trans);
  }
```

The `getRow()` method tries to obtain a row from a previous input step. If there are no more rows to receive, the method will return `null`. If the previous steps can't keep

up, the method will block until a row is made available. This will effectively throttle down this step to match the speed of the other steps in the transformation. See Chapter 15 for more information on this subject.

```
    public boolean processRow(StepMetaInterface smi, StepDataInterface sdi
        ) throws KettleException {

      HelloworldStepMeta meta = (HelloworldStepMeta) smi;
      HelloworldStepData data = (HelloworldStepData) sdi;

      Object[] row = getRow();
      if (row==null) {
        setOutputDone();
        return false;
      }

      if (first) {
        first=false;
        data.outputRowMeta = getInputRowMeta().clone();
        meta.getFields(data.outputRowMeta,
         getStepname(), null, null, this);
      }

      String value = "Hello, world!";

      Object[] outputRow = RowDataUtil.addValueData(row,
       getInputRowMeta().size(), value);

      putRow(data.outputRowMeta, outputRow);

      return true;
    }
  }
```

For performance reasons, the `getRow()` call does not give you the row metadata, only the data associated with the output of the previous step. That metadata is available using the `getInputRowMeta()` method but *only* after the first row has been received from the `getRow()` method.

The `setOutputDone()` call takes care of informing any of the following steps that there is no more output to be expected from this step. That, in turn, will allow the `getRow()` method of the next steps to return `null` for the call when the buffer between the steps is empty. By returning `false`, the transformation knows not to call the `processRow()` method of your step again.

```
  Object[] row = getRow();
  if (row==null) {
  setOutputDone();
    return false;
  }
```

The method that passes rows to the next steps is called `putRow()`. You need to pass the `RowMetaInterface` of the output along with the data to describe it in this method. If you remember from the previous section, you already have a method that calculates the output row metadata:

```
data.outputRowMeta = getInputRowMeta().clone();
meta.getFields(data.outputRowMeta, getStepname(), null, null, this);
```

Note that you make a copy with `clone()` of the input row metadata to make sure that you are absolutely not influencing any data structure in a previous step as that could lead to incorrect results during execution. The calculation of the output row metadata is something that needs to happen once and only once because the layout of all the output rows needs to be the same. Calculations that only need to happen once are usually placed in a "first" block. A member called `first` is conveniently set to `true` before execution in the `BaseStep` class so you can use that.

The last part of the code adds a string value to the input row and passes it back to the next steps:

```
String value = "Hello, world!";

Object[] outputRow = RowDataUtil.addValueData(row,
       getInputRowMeta().size(), value);

putRow(data.outputRowMeta, outputRow);
```

For pure performance, rows of data are simple Java arrays. For your convenience, you can use the static methods from the `RowDataUtil` class to manipulate the values in them. Note that there is usually excess space in the rows that are passed to prevent copying of data. Again this is done for performance reasons. If you use the static `RowDataUtil` methods, you will have no problems with that at all.

Reading Rows from Specific Steps

If you have a need to read data from specific (info) steps—for example, the Stream Lookup step—you can use the `getRowFrom()` method in a step:

```
RowSet rowSet = findInputRowSet(Source Step Name);
Object[] rowData=getRowFrom(rowSet);
```

The row metadata from those rows is available in the `rowSet` object:

```
RowMetaInterface rowMeta = rowSet.getRowMeta();
```

Writing Rows to Specific Steps

If you want to write data to specific (target) steps, as in the Filter Rows step, you can do this with the `putRowTo()` method:

```
RowSet rowSet = findOutputRowSet(Target Step Name);
```

```
...
putRowTo(outputRowMeta, rowData, rowSet);
```

Obviously, it's most efficient to obtain a handle on the input or output `RowSet` objects just once.

Writing Rows to Error Handling

If you want to allow your step to support error handling, and if your metadata class returns `true` for the `supportsErrorHandling()` method, you can write rows to the error handling channel. The following is a simple example of how to do that with the `putError()` method:

```
Object[] rowData = getRow();
...
try {
 ...
 putRow(...);
} catch(Exception e) {
 if (getStepMeta().isDoingErrorHandling()) {
 putError(getInputRowMeta(), rowData, 1, e.getMessage(),
    errorMessage, errorFieldname, errorCode);
 } else {
  throw(e);
 }
}
```

As shown in the example, you can pass an error count, field, message, and code to the step that will handle the errors of this step. Everything else regarding the error handling is handled automatically.

Identifying a Step Copy

Because a step can be executed in multiple copies, it can be useful to identify which step copy you're using. Here are a few interesting methods you can use:

- `getCopy()`: Returns the copy number of the step. Copy numbers uniquely identify a particular step copy within the collection of copies of a step, and are in the range 0...N where N is equal to `getStepMeta().getCopies() - 1`
- `getUniqueStepNrAcrossSlaves()`: The step copy number in a clustered execution (also works for local execution)
- `getUniqueStepCountAcrossSlaves()`: The total number of step copies running on a cluster

These methods can then be used to divide the work among the various step copies. For an example of this, see the source code for the "CSV file input" or "Fixed file input" steps. Both have the option to read text files in parallel by dividing the work among the step copies locally or across a cluster.

Result Feedback

To signal in the user interface or in the logging how many rows were processed, there are two metrics that are automatically incremented with the use of the `getRow()` and `putRow()` methods: the number of rows read and written, respectively. The following are a few methods to influence the metrics of a step:

- `incrementLinesRead()`: Increase the number of lines read from previous steps.
- `incrementLinesWritten()`: Increase the number of lines written to subsequent steps.
- `incrementLinesInput()`: Increase the number of lines read from files, databases, network, and so on.
- `incrementLinesOutput()`: Increase the number of lines written to files, databases, network, and so on.
- `incrementLinesUpdated()`: Increase the number of lines updated.
- `incrementLinesSkipped()`: Increase the number of lines skipped.
- `incrementLinesRejected()`: Increase the number of lines rejected.

These metrics are used to see how much work a step has done. They are visible in the transformation metrics panel in Spoon and can be logged to a database logging table as well. See Chapter 14 for more information on this topic.

It is also possible to keep track of the files that have been used by this step. You can use the method `addResultFile()` for that purpose. The list of result files can then be used by other transformations and job entries in a job. For example, take a look at this code from the "CSV file input" step:

```
ResultFile resultFile = new ResultFile(
  ResultFile.FILE_TYPE_GENERAL,
  fileObject,
  getTransMeta().getName(),
  getStepName()
 );
resultFile.setComment("File was read by a Csv input step");
addResultFile(resultFile);
```

Variable Substitution

When you have an input field that supports variable substitution (see Chapter 2 for more information), you can obtain the actual value of a variable at run-time with the `environmentSubstitute()` method. This method is available in all objects in Kettle that implement the `VariableSpace` interface. Those objects include steps, job entries, transformations, jobs, databases, and so on. For example, if you would like to support the use of variables to specify the name of your output field in the "Hello World"

example, you could change your example like this in the `getFields()` method of the `StepMetaInterface` class:

```
String realFieldName = space.environmentSubstitute(fieldName);
```

Because a step itself is a `VariableSpace` object, you can simply do the variable substitution method directly like this:

```
String value = environmentSubstitute(meta.getStringWithVariables());
```

Apache VFS

The goal of all steps in Kettle that deal with files in some form or another is to use the Apache VFS system. Apache VFS, as described in Chapter 2 in more detail, allows you to read data from not just files, as is the case with the `java.io.File` object, but from a lot of different locations, including web and FTP servers, `.zip` archives, and many more.

Apache VFS introduces the `FileObject` class that provides the abstraction layer for you. Kettle in turn creates a series of static methods in the `KettleVFS` class to make it easy for you to get your hands on `FileObject` classes for your file names. Take a look at an example:

```
FileObject fileObject = KettleVFS.getFileObject(
  "zip:http://www.example.com/archive.zip!file.txt"
  );
```

It's important that you use the `KettleVFS` wrapper as much as possible because it contains fixes and workarounds for certain issues in the Apache VFS layer. It also contains enhancements for the SFTP protocol, as explained in Appendix C.

Step Plugin Deployment

To deploy your step plugin, you have to compile the four Java classes and any other classes it depends on. Then you should place the compiled classes in a `.jar` file. You can use your IDE to do that for you (see the export functionality in Eclipse, for example), or you can manually write an Apache Ant build file to do it automatically.

The `.jar` file itself should be placed in the `plugins/steps` directory in your binary Kettle distribution. If you like, you can use a subdirectory. It is also possible to add all dependent files in a `lib/` folder where the plugin `.jar` file is located. There is no need to place them all in the common Kettle class path (the `libext/` directory) as these extra libraries will be picked up automatically by Kettle. You can place multiple step plugins in a single `.jar` file.

If you want to debug your plugin in your IDE, you can set the name of your metadata class in variable `KETTLE_PLUGIN_CLASSES` (a comma-separated list). For more information on this topic, consult the Pentaho wiki at `http://wiki.pentaho.com/display/EAI/How+to+debug+a+Kettle+4+plugin`.

The User-Defined Java Class Step

The User Defined Java Class step, UDJC for short, is a new step in version 4 of Kettle. It allows you to place the Java code program as a step plugin inside of the UDJC dialog. The code you enter will then be compiled and executed at run-time. This has the flexibility and power of a scripting language while maintaining the optimal speed of native Java code execution.

If you've read the previous section of the chapter regarding the development of step plugins, writing a UDJC is easy. The `init()`, `dispose()`, and `processRow()` methods covered can be used in exactly the same fashion in a UDJC. In fact, almost everything you've learned about writing a plugin can be applied. The main differences are the lack of metadata class and the lack of a fancy dialog to enter metadata.

Passing Metadata

Without any method of parameterization, the Java code written in a UDJC step would be inflexible and hard to maintain. That is why you can enter parameters tags and values in the Parameters tab at the bottom of the dialog. You can obtain the value for a tag in your UDJC class like this:

```
String value = getParameter("TAG");
```

Accessing Input and Fields

The UDJC includes a convenient `get()` method to make it easy to access input and output fields. Take a look at the following example that reads a `String` value from a row of data:

```
String name = get(Fields.In, "last_name").getString(rowData);
```

The same is possible for setting values in output fields that were specified in the Fields section at the bottom of the dialog:

```
get(Fields.Out, "book_name").setValue(r, "Kettle Solutions");
```

Snippets

The UDJC dialog contains a Code Snippets section in the tree on the left. You can double-click on the snippets to insert them into your code. They are basic examples showing you the various things you can do with a step.

Example

The following code performs the same function as the four classes described earlier: it simply adds a single field that contains the string "Hello, World!"You can also find

this code as part of the transformation UDJC sample.ktr in the download package for this chapter.

```
int outputRowSize;

public boolean processRow(StepMetaInterface smi, StepDataInterface sdi
             ) throws KettleException
{
  Object[] r = getRow();
  if (r == null) {
    setOutputDone();
    return false;
  }

  if (first) {
    first = false;
    outputRowSize = data.outputRowMeta.size();
  }

 r = createOutputRow(r, outputRowSize);
 get(Fields.Out, "hello").setValue(r, "Hello, World!");

 // Send the row on to the next step.
 putRow(data.outputRowMeta, r);

 return true;
}
```

As you can see, the code looks similar to the code you wrote in the `StepInterface` class of the "Hello World" step plugin shown previously. There is only one output field, called `"hello"`, specified in the Fields tab at the bottom of the UDJC step. Compared to a dialog designed for a specific set of parameters, editing the metadata is obviously less fancy inside the Java code or in the UDJC dialog in general. However, compared to writing a plugin, it takes a lot less code and time to get a solution in place.

Job Entry Plugins

Writing a job entry plugin is very similar to writing a step plugin. The main difference is that you only have to implement two interfaces instead of four:

- `JobEntryInterface`: This will take care of the metadata and execution of the job entry.
- `JobEntryDialogInterface`: Almost exactly like the step dialog in the way that it allows you to edit the metadata of the job entry.

Steps run in parallel and job entries run in sequence so there is no separation between the metadata and the execution classes in a job entry. This simplifies things a bit.

JobEntryInterface

The class that implements the `org.pentaho.di.job.entry.JobEntryInterface` interface needs to implement one important method to get the job done:

- `execute()`: This executes the job entry and returns the Result object. (See Chapter 22 for more information on the `Result` object.)

Quite a few other methods were covered earlier in the step plugin development section. They work in exactly the same way for the `JobEntryInterface`:

- `getXML()` and `loadXML()` to serialize to XML
- `saveRep()` and `readRep()` to serialize to a repository
- `getSQLStatements()` to generate SQL
- `getUsedDatabaseConnections()` to see which connections are used by the job entry
- `check()` to validate the metadata
- `getResourceDependencies()` to list dependencies
- `exportResources()` to allow linked resources (jobs and transformations, for example) to be exported
- `getDialogClassName()` to know which dialog to use to edit the metadata

In addition, two other methods are occasionally used:

- `boolean evaluates()`: Determines whether or not this step is capable of making an evaluation. Except for the rare exception (such as the Start job entry), this should always return `true`.
- `boolean isUnconditional()`: Explains to the system whether or not the job entry can allow unconditional continuation after execution. Except for the rare exception, this should also return `true`.

Let's take a look a simple "Hello World" job entry. This job entry allows the user to specify a Boolean value: `true` or `false`. If the user specifies `true`, then the job will continue to follow the green (success) job entry hop; if not, the red hop. It mimics the success or failure of a job entry.

The `@JobEntry` annotation works exactly like the `@Step` annotation described previously. It allows the plugin registry to recognize this class as a job entry plugin (you can find this code also in download file `HelloworldJobEntry.java`):

```
package org.kettlesolutions.plugin.jobentry.helloworld;

...  /* imports removed for brevity */

@JobEntry(
    id="Helloworld",
    name="HelloworldJobEntry.name",
    description="HelloworldJobEntry.description",
```

```
    categoryDescription="HelloworldJobEntry.category",
    i18nPackageName="org.ketttlesolutions.plugin.jobentry.helloworld",
    image="org/kettlesolutions/plugin/step/helloworld/HelloWorld.png"
    )
```

For your convenience you can subclass the `JobEntryBase` class to allow you to implement only those methods you actually need:

```
public class HelloworldJobEntry extends JobEntryBase
  implements JobEntryInterface {
```

Again, much like the step plugin, the class contains the actual metadata (parameters) and you have getters and setters to handle those:

```
  private boolean success;

  /**
   * @return the success
   */
  public boolean isSuccess() {
    return success;
  }

  /**
   * @param success : the success flag to set
   */
  public void setSuccess(boolean success) {
    this.success = success;
  }
```

The result of any type of execution is always going to be a `Result` object. This `Result` object is passed to the next job entry:

```
  public Result execute(Result prevResult, int nr)
    throws KettleException {
    prevResult.setResult(success);
    return prevResult;
  }
```

The only thing to keep in mind in the `execute()` method is to always make sure to use the same `Result` object. Make sure to modify it as shown in the example instead of creating a new one.

```
  public enum Tag {
    success,
  };

  public void loadXML(Node entrynode, List<DatabaseMeta> databases,
      List<SlaveServer> slaveServers, Repository rep
     ) throws KettleXMLException {
    super.loadXML(entrynode, databases, slaveServers);
```

```
      success = "Y".equalsIgnoreCase(
      XMLHandler.getTagValue(entrynode, Tag.success.name()));
    }

    public String getXML() {
      StringBuilder xml = new StringBuilder();
      xml.append(super.getXML());
      xml.append(XMLHandler.addTagValue(Tag.success.name(), success));
      return xml.toString();
    }

    public void loadRep(Repository rep, ObjectId idJobentry,
        List<DatabaseMeta> databases, List<SlaveServer> slaveServers
      ) throws KettleException {

      success =
        rep.getJobEntryAttributeBoolean(idJobentry, Tag.success.name());
    }

    public void saveRep(Repository rep, ObjectId idJob)
      throws KettleException {
      rep.saveJobEntryAttribute(idJob, getObjectId(),
        Tag.success.name(), success);
    }

  }
```

JobEntryDialogInterface

The `JobEntryDialog` interface is identical in many ways to the `StepDialogInterface` we extensively described in the step plugin section. As such, we'll only add here that you need to implement the `org.pentaho.di.job.entry.JobEntryDialogInterface` interface.

The only difference from the earlier interface is that the user edits a `JobEntryInterface` class and not a `StepMetaInterface` class. You can extend the `JobEntryDialog` class to get a good starting point for your job entry dialogs. As usual, don't forget to look at the many samples in the Kettle source code for ideas on how to construct your dialogs.

Deploying or debugging a job entry plugin happens in the exact same way as you saw with the step plugins. The only difference is that you should place the `.jar` file containing the plugins in the `plugins/jobentries` directory. All the other instructions are identical to those provided for the earlier dialog interface.

Partitioning Method Plugins

As described in Chapter 16, a partitioning method determines the partition to which a row of data belongs based on a set of rules. With a partition plugin, you can write your own set of rules for the grouping of rows.

To implement a partitioning method, you need to implement two interfaces: `Partitioner` and `StepDialogInterface`. (The latter is used for historical reasons and because the dialog interfaces usually contain only the `open()` method).

The only difference between this dialog and those covered earlier is that it will pass in the class that implemented the `Partitioner` interface instead of step or job entry metadata. Everything else about the dialog is the same as with the previous plugin types. For this reason, we won't cover the dialog here. For an example, look at file `HourPartitionerDialog.java` in the download package of this chapter or the `ModPartitionerDialog` class in the Kettle source code.

Partitioner

In addition to the usual four methods to serialize the metadata as XML or in a repository and `getDialogClassName()`, there is another main method that you need to implement to get a partitioning method to work: `getPartition()`. In the following code sample, you calculate the partition based on the last two digits before the extension of a file name. These digits represent the hour of the data for which the data in the file was generated. For example, if the file is called `data-17.txt`, it contains data from 5 to 6 p.m (the 18th hour in a 24 hour day). The `getPartition()` method should return partition 17 for this file. Since partition numbers start at zero this is the 18th partition.

The metadata of the partitioning method is the specification of the field name that contains the file name. This method initializes the number of available partitions in variable `nrPartitions` if this is not already done. This method comes from class `BasePartioner` which you subclass. You can find the next code samples in file `HourPartitioner.java` in the download package of this chapter.

```
public int getPartition(RowMetaInterface rowMeta, Object[] row
     ) throws KettleException {
  init(rowMeta);
```

For performance reasons you cache the location of the field to partition on, in this case this field will contain the file name. If a field with the specified name can't be found you throw an exception with an appropriate error message.

```
  if (partitionColumnIndex < 0) {
    partitionColumnIndex = rowMeta.indexOfValue(fieldName);
    if (partitionColumnIndex < 0) {
      throw new KettleStepException(
        BaseMessages.getString(PKG,
          "HourPartitioner.Exception.PartitioningFieldNotFound",
          fieldName,
          rowMeta.toString()));
    }
  }
```

Next you obtain the file name itself. Because data can never be trusted, you check that the input is indeed a `String` value. If this is not the case, you terminate the transformation by throwing an exception with a message that describes the problem:

```
ValueMetaInterface valueMeta =
  rowMeta.getValueMeta(partitionColumnIndex);
Object valueData = row[partitionColumnIndex];

if (!valueMeta.isString()) {
  throw new KettleException(BaseMessages.getString(PKG,
    "HourPartitioner.Exception.NotAFilename",
    valueMeta.getName()));
}
```

The last lines are responsible for the actual calculation of the partition number. It's important to return a number that is between 0 and the number of partitions minus one. To make sure that this is the case, it's convenient to use the modulo (remainder of division) calculation:

```
  String filename = valueMeta.getString(valueData);
  String hourString =
    filename.substring(filename.length()-6, filename.length()-4);
  int value = Integer.parseInt(hourString);
  int targetLocation = (int) (value % nrPartitions);

  return targetLocation;
}
```

Again, because of the presence of the uniform plugin registry, deploying a partitioning method is similar to what you've seen with the other plugins. For historical reasons, you have to deploy the `.jar` file in the same directory as the step plugins: `plugins/steps`.

Repository Type Plugins

Over the course of the last decade, new software configuration management (SCM) and content management systems (CMSs) have become available to a wide audience in the form of low-cost and open source software. Examples of popular open source SCM software titles include Concurrent Versioning System (CVS), Subversion, and Git. In the area of CMS, we can list systems such as JCR (Java Content Repository), CMIS (Content Management Interoperability Services) or even the new Google Wave protocol. Because all the listed systems would be suitable for storage and retrieval of Kettle metadata, the Kettle development team deemed it appropriate to create an abstraction layer that will allow you to create your own implementation of a repository on any of these SCM or CMS systems.

`Repository` is the name of the main interface to implement for this plugin type. It covers everything ranging from serialization of transformations to the specification of security providers. With such a wide range of topics to cover, this plugin type goes far beyond the interest range of most ETL developers. However, if you are interested in

writing a repository type plugin for Kettle, there are again samples to look at. First take a look at the Kettle Database Repository (see also the `KettleDatabaseRepository` class). That repository can serialize Kettle metadata to a relational database schema. Then you can look at the File repository (see also the `KettleFileRepository` class). It provides a repository interface layer around an Apache VFS file location such as a folder on disk or a zip file. Because of the simplicity of the File Repository we advise you to take this code as a starting point for your studies on the subject.

Because the repository type plugin system is in use for the Pentaho Enterprise Repository, you know that the capabilities of these types of plugins include advanced version management, file locking, and security. However, considerable effort is likely needed to implement all these features.

To implement advanced functionality such as file locking without this being explicitly present in the `Repository` interface, Pentaho allows the registration of services that can extend the standard capabilities. For more information, see this page on the Pentaho wiki: `http://wiki.pentaho.com/display/EAI/Registering+Services+to+the+Repository+in+Kettle`.

Finally, repository type plugins can be deployed to the `plugins/repositories` folder and will be recognized when they are annotated with the `@RepositoryPlugin` annotation.

Database Type Plugins

Relational and column store databases are in a constant state of flux. New contenders to the crown and new versions are appearing all the time, making the maintenance of the database abstraction layer not an easy task. The goal of that layer is to make it easy and convenient for the ETL developer to get connected to a database with as little hassle as possible. In the past, this has worked well for Kettle in the sense that it's easy get access to 99 percent of the most common databases. However for the odd case, the buggy drivers, and the brand new versions, it is convenient to have a way of tweaking existing functionality. That is what the database type plugin system allows you to do.

The main interface that describes a database type is `org.pentaho.di.core.database.DatabaseInterface`. This interface contains a wide range of methods that describe the behavior of the database. It is possible for a database type plugin to define or override any behavior that might be interesting to you. As an example, let's create a new driver for the MySQL 5.1 JDBC driver. As explained at the beginning of this chapter, it is possible to replace the existing MySQL driver simply by using the same existing "MYSQL" database type ID. In the example, we're creating another entry in the list of available databases in the database dialog. (You can also find this code in the `MySQL51DatabaseMeta.java` file in the download package of this chapter.)

```
package org.kettlesolutions.plugin.database.mysql51db
import org.pentaho.di.core.database.DatabaseInterface;
import org.pentaho.di.core.database.MySQLDatabaseMeta;
import org.pentaho.di.core.plugins.DatabaseMetaPlugin;

@DatabaseMetaPlugin(
```

```
    type="MYSQL51",
    typeDescription="MySQL 5.1"
  )
```

The 5.1 driver uses a different JDBC driver class by default and you also want it to support transactions:

```
public class MySQL51DatabaseMeta extends MySQLDatabaseMeta
  implements DatabaseInterface {

  public String getDriverClass() {
    return "com.mysql.jdbc.Driver";
  }

  public boolean supportsTransactions() {
   return true;
  }
}
```

To make this simple database type plugin work, you simply have to put it in a `.jar` file and deploy it in the `plugins/databases` directory in your binary Kettle distribution.

The user interface aspect of the database type plugin system is not completed at the time of this writing. Because the database dialog is a core (or *commons*) project and uses XUL, you will have to specify a value for the method `getXulOverlayFile()` in the database interface. In this example we would return `mysql`. Combined with the access method, this would result in the file `mysql_native.xul` being read. The Kettle developers expect that the specification of a plugin-defined XUL overlay file will be possible soon. For the time being, it is possible to hard-code the various interface values or to read them from a file somewhere (defined perhaps by an environment variable).

Summary

In this chapter, you learned the basics of writing a plugin for Kettle. You took a look at the plugin architecture and the setup of your development environment. Specifically, the chapter discussed:

- Implementing the four core interfaces that make up a step plugin
- Handling advanced step plugin problems like sending rows to specific steps and variable substitution
- Writing code for the User Defined Java Class step
- The basics of SWT user interface development
- Creating a job entry plugin
- Designing a custom partitioning method
- Building a repository type plugin
- Creating your own database type as a plugin

APPENDIX

A

The Kettle Ecosystem

By now you have probably discovered that developing ETL solutions using Kettle is not only easy but also a lot of fun. To make it even easier and more fun than it already is, you probably want to get in touch with other Kettle fans. In some circumstances you might also run into something that doesn't work as expected and you need to find out if it's just you that's having this problem or if it's an already known problem. Finally, you want to keep up-to-date on the latest developments, try out new features or even contribute to the Kettle community. This appendix is meant to introduce you to all the sites and tools that make up the Kettle ecosystem.

Kettle Development and Versions

The code base of the Kettle project is under heavy development and changes almost daily. Bugs are fixed, new features are added, and optimizations are being implemented constantly. As a consequence, five versions are available for use, each in its own stage of the product lifecycle. The following list explains the various options for obtaining the software and their maturity, from the GA (General Availability) to Trunk (source code you need to build and compile yourself):

- **GA (General Availability) version:** The most stable version of Kettle and probably the one you will be using in a production environment. In fact, we don't recommend any other version than GA for use in production, period. The GA version has undergone heavy unit and integration testing and is used by thousands

of people all over the world. If you go to the Pentaho Community download site and select Latest Stable Builds, the URL `http://wiki.pentaho.com/display/COM/Latest+Stable+Builds` is opened. You might notice that this overview doesn't always contain the latest versions of the product, so in order to get the actual latest GA version, it's better to go to the Pentaho SourceForge files directly, which are located at `http://sourceforge.net/projects/pentaho/files`.

- **RC (Release Candidate):** Prior to becoming GA, several Release Candidates will be published for download. RCs serve several purposes: First, they enable organizations to perform upgrade tests and make sure that existing functionality won't break. Second, they allow ETL developers to find out which new features are available. Finally, they allow Pentaho to get early feedback from customers on new features and any possible remaining issues. RC versions are code complete, meaning that there won't be a feature difference between a release candidate and the final GA release. RC versions can also be obtained from the previously mentioned SourceForge site.
- **Milestone:** During the development of a new release, several milestone releases are created, each adding more new features to the final product. The Pentaho developers have embraced the Scrum development methodology, which works from an issue backlog, where an "issue" can refer to a new feature, a product improvement, or a bug. This issue backlog is divided into so-called *sprints* and each finalized sprint results in a milestone. Sometimes these milestones are available for download as a kind of snapshot, but sometimes not. Milestone releases can be found on the wiki page `http://wiki.pentaho.com/display/COM/Latest+Milestones` and are usually kept for download on the CI site (see next bullet point) for some time as well.
- **CI (Continuous Integration):** Pentaho uses Hudson, a tool that is used for continuously building and testing software projects. The Pentaho CI environment can be accessed at `http://ci.pentaho.com` where you can find the latest builds of all Pentaho projects, including Kettle. The CI site contains the binary versions of the different Pentaho projects, but there are no installers so you need to know how to install and deploy the software yourself. In the case of Kettle, this is very straightforward; just download the latest zip file (the one with the version and build number in the file name, not the `pdi-ce-TRUNK-SNAPSHOT.zip` file), unpack it in a directory, open a terminal screen, and issue either `spoon.sh` (Linux) or `spoon.bat` (Windows). On a Mac, just click the Data Integration 64-bit.app symbol.
- **Trunk version:** If you're a developer and would like to contribute to Kettle, or would like to work with the source code directly, you can get the code from the `trunk` directory of Subversion, the version management system used by Pentaho. The Kettle code can be accessed from `http://source.pentaho.org/svnkettleroot`; the web page `http://community.pentaho.com/getthecode` offers more general information about accessing and working with the source code of the various Pentaho projects.

The Pentaho Community Wiki

Probably the first choice of documentation for anyone starting out in the Pentaho world is the Community Wiki, which you can access at `http://wiki.pentaho.com`. The wiki is divided into separate sections, so each project has a separate set of pages where you can find documentation, frequently asked questions, screenshots, and recorded demos. Although it seems like this is the documentation nirvana where you can find everything you need, you'll soon find out that reality is a bit less shiny. Many links point to almost empty pages or describe older versions of the products, and in the case of Kettle, some of the steps are not documented (yet) at all. Remember that the content of the wiki is entirely created and maintained by the Pentaho Community. That is, by the way, also the good news: As soon as you register and sign in, you can start adding or updating content and helping others become more productive using Kettle.

Using the Forums

Unlike the wiki, which is of a more static nature, the Pentaho forums can be used to interact with other Kettle fans, and are a gold mine for getting help or looking for a solution to your specific problem. The Pentaho forum is a public website that can be found at `http://forums.pentaho.org` and is open to anyone. There's only one golden rule to remember, however, not only for this forum but for any open source community: You're never on your own, but you need to give to get. A bit of patience and politeness don't hurt either.

The forums can be used to search for existing posts with a subject you're interested in, you can post new threads or reply to existing ones, and many announcements about the Pentaho community can also be found on the forums first. If you don't register and sign in, you can still use the forum to search for solutions, but if you want to post a new thread or reply to someone else's, you need to be signed in.

At your first visit to the aforementioned site, you'll immediately notice that Kettle is one of the more popular topics on the forums. The Kettle forum alone now contains over 40,000 messages, so it's likely your question was asked before by someone else. So before jumping in and posting a new thread, make sure that there isn't an existing thread with a similar question or discussion. In fact, there are some other ground rules as well, and Matt Casters, chief architect of Kettle, was kind enough to list them in the first post you should read. It's aptly called "Read me first" and can be found at `http://forums.pentaho.org/showthread.php?t=71633`.

A good way to keep track of what's going on in the forums is to subscribe to the RSS feed using a news reader like `http://reader.google.com` or any other one you prefer. Subscribing to the feed is easy. Just go to the main Kettle forum page and click the RSS button right below the New Thread button. The screenshot in Figure A-1 shows exactly where it is located.

Alternatively, you can add the feed URL directly to your reader of choice. The URL is `http://forums.pentaho.org/external.php?type=RSS&forumids=135`.

Figure A-1: Forum feed subscription

Jira

Jira is a product of the Australian company Atlassian (`http://www.atlassian.com`) and is arguably the most widely used issue tracking and project management tool in the world. Major companies such as Oracle, Cisco, and Boeing use the Atlassian tools, and even Microsoft is on their customer list. Jira is therefore not a Pentaho product, but a product used by Pentaho. It has become so well known in the community, however, that is has become a name of its own. So when someone says to "look it up in Jira" or asks you whether you have "posted it in Jira," what they actually mean is the Pentaho issue tracking site that can be found at `http://jira.pentaho.org`.

At first glance, it might seem that Jira is a bug tracker, and although this might be correct for many of the entries to be found on the site, it's not the complete story. Jira is actually the environment where you can find the Pentaho roadmaps and the planning and work in progress for the upcoming releases—and yes, bugs are posted there as well. Unlike proprietary software products, you can actually see what's going on with your favorite tool, who got assigned the open issues, and when you can expect them to be solved. Figure A-2 shows an example of the Pentaho Data Integration Task board, which shows 51 open issues on the left, 3 closed issues on the right, and no issues currently in progress.

Figure A-2: Jira Task Board

##pentaho

This somewhat cryptic title is the name of the unofficial Pentaho channel on IRC (Internet Relay Chat). IRC is one of the oldest communication channels on the Internet and has been around for over 20 years now. This predates the WWW and HTML era, so communication on an IRC channel is still text-based, although you can use URLs nowadays. IRC is a client/server type of application, so you need a client to connect to the various servers that are available worldwide. Each server hosts one or more channels you can join, and most channels have a single hash sign (#) before their name. Trying to find a #pentaho channel won't result in anything because the channel uses two hash tags: it's ##pentaho, and can be found on Freenode.net, one of the available open IRC servers on the Internet.

NOTE **A common rule on IRC is that channels having a name starting with a single hash tag (#) are general chat channels, whereas the double hash tag (##) indicates help channels.**

Getting on the channel is very straightforward: just download one of the available chat clients, connect to the Freenode server, and join the ##pentaho channel. Two of the more popular clients are XChat (`http://xchat.org`) for Linux and Windows, and mIRC (`http://www.mirc.com`) for Windows only. If you're a Mac user, you might want to take a look at XChat Aqua (`http://xchataqua.sourceforge.net`) or Babbel (`http://www.babbelirc.com`). All of these clients work in a similar way; once you connect to the channel(s) of your choice, you can simply type in what you want to say or ask in the text box, which is usually at the bottom of the screen. Figure A-3 shows a screenshot of XChat connected to the ##pentaho channel.

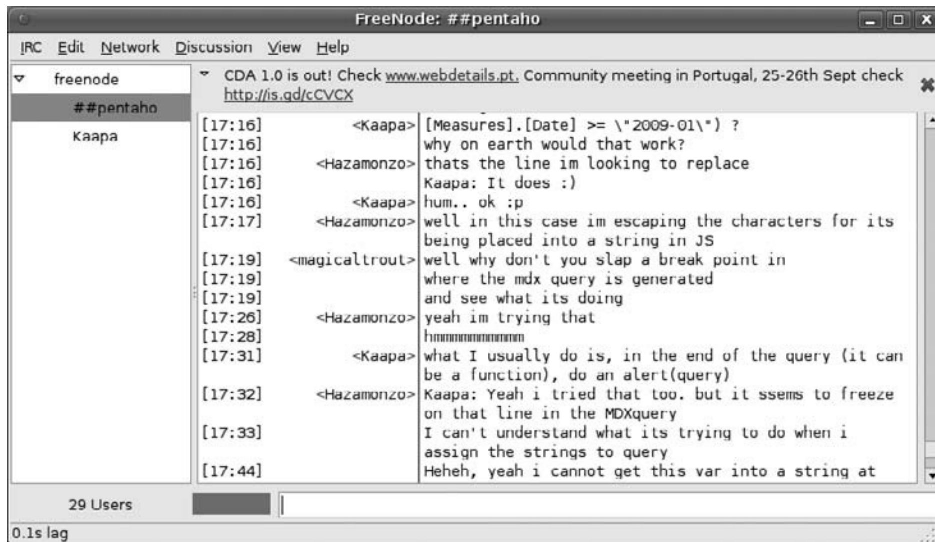


Figure A-3: Chatting with the community

As you can tell from just looking at Figure A-3, there's not much to learn here. You'll see that there are 29 users connected (this was on a Sunday), and there's an announcement about the upcoming community meeting at the top. If you engage in a private chat, the name of the user you're chatting with is displayed in the channel list on the left as well.

Usually you'll find anywhere between 25 and 40 people on the channel. Many of the Pentaho developers are active on IRC, and Doug Moran, Pentaho's community manager, is also a regular visitor. If you look at the text in the screenshot displayed in Figure A-1, you'll notice multiple people interacting and trying to tackle a problem, and usually (we must stress that again: *usually*) people are very helpful.

WARNING **Don't expect the IRC channel to be a free support channel where all your questions are answered immediately by a group of highly skilled Pentaho experts. In many cases, you'll be directed to either the forums or to Jira … or it may take hours before someone notices your question (if at all).**

In the screenshot, you'll also notice the names of the users on the channel. Most people use an alias, not a real name. Most of these users can be found on the Pentaho forum as well, and if you hang around long enough in the community you'll eventually find out who's who.

APPENDIX

B

Kettle Enterprise Edition Features

As you probably know already, Pentaho offers two versions of Kettle, an open source Community Edition (CE) and an Enterprise Edition (EE) that contains proprietary pieces of software. On the Pentaho website you can learn about the differences between CE and EE in general (`http://www.pentaho.com/products/enterprise/enterprise_comparison.php`), and as you can see from the site there's a lot to be gained from taking an Enterprise subscription.

NOTE **One thing to keep in mind about the Enterprise Edition is that you sign up for a yearly *subscription*, which is different than a software *license*. Most software licenses include the right to use the software for an infinite amount of time, and even if you stop paying for support, you can still keep using the software. A subscription works differently. Each year the subscription needs to be renewed and a new license key with an expiration date set at the current date plus one year will be issued. This means that when you decide to terminate the subscription, the software will stop working as soon as the license key expires. However, with Kettle there could be a smooth downgrade path from the Enterprise to the Community Edition; as long as you don't use any of the EE-only steps (see following), all jobs and transformations built using the Enterprise Edition will run unchanged on the Community Edition as well.**

Until version 4 of Kettle, there weren't any significant differences in functionality between Kettle CE and Kettle EE. The only difference was the extended Enterprise Console, which let you schedule and monitor jobs. This has changed considerably with

the arrival of version 4. Kettle CE version 4 is still a very powerful ETL tool with many features that can only be found in very expensive proprietary tools, but it lacks the following components that are only available via an EE subscription:

- **Installer:** Although installation of Kettle CE is a very straightforward process, the EE further eases this effort by packaging the solution in an easy-to-use installer.
- **Agile BI:** Kettle EE contains the plugin for Agile BI development (see also Chapter 11) in the installation files. CE users need to download, unpack, and deploy this component separately.
- **Integrated Scheduling:** Kettle EE contains a built-in scheduling tool, eliminating the need to use either the Enterprise Console or some third-party tool for this task.
- **Data Integration Server:** Kettle EE comes with a dedicated data integration server that can be managed independently from other Pentaho components. This also makes Kettle EE easier to deploy and manage in a corporate environment.
- **Management Console:** The Pentaho Enterprise Console can be used as a dedicated management console for Pentaho Data Integration.
- **Enterprise Repository:** Arguably the biggest differentiator between the CE and EE versions is the new repository that works closely with the new data integration server. This repository enables features that were never available in Kettle before, and are mostly targeted at supporting multi-developer teams:
  - **Versioning:** Enables storing multiple versions of a job or transformation and the ability to roll back to previous versions.
  - **Locking:** Prevents developers from overwriting someone else's work by allowing check-out/check-in of components.
  - **Security:** Supports fine-grained authorization schemes for users, roles, and permissions. By default, the data integration repository security integrates with Pentaho security, but can also leverage existing authentication and authorization providers such as LDAP and Active Directory.
- **Additional steps:** There are extra transformation steps and job entries added to the Enterprise Edition. Examples are the Google Analytics Input step, Google Docs Input step, JMS step, and many more.

As you can read on the edition comparison site, an Enterprise Edition subscription comes with toll-free telephone and e-mail support, including unlimited support cases. Even if you don't want to buy an EE subscription, you can probably still get paid support; the Kettle Community Edition is one of the products for which paid developer support was available at the time of this writing.

APPENDIX

C

Built-in Variables and Properties Reference

This appendix starts with a description of all the internal variables that are set automatically by Kettle. That is followed by a list of all the variables that you can set to influence the way that Kettle operates at run-time. Next, we discuss how you can define variables to help secure FTP authentication using VFS. Finally, we mention a few noteworthy variables from the Java Runtime Environment.

For more information on how to use variables, see Chapter 2.

Internal Variables

Table C-1 shows all the variables that are defined at run-time by the Kettle transformation or job engine.

Table C-1: Internal Variables

VARIABLE	DESCRIPTION
`Internal.Kettle.Version`	This contains the version string of Kettle, for example, 4.0.0.
`Internal.Kettle.Build.Version`	This contains the subversion revision of the Kettle source of the build you're using.

Continued

Table C-1 *(continued)*

VARIABLE	DESCRIPTION
`Internal.Kettle.Build.Date`	The build date.
`Internal.Job.Filename.Directory`	If you are running a job from a file (.kjb), this variable will contain the folder in which that file is located. This variable allows you to specify other jobs, transformations, and files in a relative fashion.
`Internal.Job.Filename.Name`	If you are running a job from a file (.kjb), this variable will contain the name of that file.
`Internal.Job.Name`	The name of the currently executing parent job.
`Internal.Job.Repository.Directory`	If you are running a job from a repository, this variable will contain the path to the repository directory from which the job was loaded.
`Internal.Transformation.Filename .Directory`	If you are running a transformation from a file (.ktr), this variable will contain the folder in which that file is located. This variable allows you to specify mappings and files in a relative fashion.
`Internal.Transformation.Filename .Name`	If you are running a transformation from a file (.ktr), this variable will contain the name of the file.
`Internal.Transformation.Name`	The name of the currently executing transformation.
`Internal.Transformation .Repository.Directory`	If you are running a transformation from a repository, this variable will contain the path to the repository directory from which the transformation was loaded.
`Internal.Step.Partition.ID`	If a step is configured to run as partitioned, multiple copies will be executed, one for each partition. This variable will contain the partition ID to which that step copy belongs.

VARIABLE	DESCRIPTION
Internal.Step.Partition.Number	If a step is configured to run as partitioned, multiple copies will be executed, one for each partition. This variable will contain the partition number to which that step copy belongs in the range between zero and the number of partitions minus one.
Internal.Slave.Transformation.Number	If a transformation is running as clustered on a slave server (not the master), this variable will contain the slave number. This value will be in the range between 0 and the number of slaves minus one.
Internal.Slave.Server.Name	If a transformation is running as clustered on a slave server (not the master), this variable will contain the name of the slave server on which it runs.
Internal.Cluster.Size	If a transformation runs as clustered, this variable will contain the number of slaves in the cluster.
Internal.Step.Unique.Number	This variable contains the unique number of step copies for a given step in a transformation. This also works when the step is executed with clustering and/or partitioning. This value will be in the range between zero and the number of step copies minus one.
Internal.Cluster.Master	If a transformation runs as clustered, this variable will contain `Y` if it is running on the master and `N` if it is running on a slave.
Internal.Step.Unique.Count	The number of unique step copies that are executed. This also works when the step is executed with clustering and/or partitioning.
Internal.Step.Name	The name of the executing step.
Internal.Step.CopyNr	The copy number in the local transformation (does not take into account clustering).

Kettle Variables

Table C-2 provides a list of variables that can be set to configure various Kettle-specific options.

Table C-2: Kettle Variables

VARIABLE	DESCRIPTION
KETTLE_SHARED_OBJECTS	The location of the shared object file for transformations and jobs. The default shared objects file is called shared.xml, located in the Kettle home directory. Setting this variable overwrites this default.
KETTLE_EMPTY_STRING_ DIFFERS_FROM_NULL	If this setting is set to Y, an empty string and null are different. Otherwise, they are not (the default).
KETTLE_MAX_LOG_SIZE_IN_LINES	The maximum number of log lines that are kept internally by Kettle. Set to 0 to keep all rows (the default).
KETTLE_MAX_LOG_TIMEOUT_IN_ MINUTES	The maximum age (in minutes) of a log line while being kept internally by Kettle. Set to 0 to keep all rows indefinitely (the default).
KETTLE_STEP_PERFORMANCE_ SNAPSHOT_LIMIT	The maximum number of step performance snapshots to keep in memory. Set to 0 to keep all snapshots indefinitely (the default).
KETTLE_PLUGIN_CLASSES	A comma-delimited list of classes to scan for plug-in annotations. See http://wiki.pentaho.com/display/EAI/How+to+debug+a+Kettle+4+plugin for more information.
KETTLE_LOG_SIZE_LIMIT	The log size limit for all transformations and jobs that don't have the log size limit in lines property set in their respective log table properties.

There are also a number of variables, listed in Table C-3, that allow you to automatically configure the various logging tables for all jobs and transformations. For the variables in that table, you have to replace ... with one of the following values:

- TRANS: For the transformation log table
- TRANS_PERFORMANCE: For the performance log table
- STEP: For the step log table
- JOB: For the job log table

- JOBENTRY: For the job entry log table
- CHANNEL: For the channel log table

For more information on log tables, see Chapter 14.

Table C-3: Kettle Log Table Variables

VARIABLE	DESCRIPTION
KETTLE_..._LOG_DB	Specifies the name of the database connection to use for the logging table.
KETTLE_..._LOG_SCHEMA	Indicates the schema to use for the logging table.
KETTLE_..._LOG_TABLE	Specifies the name of the logging table itself.

Variables for Configuring VFS

As explained in Chapter 2, it is possible to use URIs to specify various locations of files in Kettle. The underlying technology, Apache VFS, supports many different file systems such as file://, zip://, ftp://, http://, and so on. The standard way to authenticate all file systems is to place username@password before the hostname. For example:

```
ftp://john:pwd4john@example.com/pub/customer/file.txt
```

For most file systems this works fine. However, in the case of secure FTP (sftp://) you may be required to specify additional information such as the location of a private key file. This is also supported through the system of variables. If you want to configure these options, you have to follow these variable naming conventions:

- The variable must start with vfs.
- The file system scheme comes next in the variable name: sftp.
- The parameter you want to set is next:
  - StrictHostKeyChecking: If no, the certificate of any remote host will be accepted. If yes, the remote host must exist in the known hosts file (~/.ssh/known_hosts).
  - authkeypassphrase: An optional passphrase that may be required to use the identity key.
  - identity: The fully qualified path to the private key used for SFTP authentication.
- Finally, the variable name is followed by the host name or IP address for which the variable is created.

Here are a few examples:

- vfs.sftp.StrictHostKeyChecking.sftp.myhost.net
- vfs.sftp.authkeyparaphrase.sftp.example.com
- vfs.sftp.identity.192.168.1.5

NOTE **Refer to** http://wiki.pentaho.com/display/COM/Configuring+Kettle+VFS **for further information on configuring Kettle VFS.**

Noteworthy JRE Variables

For a complete listing of variables that are set by the Java Runtime Environment, please see the documentation of the getProperties() method of the System class at http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#getProperties(). Table C-4 shows a few notable variables that might be of use in your transformations and jobs.

Table C-4: Noteworthy JRE Variables

VARIABLE	DESCRIPTION
java.version	The version of the Java Runtime Environment.
java.io.tmpdir	The default temporary file directory path. This is the default for all temporary files and the Spoon logging file.
os.name	The name of the operating system.
os.arch	The architecture of the operating system.
os.version	The version of the operating system.
user.name	The user's account name.
user.home	The user's home directory.
user.dir	The current working directory.

Index

SYMBOLS

: (colon)
  object members, 522
  parameters, 322
, (comma)
  arrays, 522
  row metadata, 28
. (dot)
  row metadata, 28
  UNIX metacharacter, 507
* (asterisk), RSS Output step, 565
@ (at symbol), tokens, 536
^ (caret sign), UNIX metacharacter, 507
{} (curly brackets)
  object literals, 522
  object members, 522
- (dash)
  parameters, 322
  tokens, 536
$ (dollar sign), UNIX metacharacter, 507
$[] (dollar sign/square brackets), variable hexadecimal values, 45
= (equals sign), parameters, 322
(hash mark), chat channels, 633
%% (percent sign-double), variable names, 45
| (pipe sign), FIFO, 247–248
? (question mark), XPath, 543
" (quotes-double)
  object members, 522
  parameters, 322
  string literals, 522
' (quotes-single)
  parameters, 322
  row metadata, 28
/ (slash), parameters, 322
[] (square brackets), arrays, 522
_ (underscore), tokens, 536

A

Abort job utility, 185–186
Abort step, 186
acc, 354
Access
  Database Connection, 38, 93
  file extraction, 134
  Find Duplicate Query Wizard, 117
accumulating snapshot fact table, 260–261
  fact table loader, 121
  loading, 264–265
  partitioning, 264–265
action sequence
  schedule, 333
  transformations, 328–330
ActiveMQ, 459
Add a mini-dimension, SCD, 119
Add constant step, 178
  JavaScript, 396
add crc_src, 484
add crc_vault, 484
Add File to result, RSS Output step, 567
Add new column, SCD, 119
Add sequence step
  internal counters, 211–213
  rows, 405

Spoon, 211–217
surrogate keys, 211–217
Add to result filename, Get data from XML step, 535
Add validation msg in output, XSD Validator step, 529
Add XML step, 518, 540
export_xml_from_db, 538
streams, 538
XML, 538–541
addExport, 416
Additional fields, Split field to rows step, 499
Additional steps, Enterprise Edition, 636
addJob, 416
Addresses tab, Mail, 337
addResultFile(), StepInterface, 618
addTrans, 415
ADempiere, 15
adjacency list, 242
ADLER 32, 484
Administration Console, 330–333
aggregate builder, ETL, 123
aggregate tables, 266–267
Aggregation Designer, Mondrian, 123, 267
Agile BI, 12–14, 302
Enterprise Edition, 636
agile development
BI, 12–14
ETL, 301–306
Spoon, 301–302
Agile Manifesto, 12–13
allocateSocket, 415
Amazon, EC2, 125, 427–447
Amazon Machine Image (AMI), 439–442
master, 442–443
slaves, 443–444
Ubuntu, 442
AMI. *See* Amazon Machine Image
analyseImpact(), StepMetaInterface, 599
Ant build tool, Apache, 571
Apache
ActiveMQ, 459
Ant build tool, 571
DBCP, 400
Log4J, 366
Subversion, 343, 570
VFS, 42, 349, 517, 619
variables, 641–642
API. *See* application programming interface
application programming interface (API), 569–571
documentation, 596
Java, 570–574
jobs, 573–574
parameters, 579–580
transformations, 572–573
variables, 579–580
web services, 516
Arbor Essbase, 269
architecture
DWH, 8
logging, 364–367
plugins, 593–599
transformations, 452
archive files, 60–61
arrays, JSON, 522
Aschauer, Bernd, 140
ASCII, 15, 18
staging area, 8
at, 327
atlassian.com, 632
Attached Files tab, Mail, 339–340
attribute domain constraints, data quality, 168
attributes
HTML, 520
hubs, 468
multi-valued, 498–500
Palo, 285
Root XML element, 541
rows, 497
SQL, 484
Attributes , satellites, 470
Attunity Stream, Oracle, 163
audit dimension, 117, 192
auditing. *See also* logging
data quality, 191–192
ETL, 22
authentication, Carte, 67
authkeypassphrase, 641
authorization, Palo, 285
auto_increment, 217

B

babbelirc.com, 633
backtracking, 32–33
Backup System, 124
Badard, Thierry, 591
balanced hierarchy, 120
BAPI. *See* Business Application Programming Interface
BasePartitioner, 625
BaseStep, 614
.bat, 58

batch processing, 449
batch run, 449
batch_id (=audit key), 192
batch-level lineage extraction, 358–359
BCNF. *See* Boyce-Codd normal form
Beginning XML (Hunter), 518
behavioral testing, 306
BI. *See* business intelligence
The Big Debate, 465
BigNumber, 28
Binary, 28
BIReady, 6
black box testing, 21
Blocking Step, multi-threading, 410
Boolean, 28
  Database Connection, 38
  flags, 94
  job entries, 366
  JSON, 522
  String, 30
  valid, 160
boolean isUnconditional(), JobEntryInterface, 622
bottlenecks, transformations, 379–382
Bouman, Roland, 228, 319, 327
Boyce-Codd normal form (BCNF), 218, 226
BPM. *See* Business Process Management
branches/, 342
bridge tables, 121–122
browsers, web, 413
buffers
  logging, 364, 365–366
  performance, 380
  RowSet, 579
  transformations, 406–407
Build Model, Spoon, 302
built-in variables and properties, 637–642
Bulk Loader, MySQL, 248, 249
bulk loading
  database, 390
  fact tables, 246–251
  LucidDB, 249
  PostgresSQL, 250
  Table output step, 250
bundling, 442
  messages, 601
Bus Architecture, SCD, 118–119
Business Application Programming Interface (BAPI), 140
business intelligence (BI), 2
  Agile BI, 12–14, 302, 636
  ETL, 12
  Pentaho BI, 322, 327–333
  real-time, 450
Business key, 468
business keys, 209–210
  dimension tables, 210, 527
  DV, 467
  DWH, 209
  looking up, 210
  natural keys, 210
  Sakila, 527
  storing, 210
  surrogate keys, 210
  XML, 527
Business Objects, SQL, 9
Business Process Management (BPM), 344

C

caching, data, 453
Calculate Dimension Attributes, transformations, 85–86
calculations
  casing, 170
  Validator step, 180–181
Calculator step, 108–109, 170, 171
  Formula step, 201
  keys, 214
callback, 550
canvas
  jobs, 56
  Spoon, 318
  transformations, 56
capture groups, regular expressions, 200, 205
Carlton, 6
Carte, 41, 55, 57
  authentication, 67
  clustering, 57
  dynamic clustering, 434
  master, 441
  slave servers, 411–416, 435
  TCP/IP, 57, 417
carte-config-master-8080.xml, 434–435
Cartesian join step, RSS Ouput step, 564
Cartesian product, 87
Caserta, Joe, 113
case-sensitivity, 507
casing, calculations, 170
Casters, Matt, 167, 315
cat, 248
Catalog location, Mondrian Input step, 274
catalogs, Database Connection, 39
CDC. *See* Change Data Capture
Central storage, 41

Change Data Capture (CDC), 16
  data extraction, 450–451
  database triggers, 157–158
  dimension table keys, 80–81
  ETL, 115, 154–163
  logging, 162–163
  MySQL, 162–163
  real-time data integration, 450–451
  relational databases, 450
  RFCs, 146
  Sakila, 108
  snapshots, 146, 158–162
  source data, 155–157
  star schema, 227–228
  timestamps, 155–157, 163, 450
  triggers, 450
Change number of copies to start, User Defined Java Class step, 404
CHANNEL, 641
channel, RSS, 558–559
Channel tab, RSS Output step, 565
channel-log-table, 346
channels
  log table, 372
  logging, 366
chat channels, 633
check()
  JobEntryInterface, 622
  StepMetaInterface, 599
Check if XML file is well formed, job entries, 519
checksums, 309
  JavaScript, 395
  partitioning, 426
child key, 242
chmod, UNIX, 322
CI. *See* Continuous Integration
classpath, 70–71
cleansing. *See* data cleansing
clear-box testing, 306
clone(), StepInterface, 616
Closure Generator step, 242–243
closure table, 242
cloud computing, 433–447
cluster, kettle.pwd, 67
clustering, 18–20. *See also* dynamic clustering
  Carte, 57
  data pipelining, 425
  database, 40
  schema, 417–418
  sorting, 394
  TCP/IP, 423
  transformations, 417–425
    partitioning, 430
CMSs. *See* content management systems
Codd, E.F. (Ted), 269
Cognos, Powerplay, 269
column profiling, 17, 146
Combination lookup / update step, 99
  import_xml_into_db.ktr, 527
  Insert / Update step, 241
  junk dimensions, 241
  Spoon, 241
ComboVar, 609
Comma Separated Values (CSV), 47–50, 128
  dynamic transformations, 580–583
  sets, 498
command-line
  jobs, 322–326
  parameters, 323–324, 325–326
  transformations, 322–326
commands.jar, 596
commentary, ETL, 298–299
comments, item, 559
commit size, Table output step, 390
common.jar, 596
Community Edition, 635–636
Community Wiki, 631
compatibility mode, JavaScript, 395
complex join condition, XML Join step, 543
Compliance Reporter, ETL, 125
concatenation
  denormalization, 100–101
  Validator step, 179
Concurrent Versions System (CVS), 343
ConditionEditor, 609
configuration, 63–72
  slave servers, 411–412
conformation. *See* data conformation
connect by prior, Oracle, 20
Connection, Mondrian Input step, 274
Connection Name, Database Connection, 38, 92–93
connection pools, 400
Connection Type, Database Connection, 38, 92–93
constraints
  dependency, data validation, 183
  domain attribute constraints category, 179
  performance, 392
content management systems (CMSs), 344, 626
Content tab
  Add XML step, 539

Get data from XML step, 532–535
Regex Evaluation step, 506–507
RSS Input step, 561–562
Continuous Integration (CI), 13, 450, 630
testing, 311
control files, 247
COPY, PostgresSQL, 250
Copy rows to result step, 162
CDC, 164
job entries, 399
transformations, 576–577
Copy Table, 9
Copy Tables, 9
Copy tables wizard, Spoon, 584
Core, Java API, 571
Counter name field, Step sequence step, 212
CPU performance, 394–398
CRC. *See* Cyclic Redundancy Check
Create custom RSS, RSS Output step, 565, 567
Create new rows, SCD, 119
Create Parent folder, RSS Output step, 567
CRM. *See* Customer Relationship Management
Crockford, Douglas, 520
cron, UNIX, 326–327
crontab, UNIX, 326–327
CSV. *See* Comma Separated Values
CSV File Input step, 384–385
dynamic templates, 584
partitioning, 428
rows, 394
StepInterface, 617
CsvFileReader, 584
CsvFileReader.java, 580–583
cubes, 123
dimensions, 270–271
OLAP, 270–271
XML/A, 278
current_flag, 265
CURRENT_TIMESTAMP, 480
Customer Relationship Management (CRM), 14–15, 138–146
deduplication, 192–199
customizations.jar, 591
CVS. *See* Concurrent Versions System
Cyclic Redundancy Check (CRC)
Filter rows step, 485
Merge Join step, 485
NULL, 484
Update step, 485

D

Damerau-Levenshtein algorithm, 171
data
acquisition challenges, 14–16
caching, 453
delivery, 118
federation, 10–11
governance, 168
late-arriving, 255–260
dimensions, 256–260
ETL, 122
facts, 256
migration, 9
paths, 25
SAP, 140–145
semi-structured, 501–508
sorting, performance, 392–394
static, 397–398
streams, 577
synchronization, 9
traceability, 467
transformations, 576–580
unstructured, 501–508
Data Cleaning and Quality Screen Handler System, ETL, 116–117
Data Cleanse step, 169
data cleansing
data governance, 168
data quality, 168
data validation, 179–183
ETL, 168–183
reference tables, 172–179
regular expressions, 203–205
source data, 173
data conformation
ETL, 118
lookup tables, 172–175
reference tables, 175–179
data conversion
JavaScript, 395
transformations, 29–30
data extraction, 127–165
Access files, 134
CDC, 450–451
database, 134–136
ETL, 114–116
Excel files, 134
Freebase, 553–558
HTML, 520
HTTP client, 137–138
input steps, 128
lineage, 358–359

metadata, 359
parallelism, 538
real-time, 138
SOAP, 138
Spoon, 128
streams, 138
text files, 128–132
Web, 137
Web, 137–138
XBase files, 134
XML, 525–536
XML files, 133
data formats
HTTP, 517
non-relational, 498
non-tabular, 498
web services, 517–523
XML, 518–520
Data Grid step, 135, 173
SAP Input step, 142
Table input step, 132
testing, 311
data integration, 8, 569–592
challenges, 11–17
continuous, 450
ETL, 123
`import_xml_into_db.ktr`, 527
near real-time, 450
real-time, 449–461
Spoon, 302
streaming, 450
streams, 450
Data Integration Server, Enterprise Edition, 636
data lineage, 21, 357–363
data extraction, 358–359
impact analysis, 361–363
Data Manipulation Language (DML), 246
data mapping, 297–298
fields, 25
Sakila, 524–525
XML, 524–525
data mart
DV, 486–495
data pipelining
clustering, 425
multi-threading, 407–408
Data Profiler, Talend, 154
data profiling, 16–17, 127–128, 146–154
metadata, 17
Data Profiling System, 115
data quality
auditing, 191–192
categories, 168–169
challenges, 16–17
data cleansing, 168
Data Quality Assessment (Maydanchik), 169
Data Quality Lifecycle, 191
data types, Validator step, 179
data validation
data cleansing, 179–183
DataCleaner, 148, 153
dates, 182
deduplication, 183
dependency constraints, 183
error handling, 187–190
metadata, 182
`NULL`, 17, 180–181
rules, 180–183
`Unknown`, 17
XML Schema, 530
XSD Validator step, 530
Data Validator step, 179, 182, 187–188
Data Vault (DV), 9, 168, 465–495
business keys, 467
data mart, 486–495
database accounts, 477
dependency, 471
ETL, 477
extensibility, 471
hubs, 467–468
links, 468–469
`NULL`, 471
Sakila, 472–486
satellites, 469–471
tables, 485–486
3NF, 469
timestamps, 480
traceability, 471
data virtualization, EII, 10–11
data warehouse (DWH), 2
architecture, 8
business keys, 209
EDW, 465
jobs, 31
response time, 4
surrogate keys, 210
The Data Warehouse ETL Toolkit (Kimball and Caserta), 113, 191
Data Warehouse Lifecycle Toolkit (Kimball), 11, 113–114
Data Warehouse Toolkit (Kimball and Ross), 221, 228
Database, Java API, 571
database
accounts, 82
DV, 477

bulk loading, 390
CDC, 155
clustering, 40
connection pools, 400
connections, multi-threading, 408–409
extraction, 134–136
metadata, 588–590
OLTP, 75
partitioning, 40, 429–430
performance, 388–392
plugins, 627–628
repositories, 348–349
Sakila, 73–110
sequence, 211
surrogate keys, 217
sharding, 40
shared objects, 589
sorting, 393
time-outs, 453
triggers, CDC, 157–158
Database Connection, 37–41
DataCleaner, 150–151
Enable Connection Pooling, 400
Sakila, 90–95
transformations, 37, 90–95
Database join, 105
Database lookup step, 99, 103, 105, 178
data caching, 453
denormalization, 226
Enable cache, 253
failure, 224–225
late-arriving data, 257–258
Load all date from table, 253
source systems, 222
Stream lookup step, 253, 255
surrogate key pipeline, 252
Database Name, Database Connection, 38
Database repository, 41
DatabaseMeta, 589
DataCleaner
data validation, 153
Database Connection, 150–151
dependency, 153–154
dictionary, 153–154
eobjects.org, 147–154
JavaScript, 153
regular expressions, 151–152
Run profiling, 152
data-integration, 61
date
data validation, 182
dimensions, 239
Date, 28
Integer, conversion, 30
String, conversion, 29
Date mask matcher, DataCleaner, 149
Date of last insert (without stream field as source), Dimension lookup / update step, 238
Date of last insert or update (without stream field as source), Dimension lookup / update step, 238
Date of last update (without stream field as source), Dimension lookup / update step, 239
datetime-stamp, 484
DBCP, Apache, 400
dbhost, 354
deadlock, 383
Debian, 59
Debug option, 312–315
debugging
ETL, 21, 312–315
jobs, 56
logging, 364
real-time transformation streaming, 457–478
rows, 314
transformations, 56
decision support systems (DSS), 2
deduplication, 104
CRM, 192–199
data validation, 183
ETL, 117–118
exact duplicates, 193–194
non-exact duplicates, 194–195
transformations, 195–199
delays, parameters, 457
DELETE, 157
deleted records, CDC, 155
delimited text files, 128
DeMarco, Tom, 12
denormalization, 99
concatenation, 100–101
Database lookup step, 226
Palo, 288
star schema, 226
Denormalize Special Features, 104
dependency, 125
constraints, data validation, 183
data quality, 168
DataCleaner, 153–154
dimension tables, 219
DV, 471
profiling, 146

description, metadata, 37
Description, 332
description, 346
  channel, 559
  item, 559
descriptive fields, 318
design
  building blocks, 25–42
  flexibility, 19
  principles, 23–25
Design ETL, Spoon, 302
dev, 354
dictionary, DataCleaner, 153–154
Dictionary matcher, DataCleaner, 149
dimension(s)
  cubes, 270–271
  hierarchies, 270
  junk, 120, 241
  late-arriving data, 256–260
  mini-dimensions, 120, 239–240
  Palo, 285
  special dimension builder, 120
  static, 84–87
  user maintained, 120
Dimension lookup / update step, 99, 238
  data caching, 453
  history, 235–236
  indexes, 390–391
  Keys tab page, 234
  late-arriving data, 259
  SCD, 232–237
  SOF, 266
  surrogate keys, 234–235
dimension manager system, ETL, 122
dimension tables
  business keys, 210, 527
  dependency, 219
  ETL, 207–244
  fact tables, 251–260
  keys, 109, 208–217
    CDC, 80–81
  loading, 218–228
  natural keys, 99
  OLTP, 226
  rental star schema, 79–80
  rows, 90
  SCD, 118
  snowflakes, 97, 218–225
  star schema, 226–228
  static, 84–87
  surrogate keys, 209, 251–260
  surrogate primary keys, 80
dimension(s)static dimensions, special dimension builder, 120
dir, 324
directory, metadata, 36
dispose(), StepInterface, 614
distrib/, 571
distributed version control systems (DVCS), 344
DML. *See* Data Manipulation Language
Do Not Proceed, 90
Do not raise an error if no files, Get data from XML step, 534
docs/api, 571
Document template XML step
  export_xml_from_db, 537
  XML Join step, 542
document type definition (DTD), 519
document-all, 319
documentation, API, 596
domain attribute constraints category, 179
dotall mode, 507
Double Metaphone algorithm, 171
DQGuru, 118
drill-down, 314
driver, jdbc.properties, 65
DSS. *See* decision support systems
DTD. *See* document type definition
DTD Validator step
  job entries, 519
  XML, 133
DV. *See* Data Vault
DVCS. *See* distributed version control systems
DVD rental business, 73–110
DWH. *See* data warehouse
dynamic clustering
  Carte, 434
  cloud computing, 433–447
  master, 434
  schema, 434
dynamic ETL, 586–587
dynamic jobs, 584–586
Dynamic Slave, 445
dynamic templates, 583–584
dynamic testing, 307
dynamic transformations
  CSV, 580–583
  Spoon, 580–583
DynamicJob.java, 584

E

E4X. *See* ECMAScript for XML
EBS. *See* Elastic Block Service

E-Business Suite, Oracle, 18
EC2. *See* Elastic Computing Cloud
ec2-ami-tools, 440
ec2-bundle-vol, 442
ec2-describe-instances, 444
ECCD. *See* Extract, Cleanse, Conform, and Deliver
Eclipse, 302, 597–598, 607
ECMAScript, 394–396
ECMAScript for XML (E4X), 520
ecosystem, 629–634
Edit button, Database lookup step, 222
Edit mapping button, Insert / Update step, 231–232
EDW. *See* enterprise data warehouse
Elastic Block Service (EBS), 438
Elastic Computing Cloud (EC2), Amazon, 125, 427–447
elements
  HTML, 520
  user interface, 609
Elements name, Add XML step, 540
ELT. *See* extract, load, and transform
e-mail
  HTML, 339
  notifications, 336–340
Email Message tab, Mail, 339
embedding, 574–590
  libraries, 574
Enable cache, Database lookup step, 253
Enable Connection Pooling, Database Connection, 400
Encoding list box, Get data from XML step, 533
Encr.bat, 58
encr.sh, 58
Engine, Java API, 571
EnterListDialog, 609
EnterMappingDialog, 609
EnterNumberDialog, 609
EnterPasswordDialog, 609
enterprise data warehouse (EDW), 465
Enterprise Edition, 124, 302, 635–636
Enterprise Information Integration (EII)
  data virtualization, 10–11
  LucidDB, 10
Enterprise Repository, Enterprise Edition, 636
Enterprise Resource Planning (ERP), 127, 138–146
  metadata, 14, 139
  plugins, 140
EnterSelectionDialog, 609
entries, 346
environment, 354
environmentSubstitute(), StepInterface, 618–619
eobjects.org, DataCleaner, 147–154
ERP. *See* Enterprise Resource Planning
Error code, Data Validator step, 187
Error description, Data Validator step, 187, 188
Error fields, Data Validator step, 188
error handling
  data acquisition, 15
  data validation, 187–190
  dynamic templates, 584
  ETL, 117, 183–190
  process errors, 184–187
  StepInterface, 617
  transformations errors, 186–187
  XSD Validator step, 530
error messages, logging, 364
ErrorDialog, 609
/etc/init.d/carte, 441
ETI, 6
ETL. *See* extract, transform, and load
ETLT. *See* extract, transform, load, transform
eval(), JavaScript, 556
Excel
  file extraction, 134
  OLAP, 273
Excel Output step, 297
excludeFromCopyDistributeVerification(), StepMetaInterface, 600
excludeFromRowLayoutVerification(), StepMetaInterface, 600
ExecTrans.java, 576
execute(), JobEntryInterface, 622
Execute a transformation, Spoon, 413
Execute SQL script job, Spoon Copy tables wizard, 584
Execute SQL script step
  aggregate tables, 267
  foreign keys, 252
  import_xml_into_db.ktr, 526
Execute SQL step, multi-threading, 409–410
ExecuteJob.java, 573–574
ExecuteTrans.java, 572
export
  metadata, StepMetaInterface, 600
  repositories, 350–351
  resource exporter, 444

`exportResources()`
  `JobEntryInterface`, 622
  `StepMetaInterface`, 600
`export_xml_from_db`, 537–538
expressions, Java, 70–71
extended description, metadata, 37
extensibility, 593–628. *See also* plugins
  DV, 471
  ETL, 19–20
eXtensible Markup Language (XML)
  Add XML step, 538–541
  business keys, 527
  data extraction, 525–536
  data format, 518–520
  data mapping, 524–525
  document construction, 538
  document structure, 523–524
  documents, generating, 537–544
  ETL metadata, 24
  examples, 523–544
  file extraction, 133
  job entries, 519–520
  jobs, metadata, 346–347
  JSON, 520
  metadata, 345–347
  repositories, 344
  Sakila, 523–544
  slave servers, 413
  surrogate keys, 527
  transformations, metadata, 345–346
  VCS, 352
  Version Migration System, 352
  web services, 518–520
eXtensible Stylesheet Language (XSL), 133
Extension, RSS Output step, 567
external sort, 393
Extract, Cleanse, Conform, and Deliver (ECCD), 167
extract, load, and transform (ELT), 9
extract, transform, and load (ETL)
  aggregate builder, 123
  agile development, 301–306
  audit dimension assembler, 117
  auditing, 22
  Backup System, 124
  best practices, 296–300
  BI, 12
  building blocks, 7–8
  CDC, 115, 154–163
  commentary, 298–299
  Compliance Reporter, 125
  connectivity, 17–18
  Data Cleaning and Quality Screen Handler System, 116–117
  data cleansing, 168–183
  data conformer, 118
  data delivery, 118
  data integration manager, 123
  data migration, 9
  data paths, 25
  Data Profiling System, 115
  data synchronization, 9
  debugging, 21, 312–315
  deduplication, 117–118
  definition, 5
  design flexibility, 19
  development lifecycle, 295–320
  dimension manager system, 122
  dimension tables, 207–244
  DV, 477
  dynamic, 586–587
  error handling, 117, 183–190
  evolution, 5–6
  extensibility, 19–20
  extraction, 114–116
  Extraction System, 115–116
  fact table, loader, 121
  fact table, provider system, 122
  flow design, 300
  hierarchy dimension builder, 119–120
  impact analysis, 125
  Job Scheduler, 124
  jobs, 12, 30–36
  late-arriving data handler, 122
  Lineage and Dependency Analyzer, 125
  logging, 22
  maintainability, 300–301
  metadata, 21, 344–350
  metadata, graphical user interface, 24
  metadata, XML, 24
  Metadata Repository Manager, 125
  MOLAP, 123
  monitoring, 333–340
  multi-valued dimension bridge table builder, 121–122
  names, 24, 298–299
  Parallelizing/Pipelining System, 125
  platform independence, 18
  Problem Escalation System, 125
  RDBMS, 497
  Recovery and Restart System, 124
  reuse, 19, 300–301
  Sakila, 73–110, 81–84

scalability, 18–19
SCD, 118–119
scheduling, 321–333
scripts, 5, 200–205
Security System, 125
solution documentation, 315–320
Sort System, 124–125
special dimension builder, 120
Spoon, 81–84
subsystems, 113–126
surrogate key creation system, 119
testing, 21, 306–312
tools, 6
tools, requirements, 17–22
transformations, 12, 25–30
transformations, challenges, 20
transparency, 24
Version Control System, 124
Version Migration System, 124
Workflow Monitor, 124
extraction. *See* data extraction
Extraction System, 115–116
Extreme Programming (XP), 301

F

fact tables
accumulating snapshot fact table, 260–261
fact table loader, 121
loading, 264–265
partitioning, 264–265
bulk loading, 246–251
dimension tables, 251–260
Insert / Update step, 109
loader, 121
loading, 245–267
periodic snapshot fact tables, 260–261
fact table loader, 121
loading, 263–264
provider system, 122
rental star schema, 79
snapshots, 260–261
SOF, 261–263
loading, 265–266
transaction grain fact tables, 121
facts, late-arriving data, 256
failure hops, 90
Fetch Customer Address, 96–98
`Field description`, transformation log tables, 369
Field Splitter step, 145
Field to split, Split field to rows step, 499
fields
JavaScript, 395
maps, 25
rows, 27
Select values step, 397
text files, 384
Fields grid, Regex Evaluation step, 506
Fields tab, 237
Add XML step, 539–541
Get data from XML step, 535–536
The fields that make up the grouping, Denormalize Special Features, 104
FIFO. *See* first in, first out
`file`, 324
File locking, 42
File Output step, `export_xml_from_db`, 538
File repository, 41
File tab
Get date from XML step, 531–532
RSS Ouput step, 566–567
file-based version control systems, 342–344
filename, metadata, 36
`filename`, PostgresSQL, 250
Filename defined in a field, RSS Output step, 567
Filename field, RSS Output step, 567
`FileObject`, 619
Filter rows step, 198, 199
CRC, 485
UI, 609
Filter step, 108
Find Duplicate Query Wizard, Access, 117
first in, first out (FIFO), 247–248
Fixed File Input step, 385
`StepInterface`, 617
fixed width text files, 129
flags, `Boolean`, 94
floating point numbers, 28
JSON, 522
“follow when result is false” job hop, 32
“follow when result is true” job hop, 31–32
Force all to lower case, Database Connection, 38
Force all to upper case, Database Connection, 39
`Foreign key`, satellites, 470
foreign keys
Execute SQL script step, 252
parent key, 242
referential integrity, 251–252
Sakila, 105
SOF, 265
tables, 77, 208

forks, 591–592
FormLayout, 607–608
Formula step, Calculator step, 201
forums, 631–632
Free Software Foundation (FSF), 570
Freebase, 549–558
  data extraction, 553–558
  MQL, 551–553
  performance, 552
  read service, 550–551
  scalability, 552
  web services, 550
  Wikipedia, 549–550
FSF. *See* Free Software Foundation
full table scan, 391
Function Browser, SAP, 141
functional testing, 21, 306
fuzzy logic, 195
Fuzzy match step, 170, 171, 195–198

G

GA. *See* General Availability
gap penalty, 171
gender, coding for, 175–176
`GENERAL`, job entry results, 34
General Availability (GA), 629–630
Generate Row with URLs, Generate Rows step, 526
Generate Rows step, 314
  Add sequence step, 213
  `export_xml_from_db`, 537
  Freebase, 554
  `import_xml_into_db.ktr`, 526
  RSS Ouput step, 564
  SAP Input step, 142
  SOAP, 547
`generateJobMeta()`, 586
GeoKettle, 591
GeoNames, 137, 172
`GET`, SOAP, 548
`get()`, User Defined Java Class step, 620
Get closer value, 196–197
Get data from XML step, 361, 518, 530–536
  `import_xml_into_db.ktr`, 527
  SOAP, 548
Get Fields button, 129
  Insert / Update step, 231
Get File names step, 203
Get Lookup Fields button, Database lookup step, 223
Get rows from result step, 576–577
Get System Info step, 191
  command-line parameters, 325–326
  late-arriving data, 259
  transformation log tables, 367, 370
Get update fields, Insert / Update step, 231–232
Get variables step, JavaScript, 396
Get XPath nodes, Get data from XML step, 522
`getCopy()`, 617
`getDialogClassName()`, 622
`getEntryNr`, 587
`getExitStatus`, 587
`getFields()`
  `StepInterface`, 619
  `StepMetaInterface`, 599, 604
`getInputRowMeta()`, 615
`getLogChannelId`, 588
`getNrErrors`, 587
`getNrFilesRetrieved`, 588
`getNrLinesDeleted`, 588
`getNrLinesInput`, 587
`getNrLinesOutput`, 587
`getNrLinesRead`, 587
`getNrLinesRejected`, 588
`getNrLinesUpdated`, 588
`getNrLinesWritten`, 588
`getOptionalStreams()`, 600
`getPartition()`, 625
`getRequiredFields()`, 600
`getResourceDependencies()`
  `JobEntryInterface`, 622
  `StepMetaInterface`, 600
`getResult`, 587
`getResultFilesList`, 588
`getRow()`, 614–615, 618
`getRowFrom()`, 616
`getRows`, 588
`getRunThread(stepname, copy)`, 576
`getSlaves`, 416
`getSQLStatements()`
  `JobEntryInterface`, 622
  `StepMetaInterface`, 599
`getStepIOMeta()`, 600
`getUniqueStepCountAcrossSlaves`, `StepInterface`, 617
`getUniqueStepNrAcrossSlaves()`, 617
`getUsedDatabaseConnections()`, 622
`getusedLibraries()`, 600
`getXML()`, 583
  `JobEntryInterface`, 622
  `StepMetaInterface`, 599
`GetXMLData - Different Options.ktr`, 299
`getXulOverlayFile()`, 628

GIS. *See* Graphical Information Systems
Globally Unique identifiers (GUID), JavaScript, 395
`globalreplace.sh`, 347
GNOME, launchers, 62–63
GNU Public License (GPL), 343
GoldenGate, Oracle, 163
good-enough solutions, regular expressions, 501–502
Goodman, Nicholas, 19
Google Wave, 626
GPL. *See* GNU Public License
Graphical Information Systems (GIS), 591
graphical user interface (GUI), 24
  Java API, 571
grid-based services, 437
Guess button, Insert / Update step, 232
GUI. *See* graphical user interface
GUID. *See* Globally Unique identifiers
`guid`, `item`, 559
`.gzip`, 517

H

`handleStreamSelection()`, `StepMetaInterface`, 600
hard disks, 386–387
heaps, maximum size, 71
`HellowworldStepDialog.java`, 609–613
hierarchy
  dimension builder, 119–120
  dimensions, 270
  flattener, 20
  ragged, 120
  recursion, 120, 242–243
  variables, 120
Hillyer, Mike, 74
history
  data quality, 168
  Dimension lookup / update step, 235–236
  transformation log tables, 367–368
HOLAP. *See* Hybrid OLAP
`/home/ubuntu/runCarte.sh`, 441
`hop`, 346
hops, 7
  failure, 90
  jobs, 31–32
  loops, 27
  rows, 26, 27
  success, 90
  transformations, 25, 26–27
  unconditional, 88
`hops`, 346
Host Name, Database Connection, 38
HTML
  attributes, 520
  data extraction, 520
  elements, 520
  e-mail, 339
  JavaScript, 520
  web pages, 520
  web services, 520
HTTP. *See* Hypertext Transfer Protocol
Http Authentication, Web services lookup step, 545
HTTP client step, 516
  extraction, 137–138
  Freebase, 555
  `import_xml_into_db.ktr`, 526–527
  SOAP, 547–548
`HTTP GET`, 548
  Freebase, 550
HTTP Post step, 516, 548
`Hub Surrogate Keys()`, 469
hubs
  attributes, 468
  DV, 467–468
  Sakila, 472–473
  surrogate keys, 469
  tables, 467
Hudson, 60
Hunter, David, 518
Hybrid, SCD, 119
Hybrid OLAP (HOLAP), 272
Hyde, Julian, 458
Hypertext Transfer Protocol (HTTP), 515–517
  data formats, 517
  XML/A, 277, 278

I

IaaS. *See* Infrastructure as a Service
`../id`, XPath, 535
`../@id`, XPath, 535
IDE. *See* integrated development environment
identifiers, 317
`IDENTITY`, 217
`identity`, 641
`id_existing`, 480
Ignore comments?, Get data from XML step, 533
Ignore empty file, Get data from XML step, 534
il8n, 590
`image`, 601

Imhoff, Claudia, 296
impact analysis, 22
date lineage, 361–363
ETL, 125
StepMetaInterface, 599
import, repositories, 350–351
Import partitions button, Partitioning schema, 429
import_xml_into_db, Get data from XML step, 532
import_xml_into_db.ktr, 525–527
Execute SQL script step, 526
"in" tab, Web services lookup step, 545–546
Include date in filename, RSS Output step, 567
Include filename in result and Filename fieldname, Get data from XML step, 534
Include rownum in output, RSS Input step, 561–562
Include rownum in output?, Split field to rows step, 500
Include stepnr in filename, RSS Output step, 567
Include time in filename, RSS Output step, 567
Include url in ouput, RSS Input step, 561–562
incoming hops, 26
Increment by field, Step sequence step, 212
incrementLinesInput(), 618
incrementLinesOutput(), 618
incrementLinesRead(), 618
incrementLinesRejected(), 618
incrementLinesSkipped(), 618
incrementLinesUpdated(), 618
incrementLinesWritten(), 618
indexes
performance, 390–392
tables, 392
IndexOfValue(), 606–607
InfiniDB, 123
info, 345
InfoBright, 123
Informatica, 9
Infrastructure as a Service (IaaS), 437
init()
StepInterface, 614
User Defined Java Class step, 459
InjectDataIntoTransformation.java, 578–579
Injector step, 578
Inmon, Bill, 465
Input source step, 479
SQL, 483
Input step
extraction, 128
Mondrian, 274–275
OLAP, 274, 278–279, 281
process, 278
Input Table step, 297
Input vault step, 479
input_id, 101
input/output, 380
$, 67
inputRowMeta, StepMetaInterface, 604
Insert, Dimension lookup / update step, 238
INSERT, 157
Insert, bulk loading, 251
Insert / Update step, 101–102
accumulating snapshot fact tables, 264
CDC, 155, 163
Combination lookup / update step, 241
fact table, 109
keys, 230–231
SCD, 229–230
Update fields, 231–232
installation, 58–63
rental star schema, 81
Sakila database, 77
Installer, Enterprise Edition, 636
Integer, 27
Date, conversion, 30
integrated development environment (IDE)
plugins, 596–597
Spoon, 55–57
Integrated Scheduling, Enterprise Edition, 636
integration. *See also* data integration
testing, 307
internal counters, Add sequence step, 211–213
internal variables, 428–429
Internal.Cluster.Master, 639
Internal.Cluster.Size, 639
${Internal.Job.Filename.Directory}, 96–97
Internal.Job.Filename.Directory, 638
Internal.Job.Name, 638
Internal.Job.Repository.Directory, 638
Internal.Kettle.Build.Date, 638
Internal.Kettle.Build.Version, 637
Internal.Kettle.Version, 637
Internal.Slave.Server.Name, 639
Internal.Slave.Transformation.Number, 639
Internal.Step.CopyNr, 639
Internal.Step.Name, 639
$(Internal.Step.Partition.ID), 429

Internal.Step.Partition.ID, 638
$(Internal.Step.Partition.Number), 429
Internal.Step.Partition.Number, 639
Internal.Step.Unique.Count, 639
Internal.Step.Unique.Number, 639
${Internal.Transformation.Filename.Directory}, 96–97
$(Internal.Transformation.Filename.Directory), 530
Internal.Transformation.Filename.Directory, 638
Internal.Transformation.Name, 638
Internal.Transformation.Repository.Directory, 638
internationalization, 590
Internet Relay Chat (IRC), 633
inter-table dependencies, 183
interval logging, 453
intra-table dependencies, 183
intrusive CDC, 16, 155
IRC. *See* Internet Relay Chat
Is a file
  filename is defined in a field, XSD Validator step, 529–530
  let me specify filename, XSD Validator step, 529–530
Is defined inside XML, XSD Validator step, 529–530
ISNULL, 484
ISO8601, 170
isStopped, Result, 588
item, RSS, 559–560
Item tab, RSS Ouput step, 565–566

J

J2EE. *See* Java 2 Enterprise Edition
JAAS. *See* Java Authentication and Authorization Service
Jackrabbit, 350
jar, 587
.jar, 70, 517
  libext/, 587
Jaro and Jaro-Winkler algorithm, 171
Jaro-Winkler algorithm, 198
Java
  AMI, 440
  API, 570–574
    jobs, 573–574
    parameters, 579–580
    transformations, 572–573
    variables, 579–580
  expressions, 70–71
  installation, 58–59
  user-defined expressions and classes, 520
Java 2 Enterprise Edition (J2EE), 459
Java Authentication and Authorization Service (JAAS), 414
Java Content Repository (JCR), 350, 626
Java Development Kit (JDK), 58
  jar, 587
Java Message Service (JMS), 449, 459–461
Java Naming and Directory Interface (JNDI), 64–65, 93
Java Runtime Environment (JRE), 58
  variables, 642
Java Virtual Machine (JVM), 19, 58
  logging, 453
  rows, 397
  variables, 43
java.io.tmpdir, 642
JavaScript, 20, 202
  DataCleaner, 153
  eval(), 556
  HTML, 520
  job entries, 35–36
    logging, 366
  Mondrian, 276–277
  performance, 394–396
  variables, 43
  XML/A, 281
JavaScript Object Notation (JSON)
  example, 549–558
  Modified Java Script Value step, 522
  plugins, 522
  syntax, 521–522
  transformations, 523
  web services, 520–523
  XML, 520
java.security.auth.login.config, kettle.properties, 414
java.version, 642
JCR. *See* Java Content Repository
JDBC
  Database Connection, 93
  drivers, 72
  MySQL, 127
jdbc.properties, 64–65
  kettle.properties, 67
JD/Edwards, 18, 139
JDK. *See* Java Development Kit
jedox.com, 282
jface.jar, 596
Jira, 632
JMS. *See* Java Message Service

JNDI. *See* Java Naming and Directory Interface
JOB, 640
job, 324
job(s), 7
  canvas, 56
  command line, 322–326
  Database Connection, 37
  debugging, 56
  DWH, 31
  dynamic, 584–586
  ETL, 12, 30–36
  hops, 31–32
  Java API, 573–574
  Kitchen, 57
  log tables, 373–374
  loops, 399–400
  metadata, 36–37, 574
  Pan, 57
  parallelism, 33–34, 411
  performance, 399–400
  Run button, 83–84
  shared objects, 69
  slaves, 445
  Spoon, 82
  variables, 89
  XML, metadata, 346–347
job entries, 31
  backtracking, 32–33
  Boolean, 366
  Copy rows to result step, 399
  flow of execution, 90
  JavaScript, 35–36
    logging, 366
  log table, 373–374
  Mail, 90, 301, 337–340
  plugins, 570, 621–624
  results, 34–36
  serial execution, 90
  START, 88
  transformations, 88
  XML, 519–520
Job Scheduler, ETL, 124
JOBENTRY, 641
@JobEntry, 622
JobEntryDialogInterface, 622–624
JobEntryInterface, 622–624
jobentry-log-table, 346
job-log-table, 346
JobMeta, 574, 586
jobStatus, 416
Join comparison field, XML Join step, 543
Join condition properties, XML Join step, 542
join profile, 146
Join Rows step, 214
  Main step to read from, 398
jpalo.com, 283
JRE. *See* Java Runtime Environment
JSON. *See* JavaScript Object Notation
JSONP, 550
jtwitter, 454
junk dimensions, 241
  special dimension builder, 120
JVM. *See* Java Virtual Machine

K

Kalido, 6
.kettle, repositories.xml, 68
Kettle Logging Level, 330
KettleDatabaseRepository, 627
kettle-database-types.xml, 595
KETTLE_EMPTY_STRING_DIFFERS_FROM_NULL, 640
  NULL, 28
Kettle.exe, 55
/kettle/getSlaves, 444
KETTLE_HOME, 64
$KETTLE_HOME/.kettle/.languageChoice, 601
$KETTLE_HOME/.kettle/shared.xml, 589
kettle-job-entries.xml, 595
KETTLE_..._LOG_DB, 641
KETTLE_..._LOG_SCHEMA, 641
KETTLE_LOG_SIZE_LIMIT, 640
KETTLE_..._LOG_TABLE, 641
KETTLE_MAX_LOG_SIZE_IN_LINES, 365, 454, 640
KETTLE_MAX_LOG_TIMEOUT_IN_MINUTES, 454, 640
kettle-partition-plugins.xml, 595
KETTLE_PASSWORD, 67
KETTLE_PLUGIN_CLASSES, 619, 640
kettle.properties, 58, 66–67, 414
  logging, 365
  variables, 43
kettle.pwd, 58, 67
kettle-repositories.xml, 595
KETTLE_REPOSITORY, 67
KETTLE_SHARED_OBJECTS, 589, 640
KETTLE_STEP_PERFORMANCE_SNAPSHOT_LIMIT, 640
kettle-steps.xml, 595
KETTLE_USER, 67
KETTLEVFS, 619

The key field, Denormalize Special Features, 104
keys. *See also specific key types*
  Calculator step, 214
  dimension table, CDC, 80–81
  dimension tables, 109, 208–217
  Insert / Update step, 230–231
  JSON, 522
  SCD, 217
  source systems, 209
Keys tab page, Dimension lookup / update step, 234
key/value pairs, 508–513
  object members, 522
  Regex Evaluation step, 510–511
  text files, 509–510
Kimball, Ralph, 11, 113, 167, 191, 221, 228, 295, 465
Kitchen, 41, 44, 54, 322–326
  jobs, 57
  `level`, 336
  `logfile`, 334
  logging, 364
  transformations, 57
`Kitchen.bat`, 57
`kitchen.sh`, 57
`.ktr`, 345

L

LAF. *See* Look and Feel
`LAFpackage`, 591
Last version (without stream field as source), Dimension lookup / update step, 239
`lastmodifiedtime`, 183
`last_update`, 80
late-arriving data, 255–260
  dimensions, 256–260
  ETL, 122
  facts, 256
launchers, GNOME, 62–63
Lazy Conversion, 385, 387
lazy loading, 605
Lesser GNU Public License (LGPL), 569–570
  forks, 591
`level`, 324
  Kitchen, 336
  Pan, 336
Levenshtein algorithm, 171
LGPL. *See* Lesser GNU Public License
`libext`, 70
`libext/`, 591, 598
  `.jar`, 587
libraries
  embedding, 574
  plugins, 596
  `sapjco3.jar`, 141
  `StepMetaInterface`, 600
`libswt/`, 598
lightweight principle, 446–447
Limit, Get data from XML step, 534
Lindstedt, Dan, 9
lineage. *See* data lineage
Lineage and Dependency Analyzer, 125
`link`
  `channel`, 559
  `item`, 559
links
  DV, 468–469
  Sakila, 473–474
`link-to-link`, 472, 474
Linstedt, Dan, 466
`listdir`, 324
`listjobs`, 324
`listrep`, 324
`listtrans`, 325
`.lnk`, 62
`Load all date from table`, 253
`Load DTS`, 468, 469, 470
`Load End DTS`, 470
`load_data`, 298
loading. *See also* bulk loading
  lazy, 605
`loadXML()`, 599, 603
location outriggers, 97
Locking, Enterprise Edition, 636
logging, 333–336, 363–374
  architecture, 364–367
  avoiding, 398
  buffers, 364, 365–366
  CDC, 162–163
  channels, 366
  debugging, 364
  error messages, 364
  ETL, 22
  interval, 453
  JavaScript job entries, 366
  JVM, 453
  `kettle.properties`, 365
  Kitchen, 364
  levels, 335–336
  memory, 365
  Pan, 364
  parameters, 364
  rows, 363

Spoon, 57, 333–334, 364, 365
transformations, 453–454
variables, 367
log data change processing, 451
log tables, 367–374
channels, 372
job, 373–374
job entries, 373–374
performance, 371
step log tables, 370–371
transformation, 367–370
Log4J, Apache, 366
`logfile`, 324, 334
`loginmodulename`, 414
Look and Feel (LAF), 590
lookup cascade, 100
Lookup Language, 103
lookup mode, 232
Lookup Original Language, 103
Lookup schema field, Database lookup step, 222–223
lookup tables, 172–175
lookup values, Validator step, 180
Loop Xpath, Get data from XML step, 532–533, 535
loops
Freebase, 557
hops, 27
jobs, 399–400
`ls`, 248
LucidDB, 10, 123
bulk loading, 249
EII, 10
SQL, 10
wrappers, 10

M

Mail, 301, 336–340
Addresses tab, 337
Attached Files tab, 339–340
Email Message tab, 339
job entries, 90, 301, 337–340
Server tab, 337–338
Mail Failure step, 90, 185–186, 336–337
Mail Success step, 90, 336–337
Main step to read from, Join Rows step, 398
maintainability, ETL, 300–301
Make transformation database transactional, Database Connection, 40
`man crontab`, 327
Manage thread priorities?, Transformation Settings, 397
Management Console, Enterprise Edition, 636
Manufacturing Requirements Planning (MRP), 138
mapping. *See* data mapping
Mark Attribute rows with id of header row, Modified Java Script Value step, 511
master, 417
AMI, 442–443
Carte, 441
dynamic clustering, 434
transformations, 421–422
`<master>`, 435
Mastering Data Warehouse (Imhoff), 296
Max % errors allowed, Data Validator step, 188
Max nr errors allowed, Data Validator step, 188
Max number of articles, RSS Input step, 561–562
`MAX(id) FROM test_sequence`, 213–214
maximum heap size, 71
Maximum nr of lines in logging windows, Spoon, 365
`max_log_lines`, 412
`max_log_timeout_minutes`, 412
Maydanchik, Arkady, 168–169
MD5, 484
MDA. *See* Model Driven Architecture
MDX. *See* Multi Dimensional eXpressions
measures
Palo, 289
performance, 380–382
SOF, 265
Mechanical Turk, 437
memory
logging, 365
lookups, 253
performance, 393
Sort rows step, 453
Stream lookup step, 453
streams, 577
transformations, 452, 453
Merge join step, 479
CRC, 485
Merge Rows step, 160–161
`MERGE/UPDATE`, 249
message bundles, 601
metadata
data extraction, 359
data profiling, 17
data validation, 182
database, 588–590

description, 37
directory, 36
ERP, 14, 139
ETL, 21, 344–350
graphical user interface, 24
XML, 24
export, StepMetaInterface, 600
extended description, 37
filename, 36
jobs, 36–37, 574
names, 36
replacing, 588–590
repositories, 348–350
rows, 557, 606–607
steps, 28
spreadsheets, 297
StepMetaInterface, 599
transformations, 36–37, 421–425, 572–573
User Defined Java Class step, 620
values, 605–606
XML, 345–347
jobs, 346–347
transformations, 345–346
Metadata Repository Manager, 125
Metaphone algorithm, 171
Metaweb Query Language (MQL), 551–553
methods
partitioning, 425
plugins, partitioning, 624–626
micro-batches, 450
Microsoft SQL Server 2008 Analysis Services (MSAS), 271, 277–280
Milestone, 630
Min nr of rows to read before doing % evaluation, Data Validator step, 188
mini-dimensions, 239–240
special dimension builder, 120
mirc.com, 633
Model, Spoon, 302–303
Model Driven Architecture (MDA), 6
Modified Java Script Value step, 314
DynamicJob, 586
Freebase, 556–557
JSON, 522, 549
Mark Attribute rows with id of header row, 511
MOLAP. *See* Multi-dimensional OLAP
Mondrian, 242
Aggregation Designer, 123, 267
Input step, 274–275
JavaScript, 276–277
OLAP, 271, 273–277
Split Time step, 276
Split-field step, 276
Strings cut step, 276
MonetDB, 123
monitoring, ETL, 333–340
Monitoring tab, Transformations settings, 381
MQL. *See* Metaweb Query Language
MRP. *See* Manufacturing Requirements Planning
MSAS. *See* Microsoft SQL Server 2008 Analysis Services
Multi Dimensional eXpressions (MDX), 269–270
Query, Mondrian Input step, 274
Multi-dimensional OLAP (MOLAP), 123, 269, 272
multiline mode, 507
multi-paths, backtracking, 32–33
multiple updates, CDC, 155
multi-threading, 403–411
Blocking Step, 410
data pipelining, 407–408
database connections, 408–409
Execute SQL step, 409–410
order of execution, 409–410
row distribution, 404–407
row merging, 405–406
multi-valued attributes, 498–500
multi-valued dimension bridge table builder, ETL, 121–122
-Mxx, 60
MySQL, 73–74
Bulk Loader, 248, 249
CDC, 162–163
JDBC, 127
NOW(), 484
RDBMS, 77, 134
SET, 103, 498
SUPER, 82
mysqlbinlog, 163
mysql_native.xul, 628

N

name, 346
names
ETL, 24, 298–299
job entry results, 34
metadata, 36
parameters, 44
pipes, 248
Namespace aware?, Get data from XML step, 533

natural keys
business keys, 210
dimension tables, 99
junk dimensions, 241
near real-time data integration, 450
Needleman-Wunsch algorithm, 171
network latency, 369–370
network speed, 390
New validation button, 179
NIO buffer size, 386
non-intrusive CDC, 16, 155
non-relational data formats, 498
non-relational tabular formats, 498–501
non-tabular data formats, 498
`norep`, 323
normalization, 218
Normalize Special Features, 104
`notepads`, 346
notes, 318
transformations, 25
`NOW()`, 484
Nr of errors fieldname, Data Validator step, 188
`NULL`
Add XML step, 539, 541
CRC, 484
data profiling, 146
data validation, 17, 180–181
Database lookup step, 225
DV, 471
`KETTLE_EMPTY_STRING_DIFFERS_FROM_NULL`, 28
source data, 179
`String`, 28
`Number`, 27
Palo, 285
`String`, 29–30
Number analysis, DataCleaner, 149
numbers, JSON, 522

O

OASI. *See* One Attribute Set interface
`OBF`, 414
obfuscated passwords, 67–68, 414
object literals, JSON, 522
object members, JSON, 522
`object_timeout_minutes`, 412
OCI. *See* Oracle Call Interface
ODBC, 93
ODS. *See* operational data store
OEM version, PDI, 590–591
OLAP. *See* online analytical processing
OLTP. *See* OnLine Transaction Processing
Omit null values from XML result, Add XML step, 539
Omit XML, Add XML step, 539
One Attribute Set interface (OASI), 303
online analytical processing (OLAP), 123
aggregate tables, 266
cubes, 270–271
Input step, 274, 278–279, 281
`process`, 278
Mondrian, 271, 273–277
multidimensional, 269
Palo, 282–291
positioning, 272–273
storage types, 272
XML/A, 277–282
OnLine Transaction Processing (OLTP), 2, 269–291
database, 75
dimension tables, 226
Open Office Calc, 297
OLAP, 273
Open Symphony, 327
OpenERP, 15
operating systems, scheduling, 322
operational data store (ODS), 4, 10
Oracle
Attunity Stream, 163
connect by prior, 20
E-Business Suite, 18
GoldenGate, 163
RDBMS, 134
Spatial, 591
SQL*Loader, 247, 249
Warehouse Builder, 6
ETLT, 9
Oracle Call Interface (OCI), 247
`order`, 346
`ORDER BY`, 225, 389, 479
`org.pentaho.di.core.database.DatabaseInterface`, 627
`org.pentaho.di.trans.step.StepInterface`, 614
original transformations, 421
`os.arch`, 642
`os.name`, 642
`os.version`, 642
`OUPUT_DIR`, 319
Out of Memory, 71
outgoing hops, 26
output directory, 319
Output Fields, XSD Validator step, 529

Output one row, concatenate errors with separator, Data Validator step, 187
Output String Field, XSD Validator step, 529
Output Value, Add XML step, 539
Overwrite, SCD, 119

P

PAD. *See* Pentaho Aggregation Designer
pagila, 74
Pair letters similarity algorithm, 171
Palo, 123, 273, 274, 282–291
Palo Cell Output step, 289–291
Palo Cells Input step, 285–289
Palo Dimension Input step, 285–289
Palo Dimension Output step, 289–291
Pan, 41, 44, 54, 322–326
  jobs, 57
  `level`, 336
  `logfile`, 334
  logging, 364
  transformations, 57
`Pan.bat`, 57
`pan.sh`, 57
parallelism, 18–19
  data extraction, 538
  jobs, 33–34, 411
  performance, 385–386
  sorting, 393–394
  text files, 385–386, 387
  transformations, 27, 404
Parallelizing/Pipelining System, ETL, 125
parameters, 318
  command-line, 323–324, 325–326
  delays, 457
  Java API, 579–580
  logging, 364
  named, 44
  queries, 135–136
  SQL, 99
  transformation log tables, 369
  transformations, 579–580
  Validator step, 179
  Version Migration System, 353–355
Parameters tab, 44
parent key, foreign keys, 242
`Partitioner`, 625–626
partitioning, 18–19, 425–430
  accumulating snapshot fact tables, 264–265
  checksums, 426
  clustered transformations, 430
  CSV File Input step, 428
  database, 40, 429–430
  methods, 425
    plugins, 624–626
  plugins, 426
    methods, 624–626
  round robin, 425
  schema, 425–427
  tables, performance, 392
Partitioning schema, Import partitions button, 429
`pass`, 323
`password`, `jdbc.properties`, 65
passwords, 58
  obfuscated, 67–68, 414
  UI, 609
Pattern finder, DataCleaner, 149
patterns, 501
`pauseTrans`, 415
PDI. *See* Pentaho Data Integration
Pearson, William, 270
peer/expert reviews, 297
##pentaho, 633–634
Pentaho Aggregation Designer (PAD), 123
Pentaho BI, Quartz scheduler, 322, 327–333
Pentaho Data Integration (PDI), 60, 328–330
  AMI, 440
  DataCleaner, 148
  enterprise repository, 350
  Java API, 571
  OEM version, 590–591
  slave servers, 413
Pentaho Report Designer (PRD), 574–575
Pentaho repository, 41
Pentaho Solutions (Bouman and van Dongen), 228, 327
`PentahoSystemVersionCheck`, 330–331
Peoplesoft, 15
`perf-log-table`, 345
performance
  buffers, 380
  constraints, 392
  CPU, 394–398
  data sorting, 392–394
  database, 388–392
  Freebase, 552
  hard disks, 386–387
  indexes, 390–392
  JavaScript, 394–396
  jobs, 399–400
  log table, 371
  measures, 380–382
  memory, 393
  parallelism, 385–386
  relational databases, 390

rows, 382–383
SQL, 388
table partitioning, 392
text files, 384–387
transformations, 377–398
triggers, 392
tuning, 377–401
periodic snapshot fact tables, 260–261
fact table loader, 121
loading, 263–264
perspective, Spoon, 302
pipes, named, 248
PIT. *See* Point-In-Time
pivot fields, Palo, 288
platform independence, ETL, 18
Plug-in Registry, 595
@Step, 601
plugins, 20
architecture, 593–599
database, 627–628
ERP, 140
IDE, 596–597
JavaScript, 395
job entries, 570, 621–624
`@JobEntry`, 622
JSON, 522
LGPL, 570
libraries, 596
methods, partitioning, 624–626
partitioning, 426
repositories, 626–627
steps, 570, 619
transformation step, 599–619
types of, 594–595
`plugins`, 440
`plugins/`, 595
`plugins/steps`, 619
Point-In-Time (PIT), 472
Port Number, Database Connection, 38
`POST`
Freebase, 550
SOAP, 548
PostGIS, 591
PostgresSQL, 74
bulk loading, 250
Power*Architect, 79
Powerplay, Cognos, 269
PRD. *See* Pentaho Report Designer
`prd`, 354
preparation of statements, 388
`prepareExecution`, 415
Preview option, 312–315
`PreviewRowsDialog`, 609
`Primary key`, 468, 469, 470
primary keys
satellites, 470
source system, 210
surrogate primary keys
dimension tables, 80
tables, 77
`UPDATE`, 470
Prism, 6
privacy, 308
private schedule, 331
Problem Escalation System, 125
process, 295
error handling, 184–187
`process`, OLAP Input step, 278
`processRow()`, 614, 615
`processRows()`, 456, 460
profiling
column profiling, 17, 146
data profiling, 16–17, 127–128, 146–154
metadata, 17
dependency profiling, 146
join profile, 146
properties
built-in, 637–642
JSON, 522
Proxy Host, Web services lookup step, 545
Proxy Port, Web services lookup step, 545
Prune Path to handle large files, Get data from XML step, 534
`pubDate`, `item`, 559
public schedules, 331
Punch through, Dimension lookup / update step, 238
`putError()`, 617
`putRow()`, 362, 557, 579, 616, 618
`putRowTo()`, 616
`pwd/`, 434

Q

Quartz scheduler, Pentaho BI, 322, 327–333
queries
aggregate tables, 266
parameters, 135–136
SQL, `SELECT`, 553
Query, MDX, Mondrian Input step, 274
Quote all in database, Database Connection, 38

R

ragged hierarchy, 120
RC. *See* Release Candidate
`-RCxx`, 60

RDBMS. *See* Relational Database Management System
RDS. *See* Relational Database Service
Read articles from, RSS Input step, 561–562
read service, Freebase, 550–551
Read source as Url, Get data from XML step, 532
`readRep()`, `StepMetaInterface`, 599, 603
Really Simple Syndication (RSS), 18, 558–567
  `channel`, 558–559
  `item`, 559–560
  transformations, 563
  web services, 517
real-time business intelligence, 450
real-time data integration, 449–461
  CDC, 450–451
  source system, 451
  transformation streaming, 452–461
real-time extraction, 138
  CDC, 155, 163
real-time transformation streaming, debugging, 457–478
`Record source`, 468, 469
  satellites, 470
Recovery and Restart System, ETL, 124
Recurrence, 332
recursion, hierarchies, 120, 242–243
reference tables
  data cleansing, 172–179
  data conformation, 175–179
Referencing, 42
referential integrity, 42
  data quality, 168
  foreign keys, 251–252
RefinedSoundEx algorithm, 171
Regex Evaluation step, 204, 504–508
  key/value pairs, 510–511
Regex matcher, DataCleaner, 149
`registerSlave`, 416
regression tests, 307
regular expressions, 503–508
  capture groups, 200, 205
  data cleansing, 203–205
  DataCleaner, 151–152
  good-enough solutions, 501–502
  Validator step, 180
Relational Database Management System (RDBMS), 134
  ETL, 497
  MySQL, 77
Relational Database Service (RDS), 438
relational databases, 39, 127
  CDC, 450
  performance, 390
  transformations, 497
Relational OLAP (ROLAP), 242, 272, 274
Release Candidate (RC), 60, 630
remote execution, slave servers, 413
Remote Function Calls (RFCs), 140, 146
Remote Steps, 422
Rename fields step, XML/A, 280
rental star schema
  dimension tables, 79–80
  fact table, 79
  installation, 81
  Sakila, 78–81
`rep`, 323
repeating groups, 500–501
Replace in string step, 170, 203
Report all errors, not only the first, Data Validator step, 187
`<report_to_masters>`, 436
repositories, 41–42
  database, 348–349
  export, 350–351
  files, 349
  import, 350–351
  managing, 350–352
  metadata, 348–350
  plugins, 626–627
  upgrade, 351–352
  Version Migration System, 352–353
  XML, 344
`RepositoriesMeta.readData()`, 573
`repositories.xml`, 68, 573
`Repository`, 626–627
`Repository.loadTransformation()`, 572
`RepositoryMeta`, 573
`resetStepIOMeta()`, `StepMetaInterface`, 600
resource exporter, 444
response time, DWH, 4
`Result`, 576–577, 587–588
Result Fieldname, XSD Validator step, 529
Result stream properties, XML Join step, 543
results tab, Web services lookup step, 546
Return/remove digits, data cleansing, 170
reuse
  ETL, 19, 300–301
  shared objects, 589
Revision management, 42
`RFC_READ_TABLE`, 143–144
RFCs. *See* Remote Function Calls
ROLAP. *See* Relational OLAP
`root`, 82

Root XML element, Add XML step, 539, 540–541
Ross, Margy, 221, 228
round robin, 386
  partitioning, 425
  sorting, 394
roundtrips, 388
row(s)
  Add sequence step, 405
  attributes, 497
  CSV File Input step, 394
  debugging, 314
  dimension tables, 90
  fields, 27
  hops, 26, 27
  JavaScript, 395
  job entry results, 34
  JVM, 397
  logging, 363
  metadata, 557, 606–607
    steps, 28
  multi-threading, 404–407
  performance, 382–383
  Sort rows step, 419
  static data, 397–398
  Table input step, 424
  Text File Output step, 405
  UI, 609
  User Defined Java Class step, 404
Row denormaliser step, 511–512
  Palo, 288
Row normaliser step, 500–501
`RowDataUtil`, 616
`RowListener`, 576
`RowMetaInterface`, 604, 606–607
Rownum fieldname, Split field to rows step, 500
Rownum in output and Rownum fieldname, Get data from XML step, 535
`RowProducer`, 577, 579
`RowSet`, 579, 617
RSS. *See* Really Simple Syndication
`rss`, 558–559
RSS Input step, 561–562
RSS Output step, 562–567
`R_STEP`, 348
`R_TRANSFORMATION`, 348
Run button, 83–84
`Run profiling`, 152
`running`, 439
`runtime.jar`, 596

S

SaaS. *See* Software as a Service
Sakila
  business keys, 527
  CDC, 108
  data mapping, 524–525
  database, 73–110
    installation, 77
    subject areas, 75–76
  Database Connection, 90–95
  DV, 472–486
  ETL, 73–110, 81–84
  foreign keys, 105
  hubs, 472–473
  links, 473–474
  rental star schema, 78–81
  satellites, 474
  snowflakes, 219–221
  Spoon, 81–84
  surrogate keys, 527
  XML, 523–544
SalesForce.com input step, 140
SalesForce.com output steps, 140
SAP
  data, 140–145
  Function Browser, 141
SAP Input step, 140
  Data Grid step, 142
  Generate Rows step, 142
  `sapjco3.jar`, 141
SAP Java Connector library (`sapjco3.jar`), SAP Input step, 141
`sapjco3.jar`. *See* SAP Java Connector library
SAP/R3, 14, 18, 141
Sarbanes-Oxley Act, 308
satellites
  DV, 469–471
  primary keys, 470
  Sakila, 474
  `WHERE`, 484
`saveRep()`
  `JobEntryInterface`, 622
  `StepMetaInterface`, 599, 603
scalability
  ETL, 18–19
  Freebase, 552
SCD. *See* Slowly Changing Dimension
Schedule Creator, 331–332
Scheduling, Spoon, 302
scheduling
  action sequence, 333

ETL, 321–333
operating systems, 322
schema. *See also* XML Schema
clustering, 417–418
Database Connection, 39
DataCleaner, 148
dynamic clustering, 434
partitioning, 425–427
Schema name field, Add sequence step, 217
SCM. *See* software configuration management
screens, 191
Script Values step, 394–395
scripts, 20. *See also* JavaScript
ETL, 5, 200–205
startup, 70
Scrum, 13, 301
`searchInfoAndTargetSteps()`, `StepMetaInterface`, 600
Secure Sockets Layer (SSL), 337
Security, Enterprise Edition, 636
Security repository, 42
Security System, 125
`sed`, 347
`SELECT`, 553
Select values step, 94, 100, 397
semi-additive, 260
SOF, 265
semi-structured data, 501–508
Separate history table, SCD, 119
`sequence_value`, 213
serial execution, job entries, 90
Serialize to file step, CDC, 164
Server tab, Mail, 337–338
services. *See also* web services
grid-based, 437
slave servers, 414–416
`SET`, 499
MySQL, 103, 498
Set Environment Variables step, 354–355
Set Variables step, 216
`setDefault()`, 599, 604
SETI@Home, 433
`setOuputDone()`, 615
sets, CSV, 498
Settings tab page, Regex Evaluation step, 504–506
`.sh`, 58
SHA-1, 484
shadow copies, 31
sharding, database, 40
shared objects, 68–69
database, 589
jobs, 69
Spoon, 69
transformations, 69
`shared.xml`, 68–69
shortcuts, Spoon, 62
shrunken or rolled dimensions, special dimension builder, 120
Simple Object Access Protocol (SOAP)
accessing services directly, 546–549
examples, 544–549
extraction, 138
OLAP, 274
WDSL, 517
web services, 517
Web services lookup step, 544–546
XML/A, 277
slave(s)
AMI, 443–444
jobs, 445
transformations, 421–422
Slave Browser tab, Spoon, 457
slave servers
Carte, 411–416, 435
configuration, 411–412
PDI, 413
remote execution, 413
services, 414–416
Sort rows step, 419
Spoon, 413
Table input step, 424
XML, 413
slices, 271
Slowly Changing Dimension (SCD), 20, 228–239
Bus Architecture, 118–119
Dimension lookup / update step, 232–237
dimension tables, 118
Dimensional Data Warehouse, 118–119
ETL, 118–119
hybrid, 238–239
Insert / Update step, 229–230
keys, 217
type 1, 229–232
type 2, 232–237
type 3, 237–238
Small and Medium Business (SMB), 139
small periodic batches, 450
smart keys, 80, 108
SMB. *See* Small and Medium Business
SMTP, 337
snapshots
CDC, 146, 158–162
fact tables, 121, 260–261, 263–264

Sniff test during execution, Spoon, 457–478
sniffing, 314–315
sniffStep, 416
snippets, User Defined Java Class step, 620
snowflakes
  dimension tables, 97, 218–225
  Sakila, 219–221
SOAP. *See* Simple Object Access Protocol
soapUI.org, 547
SOF. *See* state-oriented fact tables
Software as a Service (SaaS), 437
software configuration management (SCM), 626
sorting
  clustering, 394
  data, performance, 392–394
  database, 393
  parallelism, 393–394
  round robin, 394
Sort rows step, 479
  memory, 453
  rows, 419
  slave servers, 419
  Sort size (rows in memory), 393
Sort size (rows in memory), 393
Sort System, 124–125
Sorted Merge step, 419
Soundex algorithm, 171
source code
  Java API, 570
  plugins, 594
source data
  CDC, 155–157
  data cleansing, 173
  NULL, 179
  PRD, 574
  RSS Ouput step, 564
  tabular format, 497
source system
  Database lookup step, 222
  keys, 209
  primary keys, 210
  real-time data integration, 451
Source XML field, XML Join step, 542
sourceforge.net, 59–60, 570
source_system, 178
Spatial, Oracle, 591
special dimension builder
  dimensions, 120
  ETL, 120
special_features, 103–104
Split field to rows step, 104, 499–500
Split Time step, Mondrian, 276
Split-field step, Mondrian, 276
Spoon, 41, 54
  Add sequence step, 211–217
  agile development, 301–302
  canvas, 318
  Combination lookup / update step, 241
  Copy tables wizard, 584
  dynamic transformations, 580–583
  ETL, 81–84
  Execute a transformation, 413
  extraction, 128
  IDE, 55–57
  jobs, 82
  logging, 57, 333–334, 364, 365
  perspective, 302
  Sakila, 81–84
  shared objects, 69
  shortcuts, 62
  Slave Browser tab, 457
  slave servers, 413
  Sniff test during execution, 457–478
  transformations, 57, 82
  variables, 44
Spoon.bat, 55, 62
.spoonrc, 64
spoon.sh, 55
spreadsheets
  data acquisition, 15
  metadata, 297
  testing, 311
SQL
  attributes, 484
  Business Objects, 9
  dynamic jobs, 584
  ELT, 9
  Informatica, 9
  Input source step, 483
  LucidDB, 10
  ORDER BY, 225, 479
  parameters, 99
  performance, 388
  query, SELECT, 553
  StepMetaInterface, 599
  streams, 99
  WHERE, 553
SQL Server
  RDBMS, 134
  XML/A, 278
SQL statements to execute after connecting, Database Connection, 39
SQLEditor, 609
SQL*Loader, Oracle, 247, 249
SQLPower, 118, 154

SQLStream, 458
src/, 597
SSL. *See* Secure Sockets Layer
-stable, 60
staging area, 8
  ODS, 10
standard input (STDIN), 247–248, 250
Standard measures, DataCleaner, 149
standardization, 297
star schema, 78–81. *See also* rental star schema
  CDC, 227–228
  denormalization, 226
  dimension tables, 226–228
  tables, 495
START, job entries, 88
Start at value field, Step sequence step, 212, 216
STARTDATE, 369
STARTDATE-ENDDATE, 369–370
startExec, 415
startJob, 416
startTrans, 415
startup scripts, 70
state-dependent objects, data quality, 168
state-oriented fact tables (SOF), 261–263
  loading, 265–266
static data, rows, 397–398
static dimensions
  special dimension builder, 120
  tables, 84–87
static testing, 307
static values, JavaScript, 396
status, 415
STDIN. *See* standard input
STEP, 640
step, 346
@Step, Plug-in Registry, 601
step, 557
Step name, transformation log tables, 369
Step name field, Step sequence step, 212
StepDataInterface getStepData(), 600
StepDialogInterface, 607–613
step_error_handling, 346
StepInterface, 614–619
StepInterface getStep(), 600
step-log-table, 346
stepMetaInterface, 599–607
StepMetaInterface **check**, 599
steps, 7. *See also specific steps*
  outgoing hops, 26
  plugins, 570, 619
  row metadata, 28
  shared objects, 589
  transformations, 26
  VPLs, 47–49
stopJob, 416
stopTrans, 415
stream(s), 83
  Add XML step, 538
  data, 577
  data integration, 450
  editor, 347
  extraction, 138
  memory, 577
  SQL, 99
  StepMetaInterface, 600
  Table output step, 538
  transformations, 452–461, 577
  Web services lookup step, 517
  XML Join step, 541
Stream Datefield, Dimension lookup / update step, 235–236
Stream lookup step, 173, 178, 253–255, 383
  import_xml_into_db.ktr, 527
  memory, 453
StrictHostKeyChecking, 641
String, 27
  Boolean, 30
  Date, 29
  NULL, 28
  Number, 29–30
  Palo, 285
string(s), 384
  JSON, 522
  UI, 609
String analysis, DataCleaner, 149
String getDialogClassName(), StepMetaInterface, 600
string literals, JSON, 522
Strings cut step, Mondrian, 276
structural testing, 21
Stylus Studio, 523
subscription, 635
subsystems, ETL, 113–126
subtansformation interface, 101
Subversion, Apache, 343, 570
success hops, 90
SugarCRM, 15
SUPER, 82
supportsErrorHandling(), 600
surrogate key(s), 118
  Add sequence step, 211–217
  business keys, 210
  creation system, 119
  database sequence, 217

Dimension lookup / update step, 234–235
dimension tables, 209, 251–260
DWH, 210
generating, 210–217
hubs, 469
`import_xml_into_db.ktr`, 527
pipeline, 121, 252–255
Sakila, 527
SOF, 266
XML, 527
surrogate primary keys
dimension tables, 80
tables, 77
`UPDATE`, 470
Switch/Case step, 189–190
SWT, Eclipse, 607
`swt.jar`, 596
synchronization, data, 9
Synchronize after merge step, 160–161
`sysdate`, 354

T

tab-delimited files, 128
table(s). *See also specific table types*
DataCleaner, 148
DV, 485–486
foreign keys, 77, 208
hubs, 467
indexes, 392
`link-to-link`, 472, 474
logging, 367–374
channels log tables, 372
job entries log table, 373–374
job log table, 373–374
performance log tables, 371
step log tables, 370–371
transformation log tables, 367–370
partitioning, performance, 392
star schema, 495
static dimensions, 84–87
surrogate primary keys, 77
Table daterange end, Dimension lookup / update step, 236
Table input step, 103, 596
aggregate tables, 266
CDC, 160
Data Grid step, 132
rows, 424
slave servers, 424
Stream lookup step, 254
Table output step, 216, 397
bulk loading, 250
CDC, 164
`commit size`, 390
data lineage, 358
dynamic templates, 584
`export_xml_from_db`, 538
`import_xml_into_db.ktr`, 527
streams, 538
Use batch updates for inserts, 389
`TableInput`, 595
`table_params`, 355
`TableView`, 609
tabular format
non-relational, 498–501
source data, 497
`tags/`, 342, 352
Talend, 6
Data Profiler, 154
`.tar`, 517
Target fields
Denormalize Special Features, 105
Insert / Update step, 230
Target XML field, XML Join step, 542
`.tar.gz`, 60
Task Scheduler, 327
TCP/IP
Carte, 57, 417
clustering, 423
templates, dynamic, 583–584
testing
automation, 311
CI, 311
Data Grid step, 311
dynamic, 307
ETL, 21, 306–312
integration, 307
spreadsheets, 311
static, 307
transformations, 311
upgrade, 312
`test_sequence.ktr`, 212–213, 215
text file(s)
extraction, 128–132
Web, 137
fields, 384
key/value pairs, 509–510
parallelism, 385–386, 387
performance, 384–387
reading, 384–387
writing, 387
Text file input step, 203, 384
Text file output step
CDC, 164
rows, 405

TextVar, 609
third normal form (3NF), 218
DV, 469
threads, 397. *See also* multi-threading
RowProducer, 579
3NF. *See* third normal form
time analysis, DataCleaner, 149
time dimensions, 239
time-outs, databases, 453
TIMESTAMP, 80
timestamps
CDC, 155–157, 163, 450
DV, 480
title, 559
TLS. *See* Transport Layer Security
/tmp/carte.log, 441
tokens, Get data from XML step, 536
tools, 41
ETL, 6
requirements, 17–22
top-down level-wise loading, 219
Tortoise SVN, 570
TPC-H, 253
traceability
of data, 467
DV, 471
TRANS, 640
Trans, 577
trans, 325
transaction grain fact tables, 121
<transformation>, 345
transformation(s), 7
action sequence, 328–330
architecture, 452
bottlenecks, 379–382
buffers, 406–407
Calculate Dimension Attributes, 85–86
canvas, 56
clustering, 417–425
partitioning, 430
command line, 322–326
data, 576–580
data conversion, 29–30
Database Connection, 37, 90–95
debugging, 56
deduplication, 195–199
dynamic
CSV, 580–583
Spoon, 580–583
error handling, 186–187
ETL, 12, 25–30
challenges, 20
Get data from XML step, 532
hops, 25, 26–27
Java
API, 572–573
expressions, 70–71
job entries, 88
JSON, 523
Kitchen, 57
logging, 453–454
master, 421–422
memory, 452, 453
metadata, 36–37, 421–425, 572–573
notes, 25
Pan, 57
parallelism, 27, 404
parameters, 579–580
performance, 377–398
phases, 452
relational databases, 497
RSS, 563
Run button, 83–84
shared objects, 69
slave, 421–422
Spoon, 57, 82
steps, 26
streams, 452–461, 577
testing, 311
variables, 89, 579–580
VPLs, 46
XML, metadata, 345–346
Transformation File, 330
Transformation Inputs, 330
transformation log tables, 367–370
Get System Info step, 367
history, 367–368
parameters, 369
Transformation Settings, Manage thread priorities?, 397
Transformation Step, 330
transformation step plugins, 599–619
Transformations Settings, 368
Monitoring tab, 381
transitive closure table, 242
trans-log-table, 345
TransMeta, 572, 577
TransMeta.getSQLStatements(), 566
transparency, ETL, 24
TRANS_PERFORMANCE, 640
Transport Layer Security (TLS), 337
transStatus, 415
triggers
CDC, 163, 450
database, 157–158
performance, 392

Truncate, bulk loading, 251
trunk/, 342
Trunk version, 630
trunks, 283
tst, 354
Tungsten Replicator, 163
Twitter, 454–457
type, 65
TYPE_BIGNUMBER, 606
TYPE_BINARY, 606
TYPE_BOOLEAN, 606
TYPE_DATE, 606
TYPE_INTEGER, 606
TYPE_NUMBER, 606
TYPE_STRING, 606

U

UA. *See* User Acceptance test
Ubuntu, 439
  AMI, 442
UI. *See* user interface
ui/laf.properties, 591
uname, 354
unbalanced hierarchy, 120
unconditional hops, 88
unconditional job hop, 31
UniCode, 15, 507
Uniform Resource Locators (URLs), 516
  Web services lookup step, 545
Unique rows step, 193–194
unit tests, 307
UNIX, 12, 507
  chmod, 322
  cron, 326–327
  crontab, 326–327
  Kitchen, 57
  Pan, 57
  running programs, 62
Unknown, 17
unstructured data, 501–508
Unzip, AMI, 440
UPDATE, 157, 230
  surrogate primary keys, 470
Update fields, Insert / Update step, 231–232
update mode, 232
Update step
  CRC, 485
  Dimension lookup / update step, 238
upgrade
  repositories, 351–352
  testing, 312
url, jdbc.properties, 65
URLs. *See* Uniform Resource Locators
Use batch updates for inserts, Table output step, 389
Use Kettle Repository, 330
Use tokens, Get data from XML step, 533, 535
user, 323
  jdbc.properties, 65
User Acceptance test (UA), 307
User Console, 333
User Defined Java Class step, 620–624
  Change number of copies to start, 404
  DyanicJob, 586
  get(), 620
  init(), 459
  JavaScript, 395
  metadata, 620
  rows, 404
  snippets, 620
  variables, 43
User Defined Java Expressions step, 70–71, 202–205
  data cleansing, 202–203
user interface, 24. *See also* graphical user interface
  elements, 609
  StepMetaInterface, 600
user maintained dimensions, 120
User Name and Password, Database Connection, 38
user-defined expressions and classes, Java, 520
user.dir, 642
user.home, 642
user.name, 642
UTF-8, 129
  Add XML step, 539
  RSS Ouput step, 565

V

Vaillencourt, Luc, 591
valid, 160
Validate msg field, XSD Validator step, 529
Validate XML?, Get data from XML step, 533
validation. *See* data validation
Validator step, 179–180
valid_from, 265
valid_to, 265
value(s)
  JSON, 522
  metadata, 605–606
  static, 396
Value distribution, DataCleaner, 149

Value mapper step, 94, 99, 170
Value when XML is invalid, XSD Validator step, 529
Value when XML is valid, XSD Validator step, 529
`ValueMetaInterface`, 605–606
van der Lek, Harm, 303
van Dongen, Jos, 228, 327
`VARCHAR`, 29
variables, 43
  Apache VFS, 641–642
  built-in, 637–642
  hierarchy, 120
  internal, 428–429
  Java API, 579–580
  JavaScript, 396
  jobs, 89
  JRE, 642
  `kettle.properties`, 66
  logging, 367
  Spoon, 44
  `StepInterface`, 618–619
  transformations, 89, 579–580
  using, 44–45
`VariableSpace`, 618–619
VCS. *See* Version Control System
`version`, 324
Version Control System (VCS), 341–344
  ETL, 124
  XML, 352
Version field, Dimension lookup / update step, 235
Version Migration System, 352–355
  ETL, 124
  parameters, 353–355
  repositories, 352–353
  XML, 352
Versioning, Enterprise Edition, 636
VFS. *See* Virtual File System
Virtual File System (VFS), 41, 42
  Apache, 42, 349, 517, 619
    variables, 641–642
virtual machines (VM), 438
visual programming languages (VPLs), 45–51
  steps, 47–49
  transformations, 46
Visualize, Spoon, 302–303
VM. *See* virtual machines
VPLs. *See* visual programming languages

W

Warehouse Builder, 6, 9
warnings, 405
waterfall model, 12
Wavemaker, 120
WDSL, SOAP, 517
web
  browsers, slave servers, 413
  extraction, 137–138
  pages
    HTML, 520
    web services, 515–517
  text files extraction, 137
web services, 515–568
  Apache VFS, 517
  API, 516
  data formats, 517–523
  Freebase, 550
  HTML, 520
  JSON, 520–523
  RSS, 517
  SOAP, 517
  web pages, 515–517
  XML, 518–520
Web Services Description Language (WSDL), 544
Web services lookup step, 517
  SOAP, 544–546
  streams, 517
Web services tab, Web services lookup step, 545
`wget`, 60
`WHERE`, 230
  satellites, 484
  SQL, 553
white box testing, 306
whitespace, 507
widgets, 607–608
wiki, 631
Wikipedia, 549–550
Windows, 61–62
Wintner, Robert, 140
WMS. *See* Workflow Management Systems
Workflow Management Systems (WMS), 344
Workflow Monitor, 124
wrappers, LucidDB, 10
write back, 271
WSDL. *See* Web Services Description Language

X

XBase, 134
XChat, 633
`xchataqua.soureforge.net`, 633
XML. *See* eXtensible Markup Language
XML Join step, 519, 541–544
  streams, 541
XML output step, 518
  CDC, 164
XML Schema, 518, 528
  data validation, 530
  XSD Validator step, 519
XML Schema Definition, XSD Validator step, 529–530
XML source, XSD Validator step, 529
XML source from field, Get data from XML step, 532
XML source is a filename?, Get data from XML step, 532
XML source is defined in field, Get data from XML step, 532
XML source is defined in field?, Get data from XML step, 548
XML/A
  JavaScript, 281
  MSAS, 279–280
  OLAP, 277–282
  Rename fields step, 280
`xml=Y`, 414
`-Xmx`, memory, 253
XP. *See* Extreme Programming
XPath, 518, 532
  Get data from XML step, 535
XSD Filename, XSD Validator step, 529–530
XSD Source, XSD Validator step, 529–530
XSD Validator step, 519, 528–530
  data validation, 530
  error handling, 530
  job entries, 519
  XML, 133
`xsi:schemaLocation`, 529
XSL. *See* eXtensible Stylesheet Language
XSL Transformation job entry, 519
XSL Transformation step, 518–519
XSL Transformations (XSLT), 133
XSLT. *See* XSL Transformations
`xstream.codehaus.org`, 603
XUL, 628

Y

Yourdon, Ed, 12
YouTube, 315

Z

`.zip`, 60, 517