Oracle® Endeca Information Discovery

Integrator Guide

Version 2.3.0 • April 2012

ORACLE

Copyright and disclaimer

Copyright © 2003, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. UNIX is a registered trademark of The Open Group.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Rosette® Linguistics Platform Copyright © 2000-2011 Basis Technology Corp. All rights reserved.

Teragram Language Identification Software Copyright © 1997-2005 Teragram Corporation. All rights reserved.

Table of Contents

I. CloverETL Overview 1
1. Family of CloverETL Products 2
CloverETL Designer 2
CloverETL Engine 2
CloverETL Server 2
Getting Additional Information 3
2. Integrating CloverETL Designer with CloverETL Server 4
Creating CloverETL Server project (Basic Principles) 4
Opening CloverETL Server Projects 5
Connecting via HTTP 6
Connecting via HTTPS 6
Designer has its Own Certificate 6
Designer does not have its Own Certificate 7
Connecting via proxy server 8
II. Installation Instructions 10
3. System Requirements for CloverETL Designer 11
Related Links 11
4. Downloading CloverETL 13
CloverETL Desktop Edition 13
CloverETL Desktop Trial Edition 13
CloverETL Community Edition 14
5. Starting CloverETL Designer 15
6. Installing Designer as Eclipse plugin 17
III. Getting Started 18
7. Creating CloverETL Projects 19
CloverETL Project 19
CloverETL Server Project 20
CloverETL Examples Project 22
8. Structure of CloverETL Projects 23
Standard Structure of All CloverETL Projects 24
Workspace.prm File 25
Opening the CloverETL Perspective 26
9. Appearance of CloverETL Perspective 28
CloverETL Designer Panes 28
Graph Editor with Palette of Components 29
Navigator Pane 33
Outline Pane 33
Tabs Pane 35
10. Creating CloverETL Graphs 38
Creating Empty Graphs 38
Creating a Simple Graph in a Few Simple Steps 41
11. Running CloverETL Graphs 51
Successful Graph Execution 52
Using the Run Configurations Dialog 54
IV. Working with CloverETL Designer 55
12. Using Cheat Sheets 56
13. Common Dialogs 59
URL File Dialog 59
Edit Value Dialog 60
Open Type Dialog 61
14. Import 62
Import CloverETL Projects 63
Import from CloverETL Server Sandbox 64
Import Graphs 65
Import Metadata 66

Metadata from XSD 66
Metadata from DDL 67
15. Export 68
Export Graphs 68
Export Graphs to HTML 69
Export Metadata to XSD 70
Export to CloverETL Server Sandbox 71
Export Image 72
16. Advanced Topics 73
Program and VM Arguments 73
Example of Setting Up Memory Size 75
Changing Default CloverETL Settings 76
Enlarging the Font of Displayed Numbers 78
Setting and Configuring Java 79
Setting Java Runtime Environment 79
Installing Java Development Kit 81
V. Graph Elements, Structures and Tools 83
17. Components 84
18. Edges 86
What Are the Edges? 86
Connecting Components by the Edges 86
Types of Edges 87
Assigning Metadata to the Edges 88
Propagating Metadata through the Edges 89
Colors of the Edges 89
Debugging the Edges 90
Enabling Debug 90
Selecting Debug Data 91
Viewing Debug Data 93
Turning Off Debug 95
19. Metadata 96
Data Types and Record Types 97
Data Types in Metadata 97
Record Types 98
Data Formats 99
Data and Time Format 99
Numeric Format 106
Boolean Format 110
String Format 111
Locale and Locale Sensitivity 112
Locale 112
Locale Sensitivity 116
Autofilling Functions 117
Internal Metadata 119
Creating Internal Metadata 119
Externalizing Internal Metadata 120
Exporting Internal Metadata 121
External (Shared) Metadata 122
Creating External (Shared) Metadata 122
Linking External (Shared) Metadata 122
Internalizing External (Shared) Metadata 123
Creating Metadata 124
Extracting Metadata from a Flat File 124
Extracting Metadata from an XLS(X) File 129
Extracting Metadata from a Database 130
Extracting Metadata from a DBase File 134
Creating Metadata by User 134
Extracting Metadata from Lotus Notes 134

Dynamic Metadata 136
Reading Metadata from Special Sources 137
Creating Database Table from Metadata and Database Connection 138
Metadata Editor 140
Basics of Metadata Editor 140
Record Pane 142
Details Pane 143
Changing and Defining Delimiters 145
Changing Record Delimiter 147
Changing Default Delimiter 148
Defining Non-Default Delimiter for a Field 148
Editing Metadata in the Source Code 149
20. Database Connections 150
Internal Database Connections 150
Creating Internal Database Connections 150
Externalizing Internal Database Connections 151
Exporting Internal Database Connections 152
External (Shared) Database Connections 153
Creating External (Shared) Database Connections 153
Linking External (Shared) Database Connections 153
Internalizing External (Shared) Database Connections 153
Database Connection Wizard 154
Encrypting the Access Password 157
Browsing Database and Extracting Metadata from Database Tables 158
21. JMS Connections 159
Internal JMS Connections 159
Creating Internal JMS Connections 159
Externalizing Internal JMS Connections 159
Exporting Internal JMS Connections 160
External (Shared) JMS Connections 161
Creating External (Shared) JMS Connections 161
Linking External (Shared) JMS Connection 161
Internalizing External (Shared) JMS Connections 161
Edit JMS Connection Wizard 162
Encrypting the Authentication Password 163
22. QuickBase Connections 164
23. Lotus Connections 165
24. Lookup Tables 166
Internal Lookup Tables 167
Creating Internal Lookup Tables 167
Externalizing Internal Lookup Tables 167
Exporting Internal Lookup Tables 169
External (Shared) Lookup Tables 170
Creating External (Shared) Lookup Tables 170
Linking External (Shared) Lookup Tables 170
Internalizing External (Shared) Lookup Tables 171
Types of Lookup Tables 172
Simple Lookup Table 172
Database Lookup Table 175
Range Lookup Table 176
Persistent Lookup Table 178
Aspell Lookup Table 179
25. Sequences 181
Internal Sequences 182
Creating Internal Sequences 182
Externalizing Internal Sequences 182
Exporting Internal Sequences 183
External (Shared) Sequences 184

Creating External (Shared) Sequences 184
Linking External (Shared) Sequences 184
Internalizing External (Shared) Sequences 184
Editing a Sequence 185
26. Parameters 187
Internal Parameters 187
Creating Internal Parameters 187
Externalizing Internal Parameters 188
Exporting Internal Parameters 189
External (Shared) Parameters 190
Creating External (Shared) Parameters 190
Linking External (Shared) Parameters 190
Internalizing External (Shared) Parameters 190
Parameters Wizard 192
Parameters with CTL Expressions 193
Environment Variables 193
Canonizing File Paths 193
Using Parameters 195
27. Internal/External Graph Elements 196
Internal Graph Elements 196
External (Shared) Graph Elements 196
Working with Graph Elements 196
Advantages of External (Shared) Graph Elements 196
Advantages of Internal Graph Elements 196
Changes of the Form of Graph Elements 196
28. Dictionary 198
Creating a Dictionary 198
Using the Dictionary in a Graph 200
29. Notes in the Graphs 202
30. Search Functionality 206
31. Transformations 208
32. Fact table loader 209
Launching Fact Table Loader Wizard 209
Wizard with project parameters file enabled 209
Wizard with the project parameter file disabled 211
Working with Fact Table Loader Wizard 211
Created graph 217
VI. Components Overview 219
33. Introduction to Components 220
34. Palette of Components 221
35. Common Properties of All Components 223
Edit Component Dialog 224
Component Name 227
Phases 228
Enable/Disable Component 229
PassThrough Mode 230
36. Common Properties of Most Components 231
Group Key 231
Sort Key 232
Defining Transformations 234
Return Values of Transformations 238
Error Actions and Error Log (deprecated since 3.0) 240
Transform Editor 241
Common Java Interfaces 249
37. Common Properties of Readers 250
Supported File URL Formats for Readers 251
Viewing Data on Readers 255
Input Port Reading 257

Incremental Reading 258
Selecting Input Records 258
Data Policy 260
XML Features 260
CTL Templates for Readers 261
Java Interfaces for Readers 261
38. Common Properties of Writers 262
Supported File URL Formats for Writers 263
Viewing Data on Writers 266
Output Port Writing 268
How and Where Data Should Be Written 269
Selecting Output Records 270
Partitioning Output into Different Output Files 270
Java Interfaces for Writers 272
39. Common Properties of Transformers 273
CTL Templates for Transformers 274
Java Interfaces for Transformers 275
40. Common Properties of Joiners 276
Join Types 277
Slave Duplicates 277
CTL Templates for Joiners 278
Java Interfaces for Joiners 281
41. Common Properties of Cluster Components 283
CTL Templates for Cluster Components 283
Java Interfaces for Cluster Components 283
42. Common Properties of Others 284
43. Custom Components 285
VII. Component Reference 286
44. Readers 287
CloverDataReader 289
ComplexDataReader 291
DataGenerator 298
DBFDataReader 305
DBInputTable 307
EmailReader 311
JMSReader 314
LDAPReader 317
LotusReader 320
MultiLevelReader 322
ParallelReader 326
QuickBaseRecordReader 329
QuickBaseQueryReader 331
UniversalDataReader 333
XLSDataReader 338
XMLExtract 342
XMLXPathReader 354
45. Writers 360
CloverDataWriter 362
DB2DataWriter 364
DBOutputTable 369
EmailSender 377
InfobrightDataWriter 381
InformixDataWriter 383
JMSWriter 386
LDAPWriter 389
LotusWriter 391
MSSQLDataWriter 393
MySQLDataWriter 396

OracleDataWriter 399
PostgreSQLDataWriter 403
QuickBaseImportCSV 406
QuickBaseRecordWriter 408
StructuredDataWriter 410
Trash 414
UniversalDataWriter 416
XLSDataWriter 419
XMLWriter 422
46. Transformers 439
Address Doctor 5 441
Aggregate 445
Concatenate 448
DataIntersection 449
DataSampler 452
Dedup 454
Denormalizer 456
EmailFilter 465
ExtFilter 470
ExtSort 472
FastSort 474
Merge 479
MetaPivot 481
Normalizer 484
Partition 491
Pivot 498
Reformat 502
Rollup 505
SimpleCopy 517
SimpleGather 518
SortWithinGroups 519
XSLTransformer 521
47. Joiners 523
ApproximativeJoin 524
DBJoin 532
ExtHashJoin 535
ExtMergeJoin 541
LookupJoin 546
RelationalJoin 549
48. Cluster Components 553
ClusterGather 554
ClusterPartitioner 556
49. Others 559
CheckForeignKey 560
DBExecute 564
HTTPConnector 568
JavaExecute 571
LookupTableReaderWriter 573
RunGraph 575
SequenceChecker 579
SpeedLimiter 581
SystemExecute 583
WebServiceClient 586
VIII. CTL - CloverETL Transformation Language 588
50. Overview 589
51. CTL1 vs. CTL2 Comparison 591
Typed Language 591
Arbitrary Order of Code Parts 591

Compiled Mode 591
Access to Graph Elements (Lookups, Sequences, ...) 591
Metadata 591
52. Migrating CTL1 to CTL2 595
53. CTL1 605
Language Reference 606
Program Structure 607
Comments 607
Import 607
Data Types in CTL 608
Literals 610
Variables 612
Operators 613
Simple Statement and Block of Statements 618
Control Statements 618
Error Handling 622
Functions 623
Eval 624
Conditional Fail Expression 625
Accessing Data Records and Fields 626
Mapping 629
Parameters 635
Functions Reference 636
Conversion Functions 637
Date Functions 642
Mathematical Functions 645
String Functions 649
Container Functions 657
Miscellaneous Functions 659
Dictionary Functions 661
Lookup Table Functions 662
Sequence Functions 664
Custom CTL Functions 665
54. CTL2 666
Language Reference 667
Program Structure 668
Comments 668
Import 668
Data Types in CTL2 669
Literals 672
Variables 674
Dictionary in CTL2 675
Operators 676
Simple Statement and Block of Statements 681
Control Statements 681
Error Handling 685
Functions 686
Conditional Fail Expression 687
Accessing Data Records and Fields 688
Mapping 690
Parameters 694
Functions Reference 695
Conversion Functions 697
Date Functions 705
Mathematical Functions 707
String Functions 711
Container Functions 720
Miscellaneous Functions 722

Lookup Table Functions 724
Sequence Functions 727
Custom CTL Functions 728
CTL2 Appendix - List of National-specific Characters 729
List of Figures 731
List of Tables 737
List of Examples 738

Part I. CloverETL Overview

Chapter 1. Family of CloverETL Products

This chapter is an overview of the following three products of our CloverETL software: **CloverETL Designer**, **CloverETL Engine** and **CloverETL Server**.

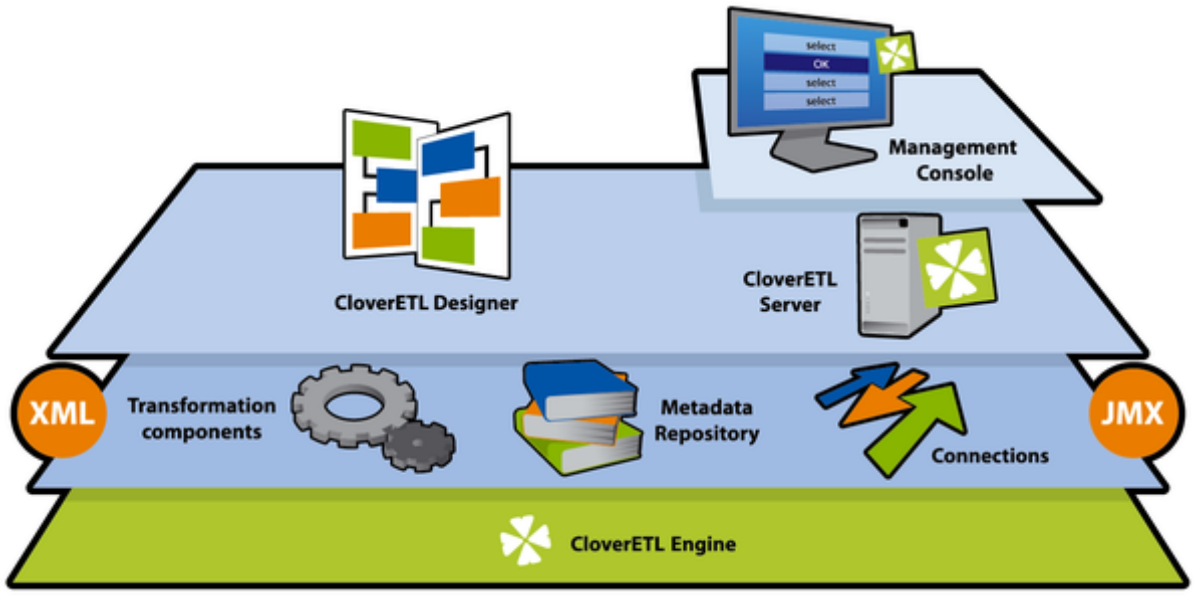


Figure 1.1. Family of CloverETL Products

CloverETL Designer

CloverETL Designer is a member of the family of **CloverETL** software products developed by Javlin. It is a powerful Java-based standalone application for data extraction, transformation and loading.

CloverETL Designer builds upon extensible **Eclipse** platform. See www.eclipse.org.

Working with **CloverETL Designer** is much simpler than writing code for data parsing. Its graphical user interface makes creating and running graphs easier and comfortable.

CloverETL Designer also allows you to work easily with **CloverETL Server**. These two products are fully integrated. You can use **CloverETL Designer** to connect to and communicate with **CloverETL Server**, create projects, graphs, and all other resources on **CloverETL Server** in the same way as if you were working with **CloverETL Designer** only locally.

See Chapter 2, Integrating CloverETL Designer with CloverETL Server (p. 4) for more information.

CloverETL Engine

CloverETL Engine is a base member of the family of **CloverETL** software products developed by Javlin. **CloverETL Engine** is a run-time layer that executes transformation graphs created in **CloverETL Designer**.

Transformation graphs are created in **CloverETL Designer** from graph elements and executed by **CloverETL Engine**.

CloverETL Engine is a Java library that can be embedded into other Java applications.

CloverETL Server

CloverETL Server is the last and newest member of **CloverETL** software products developed by Javlin. **CloverETL Server** is also based on Java.

CloverETL Designer can be used to work with **CloverETL Server**. These two products are fully integrated. You can use **CloverETL Designer** to connect to and communicate with **CloverETL Server**, create projects, graphs and all other resources on **CloverETL Server** in the same way as if you were working with the standard **CloverETL Designer** only locally.

See Chapter 2, Integrating CloverETL Designer with CloverETL Server (p. 4) for more information.

CloverETL Server allows to achieve:

- Centralized ETL job management
- Integration into enterprise workflows
- Multi-user environment
- Parallel execution of graphs
- Tracking of executions of graphs
- Scheduling tasks
- Clustering and distributed execution of graphs
- Launch services
- Load balancing and failover

Getting Additional Information

In addition to this **User's Guide**, you can find additional information on the following sites:

- **Quick Start Guide** explaing briefly the basics of **CloverETL Designer** usage.

  www.cloveretl.com/documentation/quickstart

- **FAQ** concerning various areas of **CloverETL** products.

  www.cloveretl.com/faq

- **Forum** about detailes of **CloverETL** features.

  http://forum.cloveretl.com

- **Blog** describing interesting solutions based on **CloverETL** products.

  http://blog.cloveretl.com

- **Wiki** page containing the information about **CloverETL** tools.

  http://wiki.cloveretl.com

Support

In addition to the sites mentioned above, Javlin offers a full range of support options. This **Technical Support** is designed to save you time and ensure you achieve the highest levels of performance, reliability, and uptime. **CloverCare** is intended primarily for the US and Europe.

www.cloveretl.com/services/clovercare-support

Chapter 2. Integrating CloverETL Designer with CloverETL Server

With **CloverETL Designer** and **CloverETL Server** now fully integrated, you can access **Server** sandboxes directly from the **Designer** without having to copy them back and forth manually.

Designer takes care of all the data transfers for you - you can directly edit graphs, run them on the server, edit data files, metadata, etc. You can even view live tracking of a graph execution as it runs on the **Server**.

Important

Please note that this feature works only on **Eclipse 3.5+** and Java 1.6.4+.

Rember also that version 3.0 of **CloverETL Designer** can only work with version 3.0 of **CloverETL Server**, and vice versa.

You can connect to your **CloverETL Server** by creating a **CloverETL Server Project** in **CloverETL Designer**. See CloverETL Server Project (p. 20) for detailed information.

To learn how you can interchange graphs, metadata, etc. between a **CloverETL Server** sandbox and a standard **CloverETL** project, see the following links:

- Import from CloverETL Server Sandbox (p. 64)
- Export to CloverETL Server Sandbox (p. 71)

The **User's Guide** of **CloverETL Server** can be found here: http://server-demo-ec2.cloveretl.com/clover/docs/index.html

Creating CloverETL Server project (Basic Principles)

1. As the first step, a sandbox must exist on **CloverETL Server**. To each **CloverETL Server** sandbox, only one **CloverETL Server** project can be created within the same workspace. If you want to create more than one **CloverETL Server** projects to a single **CloverETL Server** sandbox, each of these projects must be in different workspace.

2. In one workspace, you can have more **CloverETL Server projects** created using your **Designer**.

   Each of these **CloverETL Server projects** can even be linked to different **CloverETL Server**.

   However, one condition must be fulfilled: `hostname` of each **Server** must be unique. Connecting to different port numbers or using different protocol with the same `hostname` is not sufficient!

3. **CloverETL Designer** uses the HTTP/HTTPS protocols to connect **CloverETL Server**. These protocols work well with complex network setups and firewalls. Remember that each connection to any **CloverETL Server** is saved in your workspace. For this reason, you can use only one protocol in one workspace. You have your login name, password and some specified user rights and/or keys.

4. Remember that if multiple users are accessing the same sandbox (via **Designer**), they must cooperate to not overwrite their changes made to the same resources (e.g. graphs). If anyone changes the graph or any other resource on **CloverETL Server**, the other users may overwrite such resources on **Server**. However, a warning is displayed and each user must decide whether he or she really wants to overwrite such resource on **CloverETL Server**. The remote resources are not locked and user must decide what should be done in case of such conflict.

5. When you restart **CloverETL Designer**, all **CloverETL Server projects** are displayed, but all of them are closed. In order to open them, you have two possibilities:

- Double-click the project.
- Right-click the project and select **Open project** from the context menu.

See Opening CloverETL Server Projects (p. 5) for more information.

Opening CloverETL Server Projects

When you start your CloverETL Designer, you will see something like this:

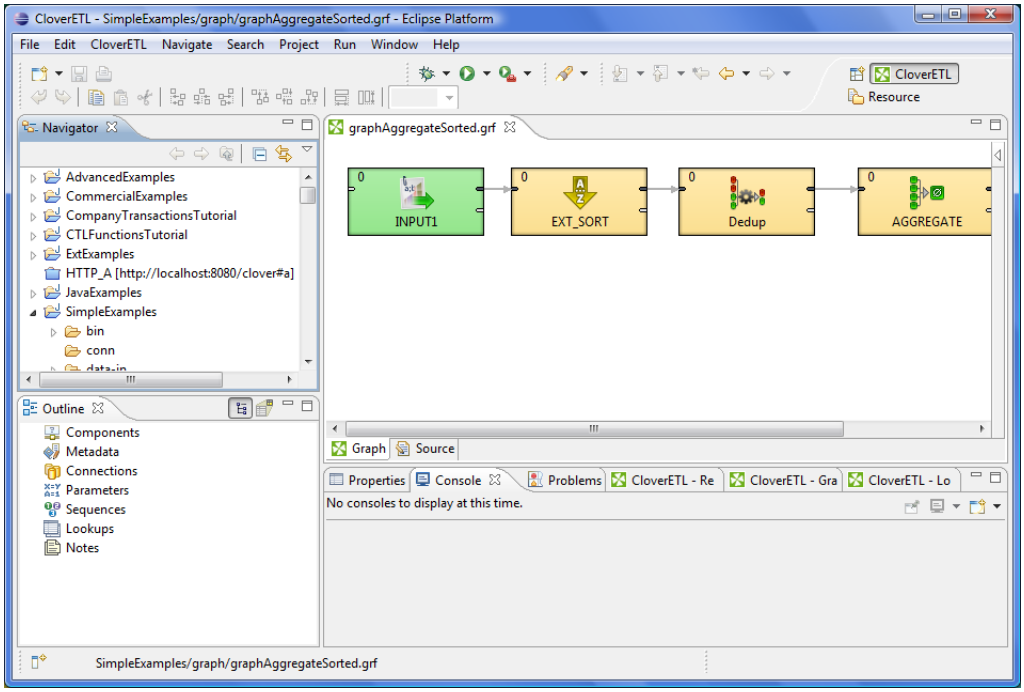


Figure 2.1. CloverETL Server Project Displayed after Opening CloverETL Designer

Note that the URL of the **Server** and the ID of the sandbox separated by hash are displayed beside the **CloverETL Server project** name. The **CloverETL Server projects** will be closed. To open these projects, do what is desribed above: Either double-click them or select **Open project** from the context menu.

You may be prompted to insert your **User ID** and **Password**, choose also whether **Password** should be saved.

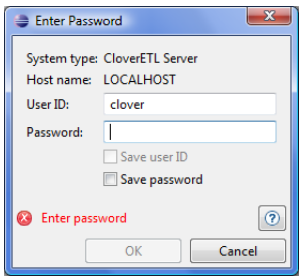


Figure 2.2. Prompt to Open CloverETL Server Project

After that, click **OK** and the **CloverETLServer project**.

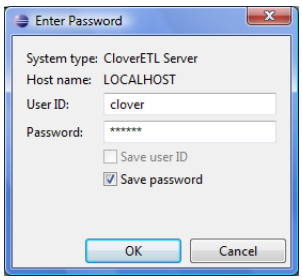


Figure 2.3. Opening CloverETL Server Project

Should you have some problem with your antivirus application, add the exceptions to your settings of the HTTP/HTTPS connections. For example, you can use `*clover*` as a mask to allow the connections to **CloverETL Servers** or **CloverETL** web pages.

Connecting via HTTP

To connect via `http`, for example, you can install **CloverETL Server** using **Tomcat**.

To do that, copy the `clover.war` and `clover_license.war` files of **CloverETL Server** to the `webapps` subdirectory of **Tomcat** and run **Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**. Once **Tomcat** starts, the two files are expanded inside its `webapps` subdirectory.

With the HTTP connection, you do not need configure **CloverETL Designer**. Simply start your **CloverETL Designer**. With this **Designer**, you can create your **CloverETL Server projects** using the following default connection to **Server**: `http://localhost:8080/clover` where both login name and password are `clover`.

Connecting via HTTPS

To connect via `https`, for example, you can install **CloverETL Server** using **Tomcat**.

As the first step, copy the `clover.war` and `clover_license.war` files of **CloverETL Server** to the `webapps` subdirectory of **Tomcat** and run **Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**. Once **Tomcat** starts, the two files are expanded inside its `webapps` subdirectory.

You need to configure both **Server** and **Designer** (in case of Designer with its own certificate) or Server alone (in case of Designer without a certificate).

Designer has its Own Certificate

In order to connect to **CloverETL Server** via `https` when Designer must have its own certificate, create client and server keystores/truststores.

To generate these keys, execute the following script (version for Unix) in the `bin` subdirectory of JDK or JRE where `keytool` is located:

```
# SERVER
# create server key-store with private-public keys
keytool -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
        -keypass semafor -storepass semafor -validity 900 \
        -dname "cn=serverDomainName, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool -exportcert -alias server -keystore serverKS.jks \
        -storepass semafor -file server.cer

# CLIENT
# create client key-store with private-public keys
```

```
keytool -genkeypair -alias client -keyalg RSA -keystore ./clientKS.jks \
        -keypass chodnik -storepass chodnik -validity 900 \
        -dname "cn=Key Owner, ou=ETL, o=Javlin, c=CR"
# exports public key to separated file
keytool -exportcert -alias client -keystore clientKS.jks \
        -storepass chodnik -file client.cer

# trust stores

# imports server cert to client trust-store
keytool -import -alias server -keystore clientTS.jks \
        -storepass chodnik -file server.cer

# imports client cert to server trust-store
keytool -import -alias client -keystore serverTS.jks \
        -storepass semafor -file client.cer
```

(In these commands, `localhost` is the default name of your **CloverETL Server**, if you want any other **Server** name, replace the `localhost` name in these commands by any other hostname.)

After that, copy the `serverKS.jks` and `serverTS.jks` files to the `conf` subdirectory of **Tomcat**.

Then, copy the following code to the `server.xml` file in this `conf` subdirectory:

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="off" />


          <Connector port="8443" maxHttpHeaderSize="7192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" disableUploadTimeout="true"
               acceptCount="100" scheme="https" secure="true"
               clientAuth="true" sslProtocol="TLS"
               SSLEnabled="true"
               protocol="org.apache.coyote.http11.Http11NioProtocol"
               keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
               keystorePass="semafor"
               truststoreFile="pathToTomcatDirectory/conf/serverTS.jks"
               truststorePass="semafor"
           />
```

Now you can run **CloverETL Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**.

Configuring CloverETL Designer

Now you need to copy the `clientKS.jks` and `clientTS.jks` files to any location.

After that, copy the following code to the end of the `eclipse.ini` file, which is stored in the `eclipse` directory:

```
-Djavax.net.ssl.keyStore=locationOfClientFiles/clientKS.jks
-Djavax.net.ssl.keyStorePassword=chodnik
-Djavax.net.ssl.trustStore=locationOfClientFiles/clientTS.jks
-Djavax.net.ssl.trustStorePassword=chodnik
```

Now, when you start your **CloverETL Designer**, you will be able to create your **CloverETL Server projects** using the following default connection to **Server**: `https://localhost:8443/clover` where both login name and password are `clover`.

Designer does not have its Own Certificate

In order to connect to **CloverETL Server** via `https` when Designer does not need to have its own certificate, you only need to create a server keystore.

To generate this key, execute the following script (version for Unix) in the `bin` subdirectory of JDK or JRE where `keytool` is located:

```
keytool -genkeypair -alias server -keyalg RSA -keystore ./serverKS.jks \
        -keypass semafor -storepass semafor -validity 900 \
        -dname "cn=serverDomainName, ou=ETL, o=Javlin, c=CR"
```

(In these commands, `localhost` is the default name of your **CloverETL Server**, if you want any other **Server** name, replace the `localhost` name in these commands by any other hostname.)

After that, copy the `serverKS.jks` file to the `conf` subdirectory of **Tomcat**.

Then, copy the following code to the `server.xml` file in this `conf` subdirectory:

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
          SSLEngine="off" />


          <Connector port="8443" maxHttpHeaderSize="7192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" disableUploadTimeout="true"
               acceptCount="100" scheme="https" secure="true"
               clientAuth="false" sslProtocol="SSL"
               SSLEnabled="true"
               protocol="org.apache.coyote.http11.Http11NioProtocol"
               keystoreFile="pathToTomcatDirectory/conf/serverKS.jks"
               keystorePass="semafor"
           />
```

Now you can run **CloverETL Server** by executing the `startup` script located in the `bin` subdirectory of **Tomcat**.

And, when you start your **CloverETL Designer**, you will be able to create your **CloverETL Server projects** using the following default connection to **Server**: `https://localhost:8443/clover` where both login name and password are `clover`.

You will be prompted to accept the Server certificate. After which, you are allowed to create a **CloverETL Server project**.

Connecting via proxy server

You can make use of your proxy server to connect to Clover Server, too.

Important

The proxy server has to support HTTP 1.1. Otherwise all connection attempts will fail.

To manage the connection, navigate to **Window →Preferences →General →Network Connections**

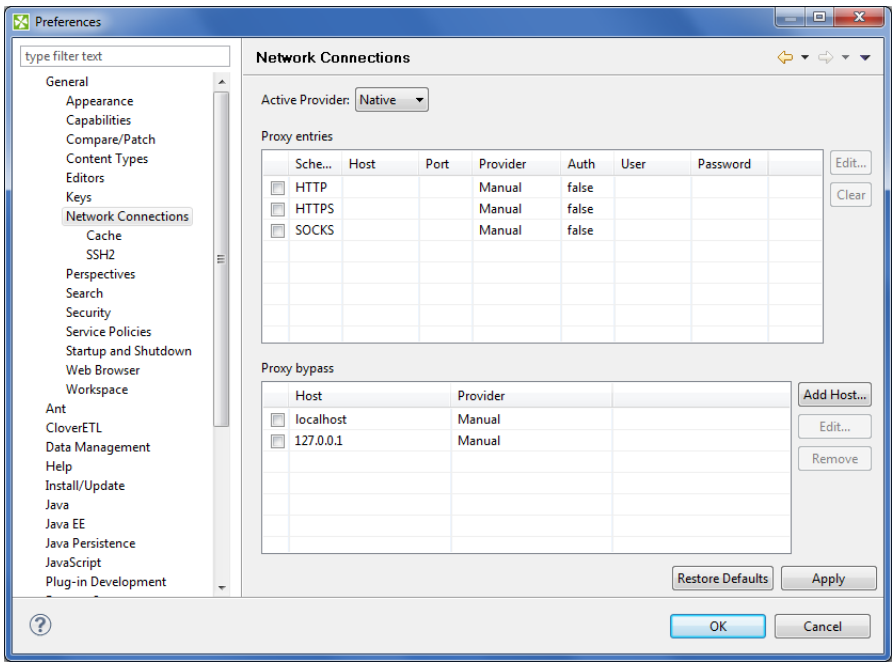


Figure 2.4. Network connections window

For more information on handling proxy settings, go to the Eclipse website.

Part II. Installation Instructions

Chapter 3. System Requirements for CloverETL Designer

CloverETL Designer is distributed in three different forms:

1. **Full installation**, which is recommended, includes all the necessary environments and applications, i.e., no prerequisites are required.
2. **Online plugin** fits if all prerequisites described below reside in your computer; **CloverETL Designer** will be installed online using the corresponding CloverETL update site.
3. **Offline plugin** is applicable under the same conditions as the online plugin but this time **CloverETL Designer** will be installed offline by executing of the previously downloaded archived file.

The following requirements must be fulfilled in order for CloverETL to run:

- supported OS are Microsoft Windows 32 bit, Microsoft Windows 64 bit, Linux 32 bit, Linux 64bit, and Mac OS X Cocoa
- at least 512MB of RAM

Full installation

- **Software requirements:**
  - None, the installer includes Eclipse Platform 3.6.2 for Java developers with RSE + GEF + Eclipse Web Tools Platform.

Plugin installation

- **Software requirements:**
  - Minimum: Eclipse Platform 3.5 + GEF. Java 6 Runtime Enviroment.

Important

If you are installing CloverETL as a plugin to Eclipse 3.5, do not install the ""CloverETL Text editor for XML properties" feature, as it requires Eclipse components that are not available in version 3.5.

  - Recommended: Eclipse Platform 3.6.2 with RSE + GEF + Eclipse Web Tools Platform. Java 6 Development Kit
  - Eclipse 3.7 is fully supported.

Important

For Mac OS users:

Please make sure your default Java system is set to 1.6 or newer. Go to **Finder** →**Applications** →**Utilities** →**Java** →**Java preferences** and reorder the available Java installations so that Java 6 is at the top of the list.

Related Links

- Eclipse Classic download page - choose the proper version for your OS

  www.eclipse.org/downloads

- JDK download page

  http://java.sun.com/javase/downloads/widget/jdk6.jsp

- Useful links for understanding the difference between Java Development Kit (JDK) and Java SE Runtime Environment (JRE)

  http://java.sun.com/javase/6/docs/

  http://java.sun.com/javase/6/webnotes/version-6.html

Chapter 4. Downloading CloverETL

There are possible two ways how to download **CloverETL Designer Edition**:

1. From your user or customer account after signing in at www.cloveretl.com/user

   This is the recommended way of installing it.

2. Using the direct http download link for your platform.

   - The full installation links are unique with respect to the operating system
   - The plugin installation links (both online and offline) are common for all the supported operating systems within the particular edition

CloverETL Desktop Edition

To log into your customer account use the login data (email address + password) you have received along with the license credentials. Later on, you will be oblidged to enter the licence number and the password to invoke the download. For more information about the installation from your customer profile see www.cloveretl.com/resources/installation-guide

Now, let's go through the list of direct download http links presented in tables below.

Get the Full Installation

OS	Download Site
Windows 32 bit	designer.cloveretl.com/update/cloveretl-designer-win32.exe
Windows 64 bit	designer.cloveretl.com/update/cloveretl-designer-win32-x68_64.exe
Linux 32 bit	designer.cloveretl.com/update/cloveretl-designer-linux-gtk.tar.gz
Linux 64 bit	designer.cloveretl.com/update/cloveretl-designer-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/update/cloveretl-designer-macosx-cocoa-x86_64.dmg.zip

Get the Eclipse Plugin

Plugin	Download Site
online	designer.cloveretl.com/update
offline	designer.cloveretl.com/update/cloveretl-designer.zip

For the plugin installation instructions go to Chapter 6, Installing Designer as Eclipse plugin (p. 17).

CloverETL Desktop Trial Edition

To get the trial version, create your user account at the company site www.cloveretl.com/user/registration. After receiving your login name with password and confirming the registration, your user account will be granted a time-limited access to the **CloverETL Desktop Trial Edition** download. Detailed information about the installation can be found at www.cloveretl.com/resources/installation-guide.

Get the Full Installation

OS	Download Site
Windows 32bit	designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32.exe
Windows 64 bit	designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32-x68_64.exe
Linux 32bit	designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk.tar.gz
Linux 64bit	designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/eval-update/cloveretl-designer-eval-macosx-cocoa-x86_64.dmg.zip

Get the Eclipse Plugin

Plugin	Download Site
online	designer.cloveretl.com/eval-update
offline	designer.cloveretl.com/eval-update/cloveretl-designer-eval.zip

See Chapter 6, Installing Designer as Eclipse plugin (p. 17) for more information about installing **CloverETL Designer** using Eclipse software update site mechanism.

CloverETL Community Edition

If you wish to get the CloverETL Community Edition, visit www.cloveretl.com/user/registration to create your user account at. The login name and password, sent by return, will authorize the download and the installation.

Get the Full Installation

OS	Download Site
Windows 32bit	designer.cloveretl.com/community-update/cloveretl-designer-community-win32.exe
Windows 64 bit	designer.cloveretl.com/community-update/cloveretl-designer-community-win32-x68_64.exe
Linux 32bit	designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk.tar.gz
Linux 64bit	designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk-x86_64.tar.gz
Mac OS Cocoa	designer.cloveretl.com/community-update/cloveretl-designer-community-macosx-cocoa-x86_64.dmg.zip

Get the Eclipse Plugin

Plugin	Download Site
online	designer.cloveretl.com/community-update
offline	designer.cloveretl.com/community-update/cloveretl-designer-community.zip

The plugin installation is described in Chapter 6, Installing Designer as Eclipse plugin (p. 17).

Chapter 5. Starting CloverETL Designer

When you start **CloverETL Designer**,you will see the following screen:

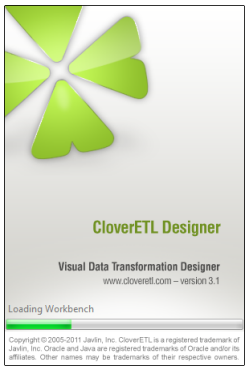


Figure 5.1. CloverETL Designer Splash Screen

The first thing you will be prompted to define after the **CloverETL Designer** launches, is the `workspace` folder. It is a place your projects will be stored at; usually a folder in the user's `home` directory (e.g., `C:\Users\your_name\workspace` or `/home/your_name/CloverETL/workspace`)

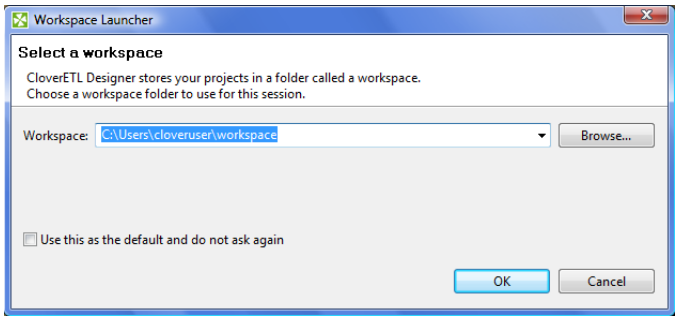


Figure 5.2. Workspace Selection Dialog

Note that the `workspace` can be located anywhere. Thus, make sure you have proper permissions to the location. If non-existing folder is specified, it will be created.

When the `workspace` is set, the welcome screen is displayed.

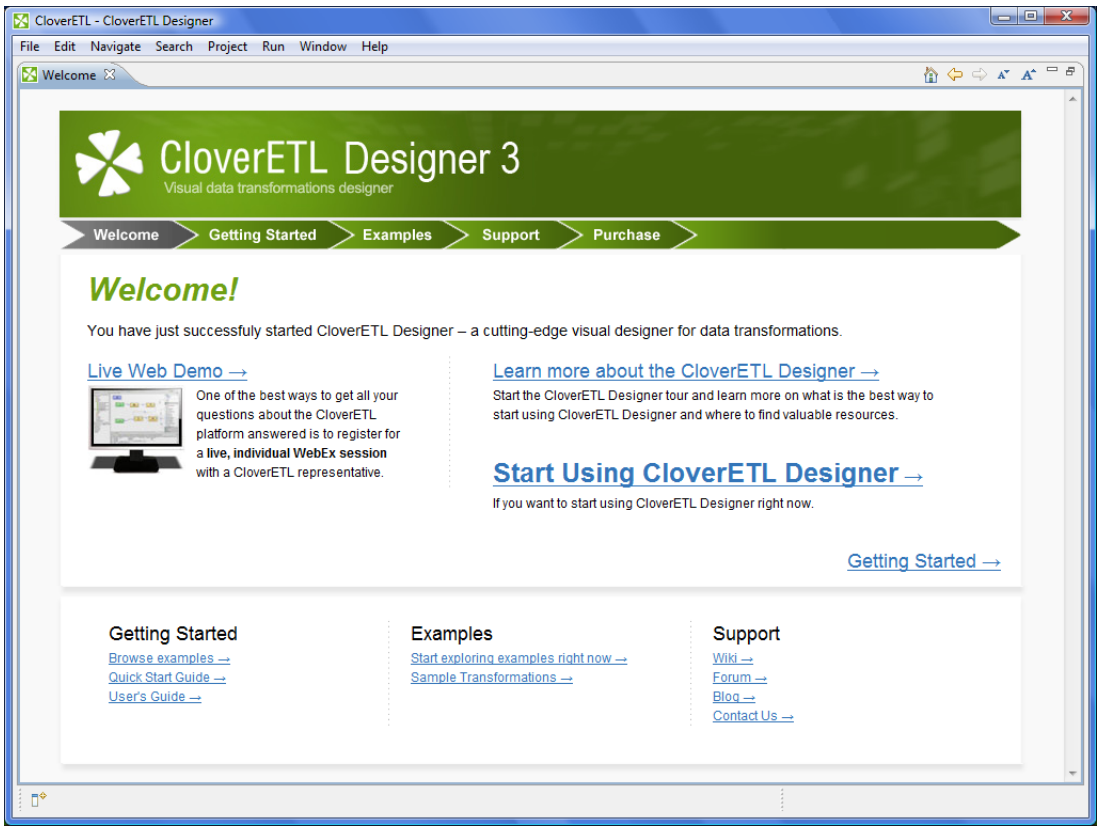


Figure 5.3. CloverETL Designer Introductory Screen

The first steps with **CloverETL Designer** are described in Chapter 7, Creating CloverETL Projects (p. 19).

Chapter 6. Installing Designer as Eclipse plugin

Eclipse allows you to install **CloverETL Designer** plugin directly into it. All required plugins will be installed during the installation of the **CloverETL Designer** plugin. You do not need to install them separately, you only need to download and install **CloverETL Designer**.

Installing the **CloverETL Designer** plugin into **Eclipse** release is considered installing new software. The **Help** →**Install New Software...** wizard displays the list of software sites and items available for installation.

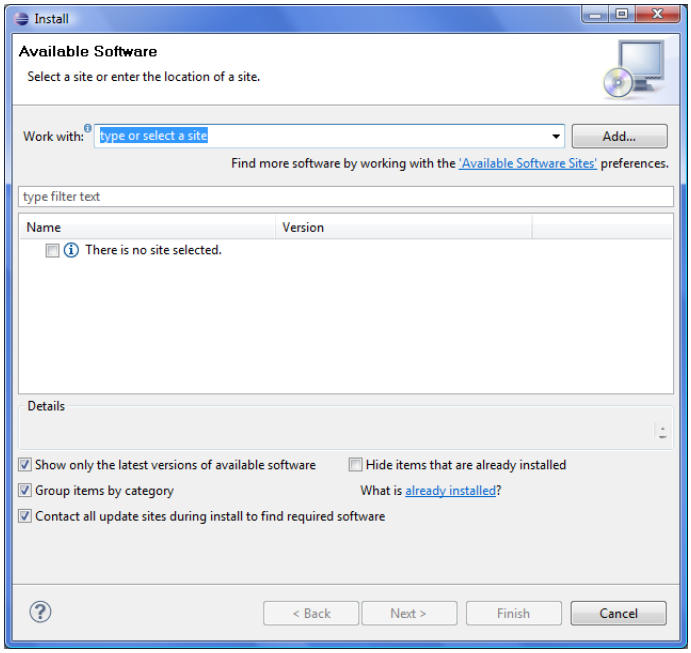


Figure 6.1. Available Software

The **Add...** button allows you to type the plugin location. The summary of **CloverETL Designer** online plugin update sites will then be provided in the table below.

Table 6.1. Sites with CloverETL

CloverETL Product	Update Site
CloverETL Desktop Edition	designer.cloveretl.com/update
CloverETL Desktop Trial Edition	designer.cloveretl.com/eval-update
CloverETL Community Edition	designer.cloveretl.com/community-update

To get the access to the **CloverETL** plugin installation, you will be prompted to enter your username and password. If you are installing **CloverETL Desktop Edition**, enter the license number for username. If you are installing any other version, use your user account login, i.e., your email address.

After checking the **CloverETL** item and accepting the terms of the license agreements in the next screen, click **Finish** to allow the downloading and installing to proceed. Later on, click **Yes** when asked to restart Eclipse SDK for the changes to take effect.

To verify that the installation had finished successfully, go to **Help** →**About Eclipse SDK**. The **CloverETL** logo should be displayed.

Part III. Getting Started

Chapter 7. Creating CloverETL Projects

This chapter describes how you can create **CloverETL** projects.

CloverETL Designer allows you to create three kinds of **CloverETL** projects:

- CloverETL Project (p. 19)

  It is a local **CloverETL** project. The whole project structure is on your local computer.

- CloverETL Server Project (p. 20)

  It is a **CloverETL** project corresponding to a **CloverETL Server** sandbox. The whole project structure is on **CloverETL Server**.

- CloverETL Examples Project (p. 22)

  In addition to the two mentioned kinds of projects, **CloverETL Designer** also allows you to create a set of prepared local **CloverETL** projects containing examples. These examples demonstrate the functionality of **CloverETL**.

We will explain how you need to create the mentioned projects from **CloverETL** perspective.

If you have installed **CloverETL Designer** as a full installation, you are already in this perspective.

If you have installed **CloverETL Designer** as a plugin into **Eclipse**, you need to change the perpsective. To do it, click the button at the top right corner and select from the menu **Others**, then choose **CloverETL**.

CloverETL Project

From the **CloverETL** perspective, select **File** →**New** →**CloverETL Project**.

Following wizard will open and you will be asked to give a name to your project:

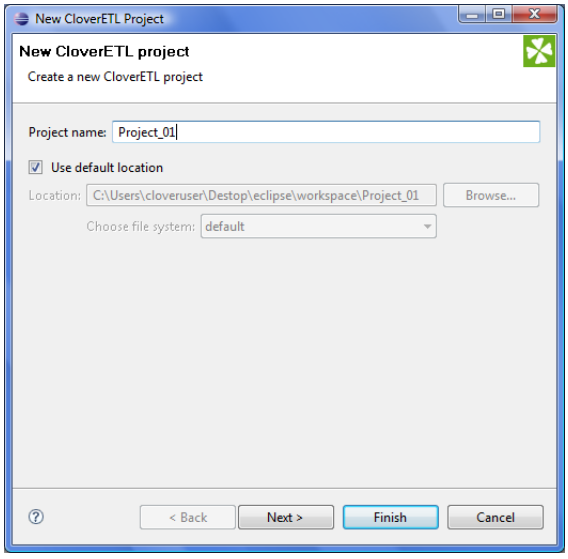


Figure 7.1. Giving a Name to a CloverETL Project

After clicking **Finish**, the selected local **CloverETL** project with the specified name will be created.

CloverETL Server Project

From the **CloverETL** perspective, select **File** →**New** →**CloverETL Server Project**.

Following wizard will open and you will be asked to do what is described below:

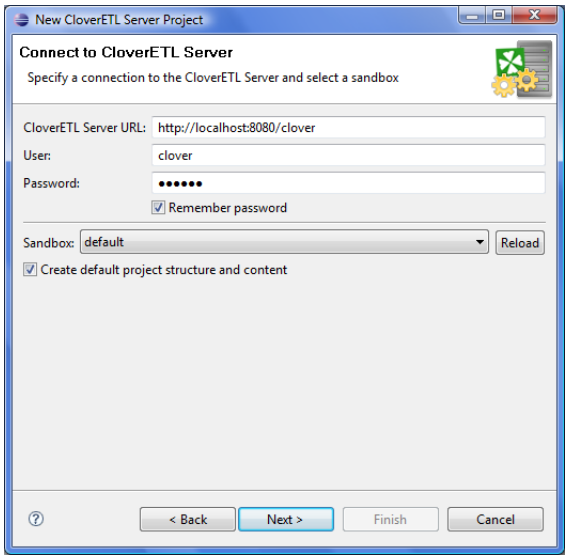


Figure 7.2. CloverETL Server Project Wizard

Fill the shown text fields: **CloverETL Server URL**, **User**, and **Password**.

Then click **Reload**, after which all **CloverETL** sandboxes will be loaded to the wizard.

Select one of these sanboxes, check the check box if you want that **Password** should be remembered in your PC. And decide whether your **CloverETL Server** project should have the default structure. Then click **Next**.

After that, enter the name of the project:

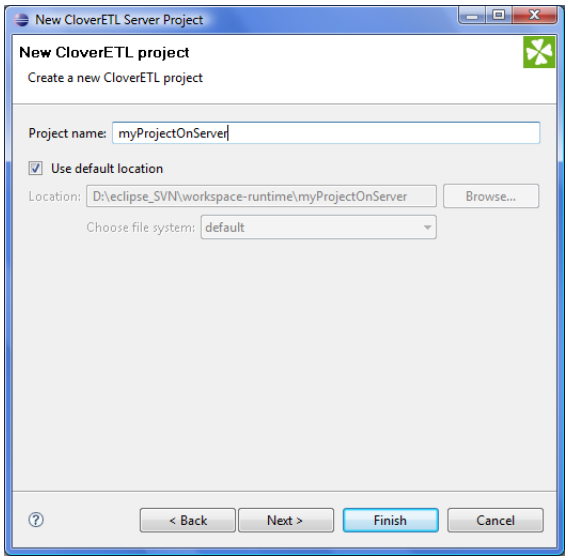


Figure 7.3. Giving a Name to the New CloverETL Server Project

Keep the other values (**Location**, and **File system**) unchanged.

After clicking **Finish**, **CloverETL Server** project will be created.

Important

In fact, any **CloverETL Server** project is only a link to an existing **CloverETL Server** sandbox.

However, it can be seen in the **Navigator** pane of **CloverETL Designer** in the same way as any other local projects, graphs can be created in this **CloverETL Designer** as if they are created locally, but they exist in the mentioned sandbox. They can also be run from this **Designer**.

If you want to create graphs within some **CloverETL Server project**, you must have the permission to write into the sandbox.

CloverETL Examples Project

If you want to create some of the prepared example projects, select **File** →**New** →**Others...**, expand the **CloverETL** category and choose **CloverETL Examples Project**.

You will be presented with the following wizard:

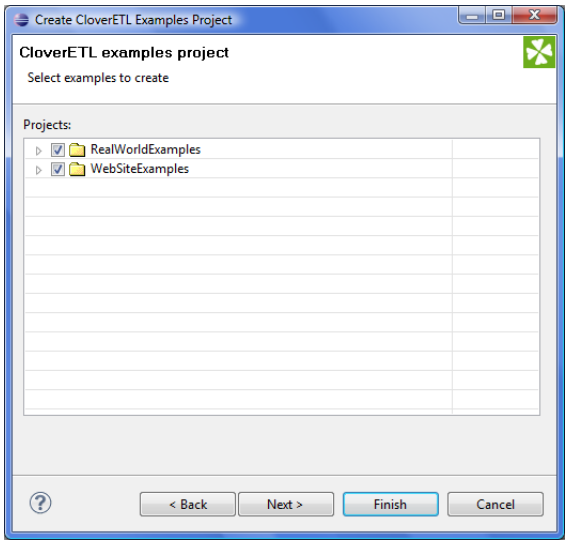


Figure 7.4. CloverETL Examples Project Wizard

You can select any of the **CloverETL** example projects by checking its checkbox.

After clicking **Finish**, the selected local **CloverETL Exaples** projects will be created.

Important

Remember that if you already have these project installed, you can click **Next** and rename them before installing. After that, you can click **Finish**.

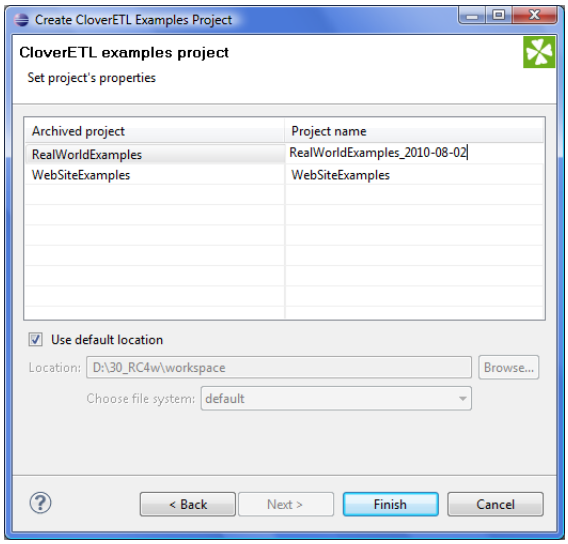


Figure 7.5. Renaming CloverETL Examples Projects

Chapter 8. Structure of CloverETL Projects

In this chapter we present only a brief overview of what happens when you are creating any **CloverETL** project.

This applies not only for local CloverETL Project (p. 19) and CloverETL Examples Project (p. 22), but also for CloverETL Server Project (p. 20).

1. Standard Structure of All CloverETL Projects (p. 24)

   Each of your **CloverETL Projects** has the standard project structure (unless you have changed it while creating the project).

2. Workspace.prm File (p. 25)

   Each of your local or remote (server) **CloverETL** projects contains the `workspace.prm` file (in the project folder) with basic information about the project.

3. Opening the CloverETL Perspective (p. 26)

   - If you installed **CloverETL Designer** as a full-installation and started the **Designer**, the **CloverETL** perspective opened by default.
   - If you installed **CloverETL Designer** as a plugin into **Eclipse** and started the **Designer**, the basic **Eclipse** perspective opened. You need to switch it to the **CloverETL** perspective.

Standard Structure of All CloverETL Projects

In the **CloverETL** perspective, there is a **Navigator** pane on the left side of the window. In this pane, you can expand the project folder. After that, you will be presented with the folder structure. There are subfolders for:

Table 8.1. Standard Folders and Parameters

Purpose	Standard folder	Standard parameter	Parameter usage [1)]
all connections	`conn`	`CONN_DIR`	`${CONN_DIR}`
input data	`data-in`	`DATAIN_DIR`	`${DATAIN_DIR}`
output data	`data-out`	`DATAOUT_DIR`	`${DATAOUT_DIR}`
temporary data	`data-tmp`	`DATATMP_DIR`	`${DATATMP_DIR}`
graphs	`graph`	`GRAPH_DIR`	`${GRAPH_DIR}`
lookup tables	`lookup`	`LOOKUP_DIR`	`${LOOKUP_DIR}`
metadata	`meta`	`META_DIR`	`${META_DIR}`
sequences	`seq`	`SEQ_DIR`	`${SEQ_DIR}`
transformation definitions (both source files and classes)	`trans`	`TRANS_DIR`	`${TRANS_DIR}`

Legend:

1): For more information about parameters, see Chapter 26, Parameters (p. 187), and about their usage, see Using Parameters (p. 195).

Important

Remember that using parameters in **CloverETL** ensures that such a graph, metadata or any other graph element can be used in any place without necessity of its renaming.

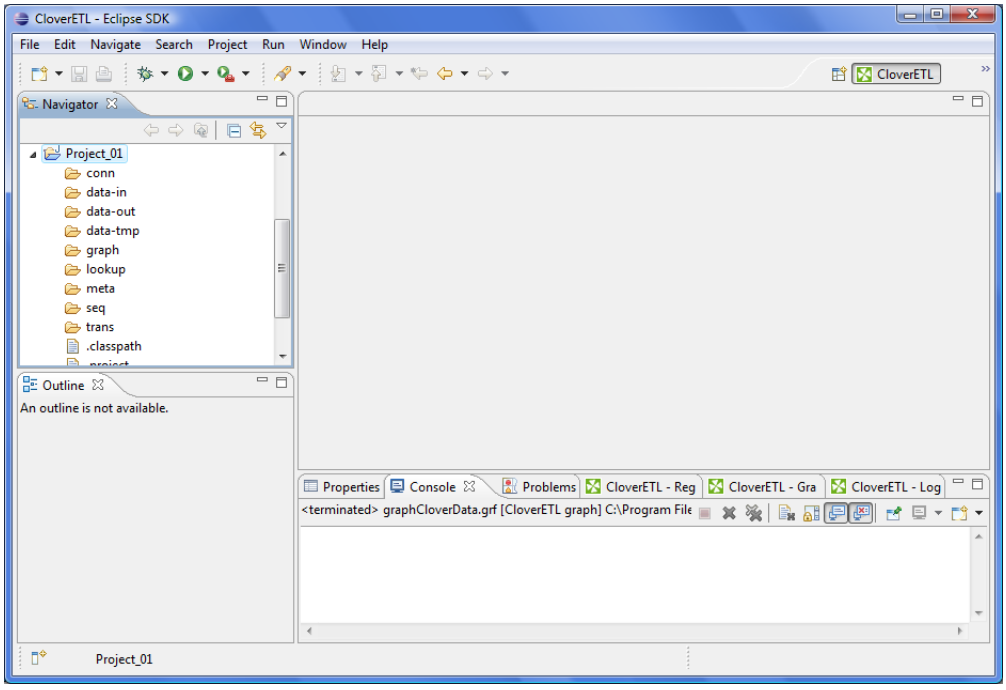


Figure 8.1. CloverETL Perspective with Highlighted Navigator Pane and the Project Folder Structure

Workspace.prm File

You can look at the `workspace.prm` file by clicking this item in the **Navigator** pane, by right-clicking and choosing **Open With →Text Editor** from the context menu.

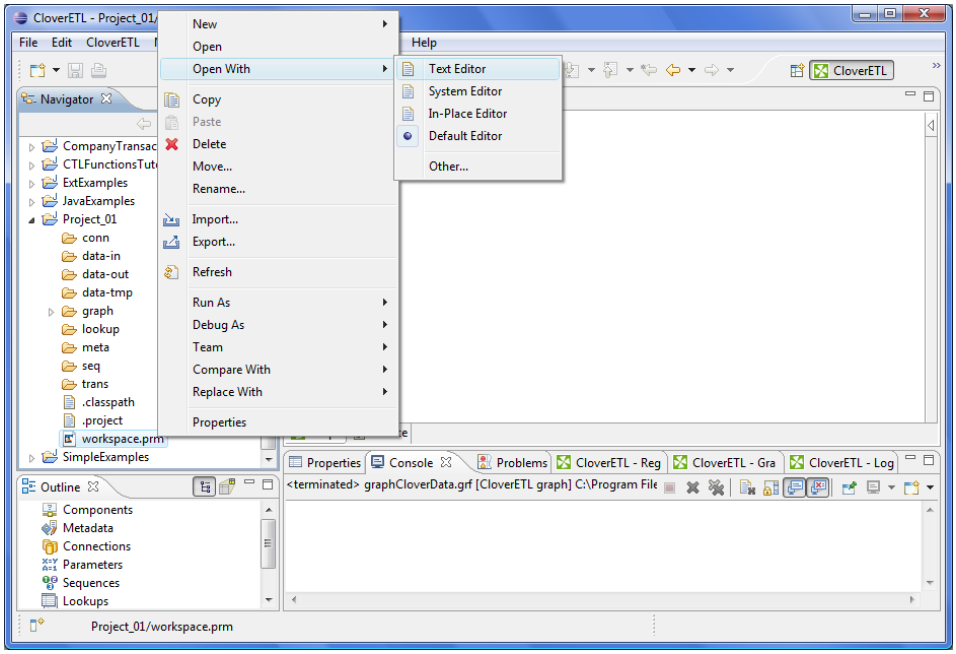


Figure 8.2. Opening the Workspace.prm File

You can see the parameters of your new project.

Note

The parameters of imported projects may differ from the default parameters of a new project.

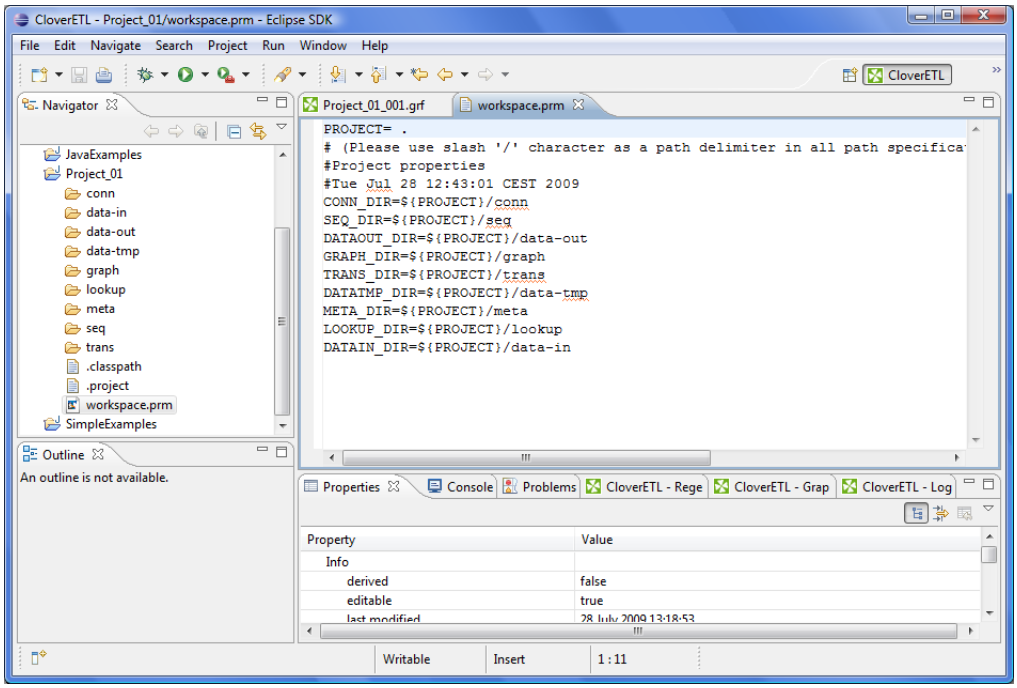


Figure 8.3. Workspace.prm File

Opening the CloverETL Perspective

As has been already said, if you installed **CloverETL Designer** as a plugin, you need to switch to the **CloverETL** perspective.

After closing the **Eclipse** welcome scren, in order to switch the perspective, click the button next to the **Java** label at the top right side of the window over the **Outline** pane and select **Other...**.

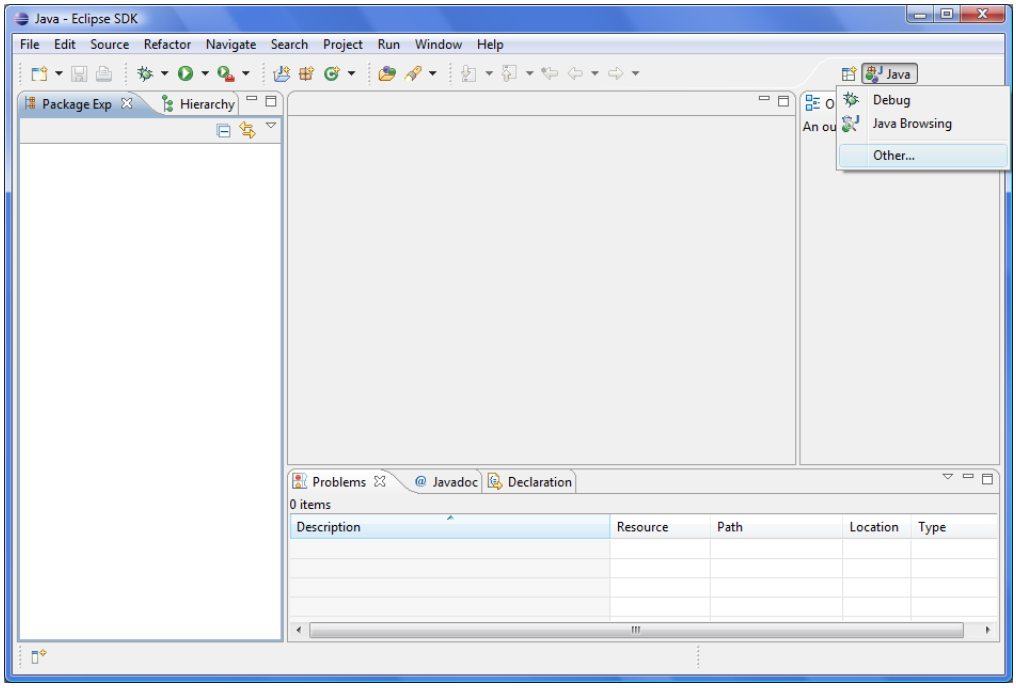


Figure 8.4. Basic Eclipse Perspective

After that, select the **CloverETL** item from the list and click **OK**.

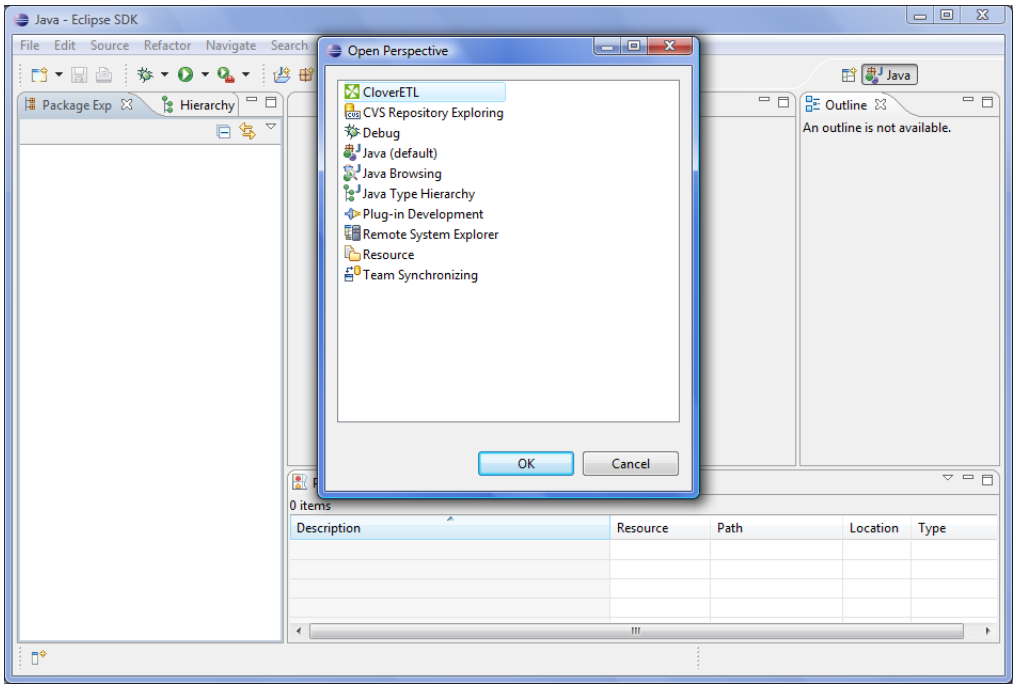


Figure 8.5. Selecting CloverETL Perspective

CloverETL perspective will open:

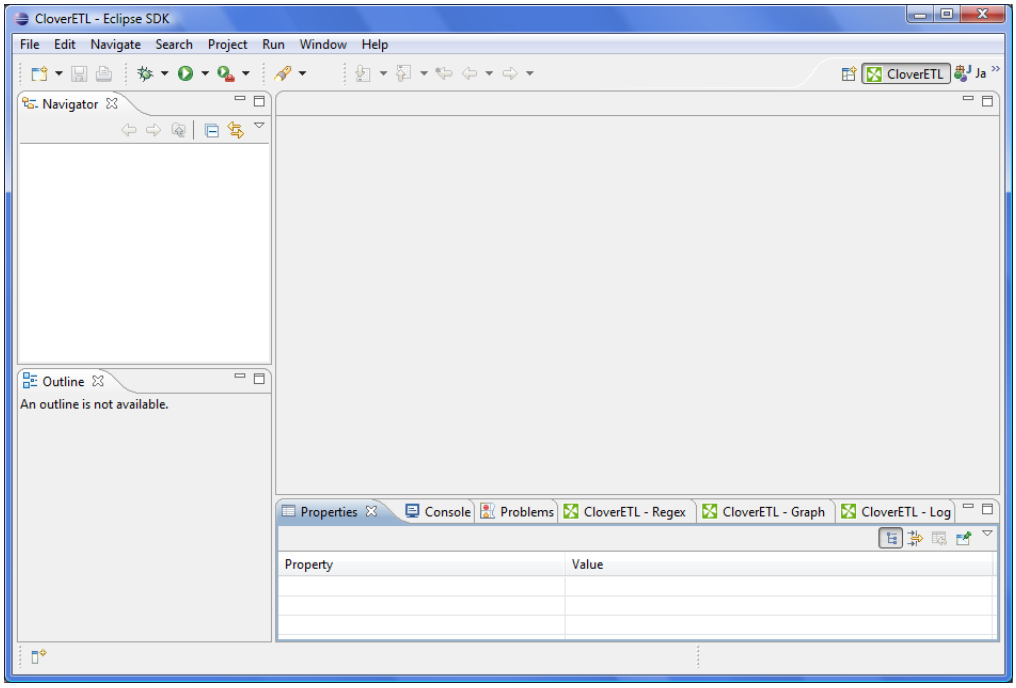


Figure 8.6. CloverETL Perspective

Chapter 9. Appearance of CloverETL Perspective

The **CloverETL** perspective consists of 4 panes:

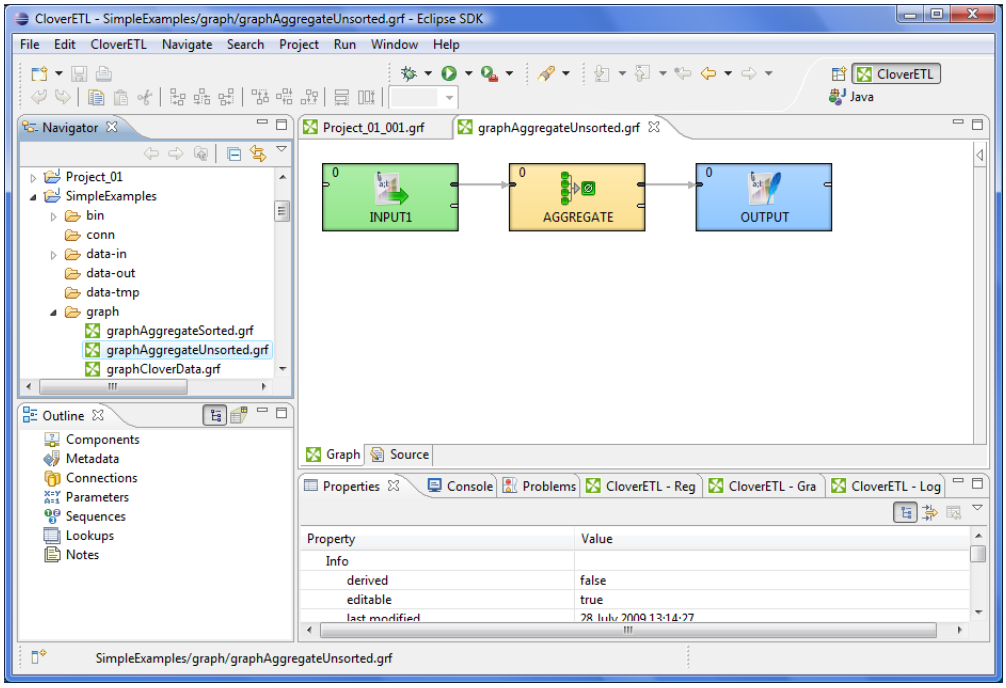


Figure 9.1. CloverETL Perspective

- **Graph Editor** with **Palette of Components** is in the upper right part of the window.

  In this pane you can create your graphs. **Palette of Components** serves to select components, move them into the **Graph Editor**, connect them by edges. This pane has two tabs. (See Graph Editor with Palette of Components (p. 29).)

- **Navigator** pane is in the upper left part of the window.

  There are folders and files of your projects in this pane. You can expand or collapse them and open any graph by double-clicking its item. (See Navigator Pane (p. 33).)

- **Outline** pane is in the lower left part of the window.

  There are all of the parts of the graph that is opened in the **Graph Editor**. (See Outline Pane (p. 33).)

- **Tabs** pane is in the lower right part of the window.

  You can see the data parsing process in these tabs. (See Tabs Pane (p. 35).)

CloverETL Designer Panes

Now we will present you a more detailed description of each pane.

The panes of **CloverETL Designer** are as follows:

- Graph Editor with Palette of Components (p. 29)
- Navigator Pane (p. 33)

- Outline Pane (p. 33)
- Tabs Pane (p. 35)

Graph Editor with Palette of Components

The most important pane is the **Graph Editor** with **Palette of Components**.

To create a graph, you need to work with the **Palette** tool. It is either opened after **CloverETL Designer** has been started or you can open it by clicking the arrow which is located above the **Palette** label or by holding the cursor on the **Palette** label. You can close the **Palette** again by clicking the same arrow or even by simple moving the cursor outside the **Palette** tool. You can even change the shape of the **Palette** by shifting its border in the **Graph Editor** and/or move it to the left side of the **Graph Editor** by clicking the label and moving it to this location.

The name of the user that has created the graph and the name of its last modifier are saved to the **Source** tab automatically.

It is the **Palette** tool from which you can select a component and paste it to the **Graph Editor**. To paste the component, you only need to click the component label, move the cursor to the **Graph Editor** and click again. After that, the component appears in the **Graph Editor**. You can do the same with the other components.

Once you have selected and pasted more components to the **Graph Editor**, you need to connect them by edges taken from the same **Palette** tool. To connect two components by an edge, you must click the **edge** label in the **Palette** tool, move the cursor to the first component, connect the edge to the output port of the component by clicking and move the cursor to the input of another component and click again. This way the two components will be connected. Once you have terminated your work with edges, you must click the **Select** item in the **Palette** window.

After creating or modifying a graph, you must save it by selecting the **Save** item from the context menu or by clicking the **Save** button in the main menu. The graph becomes a part of the project in which it has been created. A new graph name appears in the **Navigator** pane. All components and properties of the graph can be seen in the **Outline** pane when the graph is opened in the **Graph Editor**.

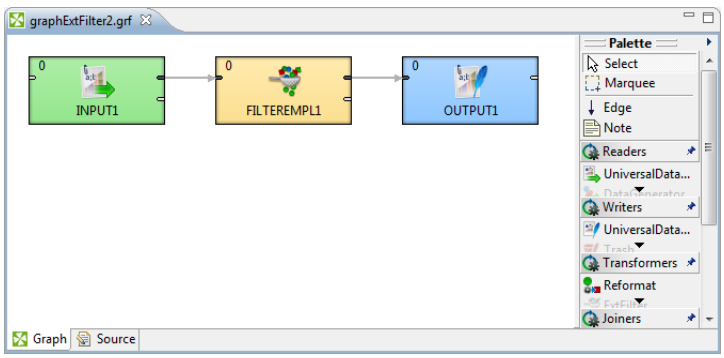


Figure 9.2. Graph Editor with an Opened Palette of Components

If you want to close any of the graphs that are opened in the **Graph Editor**, you can click the cross at the right side of the tab, but if you want to close more tabs at once, right-click any of the tabs and select a corresponding item from the context menu. There you have the items: **Close**, **Close other**, **Close All** and some other ones. See below:

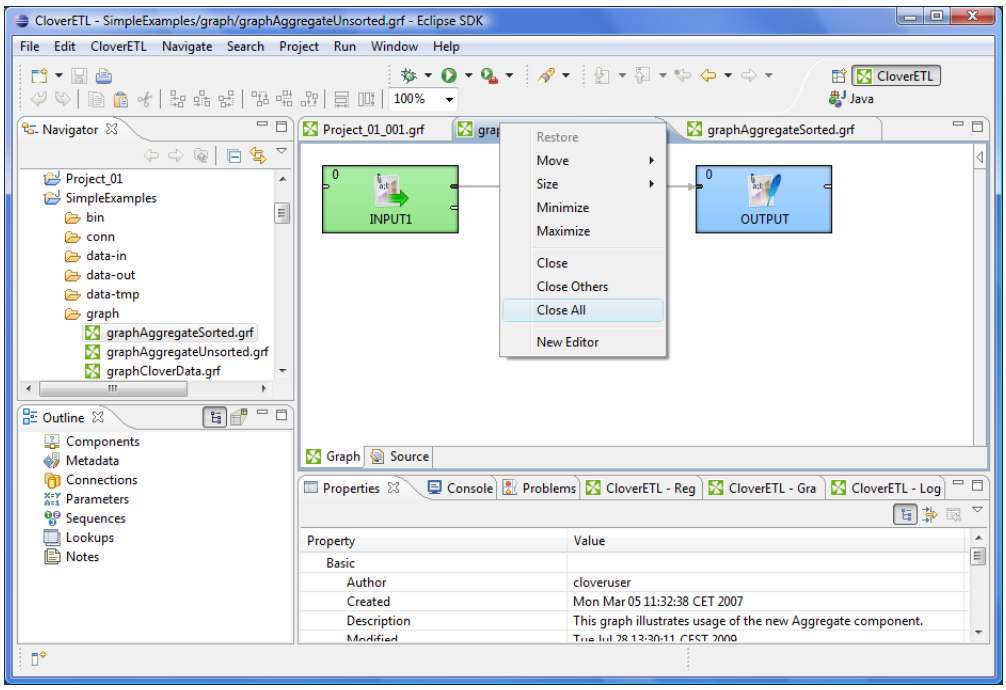


Figure 9.3. Closing the Graphs

From the main menu, you can select the **CloverETL** item (but only when the **Graph Editor** is highlighted) and you can turn on the **Rulers** option from the menu items.

After that, as you click anywhere in the horizontal or vertical rulers, there appear vertical or horizontal lines, respectively. Then, you can push any component to some of the lines and once the component is pushed to it by any of its sides, you can move the component by moving the line. When you click any line in the ruler, it can be moved throughout the **Graph Editor** pane. This way, you can align the components.

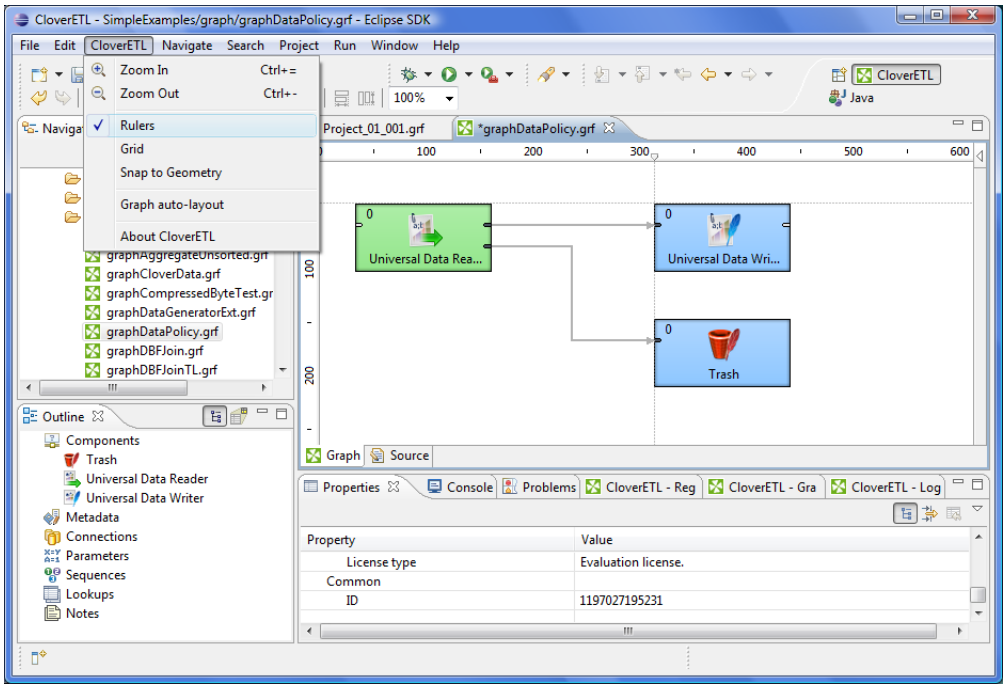


Figure 9.4. Rulers in the Graph Editor

From the main menu, you can also select the **CloverETL** item (but only when the **Graph Editor** is highlighted) and you can display a grid in the **Graph Editor** by selecting the **Grid** item from the main menu.

After that, you can use the grid to align the components as well. As you move them, the components are pushed to the lines of the grid by their upper and left sides. This way, you can align the components too.

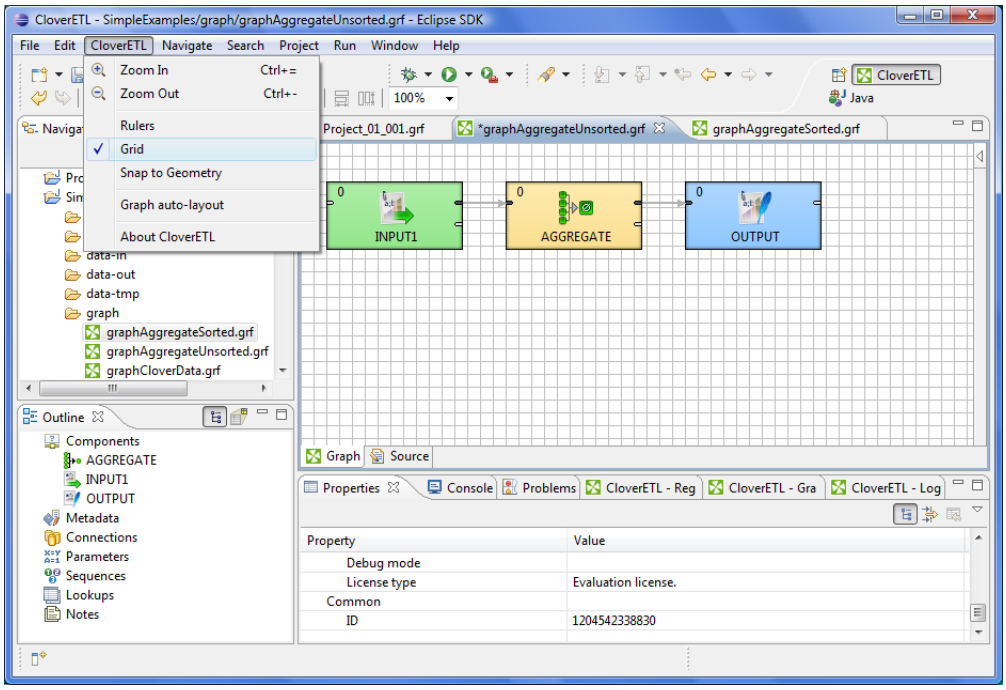


Figure 9.5. Grid in the Graph Editor

By clicking the **Graph auto-layout** item, you can change the layout of the graph. You can see how it changes when you select the **Graph auto-layout** item in case you have opened the `graphAggregateUnsorted.grf`. Before selecting this item, the graph looks like this:

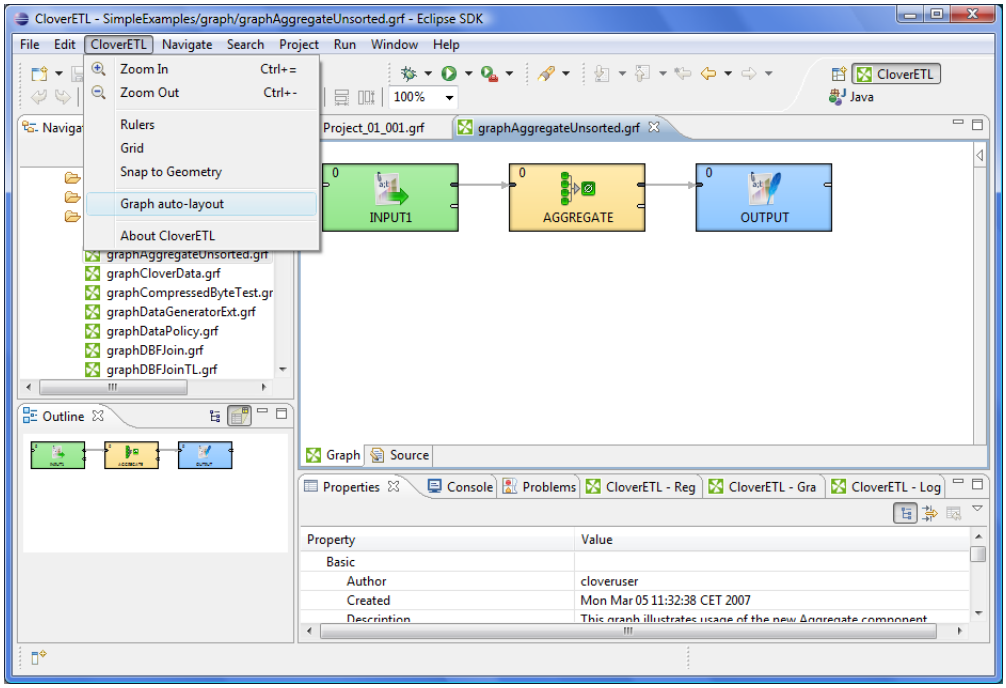


Figure 9.6. A Graph before Selecting Auto-Layout.

Once you have selected the mentioned item, graph could look like this:

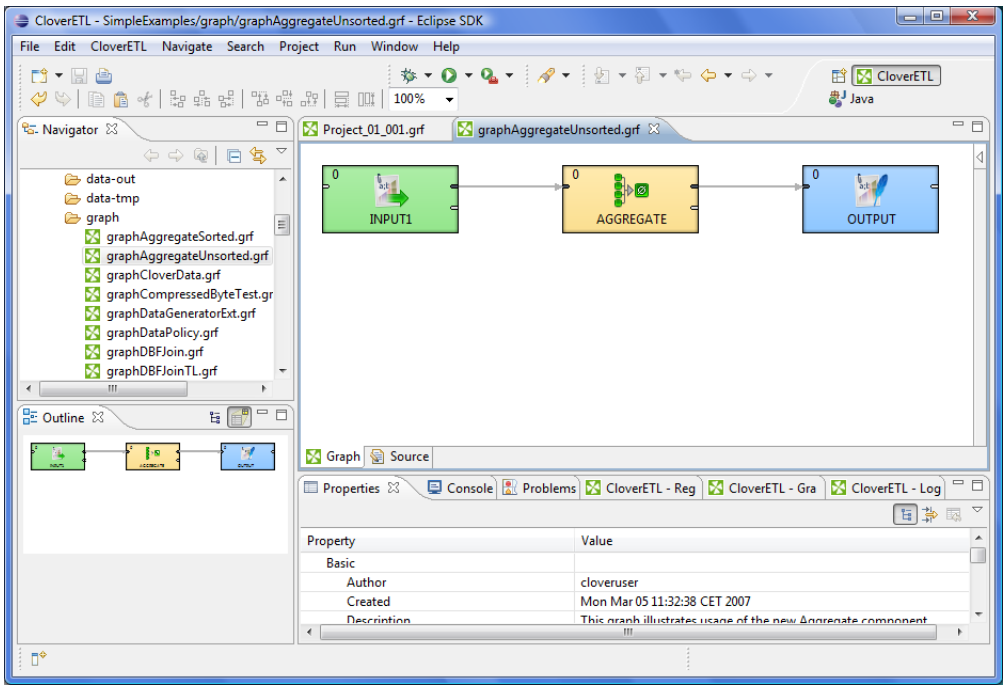


Figure 9.7. A Graph after Selecting Auto-Layout.

Another possibility of what you can do with the **Graph Editor** is the following:

When you push and hold down the left mouse button somewhere inside the **Graph Editor**, drag the mouse throughout the pane, a rectangle is created. When you create this rectangle in such a way so as to surround some of the graph components and finally release the mouse button, you can see that these components have become highlighted. (The first and second ones on the left in the graph below.) After that, six buttons (**Align Left**, **Align Center**, **Align Right**, **Align Top**, **Align Middle** and **Align Bottom**) appear highlighted in the tool bar above the **Graph Editor** or **Navigator** panes. (With their help, you can change the position of the selected components.) See below:

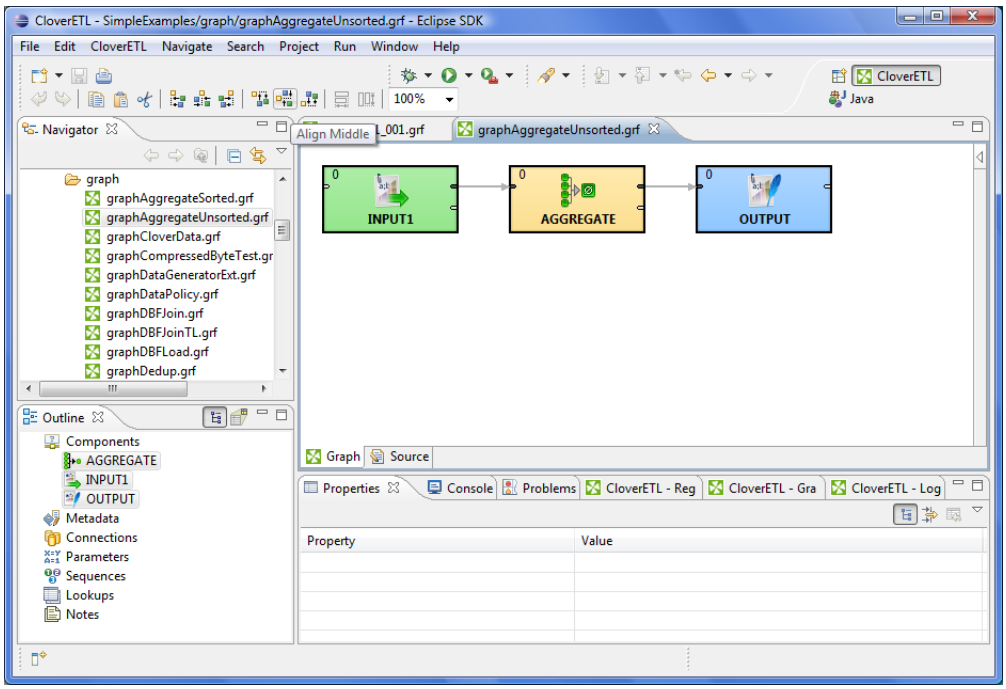


Figure 9.8. Six New Buttons in the Tool Bar Appear Highlighted (Align Middle is shown)

You can do the same by right-clicking inside the **Graph Editor** and selecting the **Alignments** item from the context menu. Then, a submenu appears with the same items as mentioned above.

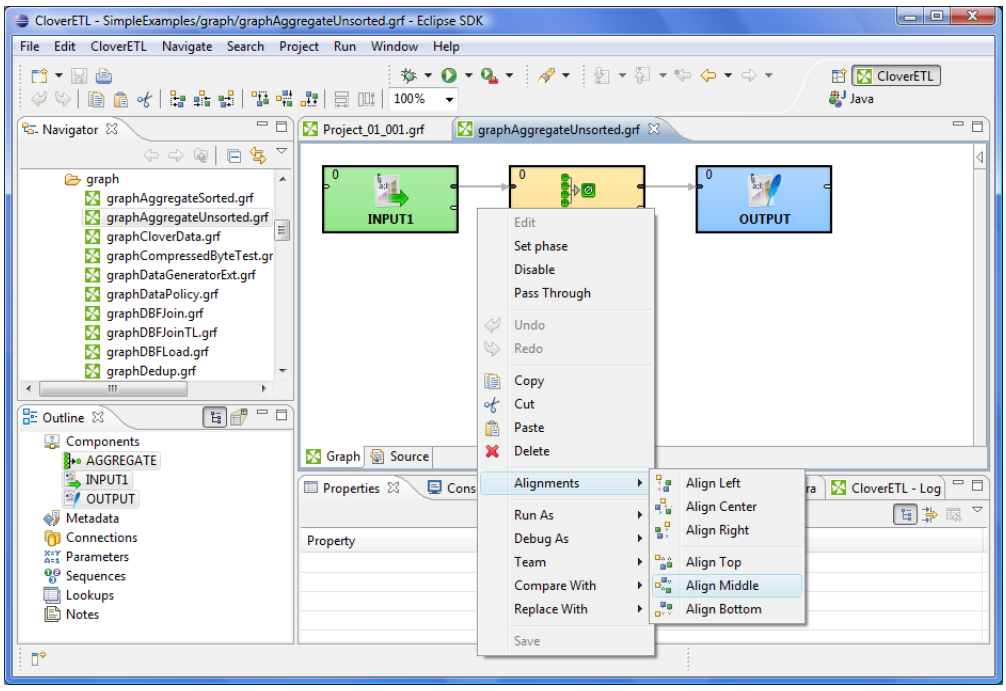


Figure 9.9. Alignments from the Context Menu

Remember that you can copy any highlighted part of any graph by pressing **Ctrl**+**C** and subsequently **Ctrl**+**V** after opening some other graph.

Navigator Pane

In the **Navigator** pane, there is a list of your projects, their subfolders and files. You can expand or collaps them, view them and open.

All graphs of the project are situated in this pane. You can open any of them in the **Graph Editor** by double-clicking the graph item.

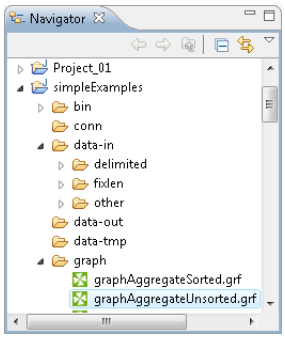


Figure 9.10. Navigator Pane

Outline Pane

In the **Outline** pane, there are shown all components of the selected graph. There you can create or edit all properties of the graph components, edges metadata, database connections or JMS connections, lookups, parameters, sequences, and notes. You can both create internal properties and link external (shared) ones. Internal

properties are contained in the graph and are visible there. You can externalize the internal properties and/or internalize the external (shared) properties. You can also export the internal metadata. If you select any item in the **Outline** pane (component, connection, metadata, etc.) and press **Enter**, its editor will open.

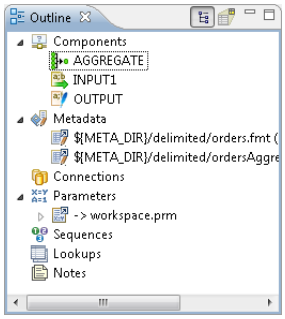


Figure 9.11. Outline Pane

Note that the two buttons in the upper right part of the **Outline** pane have the following properties:

By default you can see the tree of components, metadata, connections, parameters, sequences, lookups and notes in the **Outline** pane. But, when you click the button that is the second from the left in the upper right part of the **Outline** pane, you will be switched to another representation of the pane. It will look like this:

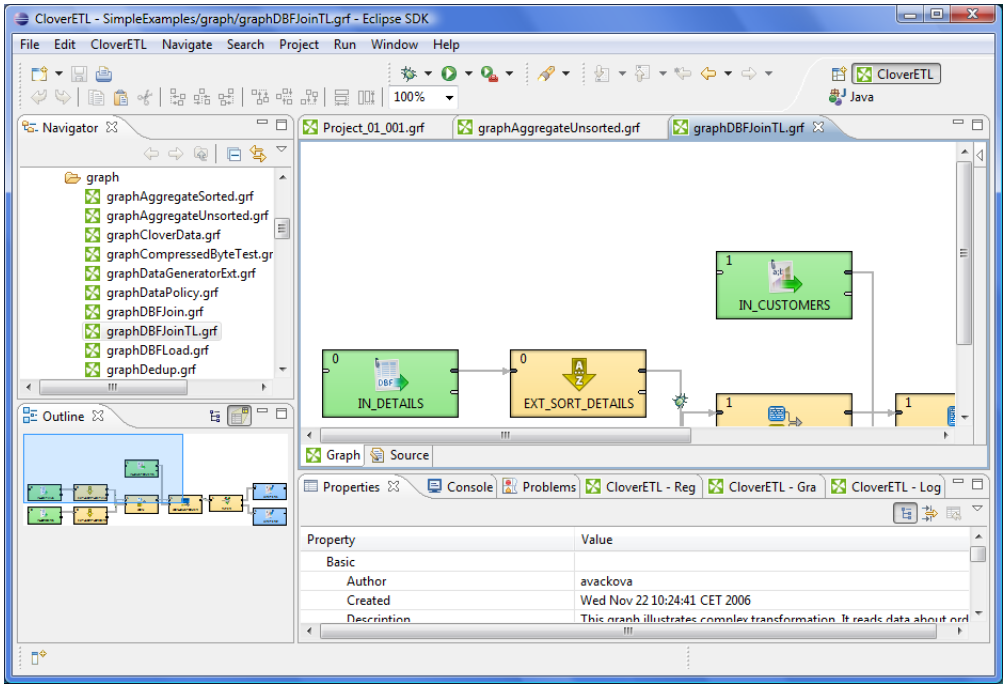


Figure 9.12. Another Representation of the Outline Pane

You can see a part of some of the example graphs in the **Graph Editor** and you can see the same graph structure in the **Outline** pane. In addition to it, there is a light-blue rectangle in the **Outline** pane. You can see exactly the same part of the graph as you can see in the **Graph Editor** within the light-blue rectangle in the **Outline** pane. By moving this rectangle within the space of the **Outline** pane, you can see the corresponding part of the graph in the **Graph Editor** as it moves along with the rectangle. Both the light blue-rectangle and the graph in the **Graph Editor** move equally.

You can do the same with the help of the scroll bars on the right and bottom sides of the **Graph Editor**.

To switch to the tree representation of the **Outline** pane, you only need to click the button that is the first from the left in the upper right part of the **Outline** pane.

Tabs Pane

In the lower right part of the window, there is a series of tabs.

Note

If you want to extend any of the tabs of some pane, you only need to double-click such a tab. After that, the pane will extend to the size of the whole window. When you double-click it again, it will return to its original size.

- **Properties** tab

  In this tab, you can view and/or edit the component properties. When you click a component, properties (attributes) of the selected component appear in this tab.

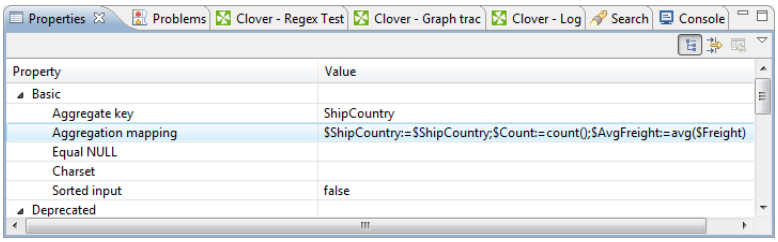


Figure 9.13. Properties Tab

- **Console** tab

  In this tab, process of reading, unloading, transforming, joining, writing, and loading data can be seen.

  By default, **Console** opens whenever **CloverETL** writes to `stdout` or `stderr` to it.

  If you want to change it, you can uncheck any of the two checkboxes that can be found when selecting **Window** →**Preferences**, expanding the **Run/Debug** category and opening the **Console** item.

  Two checkboxes that control the behavior of **Console** are the following:

  - **Show when program writes to standard out**
  - **Show when program writes to standard error**

  Note that you can also control the buffer of the characters stored in **Console**:

  There is another checkbox (**Limit console output**) and two text fields for the buffer of stored characters and the tab width.

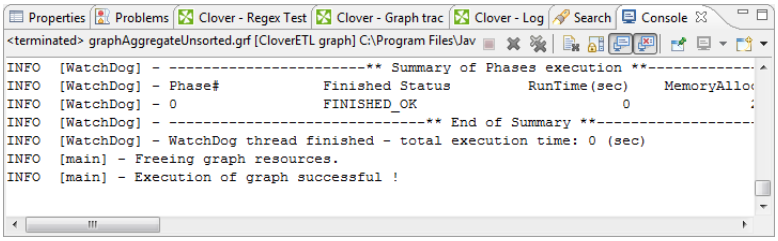


Figure 9.14. Console Tab

- **Problems** tab

  In this tab, you can see error messages, warnings, etc. When you expand any of the items, you can see their resources (name of the graph), their paths (path to the graph), their location (name of the component).

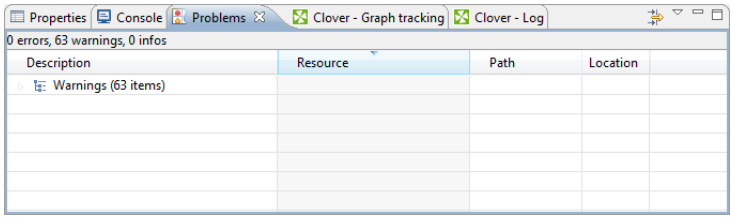


Figure 9.15. Problems Tab

- **Clover - Regex Tester** tab

  In this tab, you can work with regular expressions. You can paste or type any regular expression into the **Regular expression** text area. Content assist can be called out by pressing **Ctrl**+**Space** in this area. You need paste or type the desired text into the pane below the **Regular expression** text area. After that, you can compare the expression with the text. You can either evaluate the expression on the fly while you are changing the expression, or you can uncheck the **Evaluate on the fly** checkbox and compare the expression with the text upon clicking the **Go** button. The result will be displayed in the pane on the right. Some options are checked by default. You can also select if you want to **Find** the expression in the text, or you want to **Split** the text according the expression or you want to know whether the text **Matches** the regular expression completely. You have at your disposal a set of checkboxes. More information about regular expressions and provided options can be found at the following site: http://java.sun.com/j2se/1.6.0/docs/api/java/util/regex/Pattern.html

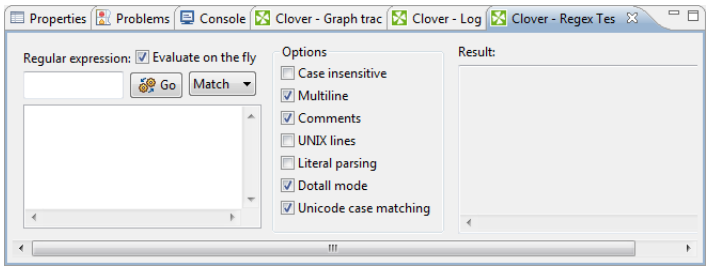


Figure 9.16. Clover - Regex Tester Tab

- **Clover - Graph tracking** tab

  In this tab, you can see a brief description of the graph that run successfully. The names of the components, grouped by phases (with their using time in seconds, their using capacity in percents), status of all components, **CPU Time** that has been used for them (in seconds), **CPU** load that has been used for them (in percents), **Byte flow** and **Byte flow peak** (in Bytes per second), **Record flow** and **Record flow peak** (in records per second), **Total bytes** processed (in Bytes, Kilobytes, etc.), **Total records** processed (in records). These properties are displayed in the tab by default.

  In addition to them, also **User time**, **Peak CPU**, **Waiting records** and **Average waiting records** can be displayed. These should be added to those already mentioned by selecting **Window →Preferences**. Then you need to expand the **CloverETL** group and select the **Tracking** item. In the pane, you can see a list of log columns. You can add or remove some of them by clicking the **Next...** or **Remove** buttons, respectively. You can also reorder the columns within the **Graph tracking** tab by clicking the **Up** or **Down** buttons, respectively.

  You can also turn off the tracking of graphs by unchecking the **Show tracking during graph run** checkbox.

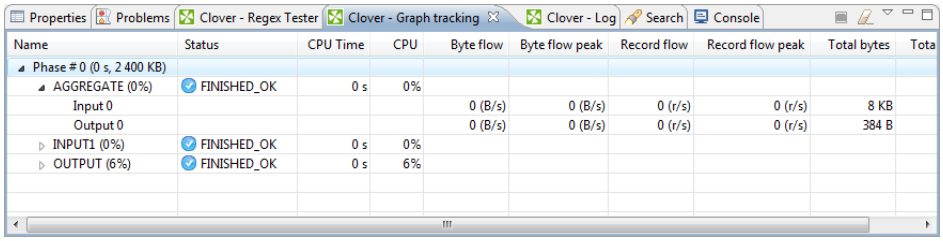


Figure 9.17. Clover - Graph Tracking Tab

- **Clover - Log** tab

  In this tab, you can see the entire log from the process of data parsing that is created after running a graph. There can be a set of logs from more runs of graphs.

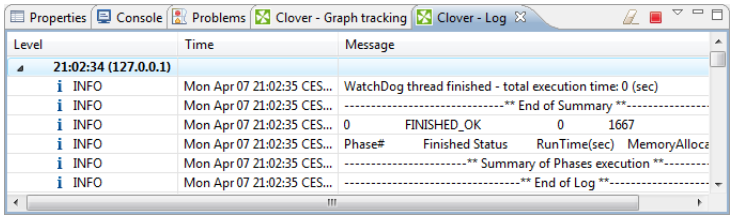


Figure 9.18. Clover - Log Tab

Chapter 10. Creating CloverETL Graphs

Within any **CloverETL** project, you need to create **CloverETL** graph. In the following sections we are going to describe how you can create your graphs:

1. As the first step, you must create an empty graph in a project. See Creating Empty Graphs (p. 38).

2. As the second step, you must create the transformation graph by using graph components, elements and others tools. See Creating a Simple Graph in a Few Simple Steps (p. 41) for an example in which we want to show you an example of the transformation graph creation.

Note

Remember that once you have already some **CloverETL** project in you workspace and have opened the **CloverETL** perspective, you can create your next **CloverETL** projects in a slightly different way:

- You can create directly a new **CloverETL** project from the main menu by selecting **File** →**New** →**CloverETL Project** or select **File** →**New** →**Project...** and do what has been described above.

- You can also right-click inside the **Navigator** pane and select either directly **New** →**CloverETL Project** or **New** →**Project...** from the context menu and do what has been described above.

Creating Empty Graphs

After that, you can create **CloverETL** graphs for any of your **CloverETL** projects. For example, you can create a graph for the `Project_01` by choosing **File** →**New** →**ETL Graph**. You can also right-click the desired project in the **Navigator** pane and select **New** →**ETL Graph** from the context menu.

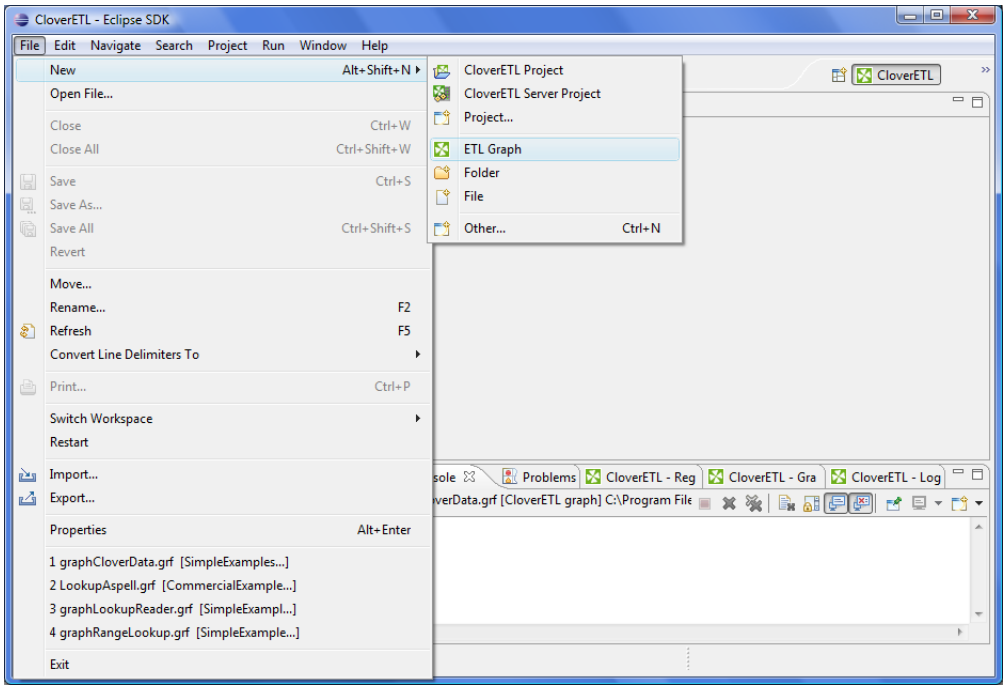


Figure 10.1. Creating a New Graph

After clicking the item, you will be asked to give a name to the graph. For example, the name can be `Project_01` too. But, in most cases your project will contain more graphs and you can give them names such as `Project_01_###`, for example. Or any other names which would describe what these graphs should do.

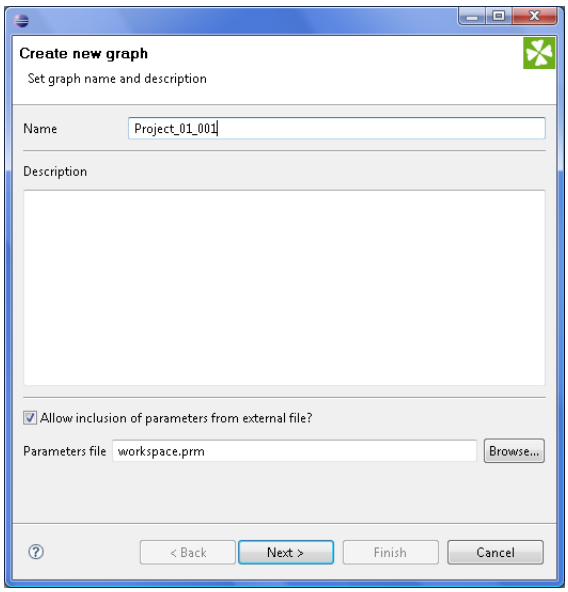


Figure 10.2. Giving a Name to a New CloverETL Graph

Remember that you can decide what parameters file should be included to this project along with the graph. This selection can be done in the text area at the bottom of this window. You can locate some other file by clicking the **Browse...** button and searching for the right one. Or, you can even uncheck the checkbox leaving the graph without a parameters file included.

We decided to have the `workspace.prm` file included.

At the end, you can click the **Next** button. After that, the extension `.grf` will be added to the selected name automatically.

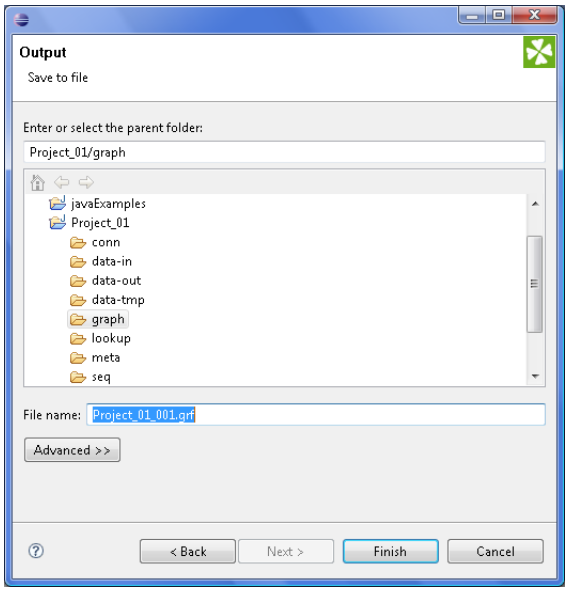


Figure 10.3. Selecting the Parent Folder for the Graph

By clicking **Finish**, you save the graph in the `graph` subfolder. Then, an item `Project_01_001.grf` appears in the **Navigator** pane and a tab named **Project_01_001.grf** appears on the window.

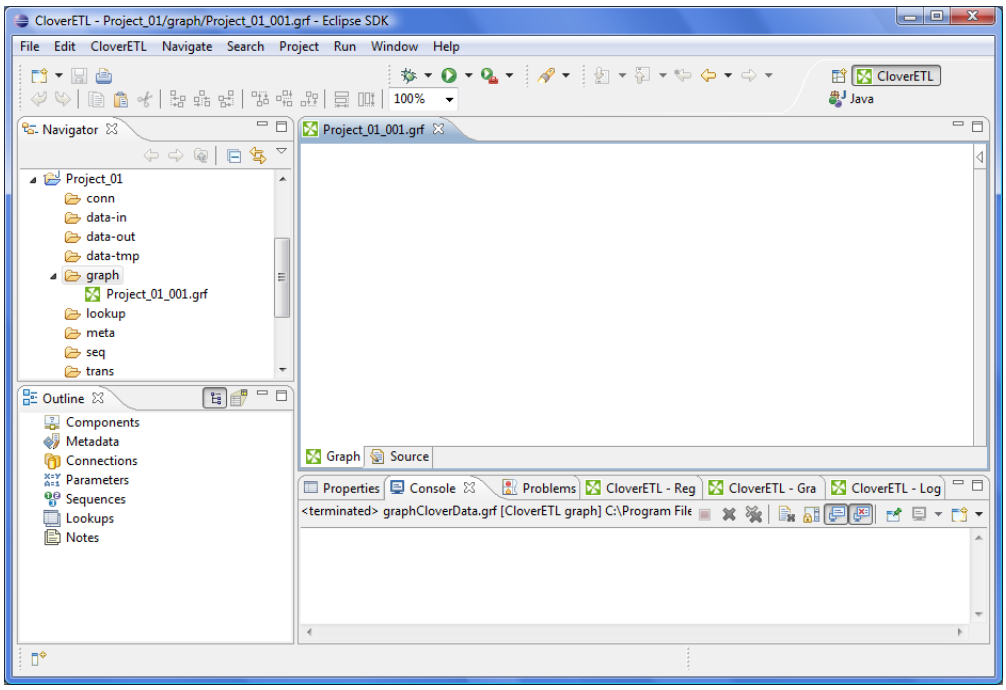


Figure 10.4. CloverETL Perspective with Highlighted Graph Editor

You can see that there is a palette of components on the right side of the graph. This palette can be opened and closed by clicking the **Triangle** button. If you want to know what components are, see Part VI, Components Overview (p. 219) for information.

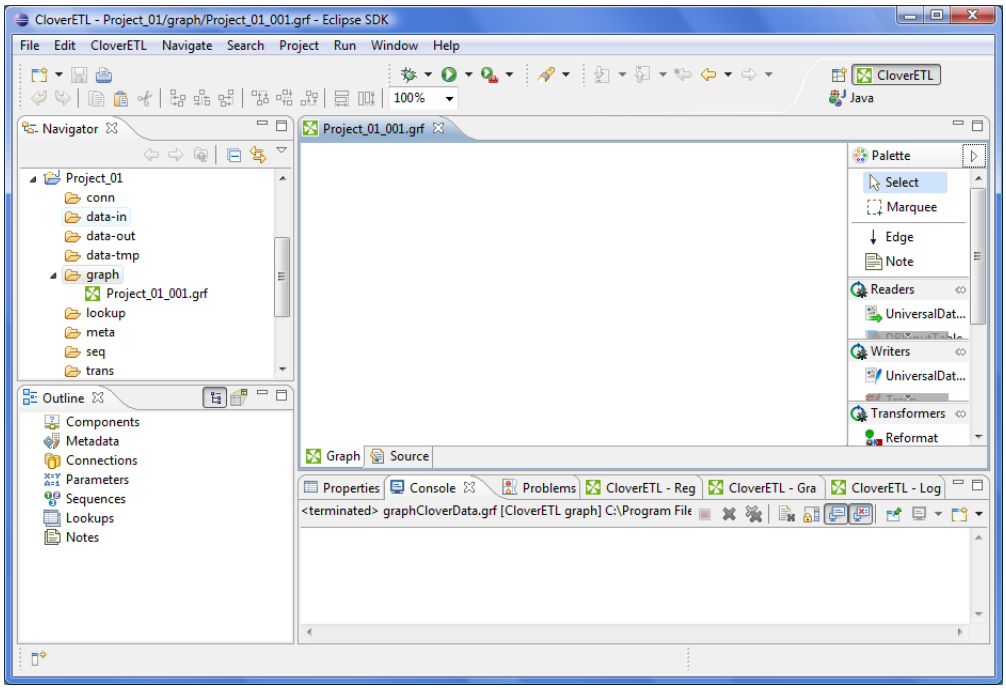


Figure 10.5. Graph Editor with a New Graph and the Palette of Components

Creating a Simple Graph in a Few Simple Steps

After creating a new **CloverETL** graph, it is an empty pane. In order to create a non-empty graph, you must fill the empty graph with components and other graph elements. You need to select graph components, set up their properties (attributes), connect these components by edges, select data files and/or database tables that should be read or unloaded from, written or loaded to, create metadata describing data, assign them to edges, create database connections or JMS connections, create lookup tables and/or create sequences and parameters. Once all of it is done, you can run the graph.

If you want to know what edges, metadata, connections, lookup tables, sequences or parameters are, see Part V, Graph Elements, Structures and Tools (p. 83) for information.

Now we will present you a simple example of how **CloverETL** transformation graphs can be created using **CloverETL Designer**. We will try to make the explanation as clear as possible.

First, you need to select components from the **Palette of Components**.

To select any component, click the triangle on the upper right corner of the **Graph Editor** pane. The **Palette of Components** will open. Select the components you want by clicking and then drag-and-dropping them to the **Graph Editor** pane.

For our demonstration purposes, select **UniversalDataReader** from the **Readers** category of the **Palette**. Select also the **ExtSort** component from the **Transformers** category and **UniversalDataWriter** from the **Writers** category.

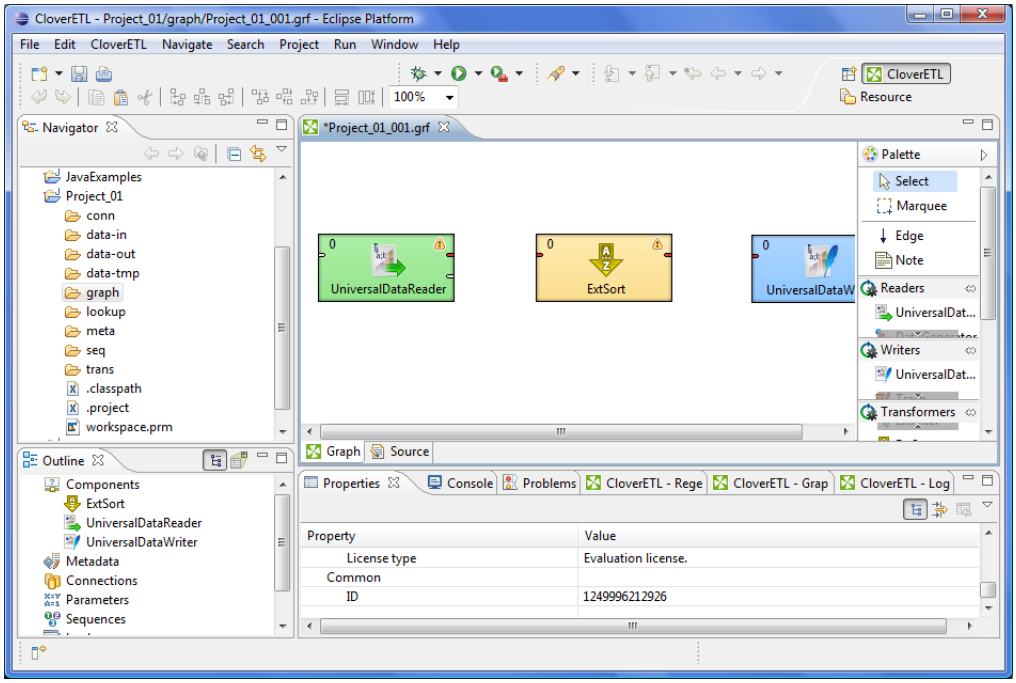


Figure 10.6. Components Selected from the Palette

Once you have inserted the components to the **Graph Editor** pane, you need to connect them by edges. Select the **Edge** tool on the **Palette** and click the output port of one component and connect it with the input port of another by clicking again. Do the same with all selected components. The newly connected edges are still dashed. Close the **Palette** by clicking the triangle at its upper right corner. (See Chapter 18, Edges (p. 86) for more information about **Edges**.)

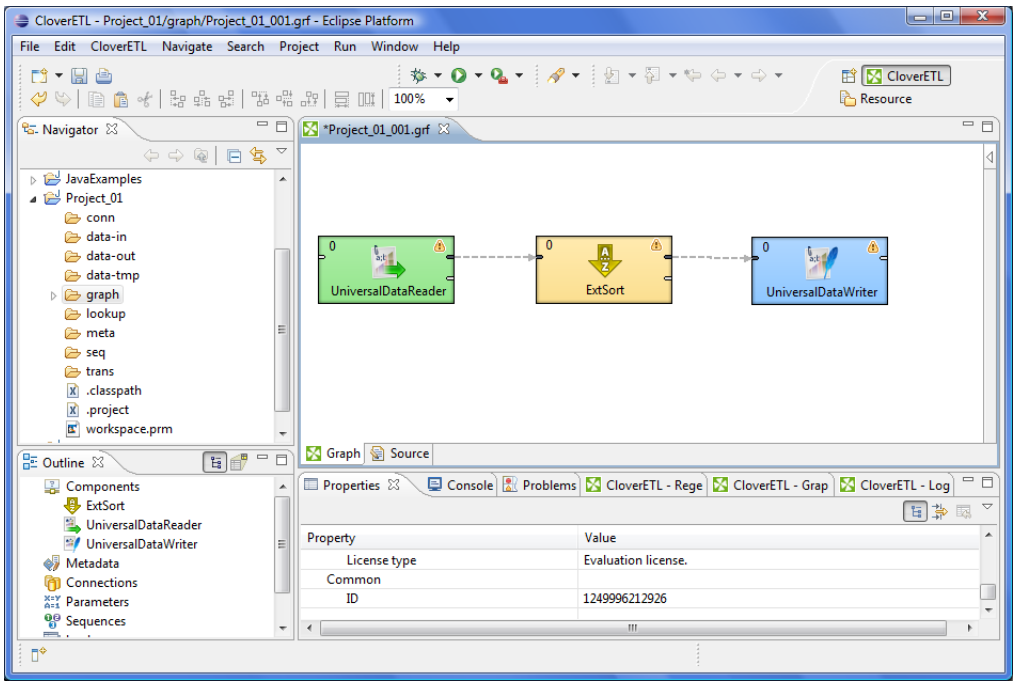


Figure 10.7. Components are Connected by Edges

Now you need to prepare some input file. Move to the **Navigator** pane, which is on the left side of **Eclipse** window. Right-click the `data-in` folder of your project and select **New** →**File**.

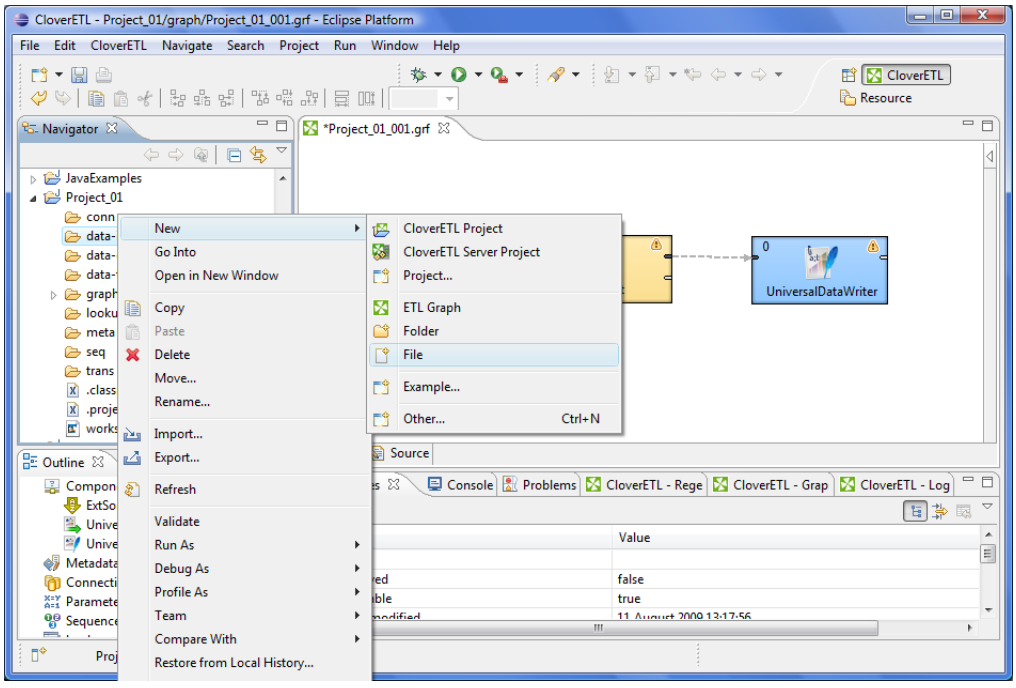


Figure 10.8. Creating an Input File

Once the new window appears, select the name of your input file in this window. For example, its name can be `input.txt`. Click **Finish**. The file will open in the **Eclipse** window.

Type some data in this file, for example, you can type pairs of firstname and surname like this: `John|Brown`. Type more rows whose form should be similar. Do not forget to create also a new empty row at the end. The rows (records) will look like this:

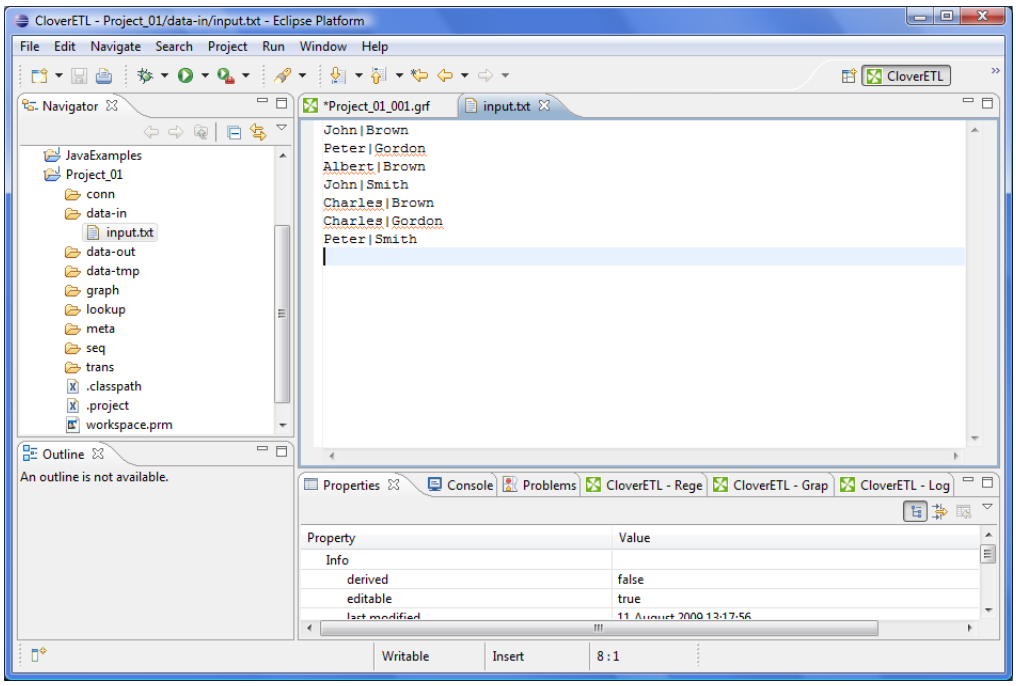


Figure 10.9. Creating the Contents of the Input File

You need to save the file by pressing **Ctrl**+**S**.

After that, double-click the first edge from the left and select **Create metadata** from the menu that appears beside the edge. In the **Metadata editor**, click the green **Plus sign** button. Another (second) field appears. You can click any of the two fields and rename them. By clicking any of them, it turns blue, you can rename it and press **Enter**. (See Chapter 19, Metadata (p. 96) for more information about creating **Metadata**.)

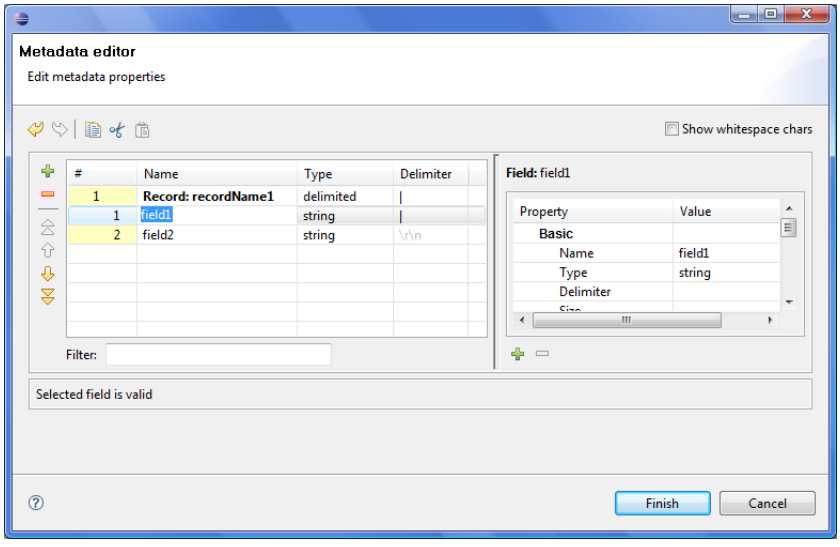


Figure 10.10. Metadata Editor with Default Names of the Fields

After doing that, the names of the two fields will be **Firstname** and **Surname**, respectively.

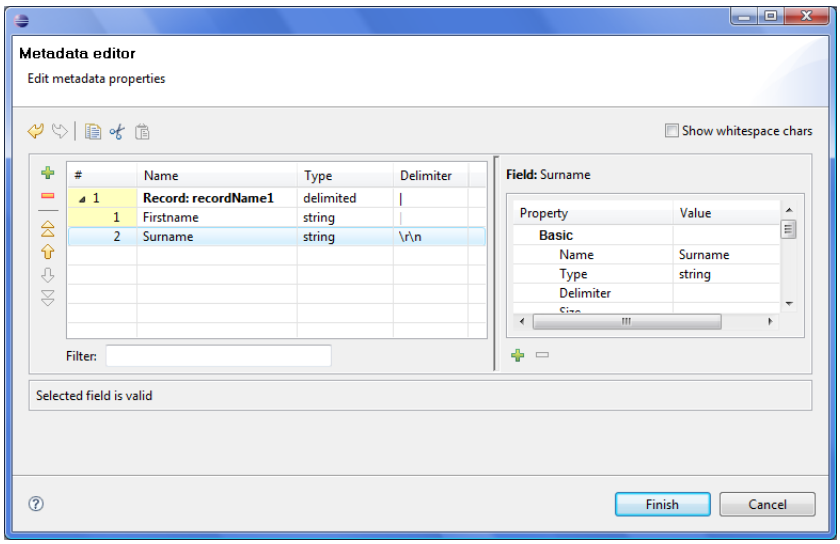


Figure 10.11. Metadata Editor with New Names of the Fields

After clicking **Finish**, metadata is created and assigned to the edge. The edge is solid now.

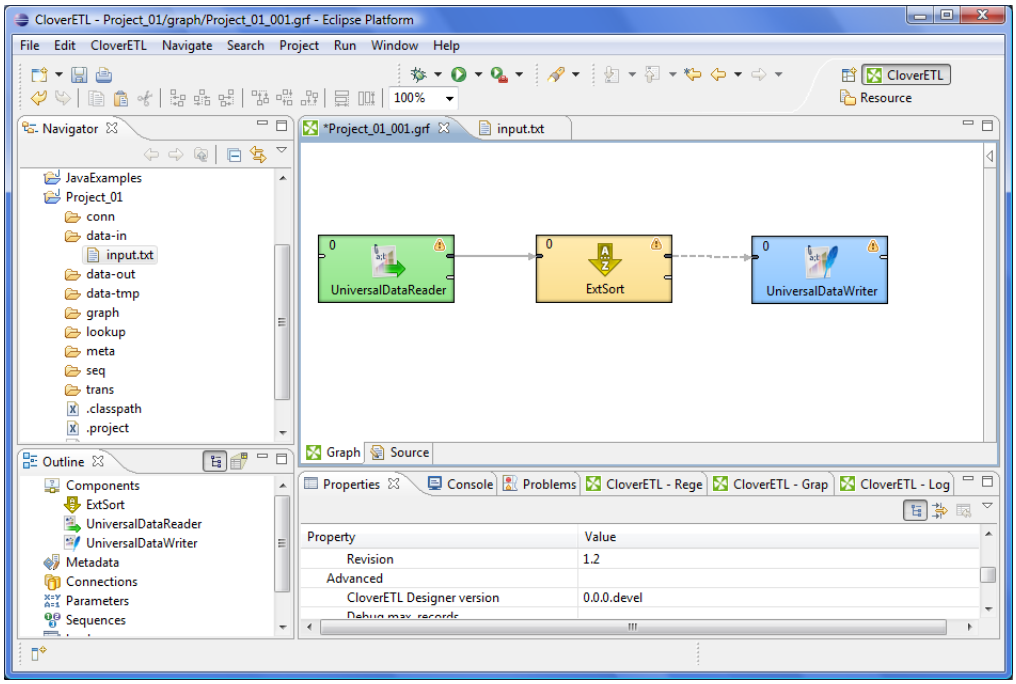


Figure 10.12. Edge Has Been Assigned Metadata

Now right-click the first edge and select **Propagate metadata** from the context menu. The second edge also becomes solid.

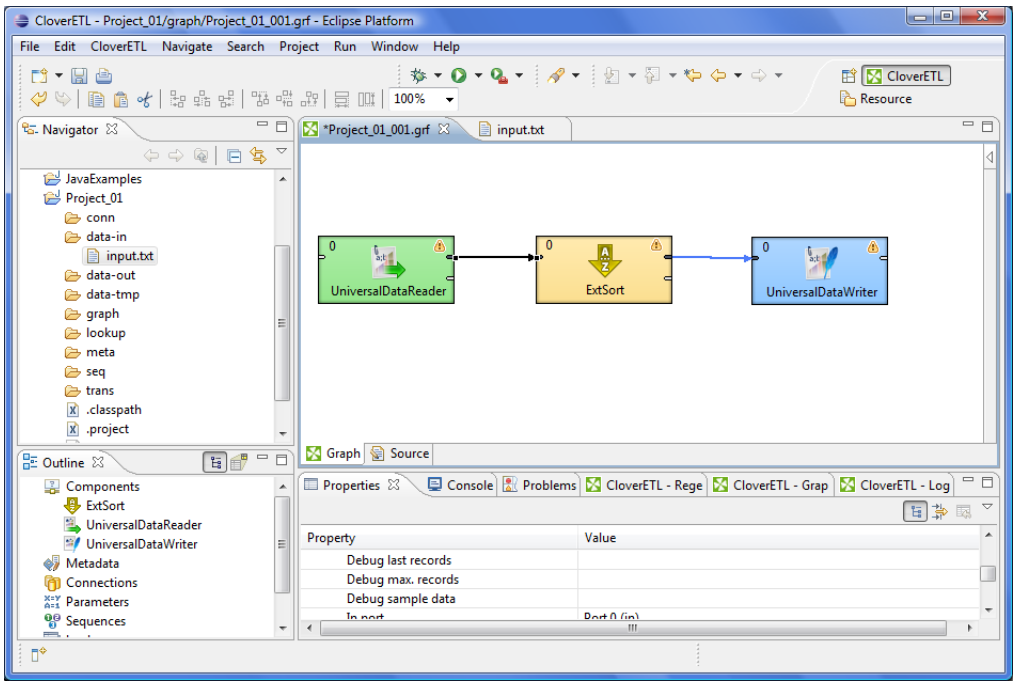


Figure 10.13. Metadata Have Been Propagated through the Component

Now, double-click **UniversalDataReader**, click the **File URL** attribute row and click the button that appears in this **File URL** attribute row.

(You can see UniversalDataReader (p. 333) for more information about **UniversalDataReader**.)

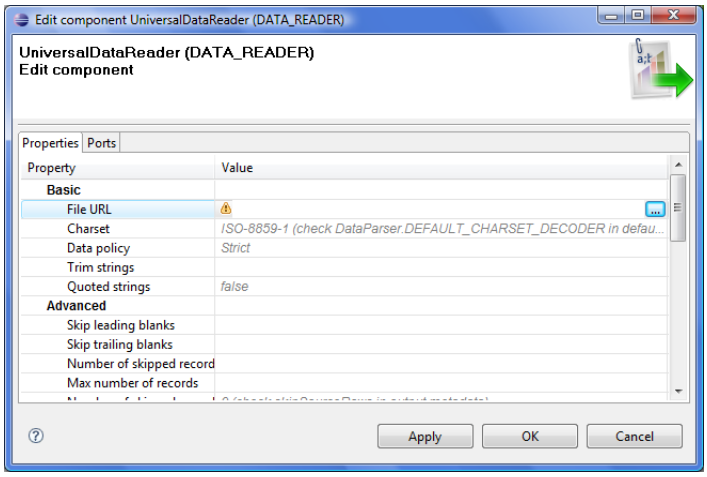


Figure 10.14. Opening the Attribute Row

After that, URL File Dialog (p. 59) will open. Double-click the `data-in` folder and double-click the `input.txt` file inside this folder. The file name appears in the right pane.

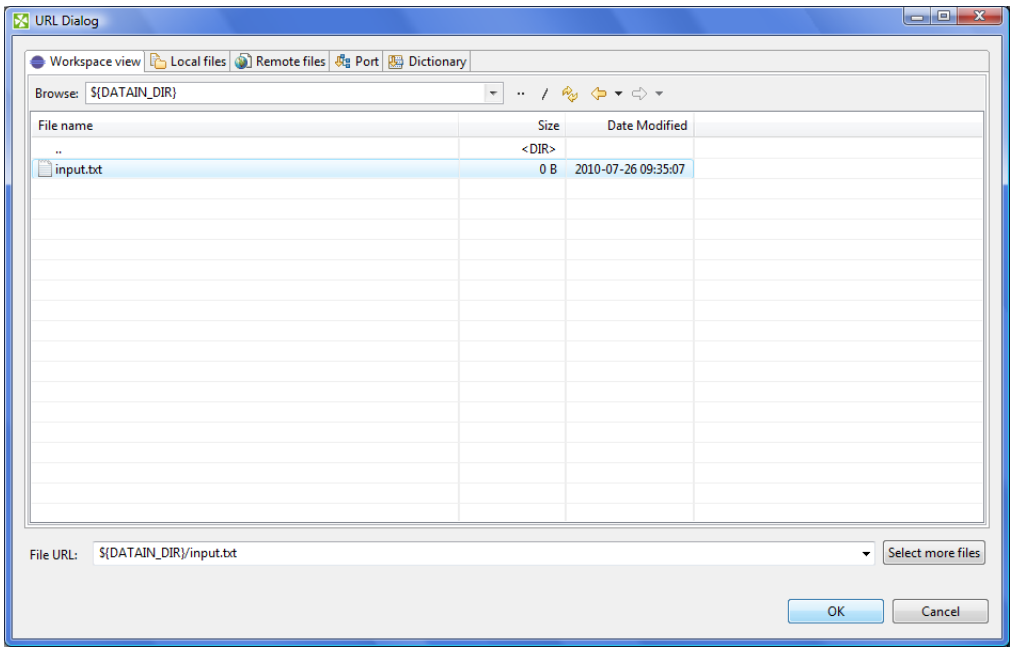


Figure 10.15. Selecting the Input File

Then click **OK**. The **File URL** attribute row will look like this:

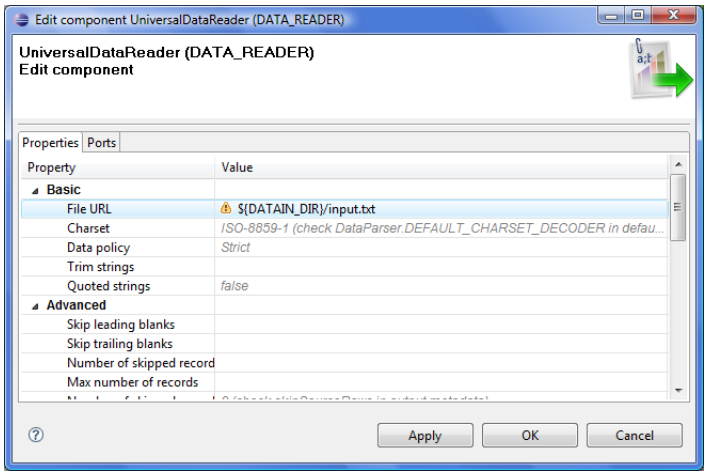


Figure 10.16. Input File URL Attribute Has Been Set

Click **OK** to close the **UniversalDataReader** editor.

Then, double click **UniversalDataWriter**.

(You can see UniversalDataWriter (p. 416) for more information about **UniversalDataWriter**.)

Click the **File URL** attribute row and click the button that appears in this **File URL** attribute row. After that, URL File Dialog (p. 59) will open. Double-click `data-out` folder. Then click **OK**. The **File URL** attribute row will look like this:

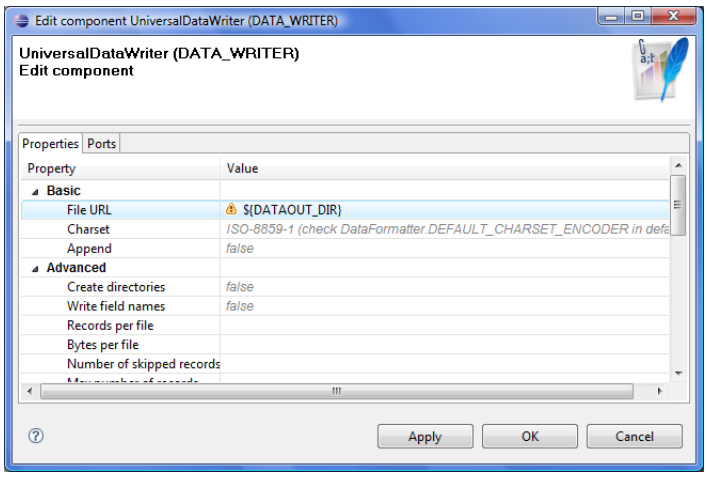


Figure 10.17. Output File URL without a File

Click twice the **File URL** attribute row and type `/output.txt` there. The result will be as follows:

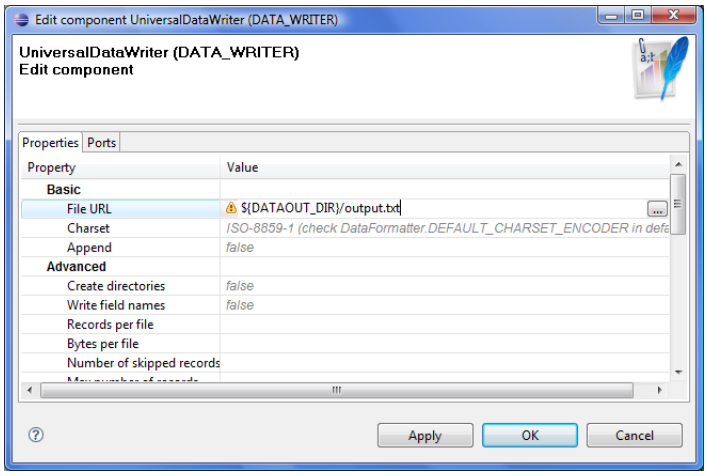


Figure 10.18. Output File URL with a File

Click **OK** to close the **UniversalDataWriter** editor.

Now you only need to set up the **ExtSort** component.

(You can see ExtSort (p. 472) for more information about **ExtSort**.)

Double-click the component and its **Sort key** attribute row. After that, move the two metadata fields from the left pane (**Fields**) to the right pane (**Key parts**). Move **Surname** first, then move **Firstname**.

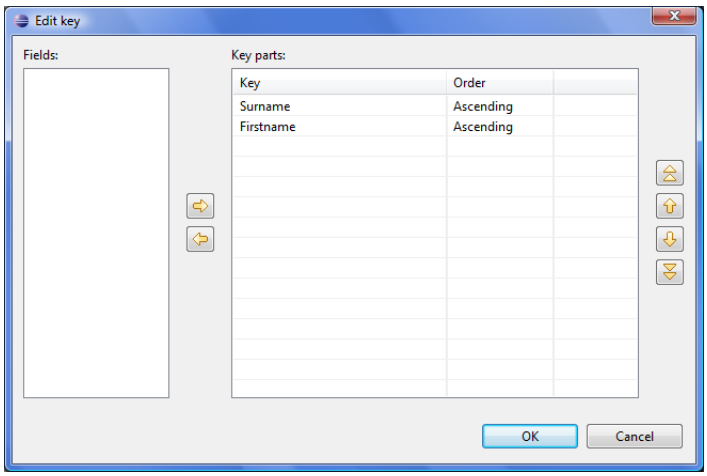


Figure 10.19. Defining a Sort Key

When you click **OK**, you will see the **Sort key** attribute row as follows:

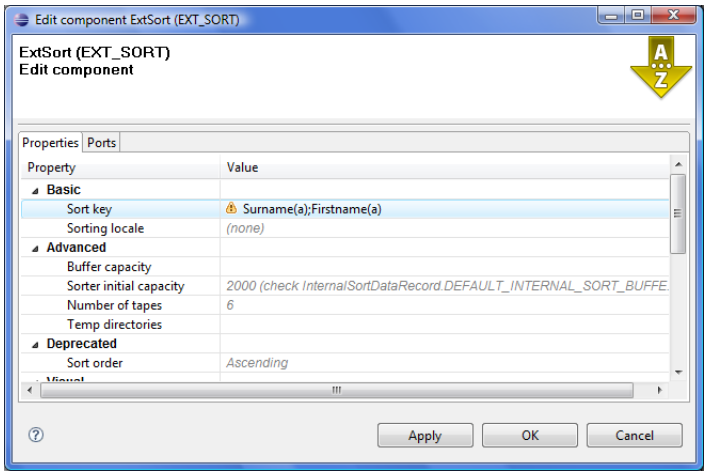


Figure 10.20. Sort Key Has Been Defined

Click **OK** to close the **ExtSort** editor and save the graph by pressing **Ctrl**+**S**.

Now right-click in any place of the **Graph Editor** (outside any component or edge) and select **Run As** →**CloverETL graph**.

(Ways how graphs can be run are described in Chapter 11, Running CloverETL Graphs (p. 51).)

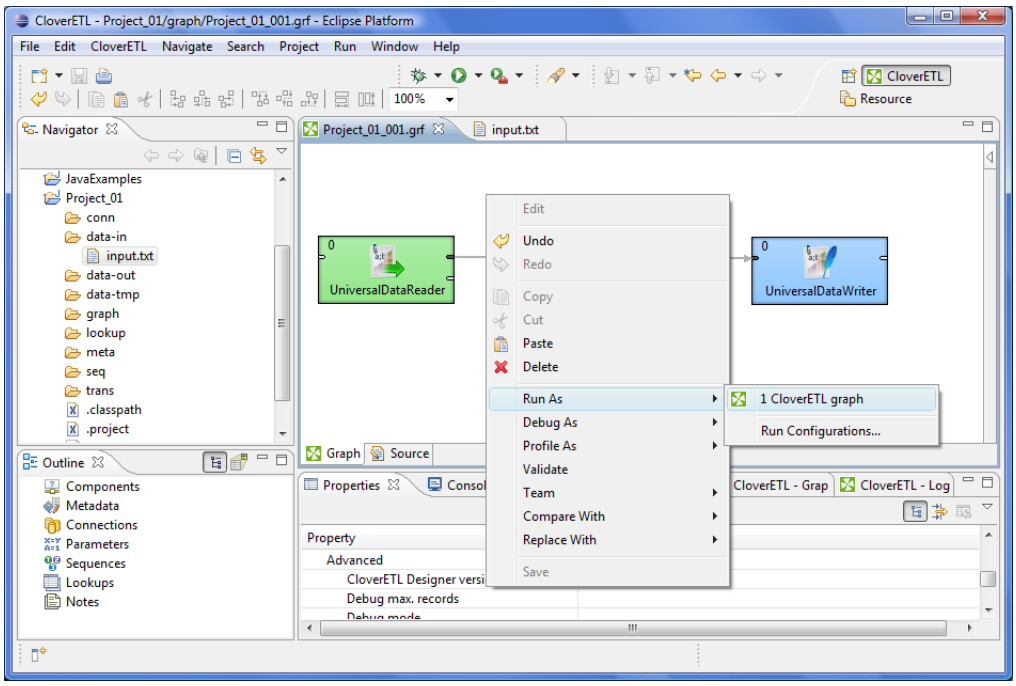


Figure 10.21. Running the Graph

Once graph runs successfully, blue circles are displayed on the components and numbers of parsed records can be seen below the edges:

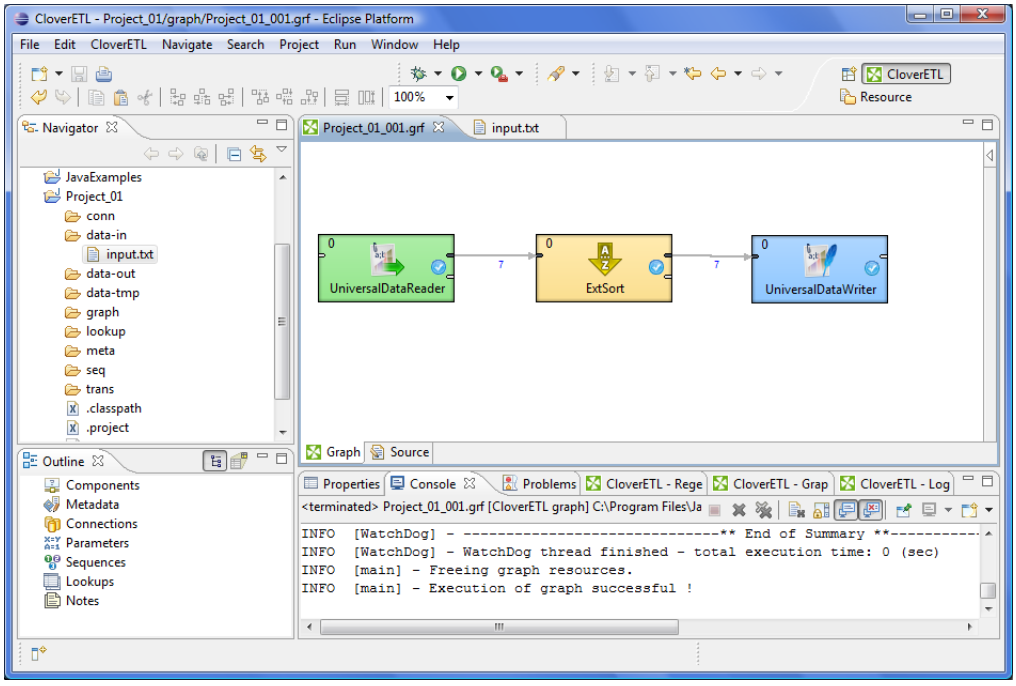


Figure 10.22. Result of Successful Run of the Graph

When you expand the `data-out` folder in the **Navigator** pane and open the output file, you can see the following contents of the file:

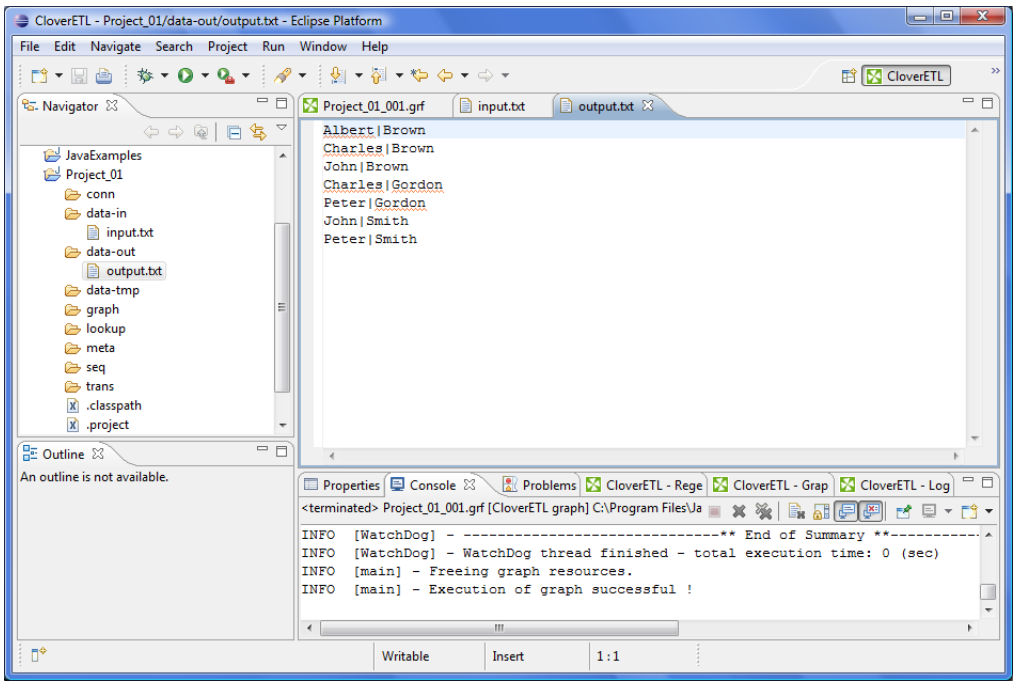


Figure 10.23. Contents of the Output File

You can see that all persons have been sorted alphabetically. Surnames first, firstnames last. This way, you have built and run your first graph.

Chapter 11. Running CloverETL Graphs

As was already mentioned, in addition to the context menu from which you can run graphs, you can run them from other places:

When you have already created or imported graphs into your projects, you can run them in various ways

There are four simpliest ways of running a graph:

- You can select **Run →Run as →CloverETL graph** from the main menu.
- Or you can right-click in the **Graph editor**, then select **Run as** in the context menu and click the **CloverETL graph** item.
- Or you can click the green circle with white triangle in the tool bar located in the upper part of the window.

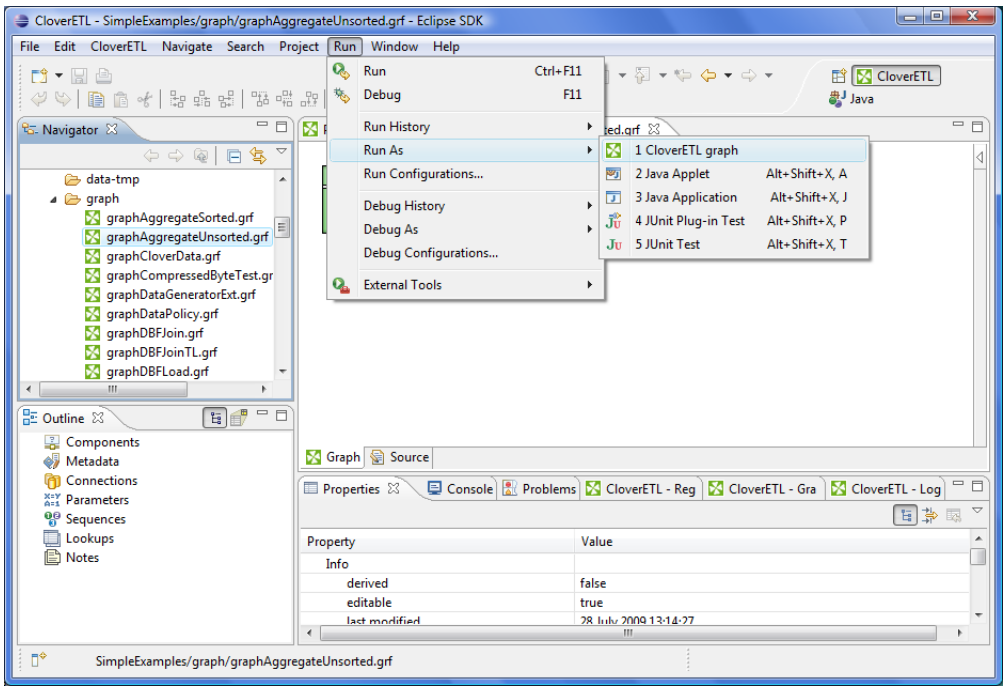


Figure 11.1. Running a Graph from the Main Menu

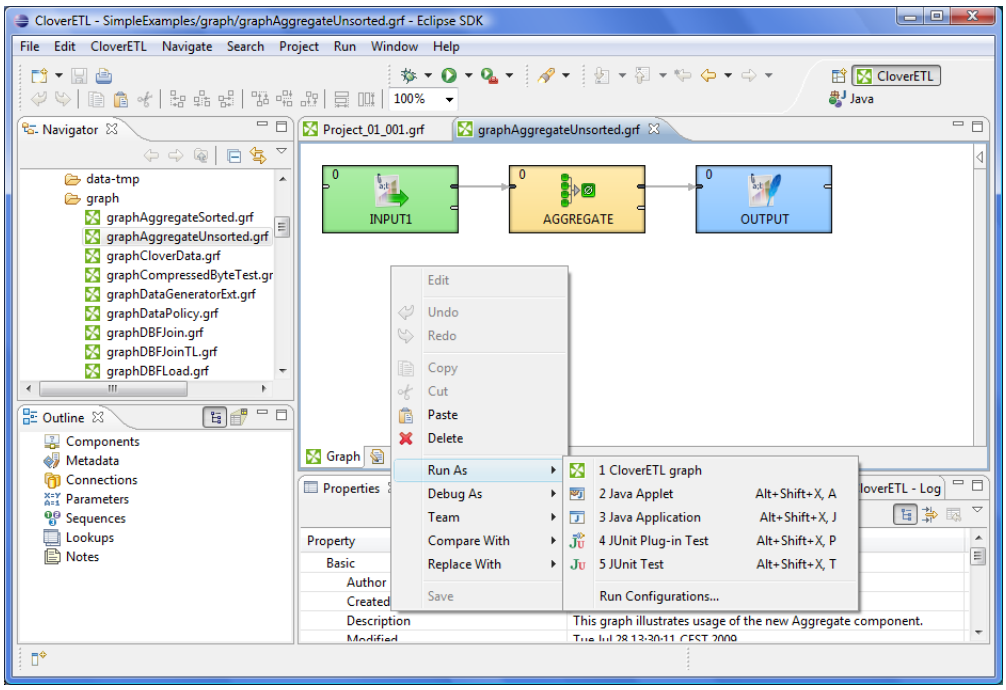


Figure 11.2. Running a Graph from the Context Menu

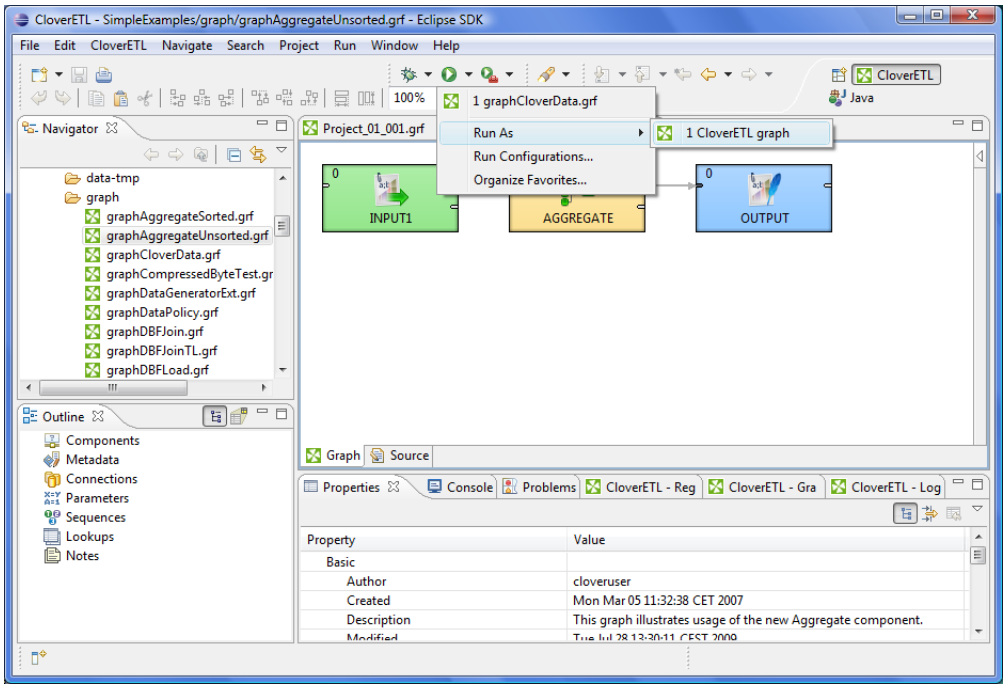


Figure 11.3. Running a Graph from the Upper Tool Bar

Successful Graph Execution

After running any graph, the process of the graph execution can be seen in the **Console** and the other tabs. (See Tabs Pane (p. 35) for detailed information.)

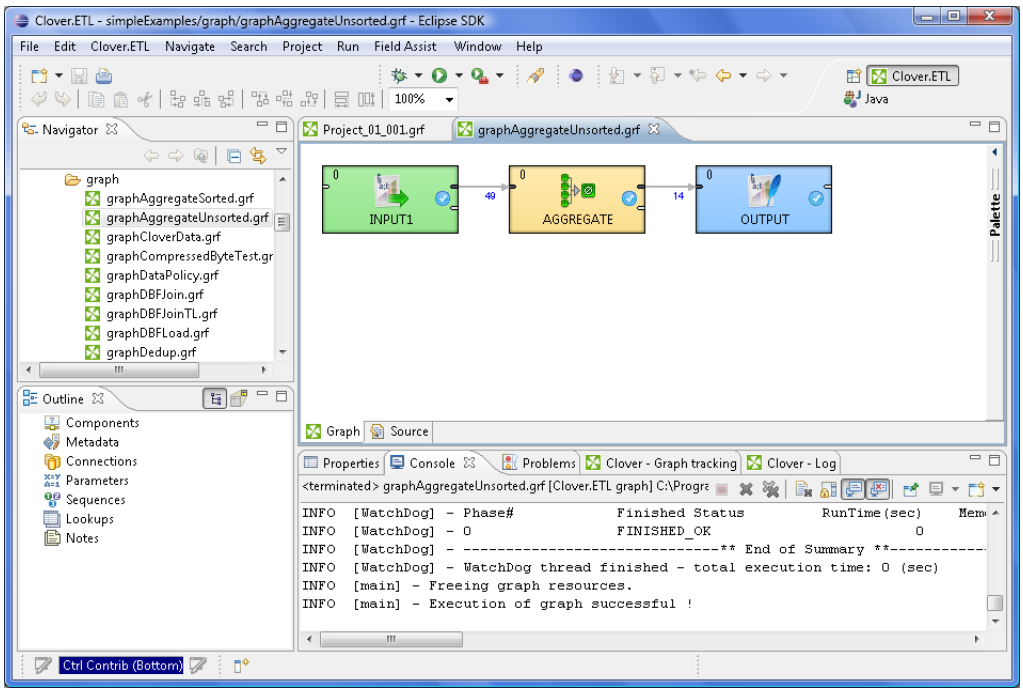


Figure 11.4. Successful Graph Execution

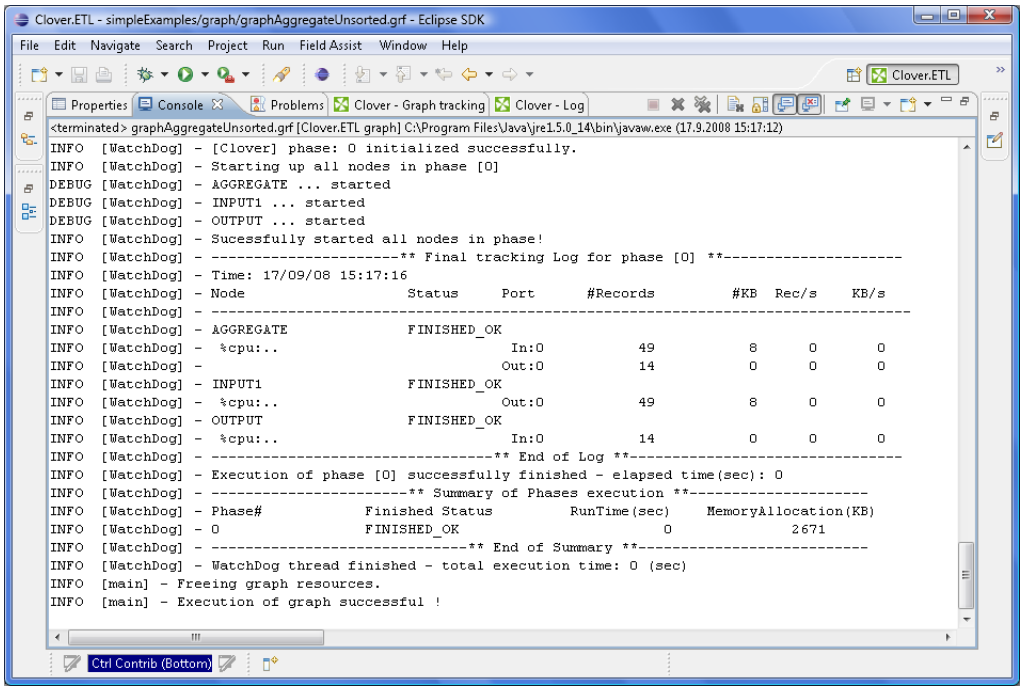


Figure 11.5. Console Tab with an Overview of the Graph Execution

And, below the edges, counts of processed data should appear:

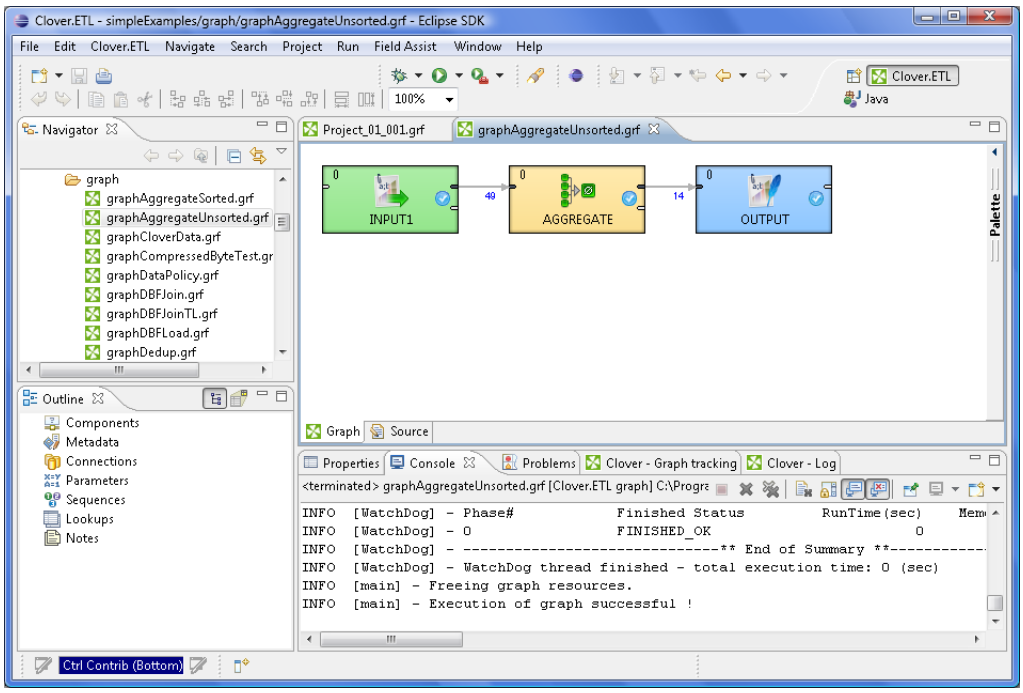


Figure 11.6. Counting Parsed Data

Using the Run Configurations Dialog

In addition to the options mentioned above, you can also open the **Run Configurations** dialog, fill in the project name, the graph name and set up program and vm arguments, parameters, etc. and click the **Run** button.

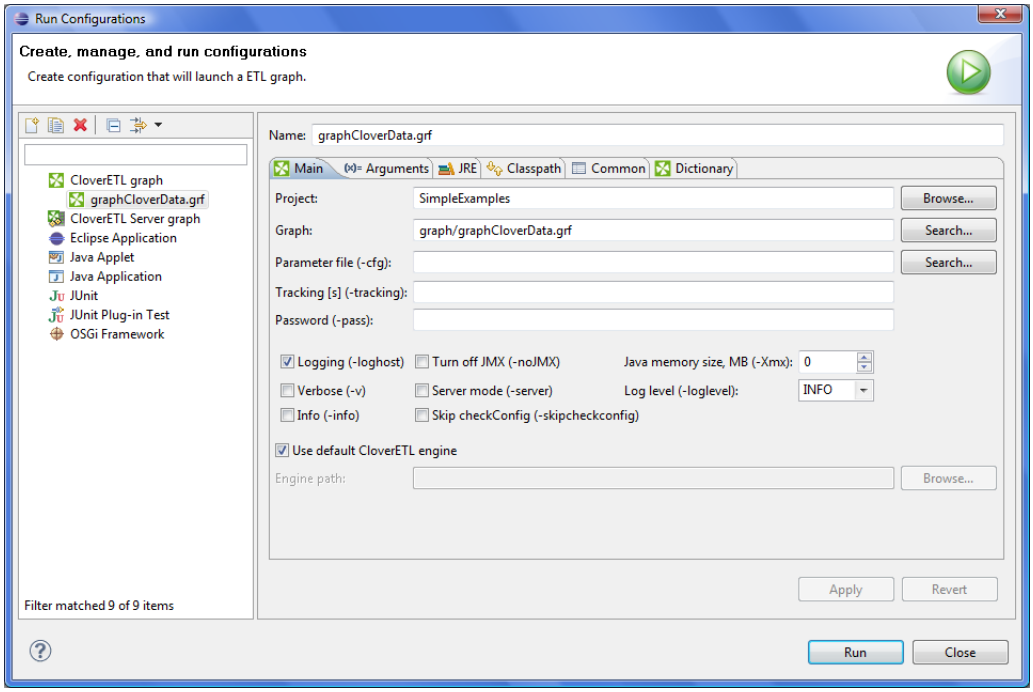


Figure 11.7. Run Configurations Dialog

More details about using the **Run Configurations** dialog can be found in Chapter 16, Advanced Topics (p. 73).

Part IV. Working with CloverETL Designer

Chapter 12. Using Cheat Sheets

Cheat Sheets are interactive tools for learning to use **CloverETL Designer** or other applications and for performing prepared tasks and creating newer ones.

CloverETL Designer includes two cheat sheets. You can view them by selecting **Help →Cheat Sheets...** from the main menu.

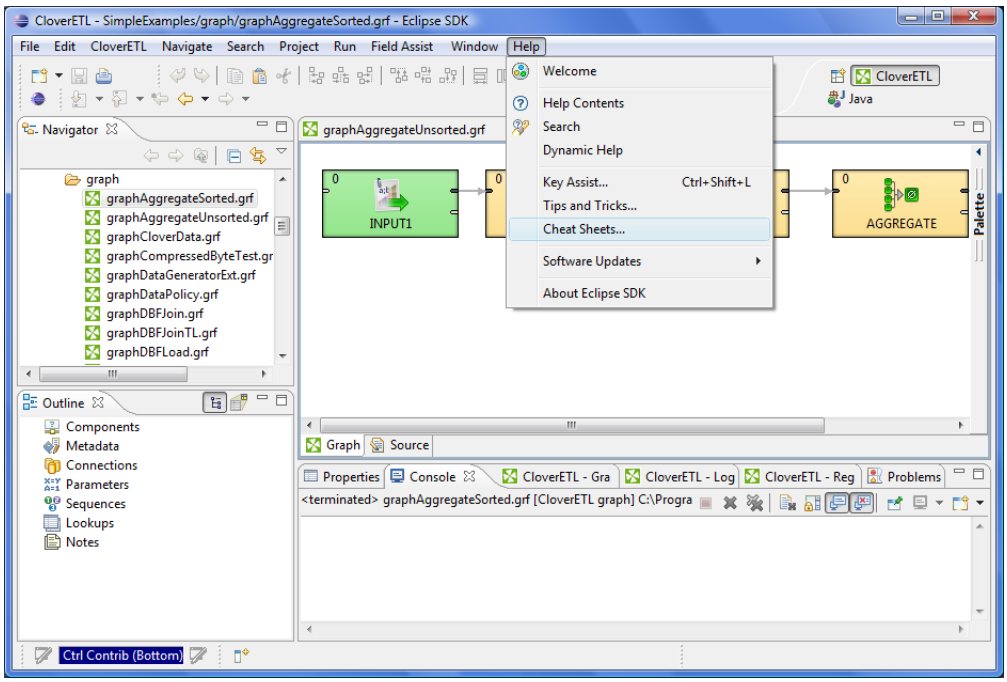


Figure 12.1. Selecting Cheat Sheets

After that, the following window will open:

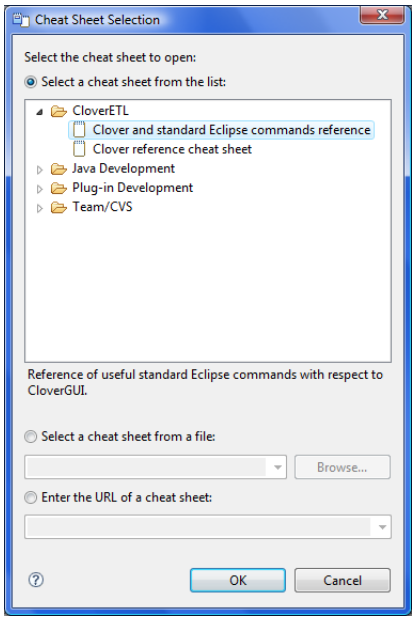


Figure 12.2. The Cheat Sheet Selection Wizard

The two cheat sheets of **CloverETL Designer** are as follows:

- Clover and standard Eclipse commands reference
- Clover reference cheat sheet

The first of them describes and teaches useful standard **Eclipse** commands with respect to **CloverETL Designer**.

The second one presents a guide of how **CloverETL Designer** cheat sheets can be written.

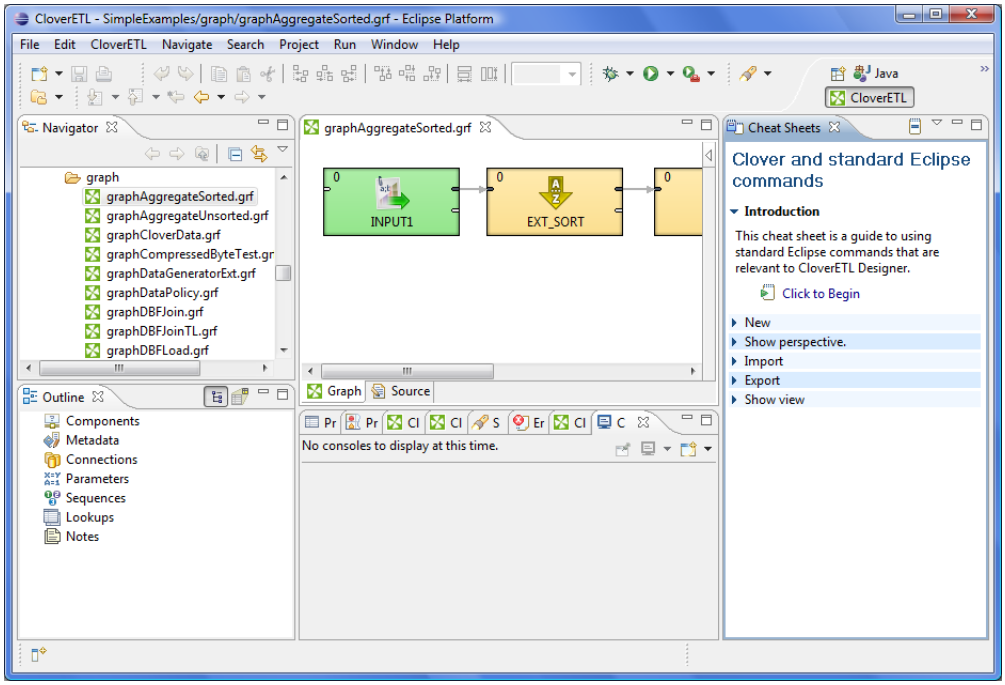


Figure 12.3. CloverETL and Standard Eclipse Commands (Collapsed)

When you expand the items, you can see an overview of **CloverETL** and/or **Eclipse** commands:

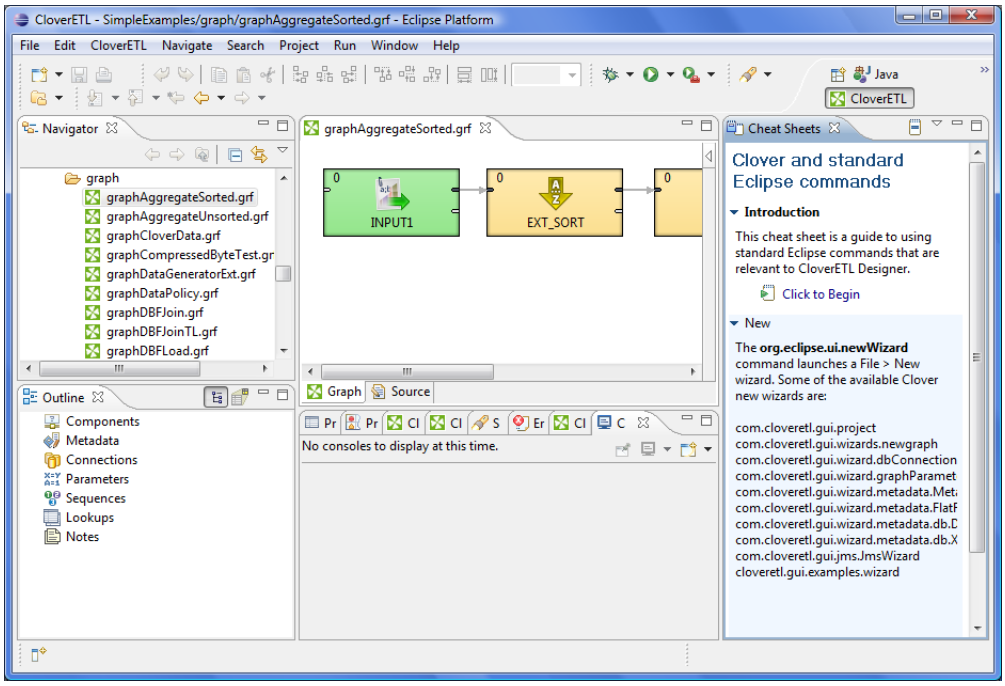


Figure 12.4. CloverETL and Standard Eclipse Commands (Expanded)

The second cheat sheet can teach you how to work with **CloverETL Designer**. You can follow the instructions and create a new project, a new graph, etc.:

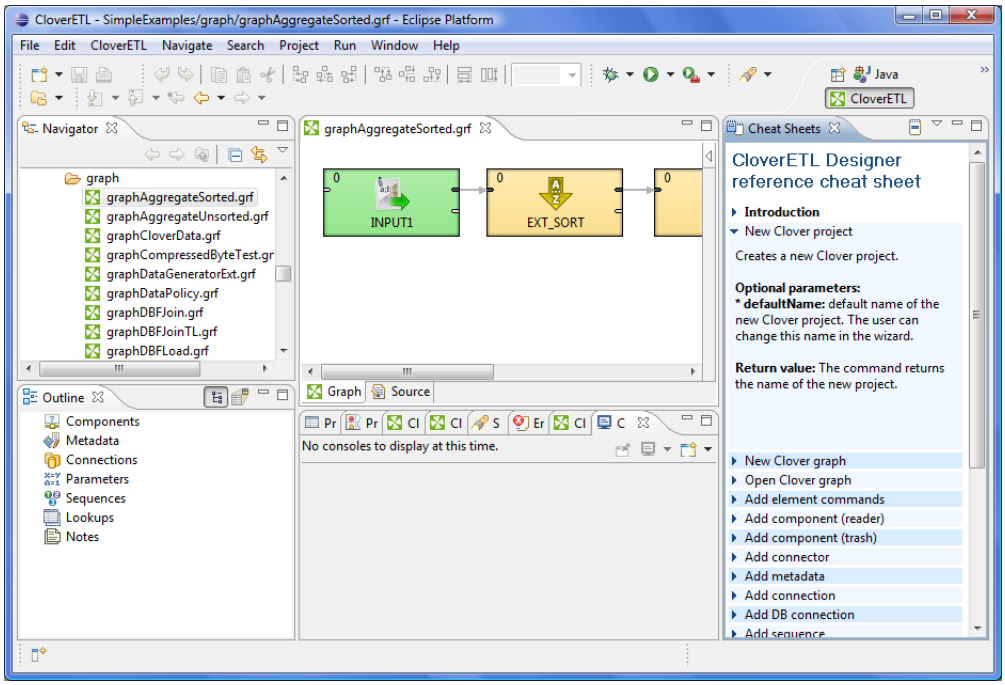


Figure 12.5. CloverETL Designer Reference Cheat Sheet

The basis of these two mentioned cheat sheets are two `.xml` files. They are included in the `com.cloveretl.gui.cheatsheet_2.9.0.jar` file located in the `plugins` directory of **CloverETL Designer** and situated within the `cheatsheets` subdirectory of this `.jar` file. They are `EclipseReference.xml` and `Reference.xml` files.

In addition to the prepared cheat sheets, you can write another cheat sheet of your own using these two files as a pattern. Once you have written your custom cheat sheet, you only need to switch on the **Select a cheat sheet from a file** radio button. Then you can browse your disk and locate the custom cheat sheet `.xml` file. After clicking **OK**, your cheat sheet will appear in the same way as any of the two prepared cheat sheets on the right side of **CloverETL Designer** window.

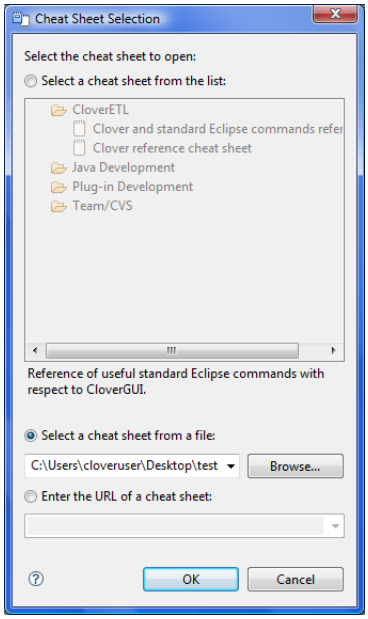


Figure 12.6. Locating a Custom Cheat Sheet

Chapter 13. Common Dialogs

Here we provide the list of the most common dialogs:

- URL File Dialog (p. 59)
- Edit Value Dialog (p. 60)
- Open Type Dialog (p. 61)

URL File Dialog

In most of the components you must also specify URL of some files. These files can serve to locate the sources of data that should be read, the sources to which data should be written or the files that must be used to transform data flowing through a component and some other file URL. To specify such a file URL, you can use the **URL File Dialog**.

When you open the **URL File Dialog**, you can see tabs on it.

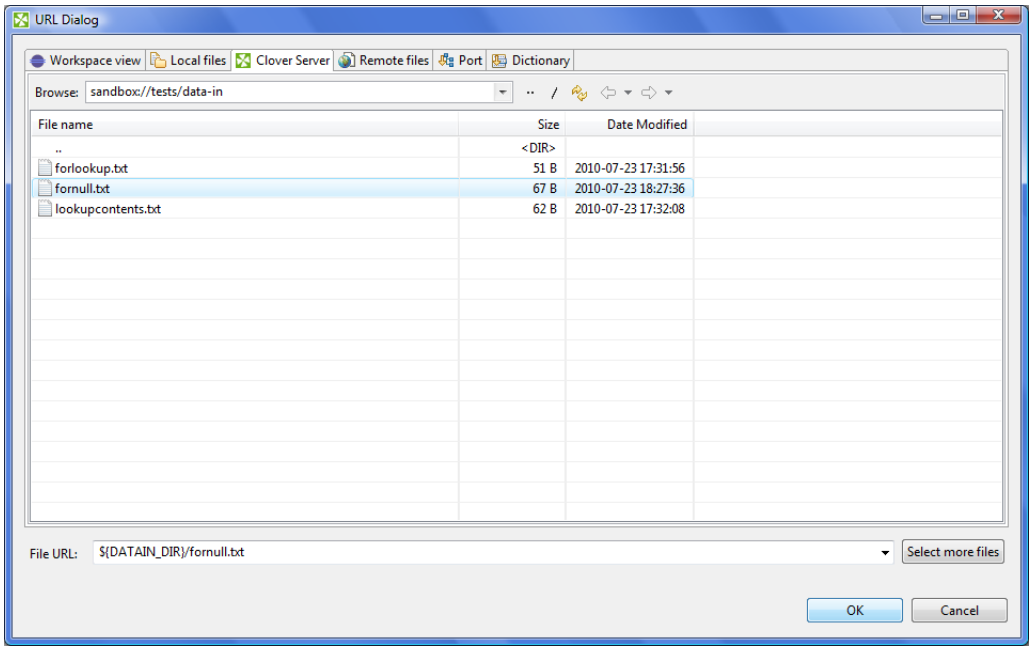


Figure 13.1. URL File Dialog

- **Workspace view**

  Serves to locate files in a workspace of local **CloverETL** project.

- **Local files**

  Serves to locate files on localhost. Combo contains disks and parameters. Can be used to specify both **CloverETL projects** and any other local files.

- **Clover Server**

  Serves to locate files of all opened **CloverETL Server projects**. Available only for **CloverETL Server** projects.

- **Remote files**

Serves to locate files on a remote computer or on the Internet. You can specify properties of connection, proxy settings, and http properties.

- **Port**

  Serves to specify fields and processing type for port reading or writing. Opens only in those component that allow such data source or target.

- **Dictionary**

  Serves to specify dictionary key value and processing type for dictionary reading or writing. Opens only in those component that allow such data source or target.

Important

To ensure graph portability, forward slashes are used for defining the path in URLs (even on Microsoft Windows).

More detailed information of **URL**s for each of the tabs described above is provided in sections

- Supported File URL Formats for Readers (p. 251)
- Supported File URL Formats for Writers (p. 263)

Edit Value Dialog

The **Edit Value** dialog contains a simple text area where you can write the transformation code in **JMSReader**, **JMSWriter** and **JavaExecute** components.

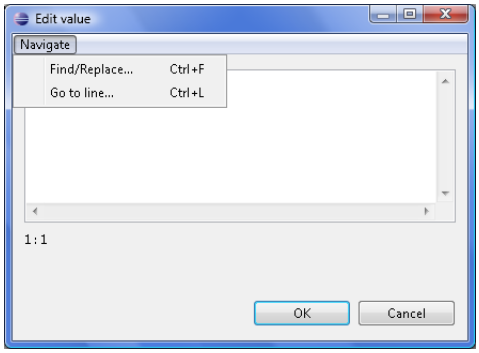


Figure 13.2. Edit Value Dialog

When you click the **Navigate** button at the upper left corner, you will be presented with the list of possible options. You can select either **Find** or **Go to line**.

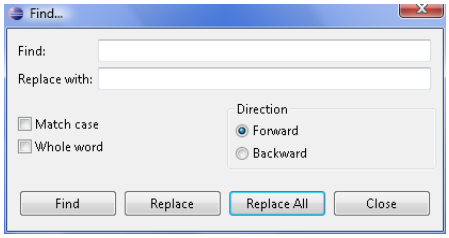


Figure 13.3. Find Wizard

If you click the **Find** item, you will be presented with another wizard. In it you can type the expression you want to find (**Find** text area), decide whether you want to find the whole word only (**Whole word**), whether the cases

should match or not (**Match case**), and the **Direction** in which the word will be searched - downwards (**Forward**) or upwards (**Backward**). These options must be selected by checking the presented checkboxes or radio buttons.

If you click the **Go to line** item, a new wizard opens in which you must type the number of the line you want to go to.

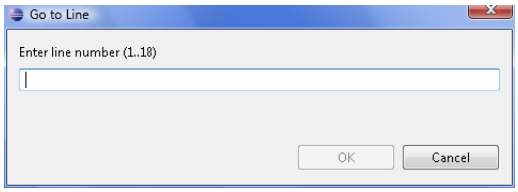


Figure 13.4. Go to Line Wizard

Open Type Dialog

This dialog serves to select some class (**Transform class, Denormalize class, etc.**) that defines the desired transformation. When you open it, you only need to type the beginning of the class name. By typing the beginning, the classes satisfying to the written letters appear in this wizard and you can select the right one.

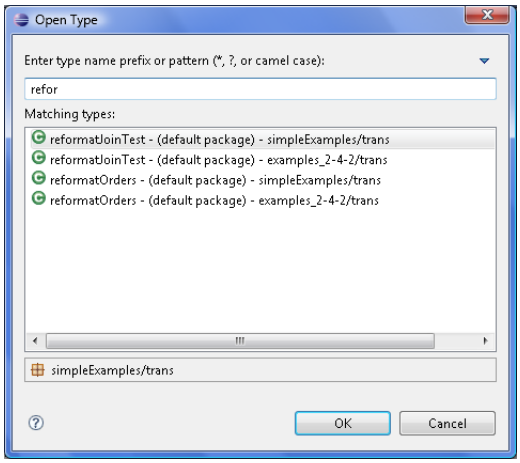


Figure 13.5. Open Type Dialog

Chapter 14. Import

CloverETL Designer allows you to import already prepared **CloverETL** projects, graphs and/or metadata. If you want to import something, select **File →Import...** from the main menu.

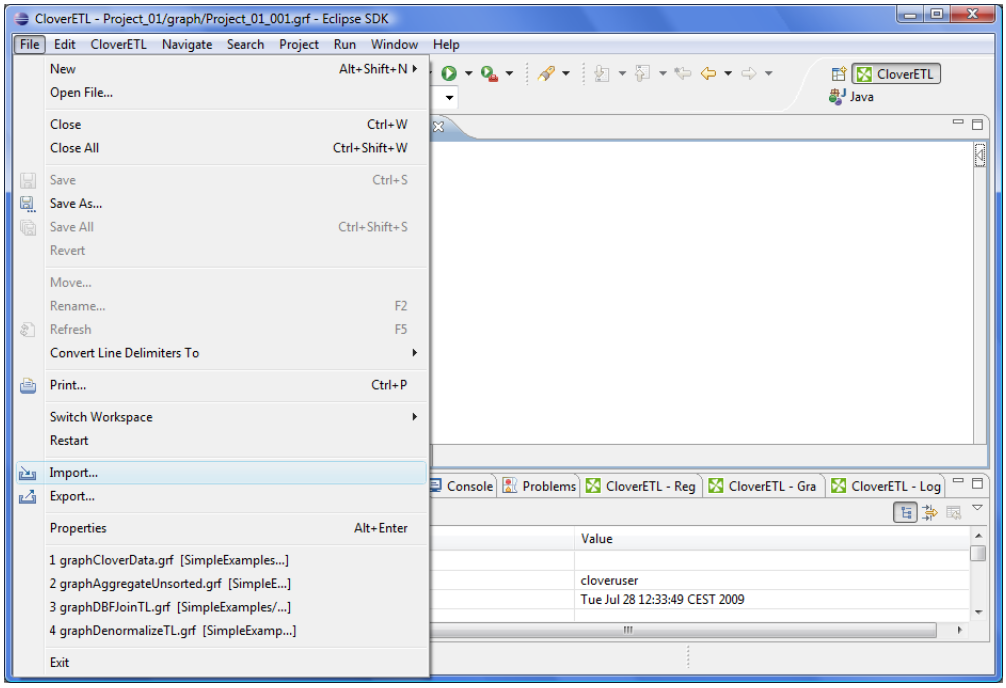


Figure 14.1. Import (Main Menu)

Or right-click in the **Navigator** pane and select **Item...** from the context menu.

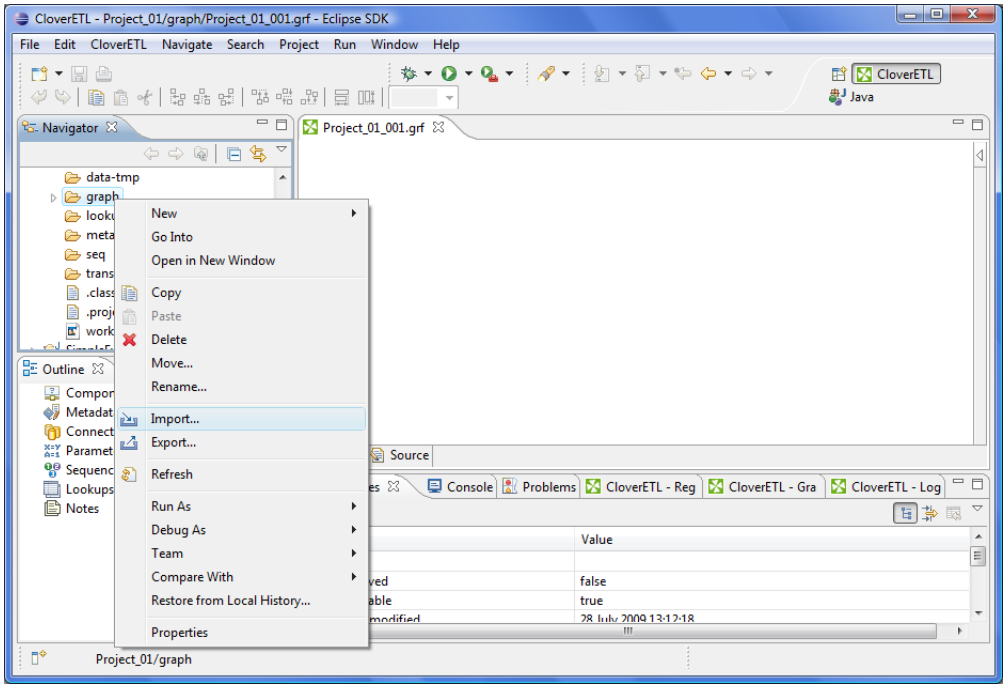


Figure 14.2. Import (Context Menu)

After that, the following window opens. When you expand the **Clover ETL** category, the window will look like this:

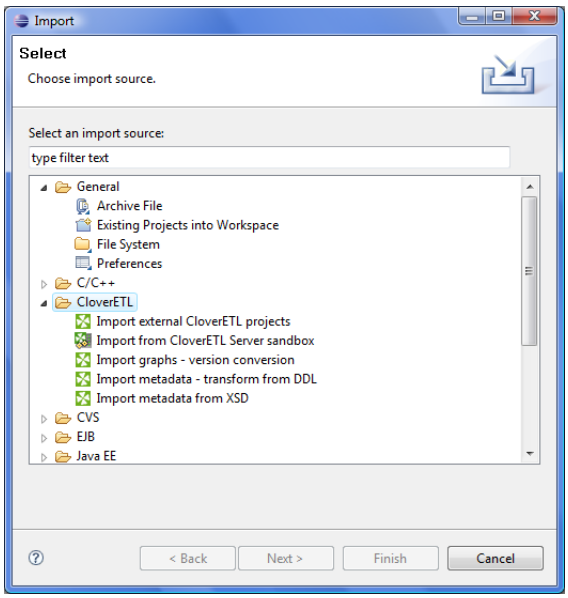


Figure 14.3. Import Options

Import CloverETL Projects

If you select the **Import external CloverETL projects** item, you can click the **Next** button and you will see the following window:

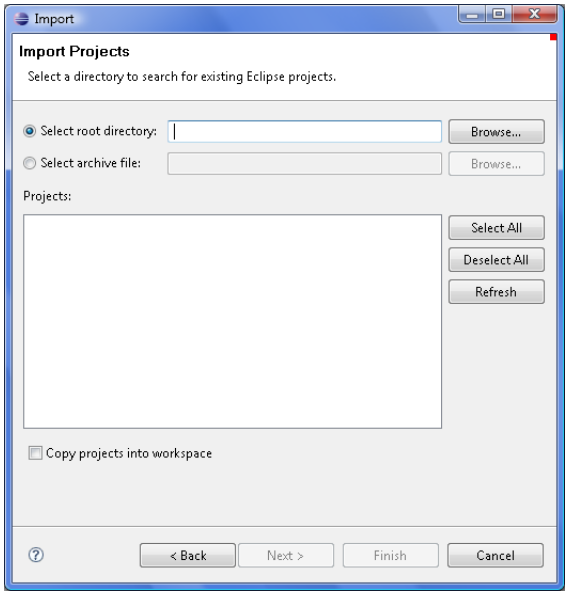


Figure 14.4. Import Projects

You can find some directory or compressed archive file (the right option must be selected by switching the radio buttons). If you locate the directory, you can also decide whether you want to copy or link the project to your workspace. If you want the project be linked only, you can leave the **Copy projects into workspace** checkbox unchecked. Otherwise, it will be copied. Linked projects are contained in more workspaces. If you select some archive file, the list of projects contained in the archive will appear in the **Projects** area. You can select some or all of them by checking the checkboxes that appear along with them.

Import from CloverETL Server Sandbox

CloverETL Designer now allows you to import any part of **CloverETL Server** sandboxes. To import, select the **Import from CloverETL Server Sandbox** option. After that, the following wizard will open:

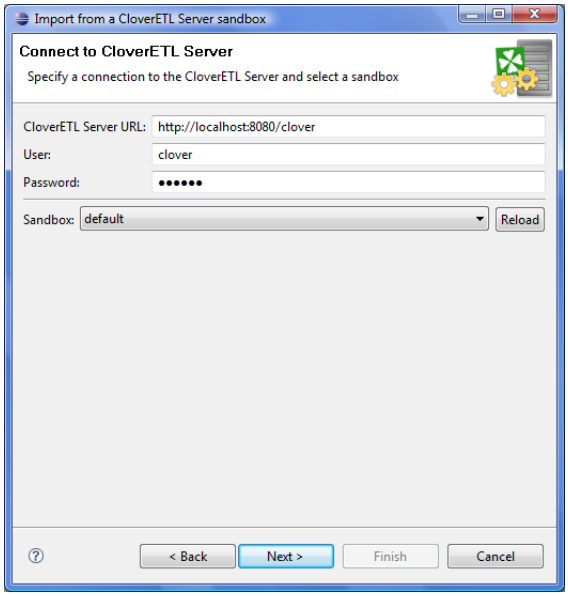


Figure 14.5. Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server)

Specify the following three items: **CloverETL Server URL**, your **username** and **password**. Then click **Reload**. After that, a list of sandboxes will be available in the **Sandbox** menu. Select any of them and click **Next**. A new wizard will open:

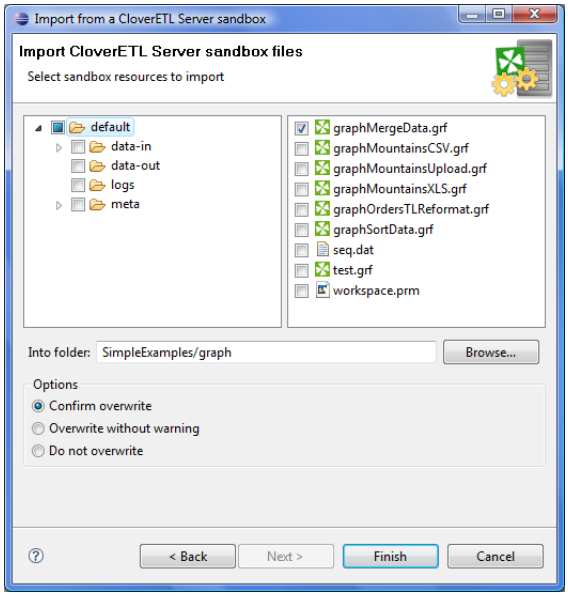


Figure 14.6. Import from CloverETL Server Sandbox Wizard (List of Files)

Select the files and/or directories that should be imported, select the folder into which they should be imported and decide whether the files and/or directories with identical names should be overwritten without warning or whether overwriting should be confirmed or whether the files and/or directories with identical names should not be overwritten at all. Then click **Finish**. Selected files and/or directories will be imported from **CloverETL Server** sandbox.

Import Graphs

If you select the **Import graphs - version conversion** item, you can click the **Next** button and you will see the following window:

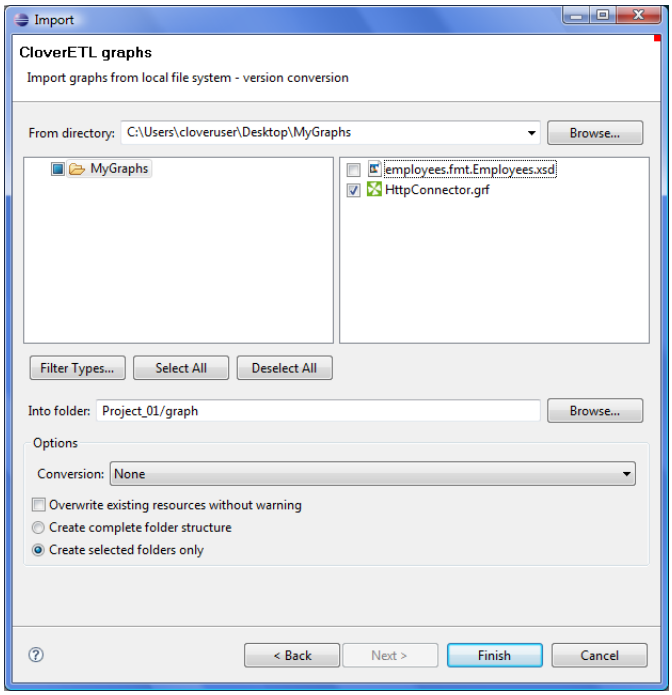


Figure 14.7. Import Graphs

You must select the right graph(s) and specify from which directory into which folder the selected graph(s) should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite the existing sources without warning.

Note

You can also convert older graphs from 1.x.x to 2.x.x version of **CloverETL Designer** and from 2.x.x to 2.6.x version of **CloverETL Engine**.

Import Metadata

You can also import metadata from XSD or DDL.

If you want to know what metadata is and how it can be created, see Chapter 19, Metadata (p. 96) for more information.

Metadata from XSD

If you select the **Import metadata from XSD** item, you can click the **Next** button and you will see the following window:

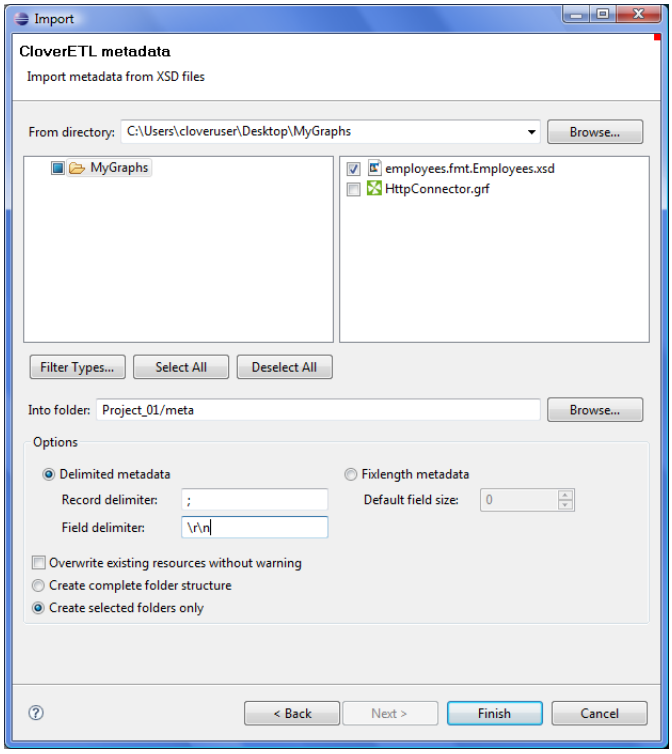


Figure 14.8. Import Metadata from XSD

You must select the right metadata and specify from which directory into which folder the selected metadata should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite existing sources without warning. You can specify the delimiters or default field size.

Metadata from DDL

If you select the **Import metadata - transform from DDL** item, you can click the **Next** button and you will see the following window:

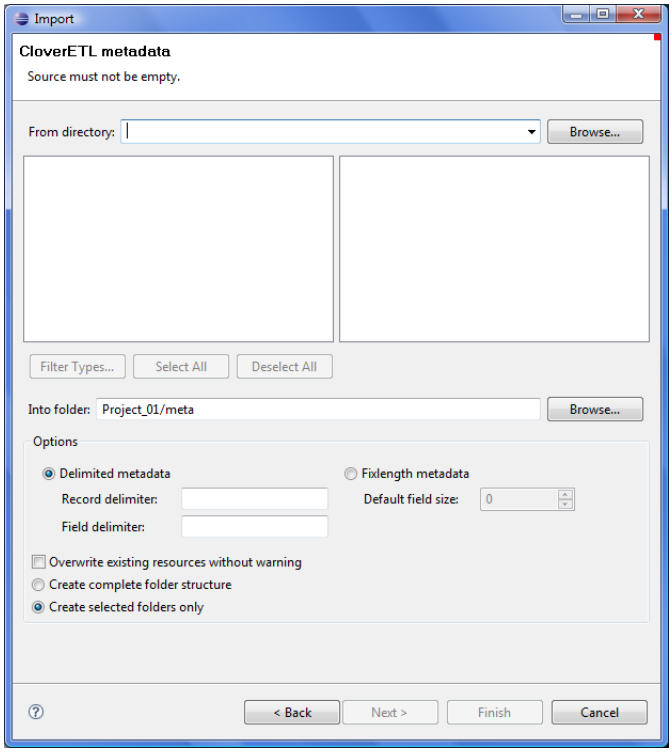


Figure 14.9. Import Metadata from DDL

You must select the right metadata and specify from which directory into which folder the selected metadata should be copied. By switching the radio buttons, you decide whether complete folder structure or only selected folders should be created. You can also order to overwrite existing sources without warning. You need to specify the delimiters.

Chapter 15. Export

CloverETL Designer allows you to export your own **CloverETL** graphs and/or metadata. If you want to export something, select **File** →**Export...** from the main menu. Or right-click in the **Navigator** pane and select **Item...** from the context menu. After that, the following window opens. When you expand the **CloverETL** category, the window will look like this:

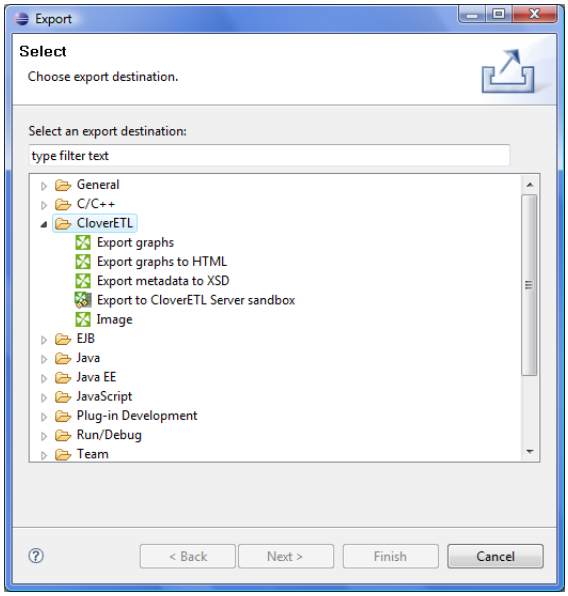


Figure 15.1. Export Options

Export Graphs

If you select the **Export graphs** item, you can click the **Next** button and you will see the following window:

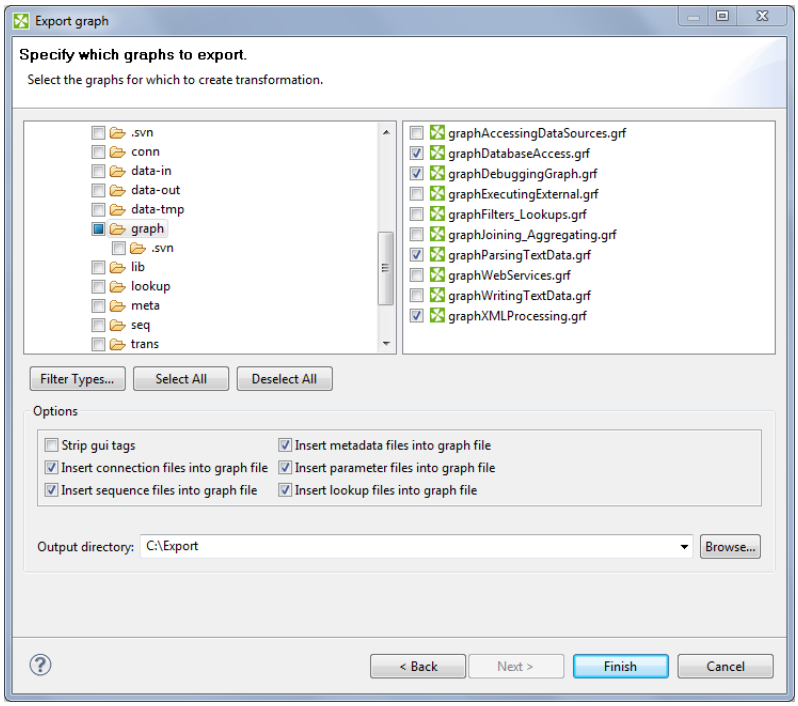


Figure 15.2. Export Graphs

Check the graph(s) to be exported in the right-hand pane. You have to locate the output directory as well. In addition to that, you can select whether external (shared) metadata, connections, parameters, sequences and lookups should be internalized and inserted into graph(s). This has to be done by checking corresponding checkboxes. You can also remove gui tags from the output file by checking the **Strip gui tags** checkbox.

Export Graphs to HTML

If you select the **Export graphs to HTML** item, you can click the **Next** button and you will see the following window:

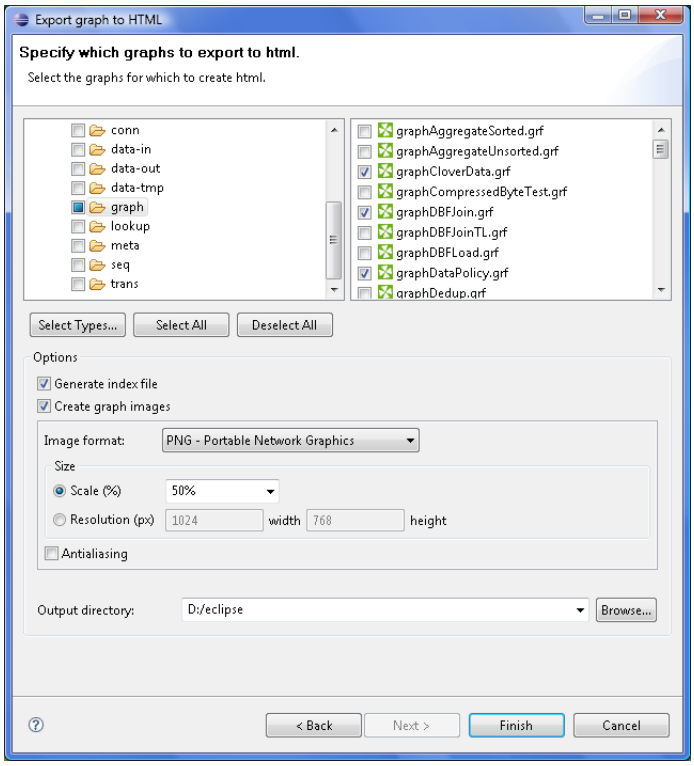


Figure 15.3. Export Graphs to HTML

You must select the graph(s) and specify to which output directory the selected graph(s) should be exported. You can also generate index file of the exported pages and corresponding graphs and/or images of the selected graphs. By switching the radio buttons, you are selecting either the scale of the output images, or the width and the height of the images. You can decide whether antialiasing should be used.

Export Metadata to XSD

If you select the **Export metadata to XSD** item, you can click the **Next** button and you will see the following window:

If you want to know what metadata are and how they can be created, see Chapter 19, Metadata (p. 96) for more information.

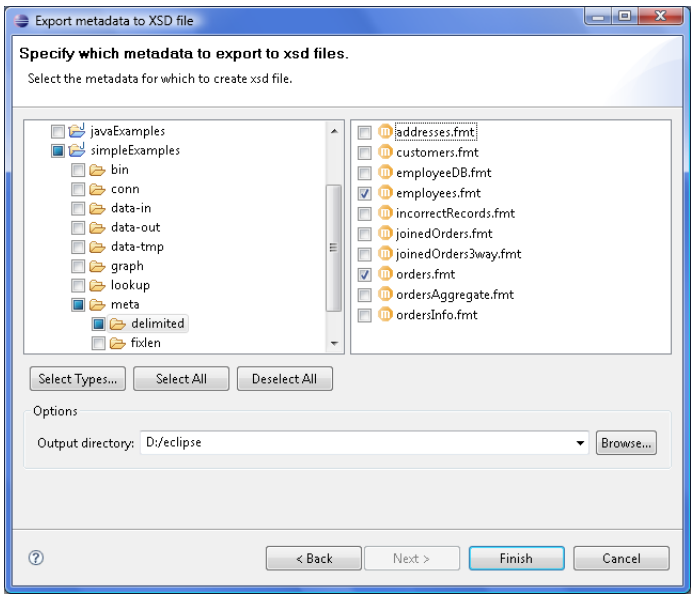


Figure 15.4. Export metadata to XSD

You must select the metadata and specify to which output directory the selected metadata should be exported.

Export to CloverETL Server Sandbox

CloverETL Designer now allows you to export any part of your projects to **CloverETL Server** sanboxes. To export, select the **Export to CloverETL Server sandbox** option. After that, the following wizard will open:

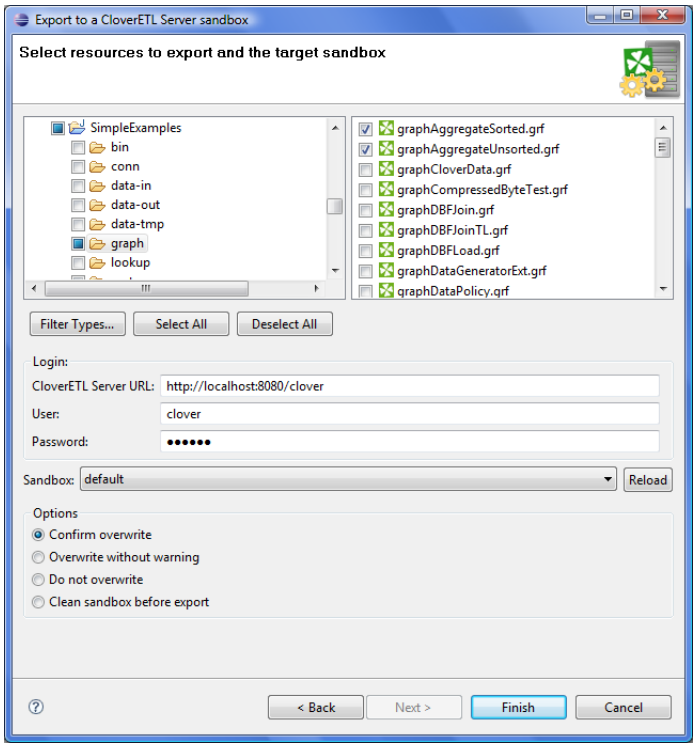


Figure 15.5. Export to CloverETL Server Sandbox

Select the files and/or directories that should be exported and decide whether the files and/directories with identical names should be overwritten without warning or whether overwriting should be confirmed or whether the files and/or directories with identical names should not be overwritten at all and also decide whether sandbox should be cleaned before export.

Specify the following three items: **CloverETL Server URL**, your **username** and **password**. Then click **Reload**. After that, a list of sandboxes will be available in the **Sandbox** menu.

Select a sandbox. Then click **Finish**. Selected files and/or directories will be exported to the selected **CloverETL Server** sandbox.

Export Image

If you select the **Export image** item, you can click the **Next** button and you will see the following window:

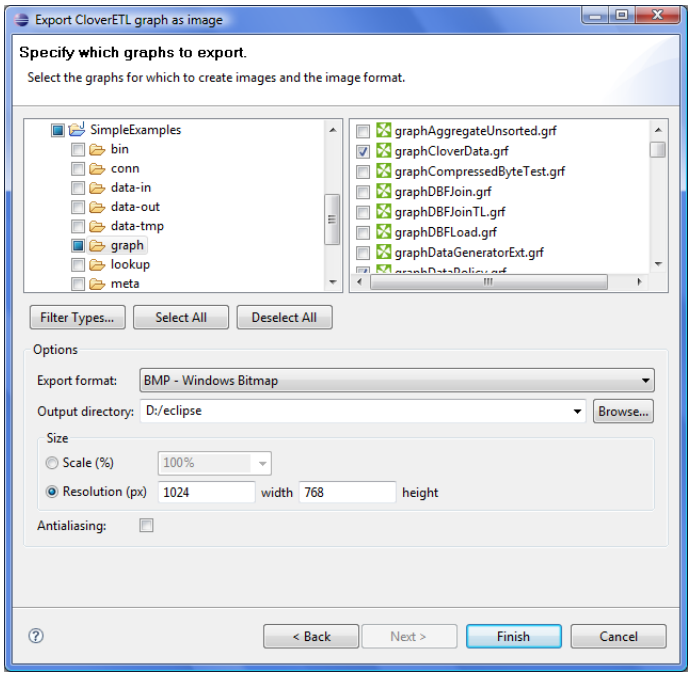


Figure 15.6. Export Image

This option allows you to export images of the selected graphs only. You must select the graph(s) and specify to which output directory the selected graph(s) images should be exported. You can also specify the format of output files - `bmp`, `jpeg` or `png`. By switching the radio buttons, you are selecting either the scale of the output images, or the width and the height of the images. You can decide whether antialiasing should be used.

Chapter 16. Advanced Topics

As was already described in Using the Run Configurations Dialog (p. 54), you know that you can use the **Run Configurations** dialog to set more advanced options in order to run the graphs.

You can set there:

- Program and VM Arguments (p. 73)
- Changing Default CloverETL Settings (p. 76)

You can also change the size of displayed numbers of processed records:

- Enlarging the Font of Displayed Numbers (p. 78)

Another advanced option is using Java. In the following sections you can see how JRE and/or JDK can be set:

- Setting and Configuring Java (p. 79)
  - Setting Java Runtime Environment (p. 79)
  - Installing Java Development Kit (p. 81)

Program and VM Arguments

If you want to specify some arguments during run of the graph, select **Run Configurations** from the context menu and set up some options in the **Main** tab.

You can enter the following three **Program arguments** in the tab:

- **Program file** (`-cfg <filename>`)

  In the specified file, more parameter values can be defined. Instead of a sequence of expressions like the following: `-P:parameter1=value1 -P:parameter2=value2 ... P:parameterN=valueN` (see below), you can define these values of these parameters in a single file. Each row should define one parameter as follows: `parameterK=valueK`. Then you only need to specify the name of the file in the expression above.

  - **Priorities**

    The parameters specified in this tab have higher priority than the parameters specified in the **Arguments** tab, those linked to the graph (external parameters) or specified in the graph itself (internal parameters) and they can also overwrite any environment variable. Remember also that if you specify any parameter twice in the file, only the last one will be applied.

- **Tracking [s]** (`-tracking <filename>`)

  Sets the frequency of printing the graph processing status.

- **Password** (`-pass`)

  Enters a password for decrypting the encrypted connection(s). Must be identical for all linked connections.

- **Checkconfig** (`-checkconfig`)

  Only checks the graph configuration without running the graph.

You can also check some checkboxes that define the following **Program arguments**:

- **Logging** (`-loghost`)

Defines host and port for socket appender of `log4j`. The `log4j` library is required. For example, `localhost:4445`.

You can specify the port when selecting **Windows** →**Preferences**, choosing the **Logging** item within the **CloverETL** category and setting the port.

- **Verbose** (`-v`)

  Switches on verbose mode of running the graph.

- **Info** (`-info`)

  Prints out the information about Clover library version.

- **Turn off JMX** (`-noJMX`)

  Turns off sending tracking information through JMX bean, which can make the performance better.

- **Log level** (`-loglevel <option>`)

  Defines one of the following: `ALL | TRACE | DEBUG | INFO | WARN | ERROR | FATAL | OFF`.

  Default **Log level** is `INFO` for **CloverETL Designer**, but `DEBUG` for **CloverETL Engine**.

- **Skip checkConfig** (`-skipcheckconfig`)

  Skips checking the graph configuration before running the graph.

Two checkboxes define **VM arguments**:

- **Server mode** (`-server`)

  The client system (default) is optimal for applications which need fast start-up times or small footprints. Switching to server mode is advantageous to long-running applications, for which reaching the maximum program execution speed is generally more important than having the fastest possible start-up time. To run the server system, Java Development Kit (JDK) needs to be downloaded.

- **Java memory size, MB** (`-Xmx`)

  Specifies the maximum size of the memory allocation pool (memory available during the graph run). Default value of Java heap space is 68MB.

All of these arguments can also be specified using the expressions in the parentheses shown above when typing them in the **Program arguments** pane or **VM arguments** of the **Arguments** tab.

In addition to the arguments mentioned above, you can also switch to the **Arguments** tab and type in the **Program arguments** pane:

- `-P:<parameter>=<value>`

  Specifies the `value` of a `parameter`. White spaces must not be used in this expression.

  Priorities

  - More of these expressions can be used for the graph run. They have higher priority than the parameters linked to the graph (external parameters), those specified in the graph itself (internal parameters) and they can also overwrite any environment variable. However, they have less priority than the same parameters specified in the **Main**tab. Remember also that if you specify any parameter twice in the **Arguments** tab, only the last one will be applied.

- `-config <filename>`

Loads the default **CloverETL Engine** properties from the specified file. Overwrites the same properties definitions contained in the `defaultProperties` file. The name of the file can be selected arbitrarily and the file can only redefine selected default properties.

- `-logcfg <filename>`

  Loads `log4j` properties from the specified file. If not specified, `log4j.properties` should be in the classpath.

Example of Setting Up Memory Size

In the **Run Configurations** dialog, you can set the Java memory size in Megabytes. It is important to define some memory size because Java Virtual Machine needs this memory capacity to run the graphs. You must define maximum memory size for JVM by selecting the proper value:

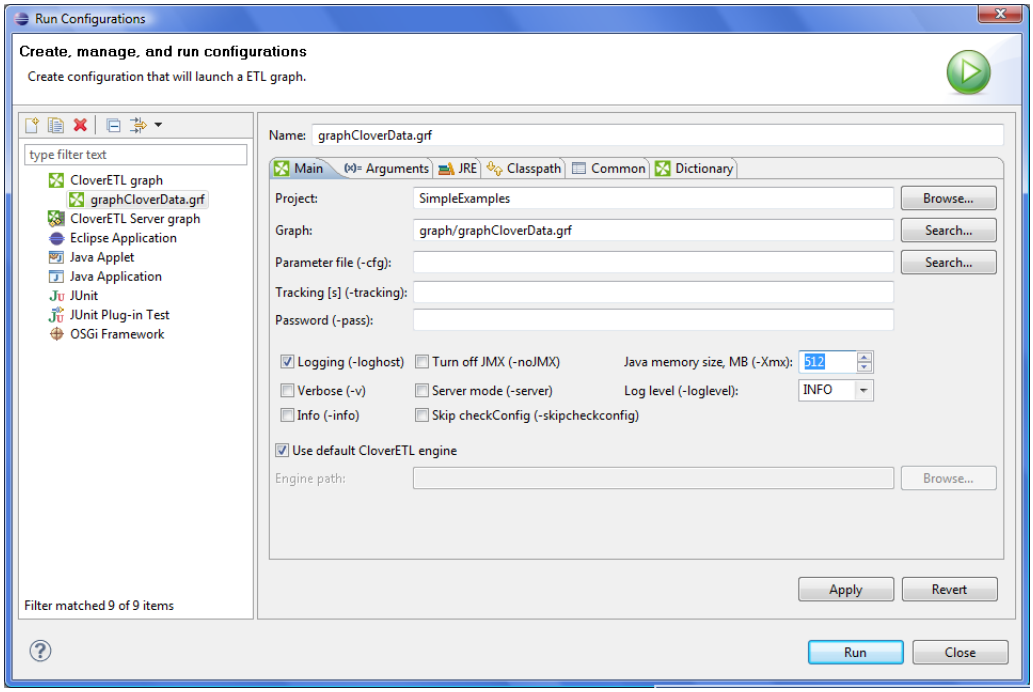


Figure 16.1. Setting Up Memory Size

Changing Default CloverETL Settings

CloverETL internal settings (defaults) are stored in `defaultProperties` file located in the **CloverETL** engine (`plugins/com.cloveretl.gui/lib/lib/cloveretl.engine.jar`) in its `org/jetel/data` subfolder. This source file contains various parameters that are loaded at run-time and used during the transformation execution.

If you modify the values right in the `defaultProperties` file, such change will be applied for all graph runs.

To change the values just for the current graph(s), create a local file with only those properties you need to override. Place the file in the project directory. To instruct **CloverETL** to retrieve the properties from this local file, use the `-config` switch. Go to **Run Configurations...**, to the **Arguments** tab and type the following in the **Program arguments** pane: use `-config <file_with_overriden_properties>` switch.

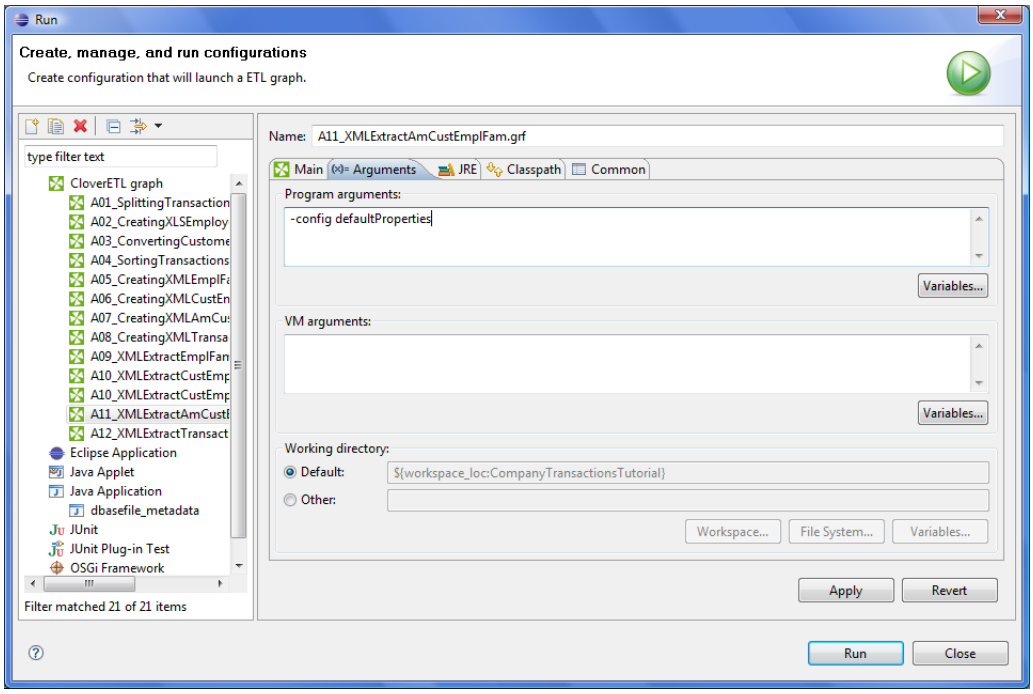


Figure 16.2. Custom Clover Settings

Here we present some of the properties and their values as they are presented in the `defaultProperties` file:

- `Record.MAX_RECORD_SIZE = 12288`

  This limits the maximum size of data record in the **CloverETL** internal binary form, e.g., strings are stored in unicode (16bits per character). If you start getting buffer overflow or similar errors, increase this value. The limit is theoretically 2^{31}, but you should keep it under 64K. Increasing this value increases memory utilization.

- `Record.DEFAULT_COMPRESSION_LEVEL=5`

  This sets the compression level for compressed data fields (`cbyte`).

- `DataParser.FIELD_BUFFER_LENGTH = 4096`

  It determines the maximum size of each data text format field for **Readers**. If you have data containing long text strings, increase this value. The impact on memory utilization is low.

- `DataFormatter.FIELD_BUFFER_LENGTH = 4096`

  The same as above but for **Writers**

- `DEFAULT_INTERNAL_IO_BUFFER_SIZE = 49152`

  It determines the internal buffer size the components allocate for I/O operations. Increasing this value affects performance negligibly. The value must be at least 2*`MAX_RECORD_SIZE`.

- `DEFAULT_DATE_FORMAT = yyyy-MM-dd`
- `DEFAULT_TIME_FORMAT = HH:mm:ss`
- `DEFAULT_DATETIME_FORMAT = yyyy-MM-dd HH:mm:ss`
- `DEFAULT_REGEXP_TRUE_STRING = true|T|TRUE|YES|Y|t|1|yes|y`
- `DEFAULT_REGEXP_FALSE_STRING = false|F|FALSE|NO|N|f|0|no|n`
- `DataParser.DEFAULT_CHARSET_DECODER = ISO-8859-1`
- `DataFormatter.DEFAULT_CHARSET_ENCODER = ISO-8859-1`
- `Data.DATA_RECORDS_BUFFER_SIZE = 10485760`

  It determines the maximum size of each buffer used for sorting.

- `Data.MAX_BUFFERS_ALLOCATED = 99`

  It determines the maximum number of buffers allocated for sorting.

- `Lookup.LOOKUP_INITIAL_CAPACITY = 512`

  The initial capacity of a lookup table when created without specifying the size.

- `DataFieldMetadata.DECIMAL_LENGTH = 8`

  It determines the default maximum precision of decimal data field metadata. Precision is the number of digits in a number, e.g., the number 123.45 has a precision of 5.

- `DataFieldMetadata.DECIMAL_SCALE = 2`

  It determines the default scale of decimal data field metadata. Scale is the number of digits to the right of the decimal point in a number, e.g., the number 123.45 has a scale of 2.

Important

Among many other properties, there is also another one that allows to define locale that should be used as the default one.

The setting is the following:

```
# DEFAULT_LOCALE = en.US
```

By default, system locale is used by **CloverETL**. If you uncomment this row you can set the `DEFAULT_LOCALE` property to any locale supported by **CloverETL**, see the List of all Locale (p. 112)

Enlarging the Font of Displayed Numbers

If you want, you can enlarge the font of the numbers that appear along edges to tell you how many records went along them. To do that, select **Window →Preferences...**

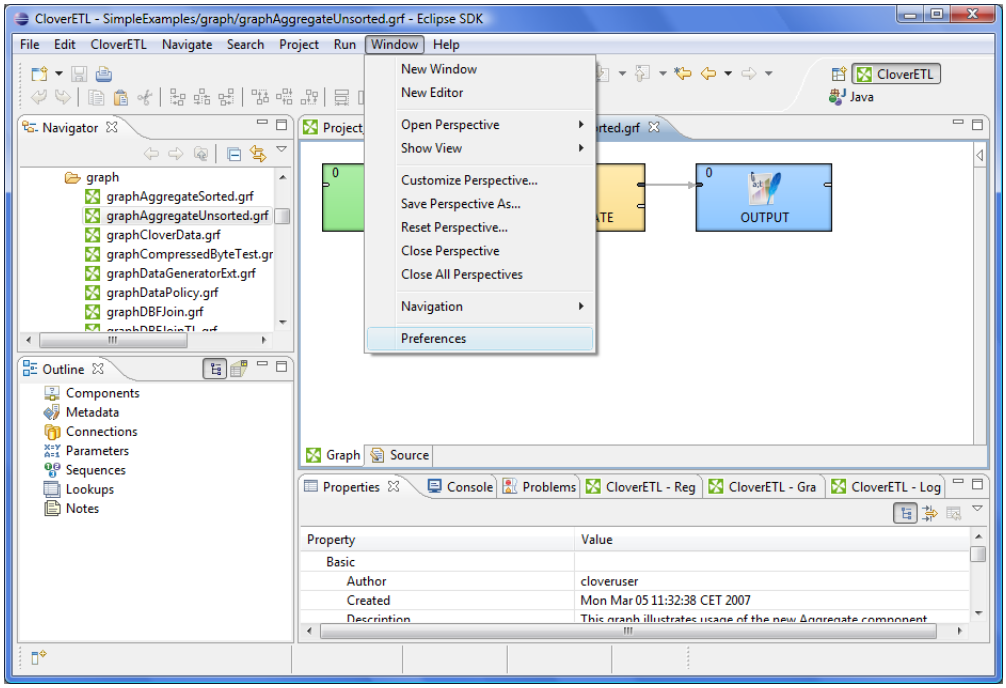


Figure 16.3. Enlarging the Font of Numbers

Then, expand the **CloverETL** item, select **Tracking** and select the desired font size in the **Record number font size** area. By default, it is set to 7.

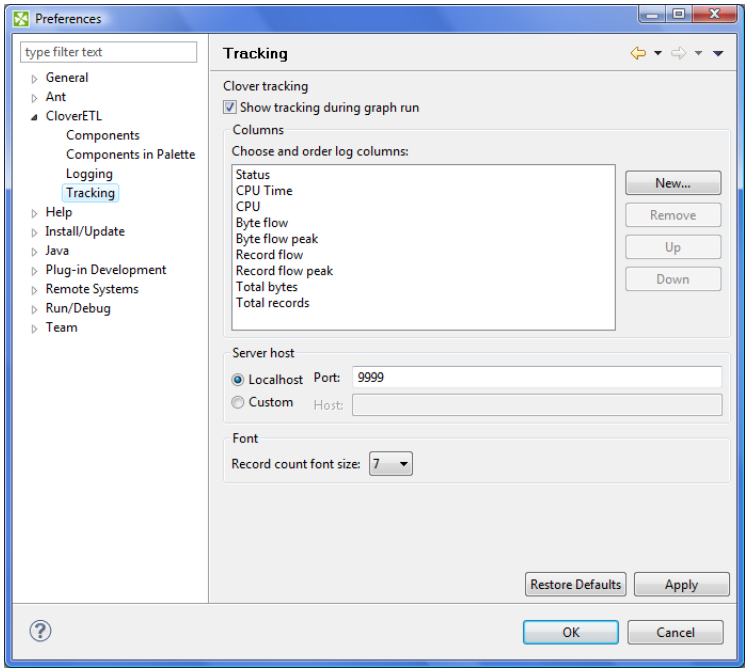


Figure 16.4. Setting the Font Size

Setting and Configuring Java

If you want to set up JRE or add JDK libraries, you can do it as shown here.

Note

Remember that you should use Java 1.6+!

- For detailed information about setting JRE see Setting Java Runtime Environment (p. 79).
- For detailed information about installing JDK see Installing Java Development Kit (p. 81).

Setting Java Runtime Environment

If you want to set the JRE, you can do it by selecting **Window →Preferences**.

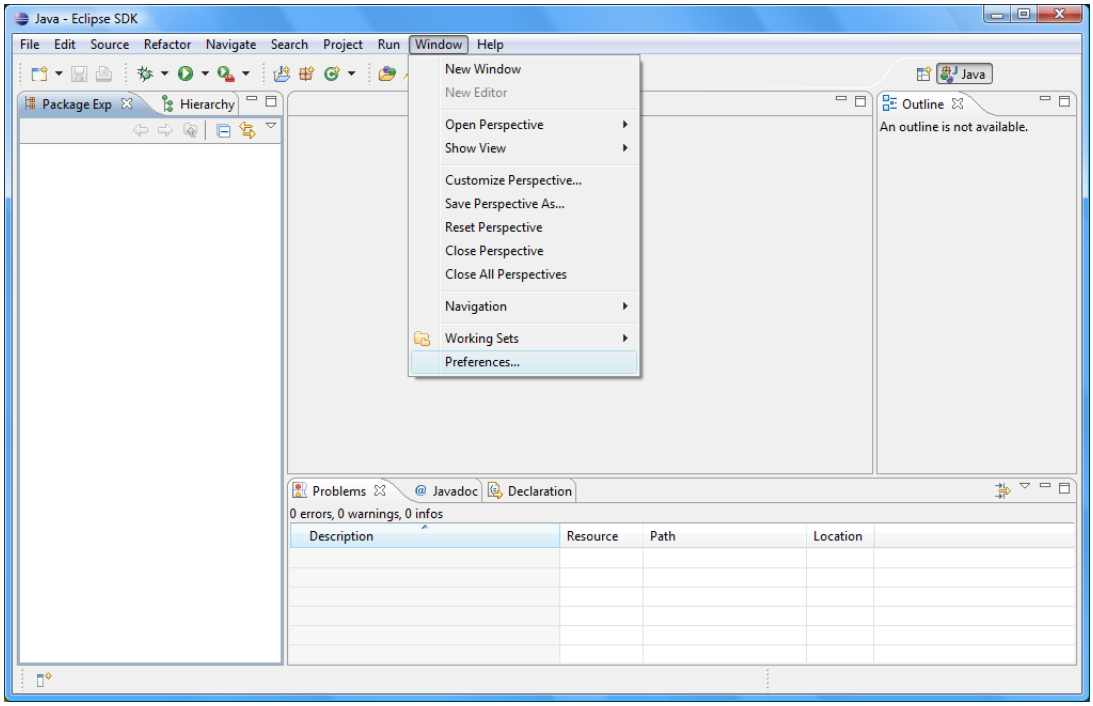


Figure 16.5. Setting The Java Runtime Environment

After clicking the option, you see the following window:

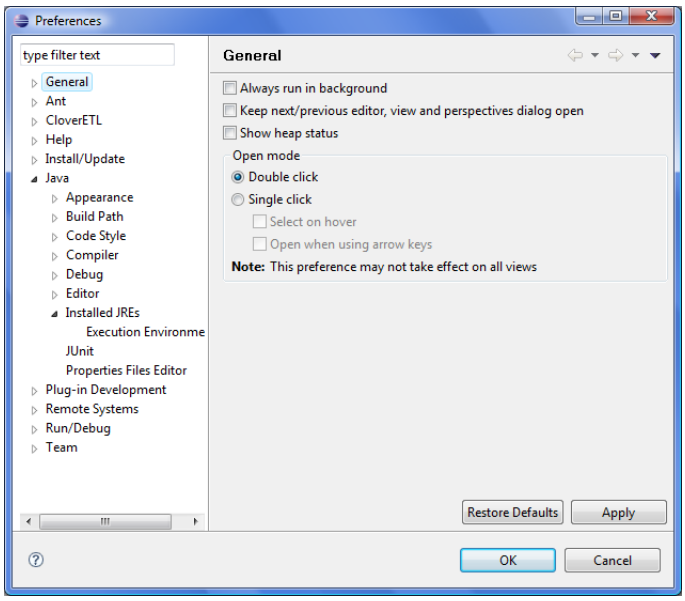


Figure 16.6. Preferences Wizard

Now you must expand the **Java** item and select the **Installed JREs** item as shown above. If you have installed JRE 1.6 already, you will see the following window:

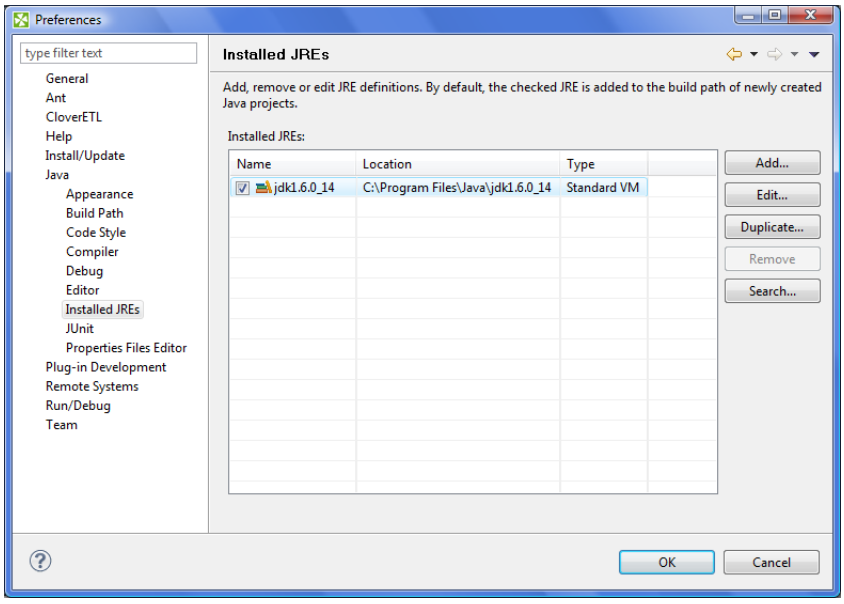


Figure 16.7. Installed JREs Wizard

Installing Java Development Kit

If you want to write and compile transformations in Java, you can install a JDK and add it to the project by selecting the project, right-clicking it and selecting the **Properties** item in the context menu. We suggest once again that you use JDK 1.6+.

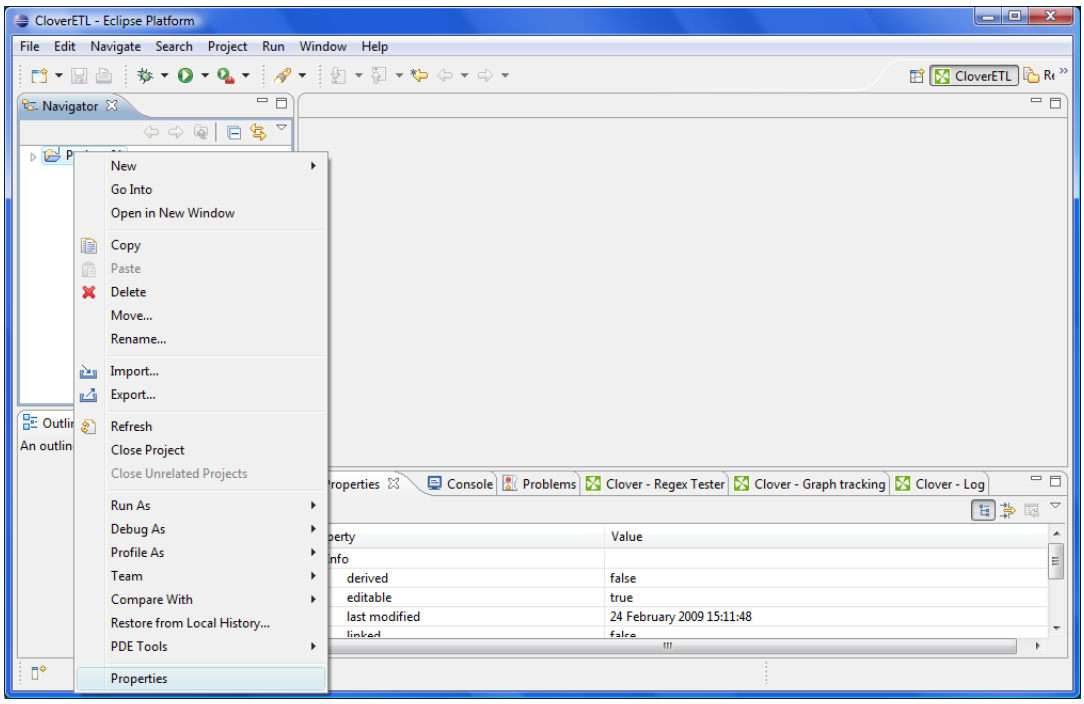


Figure 16.8. Adding a Java Development Kit

Then you select the **Java Build Path** item and its **Libraries** tab. You the click the **Add External JARs...** button.

Important

For Mac users: navigate to this menu directly after the installation. Then search a correct Java version (1.6 or higher) and select it. This has to be done on Mac because of a known issue when obsolete Java versions were selected by default after the installation.

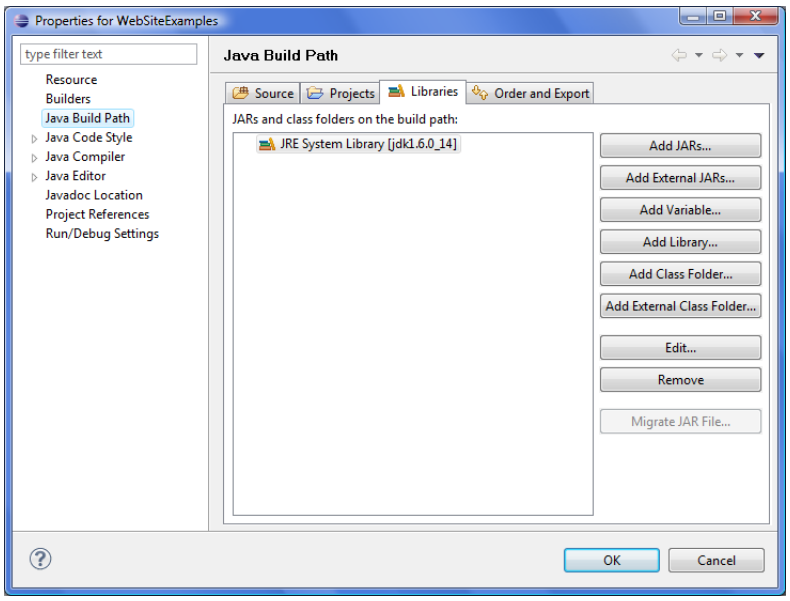


Figure 16.9. Searching for JDK Jars

You can add all `.jar` files contained in the selected `jdk` folder into the **Libraries** tab.

After confirming the selection, the `.jar` files will be added to the project as shown below.

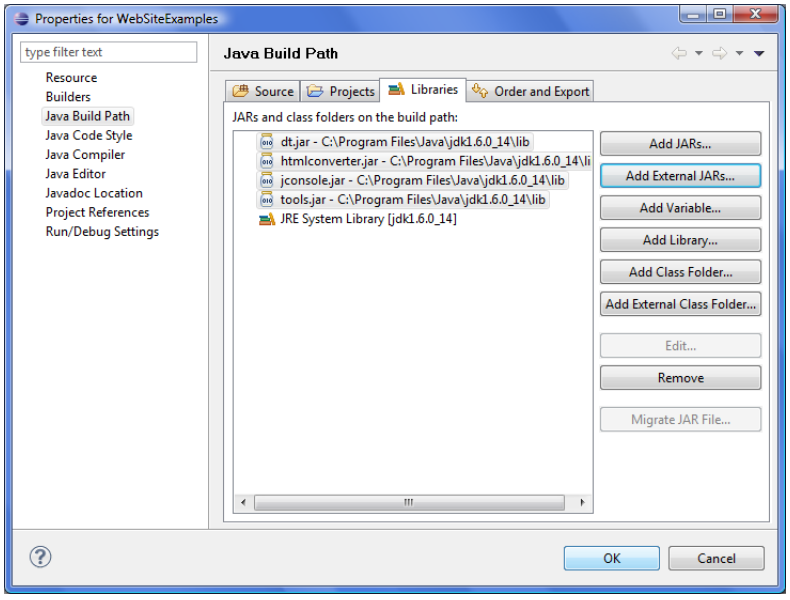


Figure 16.10. Adding JDK Jars

Part V. Graph Elements, Structures and Tools

Chapter 17. Components

The most important graph elements are components (nodes). They all serve to process data. Most of them have ports through which they can receive data and/or send the processed data out. Most components work only when edges are connected to these ports. Each edge in a graph connected to some port must have metadata assigned to it. Metadata describes the structure of data flowing through the edge from one component to another.

All components can be divided into five groups:

- Readers (p. 287)

  These components are usually the initial nodes of a graph. They read data from input files (either local or remote), receive it from a connected input port, read it from a dictionary, or generate data. Such nodes are called **Readers**.

- Writers (p. 360)

  Other components are the terminal nodes of a graph. They receive data through their input port(s) and write it to files (either local or remote), send it out through a connected output port, send e-mails, write data to a dictionary, or discard the received data. Such nodes are called **Writers**.

- Transformers (p. 439)

  These components are intermediate nodes of a graph. They receive data and copy it to all output ports, deduplicate, filter or sort data, concatenate, gather, or merge received data through many ports and send it out through a single output port, distribute records among many connected output ports, intersect data received through two input ports, aggregate data to get new information or transform data in a more complicated way. Such nodes are called **Transformers**.

- Joiners (p. 523)

  Joiners are also intermediate nodes of a graph. They receive data from two or more sources, join them according to a specified key, and send the joined data out through the output ports.

- Cluster Components (p. 553)

  The two **Cluster Components** serve to distribute data records among various nodes of a Cluster of **CloverETL Server** instances, or to gather these records together.

  Such graphs run in parallel in a Cluster.

- Others (p. 559)

  The **Others** group is a heterogeneous group of components. They can perform different tasks - execute system, Java, or DB commands; run **CloverETL** graphs, or send HTTP requests to a server. Other components of this group can read from or write to lookup tables, check the key of some data and replace it with another one, check the sort order of a sequence, or slow down processing of data flowing through the component.

- Some properties are common to all components.

  Common Properties of All Components (p. 223)

- Some are common to most of them.

  Common Properties of Most Components (p. 231)

- Other properties are common to each of the groups:

  - Common Properties of Readers (p. 250)
  - Common Properties of Writers (p. 262)
  - Common Properties of Transformers (p. 273)

- Common Properties of Joiners (p. 276)
- Common Properties of Cluster Components (p. 283)
- Common Properties of Others (p. 284)

For information about these common properties see Part VI, Components Overview (p. 219).

For information about individual components see Part VII, Component Reference (p. 286).

Chapter 18. Edges

This chapter presents an overview of the edges. It describes what they are, how they can be connected to the components of a graph, how metadata can be assigned to them and propagated through them, how the edges can be debugged and how the data flowing through the edges can be seen.

What Are the Edges?

Edges represent data flowing from one component to another.

The following are properties of edges:

- Connecting Components by the Edges (p. 86)

  Each edge must connect two components.

- Types of Edges (p. 87)

  Each edge is of one of the four types.

- Assigning Metadata to the Edges (p. 88)

  Metadata must be assigned to each edge, describing the data flowing through the edge.

- Propagating Metadata through the Edges (p. 89)

  Metadata can be propagated through some components from their input port to their output ports.

- Colors of the Edges (p. 89)

  Each edge changes its color upon metadata assignment, edge selection, etc.

- Debugging the Edges (p. 90)

  Each edge can be debugged.

Connecting Components by the Edges

When you have selected and pasted at least two components to the **Graph Editor**, you must connect them by edges taken from the **Palette** tool. Data will flow from one component to the other in this edge. For this reason, each edge must have assigned some metadata describing the structure of data records flowing in the edge.

There are two ways to create an edge between two components, you can click the **edge** label in the **Palette** tool, then move the cursor over the source component, the one you want the edge to start from, then left-click to start the edge creation. Then, move the cursor over to the target component, the one you want the edge to end at and click again. This creates the edge. The second way short-cuts the tool selection. You can simply mouse over the output ports of any component, and Clover will automatically switch to the **edge** tool if you have the **selection** tool currently selected. You can then click to start the edge creation process, which will work as above.

Some components only receive data from their input port(s) and write it to some data sources (**Writers**, including **Trash**), other components read data from data sources or generate data and send it out through their output port(s) (**Readers**, including **DataGenerator**), and other components both receive data and send it to other components (**Transformers** and **Joiners**). And the last group of components either must be connected to some edges (non-executing components such as **CheckForeignKey**, **LookupTableReaderWriter**, **SequenceChecker**, **SpeedLimiter**) or can be connected (the **Executing Components**).

When pasting an edge to the graph, as described, it always bounds to a component port. The number of ports of some components is strictly specified, while in others the number of ports is unlimited. If the number of ports is unlimited, a new port is created by connecting a new edge. Once you have terminated your work with edges, you must click the **Select** item in the **Palette** tool or click **Esc** on the keyboard.

If you have already connected two components by an edge, you can move this edge to any other component. To do that, you can highlight the edge by clicking, then move to the port to which the edge is connected (input or output) until the arrow mouse cursor turns to a cross. Once the cross appears, you can drag the edge to some of the other free ports of any component. If you mouse over the port with the selecton tool, it will automatically select the edge for you, so you can simply click and drag. Remember that you can only replace output port by another output port and input port by another input port.

Edge Auto-routing or Manual Routing

When two components are connected by an edge, sometimes the edge might overlap with other elements, like other components, notes, etc. In this case you may want to switch from default auto-routing to manual routing of the edge - in this mode you have control over where the edge is displayed. To achieve this, right-click the edge and uncheck the **Edge Autorouting** from the context menu.

After that, a point will appear in the middle of each straight part of the edge.

When you move the cursor over such point, the cursor will be replaced with either horizontal or vertical resize cursor, and you will be able to drag the corresponding edge section horizontally or vertically.

This way you can move the edges away from problematic areas.

You can also select an edge and then press **Ctrl**+**R** which toggles between edge auto-routing and manual mode.

Types of Edges

There are four types of edges, three of which have some internal buffer. You can select among edges by right clicking on an edge, then clicking the **Select edge** item and clicking one of the presented types.

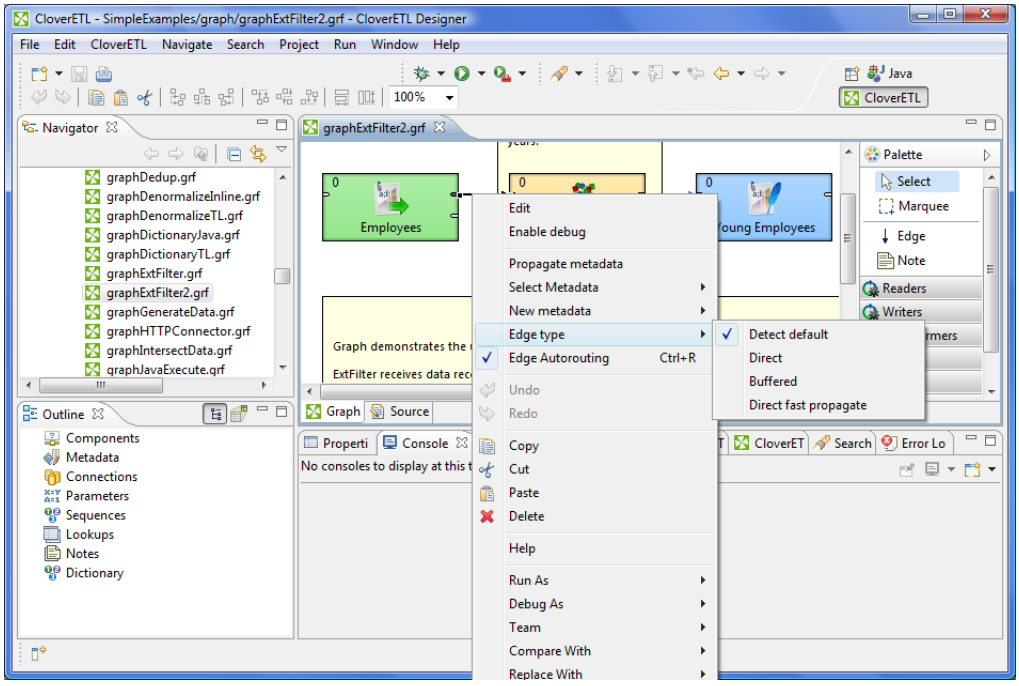


Figure 18.1. Selecting the Edge Type

Edges can be set to any of the following types:

- **Direct edge**: This type of edge has a buffer in memory, which helps data flow faster. This is the default edge type.
- **Buffered edge**: This type of edge has also a buffer in memory, but, if necessary, it can store data on disk as well. Thus the buffer size is unlimited. It has two buffers, one for reading and one for writing.

- **Direct fast propagate edge**. This is an alternative implementation of the **Direct edge**. This edge type has no buffer but it still provides a fast data flow. It sends each data record to the target of this edge as soon as it receives it.

- **Phase connection edge**. This edge type cannot be selected, it is created automatically between two components with different phase numbers.

If you do not want to specify an explicit edge type, you can let Clover decide by selecting the option **Detect default**.

Assigning Metadata to the Edges

Metadata are structures that describe data. At first, each edge will appear as a dashed line. Only after a metadata has been created and assigned to the edge, will the line becomes continuous.

You can create metadata as shown in corresponding sections below, however, you can also double-click the empty (dashed) edge and select **Create metadata** from the menu, or link some existing external metadata file by selecting **Link shared metadata**.

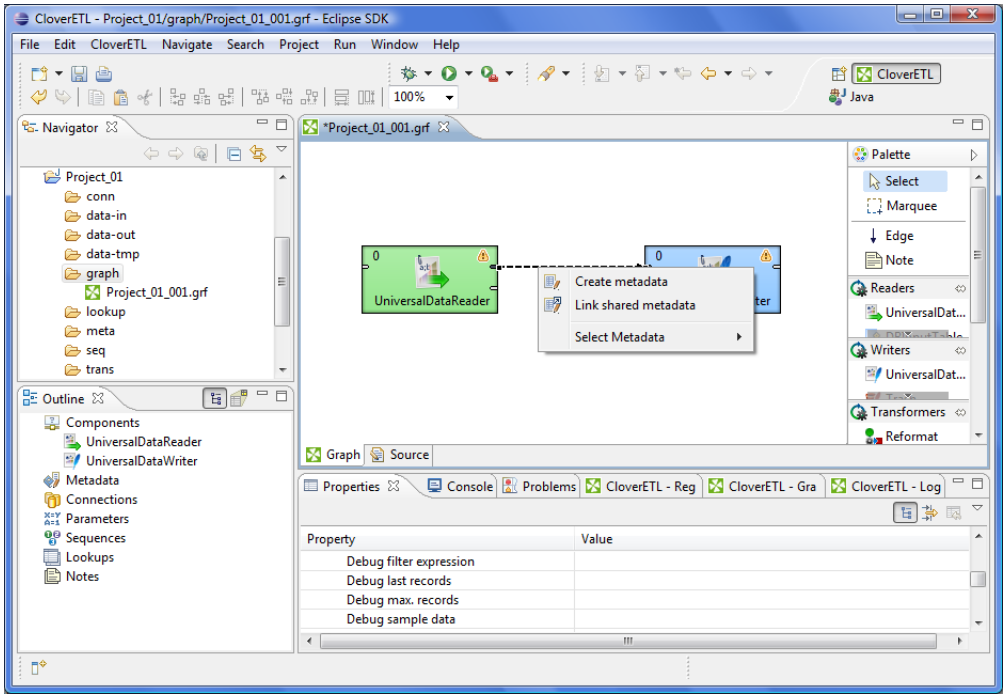


Figure 18.2. Creating Metadata on an empty Edge

You can also assign metadata to an edge by right-clicking the edge, choosing the **Select metadata** item from the context menu and selecting the desired metadata from the list. This can also be accomplished by dragging a metadata's entry from the **Outline** onto an edge.

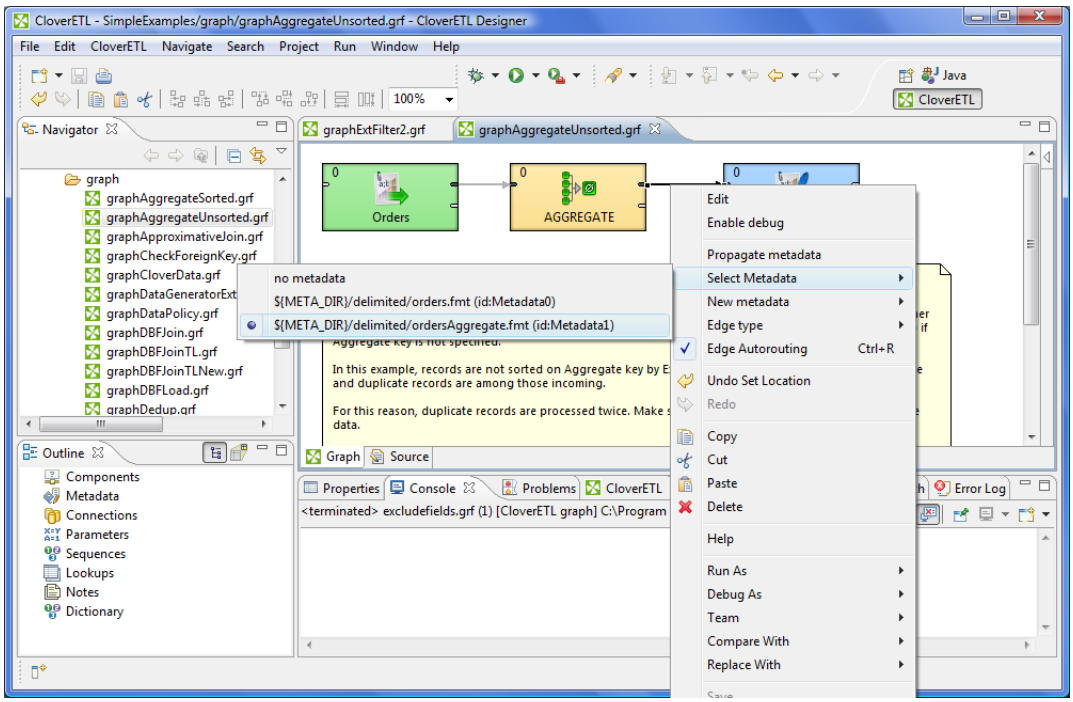


Figure 18.3. Assigning Metadata to an Edge

You can also select a metadata to be be automatically applied to edges as you create them. You choose this by right-clicking on the edge tool in the **Palette** and then selecting the metadata you want, or **none** if you want to remove the selection.

Propagating Metadata through the Edges

When you have already assigned metadata to the edge, you need to propagate the assigned metadata to other edges through a component.

To propagate metadata, you must also open the context menu by right-clicking the edge, then select the **Propagate metadata** item. The metadata will be propagated until it reaches a component in which metadata can be changed (for example: **Reformat**, **Joiners**, etc.).

For the other edges, you must define another metadata and propagate it again if desired.

Colors of the Edges

- When you connect two components by an edge, it is gray and dashed.
- After assigning metadata to the edge, it becomes solid, but still remains gray.
- When you click any metadata item in the **Outline** pane, all edges with the selected metadata become blue.
- If you click an edge in the **Graph Editor**, the selected edge becomes black and all of the other edges with the same metadata become blue. (In this case, metadata are shown in the edge tooltip as well.)

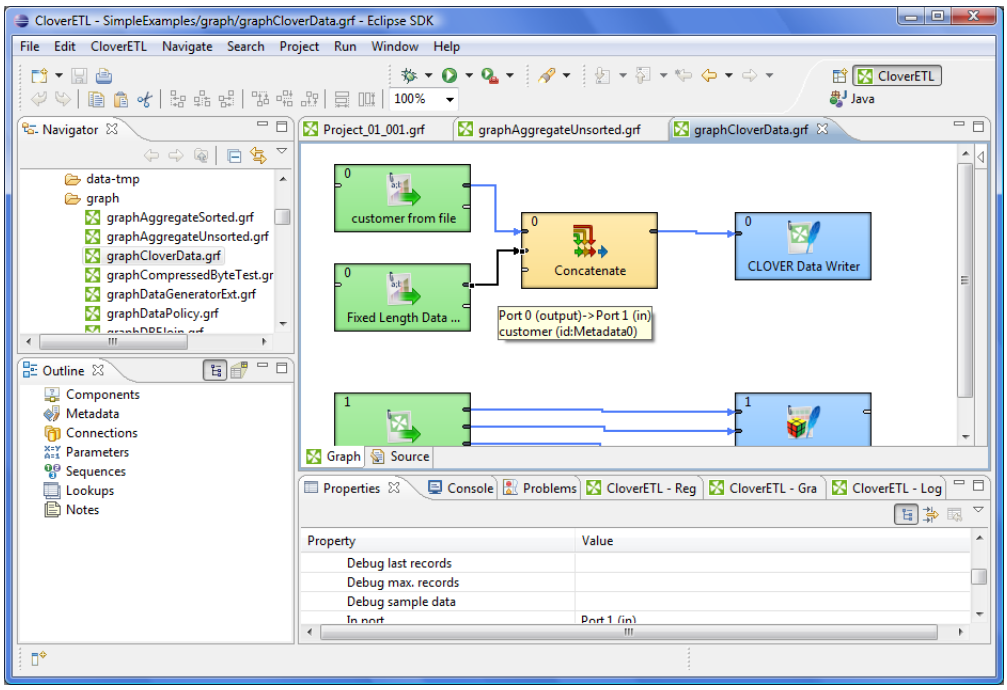


Figure 18.4. Metadata in the Tooltip

Debugging the Edges

If you obtain incorrect or unexpected results when running some of your graphs and you want to know what errors occur and where, you can debug the graph. You need to guess where the problem may arise from and, sometimes, you also need to specify what records should be saved to debug files. If you are not processing large numbers of records, you should not need to limit the number that should be saved to debug files, however, in case you are processing large numbers of records, this option may be useful.

To debug an edge, you can:

1. Enable debug. See Enabling Debug (p. 90).

2. Select debug data. See Selecting Debug Data (p. 91),

3. View debug data. See Viewing Debug Data (p. 93).

4. Turn off the debug. See Turning Off Debug (p. 95).

Enabling Debug

- To debug the graph, right-click the edges that are under suspicion and select the **Enable debug** option from the context menu. After that, a bug icon appears on the edge meaning that a debugging will be performed upon the graph execution.

- The same can be done if you click the edge and switch to the **Properties** tab of the **Tabs** pane. There you only need to set the **Debug mode** attribute to `true`. By default, it is set to `false`. Again, a bug icon appears on the edge.

When you run the graph, for each debug edge, one debug file will be created. After that, you only need to view and study the data records from these debug files (`.dbg` extension).

Selecting Debug Data

If you do not do anything else than select the edges that should be debugged, all data records that will go through such edges will be saved to debug files.

Nevertheless, as has been mentioned above, you can restrict those data records that should be saved to debug files.

This can be done in the **Properties** tab of any debug edge or by selecting **Debug properties** from the context menu after right-clicking the debug edge.

You can set any of the following four edge attributes either in the **Properties** tab or in the **Debug properties** wizard.

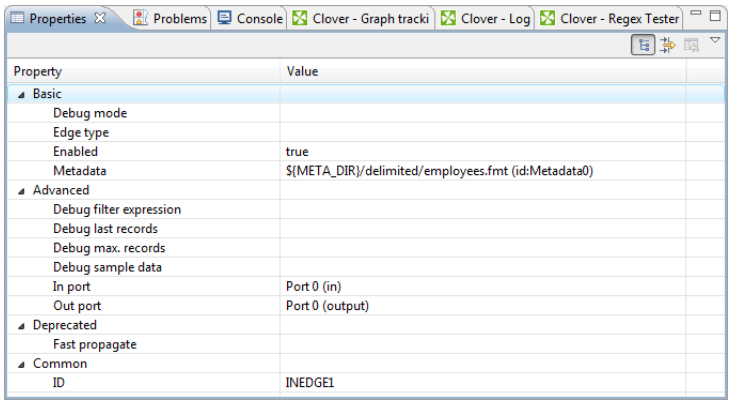


Figure 18.5. Properties of an Edge

- **Debug filter expression**

  If you specify some filter expression for an edge, data records that satisfy the specified filter expression will be saved to the debug file. The others that do not satisfy the expression will be ignored.

  Remember also that if a filter expression is defined, either all records that satisfy the expression (**Debug sample data** is set to `false`) or only a sample of them (**Debug sample data** is set to `true`) will be saved.

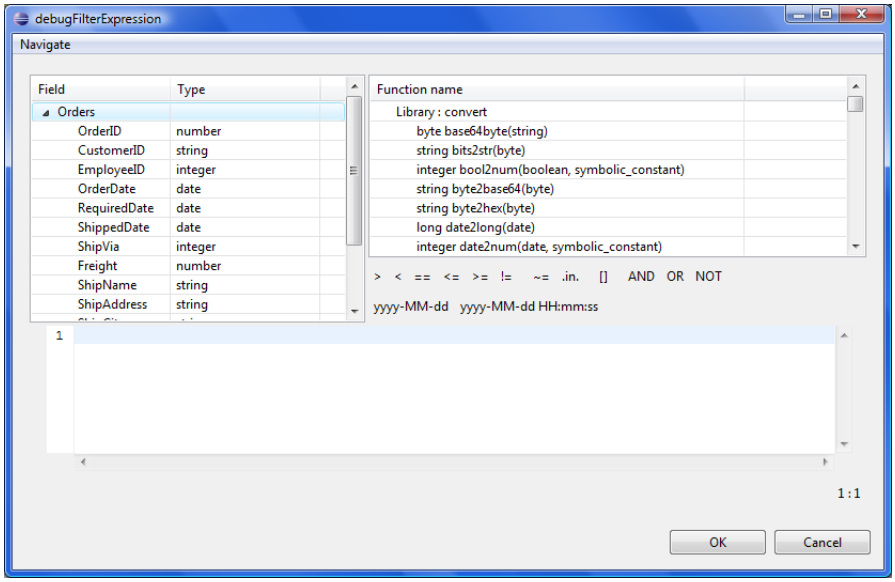


Figure 18.6. Filter Editor Wizard

This wizard consists of three panes. The left one displays the list of record fields, their names and data types. You can select any of them by double-clicking or dragging and dropping. Then the field name appears in the bottom area with the port number preceded by dollar sign that are before that name. (For example, `$0.employee`.) You can also use the functions selected from the right pane of the window. Below this pane, there are both comparison signs and logical connections. You can select any of the names, functions, signs and connections by double-clicking. After that, they appear in the bottom area. You can work with them in this area and complete the creation of the filter expression. You can validate the expression, exit the creation by clicking **Cancel** or confirm the expression by clicking **OK**.

Important

You can use either CTL1, or CTL2 in **Filter Editor**.

The following two options are equivalent:

1. **For CTL1**

```
is_integer($0.field1)
```

2. **For CTL2**

```
//#CTL2
isInteger($0.field1)
```

- **Debug last records**

  If you set the **Debug last records** property to `false`, data records from the beginning will be saved to the debug file. By default, the records from the end are saved to debug files. Default value of **Debug last records** is `true`.

  Remember that if you set the **Debug last records** attribute to `false`, data records will be selected from the beginning with greater frequency than from the end. And, if you set the **Debug last records** attribute to `true` or leave it unchanged, they will be selected more frequently from the end than from the beginning.

- **Debug max. records**

  You can also define a limit of how many data records should be saved to a debug file at most. These data records will be taken from either the beginning (**Debug last records** is set to `false`) or the end (**Debug last records** has the default value or it is set to `true` explicitly).

- **Debug sample data**

  If you set the **Debug sample data** attribute to `true`, the **Debug max. records** attribute value will only be the threshold that would limit how many data records could be saved to a debug file. Data records will be saved at random, some of them will be omitted, others will be saved to the debug file. In this case, the number of data records saved to a debug file will be less than or equal to this limit.

  If you do not set any value of **Debug sample data** or if you set it to `false` explicitly, the number of records saved to the debug file will be equal to the **Debug max. records** attribute value (if more records than **Debug max. records** go through the debug edge).

The same properties can also be defined using the context menu by selecting the **Debug properties** option. After that, the following wizard will open:

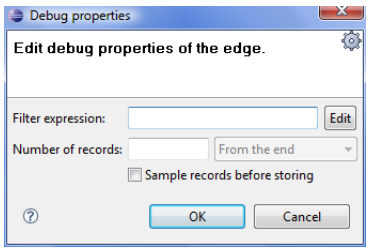


Figure 18.7. Debug Properties Wizard

Viewing Debug Data

In order to view the records that have gone through the edge and met the filter expression and have been saved, you must open the context menu by right-clicking. Then you must click the **View data** item. After that, a **View data** dialog opens. Note, that you can create a filter expression here in the same way as described above.

You must select the number of records that should be displayed and confirm it by clicking **OK**.

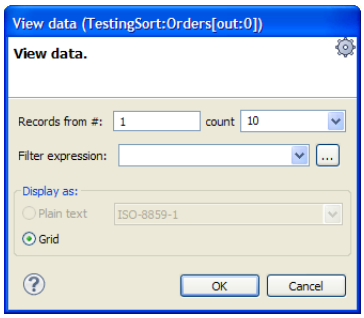
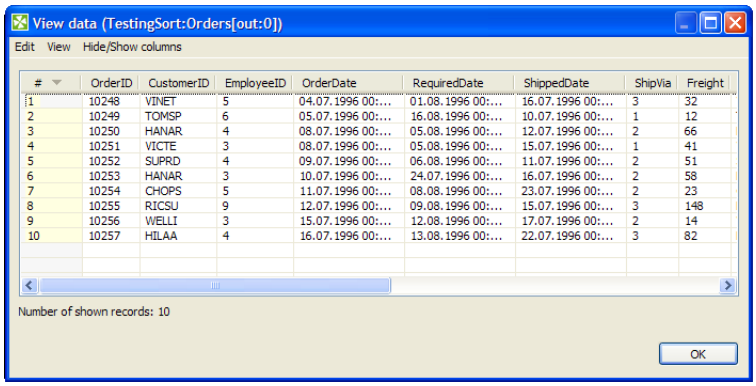


Figure 18.8. View Data Dialog

CloverETL Designer remembers the selected count and after the **View data** dialog is opened again, the same count is offered.

The records are shown in another **View data** dialog. This dialog has grid mode. You can sort the records in any of its columns in ascending or descending order by simply clicking its header. Dialog acts is modal window, so that user can view data on more edges at the same time. To differ between dialogs window title provides info about viewing edge in format GRAPH.name:COMPONENT.name[out: PORT.id].



#	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight
1	10248	VINET	5	04.07.1996 00:...	01.08.1996 00:...	16.07.1996 00:...	3	32
2	10249	TOMSP	6	05.07.1996 00:...	16.08.1996 00:...	10.07.1996 00:...	1	12
3	10250	HANAR	4	08.07.1996 00:...	05.08.1996 00:...	12.07.1996 00:...	2	66
4	10251	VICTE	3	08.07.1996 00:...	05.08.1996 00:...	15.07.1996 00:...	1	41
5	10252	SUPRD	4	09.07.1996 00:...	06.08.1996 00:...	11.07.1996 00:...	2	51
6	10253	HANAR	3	10.07.1996 00:...	24.07.1996 00:...	16.07.1996 00:...	2	58
7	10254	CHOPS	5	11.07.1996 00:...	08.08.1996 00:...	23.07.1996 00:...	2	23
8	10255	RICSU	9	12.07.1996 00:...	09.08.1996 00:...	15.07.1996 00:...	3	148
9	10256	WELLI	3	15.07.1996 00:...	12.08.1996 00:...	17.07.1996 00:...	2	14
10	10257	HILAA	4	16.07.1996 00:...	13.08.1996 00:...	22.07.1996 00:...	3	82

Figure 18.9. Viewing Data

Above the grid, there are three labels: **Edit**, **View**, **Hide/Show columns**.

By clicking the **Hide/Show columns** label, you can select which columns should be displayed: all, none, only selected. You can select any option by clicking.

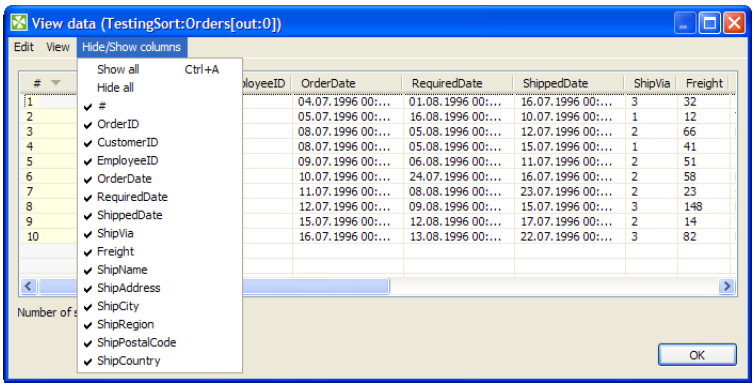


Figure 18.10. Hide/Show Columns when Viewing Data

By clicking the **View** label, you are presented with two options: You can decide whether you want to view the unprintable characters, or not. You can also decide whether you want to view only one record separately. Such a record appears in the **View record** dialog. At the bottom of this dialog, you can see some arrow buttons. They allow user to browse the records and view them in sequence. Note that by clicking the button most on the right, you can see the last record of the displayed records, but it does not necessarily display the record that is the last processed.

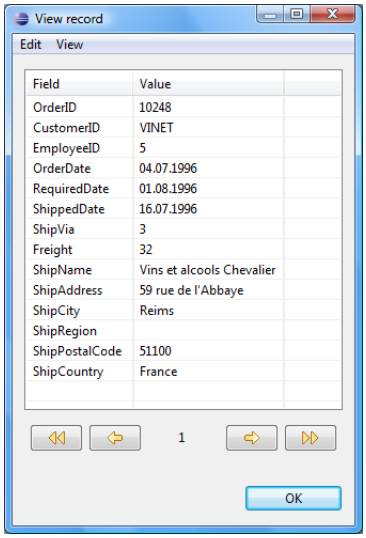


Figure 18.11. View Record Dialog

By clicking the **Edit** label, you are presented with four options.

- You can select the number of record or line you want to see. Such a record will be highlighted after typing its number and clicking **OK**.

- Another option opens the **Find** dialog. First of all, this wizard contains a text area you can type an expression into. Then, if you check the **Match case** checkbox, the search will be case sensitive. If you check the **Entire cells** checkbox, only the cells that meet the expression completely will be highlighted. If you check the **Regular expression** checkbox, the expression you have typed into the text area will be used as a regular expression. You can also decide whether you want to search some expression in the direction of rows or columns. You can also select what column it will be searched in: all, only visible, one column from the list. And, as the last option, you can select whether you want to find all cells that meet some criterion or only one of the cells.

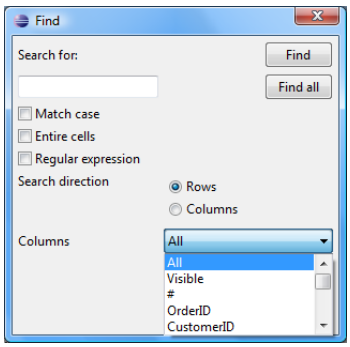


Figure 18.12. Find Dialog

- As the last option, you can copy some of your records or a part of a record. You need to select whether you want to copy either the entire record (either to string, or as a record - in this last case you can select the delimiter as well) or only some of the record fields. The selected option is enabled, the other one is disabled. After clicking the **OK** button, you only need to choose the location where it shall be copied into and past it there.

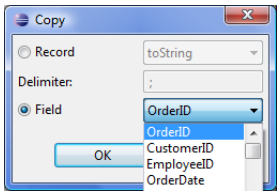


Figure 18.13. Copy Dialog

Turning Off Debug

If you want to turn off debugging, you can click the **Graph editor** in any place outside the components and the edges, switch to the **Properties** tab and set the **Debug mode** attribute to `false`. This way you can turn off all debugging at a time.

Also, if you have not defined the **Debug max. records** attribute, you could specify it in this **Properties** tab (if **Debug mode** is empty or set to `true`) for all debug edges at a time. But remember that if any edge has its own **Debug max. records** attribute value defined, the global value of this attribute will be ignored and that of the edge will be applied.

Chapter 19. Metadata

Every edge of a graph carries some data. This data must be described using metadata. These metadata can be either internal, or external (shared).

For information about data types and record types that can be used in metadata see Data Types and Record Types (p. 97).

When working with various data types, the formatting or locale can also be specified. See:

- Data and Time Format (p. 99)
- Numeric Format (p. 106)
- Locale (p. 112)

Some of the components may also use the **Autofilling** functionality in **Metadata**.

See Autofilling Functions (p. 117).

Each metadata can be created as:

- **Internal**: See Internal Metadata (p. 119).

  Internal metadata can be:

  - **Externalized**: See Externalizing Internal Metadata (p. 120).
  - **Exported**: See Exporting Internal Metadata (p. 121).
- **External (shared)**: See External (Shared) Metadata (p. 122).

  External (shared) metadata can be:

  - **Linked to the graph**: See Linking External (Shared) Metadata (p. 122).
  - **Internalized**: See Internalizing External (Shared) Metadata (p. 123).

Metadata can be created from:

- **Flat file**: See Extracting Metadata from a Flat File (p. 124).
- **XLS(X) file**: See Extracting Metadata from an XLS(X) File (p. 129).
- **DBase file**: See Extracting Metadata from a DBase File (p. 134).
- **Database**: See Extracting Metadata from a Database (p. 130).
- **By user**: See Creating Metadata by User (p. 134).
- **Lotus Notes**: See Extracting Metadata from Lotus Notes (p. 134).

Metadata can also be created dynamically or read from remote sources:

- **Dynamic metadata**: See Dynamic Metadata (p. 136).
- **Read from special sources**: See Reading Metadata from Special Sources (p. 137).

Metadata editor is described in Metadata Editor (p. 140).

For detailed information about changing or defining delimiters in `delimited` or `mixed` record types see Changing and Defining Delimiters (p. 145).

Metadata can also be edited in its source code. See Editing Metadata in the Source Code (p. 149).

Metadata can serve as a source for creating a database table. See Create Database Table from Metadata (p. 138).

Data Types and Record Types

Data flowing through the edges must be described using metadata. Metadata describes both the record as a whole and all its fields.

Clover data types are described in following sections:

- Data Types in Metadata (p. 97)
- Data Types in CTL (p. 608) for CTL1
- Data Types in CTL2 (p. 669) for CTL2

Data Types in Metadata

Following are the types of record fields used in metadata:

Table 19.1. Data Types in Metadata

Data type	Size [5]	Range or values	Default value
boolean	Represents 1 bit. Its size is not precisely defined.	`true` | `false` | `1` | `0`	`false` | `0`
byte	Depends on the actual data length.	from -128 to 127	`null`
cbyte	Depends on the actual data length and success of compression.	from -128 to 127	`null`
date	64 bits [1]	Starts January 1, 1970, 00:00:00 GMT and is incremented by 1 ms.	current date and time
decimal	Depends on `Length` and `Scale`. (The former is the maximum number of all digits, the latter is the maximum number of digits after the decimal dot. Default values are 8 and 2, respectively.) [2], [3]	`decimal(6,2)` (They can have values from -9999.99 to 9999.99, length and scale can only be defined in CTL1)	`0.00`
integer	32 bits [2]	From `Integer.MIN_VALUE` to `Integer.MAX_VALUE` (according to the Java integer data type): From -2^{31} to $2^{31}-1$. `Integer.MIN_VALUE` is interpreted as `null`.	`0`
long	64 bits [2]	From `Long.MIN_VALUE` to `Long.MAX_VALUE` (according to the Java long data type): From -2^{63} to $2^{63}-1$. `Long.MIN_VALUE` is interpreted as `null`.	`0`
number	64 bits [2]	Negative values are from $-(2-2^{-52}).2^{1023}$ to -2^{-1074}, another value is 0, and positive values are from 2^{-1074} to $(2-2^{-52}).2^{1023}$. Three special values: `NaN`, `-Infinity`, and `Infinity` are defined.	`0.0`

Data type	Size [5]	Range or values	Default value
string	Depends on the actual data length. Each character is stored in 16 bits.	Obviously you cannot have infinite strings. Instead of limiting how many characters each string can consist of (theoretically up to 64K), think about memory requirements. A string takes (number of characters) * 2 bytes of memory. At the same time, no record can take more than `MAX_RECORD_SIZE` of bytes, see Chapter 16, Advanced Topics (p. 73).	`null` [4]

Legend:

1): Any date can be parsed and formatted using date and time format pattern. See Data and Time Format(p. 99). Parsing and formatting can also be influenced by locale. See Locale (p. 112).

2): Any numeric data type can be parsed and formatted using numeric format pattern. See Numeric Format (p. 106). Parsing and formatting may also be influenced by locale. See Locale (p. 112).

3): The default *`length`* and *`scale`* of a `decimal` are `8` and `2`, respectively. These default values of DECIMAL_LENGTH and DECIMAL_SCALE are contained in the `org.jetel.data.defaultProperties` file and can be changed to other values.

4): By default, if a field which is of the string data type of any metadata is an empty string, such field value is converted to `null` instead of an empty string (`""`) unless you set the **Null value** property of the field to any other value.

5): This column may look like an implementation detail but it is not so true. **Size** lets you estimate how much memory your records are going to need. To do that, take a look at how many fields your record has, which data types they are and then compare the result to the `MAX_RECORD_SIZE` property (the maximum size of a record in bytes, see Chapter 16, Advanced Topics (p. 73)). If your records are likely to have more bytes than that, simply raise the value (otherwise buffer overflow will occur).

For other information about these data types and other data types used in Clover transformation language (CTL) see Data Types in CTL (p. 608) for CTL1 or Data Types in CTL2 (p. 669) for CTL2.

Record Types

Each record is of one of the following three types:

- **Delimited**. This is the type of records in which every two adjacent fields are separated from each other by a delimiter and the whole record is terminated by record delimiter as well.
- **Fixed**. This is the type of records in which every field has some specified length (size). It is counted in numbers of characters.
- **Mixed**. This is the type of records in which fields can be separated from each other by a delimiter and also have some specified length (size). The size is counted in number of characters. This record type is the mixture of the two cases above. Each individual field may have different properties. Some fields may only have a delimiter, others may have specified size, the rest of them may have both delimiter and size.

Data Formats

Sometimes **Format** may be defined for parsing and formatting data values.

1. Any date can be parsed and/or formatted using date and time format pattern. See Data and Time Format (p. 99).

   Parsing and formatting can also be influenced by locale (names of months, order of day or month information, etc.). See Locale (p. 112).

2. Any numeric data type (`decimal`, `integer`, `long`, `number`) can be parsed and/or formatted using numeric format pattern. See Numeric Format (p. 106).

   Parsing and formatting can also be influenced by locale (e.g., decimal dot or decimal comma, etc.). See Locale (p. 112).

3. Any boolean data type can be parsed and formatted using boolean format pattern. See Boolean Format (p. 110).

4. Any string data type can be parsed using string format pattern. See String Format (p. 111).

Note

Remember that both date and time formats and numeric formats are displayed using system **Locale** value or the **Locale** specified in the `defaultProperties` file, unless another **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Data and Time Format

A formatting string describes how a date/time values should be read and written from(to) string representation (flat files, human readable output, etc.).

A format can also specify an engine which **CloverETL** will use by specifying a prefix (see below). There are two built-in date engines available: standard Java and third-party Joda (http://joda-time.sourceforge.net).

Table 19.2. Available date engines

Date engine	Prefix	Default	Description	Example
Java	`java:`	yes - when no prefix is given	Standard Java date implementation. Provides lenient, error-prone and full-featured parsing and writing. It has moderate speed and is generally a good choice unless you need to work with large quantities of date/time fields. For advanced study please refer to Java SimpleDateFormat documentation.	`java:yyyy-MM-dd HH:mm:ss`

Date engine	Prefix	Default	Description	Example
Joda	`joda:`		An improved third-party date library. Joda is more strict on input data accuracy when parsing and does not work well with time zones. It does, however, provide a 20-30% speed increase compared to standard Java. For further reading please visit the project site at http://joda-time.sourceforge.net). Joda may be convenient for AS/400 machines. On the other hand, Joda is unable to read time zone expressed with any number of `z` letters and/or at least three `Z` letters in a pattern.	`joda:yyyy-MM-dd HH:mm:ss`

Please note, that actual format strings for Java and Joda are almost 100% compatible with each other - see tables below.

Important

The format patterns described in this section are used both in metadata as the **Format** property and in CTL.

At first, we provide the list of pattern syntax, the rules and the examples of its usage for Java:

Table 19.3. Date Format Pattern Syntax (Java)

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
Y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; VII; 07; 7
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	970

Letter	Date or Time Component	Presentation	Examples
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
'	Escape for text/id	Delimiter	(none)
''	Single quote	Literal	'

The number of symbol letters you specify also determines the format. For example, if the "zz" pattern results in "PDT", then the "zzzz" pattern generates "Pacific Daylight Time". The following table summarizes these rules:

Table 19.4. Rules for Date Format Usage (Java)

Presentation	Processing	Number of Pattern Letters	Form
Text	Formatting	1 - 3	short or abbreviated form, if one exists
Text	Formatting	>= 4	full form
Text	Parsing	>= 1	both forms
Year	Formatting	2	truncated to 2 digits
Year	Formatting	1 or >= 3	interpreted as Number.
Year	Parsing	1	intepreted literally
Year	Parsing	2	interpreted relative to the century within 80 years before or 20 years after the time when the `SimpleDateFormat` instance is created
Year	Parsing	>= 3	intepreted literally
Month	Both	1-2	interpreted as a Number
Month	Parsing	>= 3	interpreted as Text (using Roman numbers, abbreviated month name - if exists, or full month name)
Month	Formatting	3	interpreted as Text (using Roman numbers, or abbreviated month name - if exists)
Month	Formatting	>= 4	interpreted as Text (full month name)
Number	Formatting	minimum number of required digits	shorter numbers are padded with zeros
Number	Parsing	number of pattern letters is ignored (unless needed to separate two adjacent fields)	any form

Presentation	Processing	Number of Pattern Letters	Form
General time zone	Both	1-3	short or abbreviated form, if has a name. Otherwise, GMT offset value (GMT[sign][[0]0-23]:[00-59])
General time zone	Both	>= 4	full form, , if has a name. Otherwise, GMT offset value (GMT[sign][[0]0-23]:[00-59])
General time zone	Parsing	>= 1	RFC 822 time zone form is allowed
RFC 822 time zone	Both	>= 1	RFC 822 4-digit time zone format is used ([sign][0-23][00-59])
RFC 822 time zone	Parsing	>= 1	General time zone form is allowed

Examples of date format patterns and resulting dates follow:

Table 19.5. Date and Time Format Patterns and Results (Java)

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, "yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o"clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

The described format patterns are used both in metadata as the **Format** property and in CTL.

Now the list of format pattern syntax for Joda follows:

Table 19.6. Date Format Pattern Syntax (Joda)

Symbol	Meaning	Presentation	Examples
G	Era designator	Text	AD
C	Century of era (>=0)	Number	20
Y	Year of era (>=0)	Year	1996
y	Year	Year	1996
x	Week of weekyear	Year	1996
M	Month of year	Month	July; Jul; 07
w	Week of year	Number	27
D	Day of year	Number	189
d	Day of month	Number	10
e	Day of week	Number	2
E	Day of week	Text	Tuesday; Tue
a	Halfday of day	Text	PM
H	Hour of day (0-23)	Number	0
k	Clockhour of day (1-24)	Number	24
K	Hour of halfday (0-11)	Number	0
h	Clockhour of halfday (1-12)	Number	12
m	Minute of hour	Number	30
s	Second of minute	Number	55
S	Fraction of second	Number	970
z	Time zone	Text	Pacific Standard Time; PST
Z	Time zone offset/id	Zone	-0800; -08:00; America/Los_Angeles
'	Escape for text/id	Delimiter	(none)
''	Single quote	Literal	'

The number of symbol letters you specify also determines the format. The following table summarizes these rules:

Table 19.7. Rules for Date Format Usage (Joda)

Presentation	Processing	Number of Pattern Letters	Form
Text	Formatting	1 - 3	short or abbreviated form, if one exists
Text	Formatting	>= 4	full form
Text	Parsing	>= 1	both forms
Year	Formatting	2	truncated to 2 digits
Year	Formatting	1 or >= 3	interpreted as Number.
Year	Parsing	>= 1	intepreted literally

Presentation	Processing	Number of Pattern Letters	Form
Month	Both	1-2	interpreted as a Number
Month	Parsing	>= 3	interpreted as Text (using Roman numbers, abbreviated month name - if exists, or full month name)
Month	Formatting	3	interpreted as Text (using Roman numbers, or abbreviated month name - if exists)
Month	Formatting	>= 4	interpreted as Text (full month name)
Number	Formatting	minimum number of required digits	shorter numbers are padded with zeros
Number	Parsing	>= 1	any form
Zone name	Formatting	1-3	short or abbreviated form
Zone name	Formatting	>= 4	full form
Time zone offset/id	Formatting	1	Offset without a colon between hours and minutes
Time zone offset/id	Formatting	2	Offset with a colon between hours and minutes
Time zone offset/id	Formatting	>= 3	Full textual form like this: "Continent/City"
Time zone offset/id	Parsing	1	Offset without a colon between hours and minutes
Time zone offset/id	Parsing	2	Offset with a colon between hours and minutes

Important

Remember that parsing with any number of "z" letters is not allowed. And neither parsing with the number of "Z" letters greater than or equal to 3 is allowed.

See information about data types in metadata and CTL1 and CTL2:

- Data Types and Record Types (p. 97)
- **For CTL1:**

  Data Types in CTL (p. 608)
- **For CTL2:**

  Data Types in CTL2 (p. 669)

They are also used in CTL1 and CTL2 functions. See:

For CTL1:

- Conversion Functions (p. 637)
- Date Functions (p. 642)

- String Functions (p. 649)

For CTL2:

- Conversion Functions (p. 697)
- Date Functions (p. 705)
- String Functions (p. 711)

Numeric Format

When a text is parsed as any numeric data type or any numeric data type should be formatted to a text, format pattern must be specified.

Parsing and formatting is locale sensitive.

In **CloverETL**, Java decimal format is used.

Table 19.8. Numeric Format Pattern Syntax

Symbol	Location	Localized?	Meaning
#	Number	Yes	Digit, zero shows as absent
0	Number	Yes	Digit
.	Number	Yes	Decimal separator or monetary decimal separator
-	Number	Yes	Minus sign
,	Number	Yes	Grouping separator
E	Number	Yes	Separates mantissa and exponent in scientific notation. *Need not be quoted in prefix or suffix.*
;	Subpattern boundary	Yes	Separates positive and negative subpatterns
%	Prefix or suffix	Yes	Multiply by 100 and show as percentage
‰ (\u2030)	Prefix or suffix	Yes	Multiply by 1000 and show as per mille value
¤ (\u00A4)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
'	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, "'#'#" formats 123 to "#123". To create a single quote itself, use two in a row: "# o"clock".

- Both prefix and suffix are Unicode characters from \u0000 to \uFFFD, including the margins, but excluding special characters.

Format pattern composes of subpatterns, prefixes, suffixes, etc. in the way shown in the following table:

Table 19.9. BNF Diagram

Format	Components
pattern	subpattern{;subpattern}
subpattern	{prefix}integer{.fraction}{suffix}
prefix	'\\u0000'..'\\uFFFD' - specialCharacters
suffix	'\\u0000'..'\\uFFFD' - specialCharacters
integer	'#'* '0'* '0'
fraction	'0'* '#'*

Explanation of these symbols follow:

Table 19.10. Used Notation

Notation	Description
X*	0 or more instances of X
(X | Y)	either X or Y
X..Y	any character from X up to Y, inclusive
S - T	characters in S, except those in T
{X}	X is optional

Important

The grouping separator is commonly used for thousands, but in some countries it separates ten-thousands. The grouping size is a constant number of digits between the grouping characters, such as 3 for 100,000,000 or 4 for 1,0000,0000. If you supply a pattern with multiple grouping characters, the interval between the last one and the end of the integer is the one that is used. So "#,##,###,####" == "######,####" == "##,####,####".

Remember also that formatting is locale senistive. See the following table in which results are different for different locales:

Table 19.11. Locale-Sensitive Formatting

Pattern	Locale	Result
###,###.###	en.US	123,456.789
###,###.###	de.DE	123.456,789
###,###.###	fr.FR	123 456,789

Note

For a deeper look on handling numbers, consult the official Java documentation.

Scientific Notation

Numbers in scientific notation are expressed as the product of a mantissa and a power of ten.

For example, `1234` can be expressed as `1.234 x 10`^{3}.

The mantissa is often in the range `1.0 <= x < 10.0`, but it need not be.

Numeric data types can be instructed to format and parse scientific notation only via a pattern. In a pattern, the exponent character immediately followed by one or more digit characters indicates scientific notation.

Example: "0.###E0" formats the number 1234 as "1.234E3".

Examples of numeric pattern and results follow:

Table 19.12. Numeric Format Patterns and Results

Value	Pattern	Result
1234	0.###E0	1.234E3
12345	##0.#####E0[1)]	12.345E3
123456	##0.#####E0[1)]	123.456E3
1234567	##0.#####E0[1)]	1.234567E6
12345	#0.#####E0[2)]	1.2345E4
123456	#0.#####E0[2)]	12.3456E4
1234567	#0.#####E0[2)]	1.234567E6
0.00123	00.###E0[3)]	12.3E-4
123456	##0.##E0[4)]	12.346E3

Legend:

1): Maximum number of integer digits is 3, minimum number of integer digits is 1, maximum is greater than minimum, thus exponent will be a multiplicate of three (maximum number of integer digits) in each of the cases.

2): Maximum number of integer digits is 2, minimum number of integer digits is 1, maximum is greater than minimum, thus exponent will be a multiplicate of two (maximum number of integer digits) in each of the cases.

3): Maximum number of integer digits is 2, minimum number of integer digits is 2, maximum is equal to minimum, minimum number of integer digits will be achieved by adjusting the exponent.

4): Maximum number of integer digits is 3, maximum number of fraction digits is 2, number of significant digits is sum of maximum number of integer digits and maximum number of fraction digits, thus, the number of significant digits is as shown (5 digits).

Binary Formats

The table below presents a list of available formats:

Table 19.13. Available Binary Formats

Type	Name	Format	Length
integer	BIG_ENDIAN	two's-complement, big-endian	variable
	LITTLE_ENDIAN	two's-complement, little-endian	
	PACKED_DECIMAL	packed decimal	
floating-point	DOUBLE_BIG_ENDIAN	IEEE 754, big-endian	8 bytes
	DOUBLE_LITTLE_ENDIAN	IEEE 754, little-endian	
	FLOAT_BIG_ENDIAN	IEEE 754, big-endian	4 bytes
	FLOAT_LITTLE_ENDIAN	IEEE 754, little-endian	

The floating-point formats can be used with `numeric` and `decimal` datatypes. The integer formats can be used with `integer` and `long` datatypes. The exception to the rule is the `decimal` datatype, which also supports integer formats (`BIG_ENDIAN`, `LITTLE_ENDIAN` and `PACKED_DECIMAL`). When an integer format is used with the `decimal` datatype, implicit decimal point is set according to the **Scale** attribute. For example, if the stored value is 123456789 and **Scale** is set to 3, the value of the field will be 123456.789.

To use a binary format, create a metadata field with one of the supported datatypes and set the **Format** attribute to the name of the format prefixed with `"BINARY:"`, e.g. to use the `PACKED_DECIMAL` format, create a decimal field and set its **Format** to `"BINARY:PACKED_DECIMAL"` by choosing it from the list of available formats.

For the fixed-length formats (double and float) also the **Size** attribute must be set accordingly.

Currently, binary data formats can only be handled by ComplexDataReader (p. 291) and the deprecated FixLenDataReader.

Boolean Format

Format for boolean data type specified in **Metadata** consists of up to four parts separated from each other by the same delimiter.

This delimiter must also be at the beginning and the end of the **Format** string. On the other hand, the delimiter must not be contained in the values of the boolean field.

Important

If you do not use the same character at the beginning and the end of the **Format** string, the whole string will serve as the regular expression for the `true` value. The default values (`false|F|FALSE|NO|N|f|0|no|n`) will be the only ones that will be interpreted as `false`.

Values that match neither the **Format** regular expression (interpreted as `true` only) nor the mentioned default values for `false` will be interpreted as error. In such a case, graph would fail.

If we symbolically display the format as:

`/A/B/C/D/`

th meaning of each part is as follows:

1. If the value of the boolean field matches the pattern of the first part (A) and does not match the second part (B), it is interpreted as `true`.
2. If the value of the boolean field does not match the pattern of the first part (A), but matches the second part (B), it is interpreted as `false`.
3. If the value of the boolean field matches both the pattern of the first part (A) and, at the same time, the pattern of the second part (B), it is interpreted as `true`.
4. If the value of the boolean field matches neither the pattern of the first part (A), nor the pattern of the second part (B), it is interpreted as error. In such a case, the graph would fail.

All parts are optional, however, if any of them is omitted, all of the others that are at its right side must also be omitted.

If the second part (B) is omitted, the following default values are the only ones that are parsed as boolean `false`:

`false|F|FALSE|NO|N|f|0|no|n`

If there is not any **Format**, the following default values are the only ones that are parsed as boolean `true`:

`true|T|TRUE|YES|Y|t|1|yes|y`

- The third part (C) is a formatting string used to express boolean `true` for all matched strings. If the third part is omitted, either the `true` word is used (if the first part (A) is complicated regular expression), or the first substring from the first part is used (if the first part is a serie of simple substrings separated by pipe, e.g.: `Iagree|sure|yes|ok` - all these values would be formatted as `Iagree`).
- The fourth part (D) is a formatting string used to express boolean `false` for all matched strings. If the fourth part is omitted, either the `false` word is used (if the second part (B) is complicated regular expression), or the first substring from the second part is used (if the second part is a serie of simple substrings separated by pipe, e.g.: `Idisagree|nope|no` - all these values would be formatted as `Idisagree`).

String Format

Such string pattern is a regular expression that allows or prohibits parsing of a string.

Example 19.1. String Format

If an input file contains a string field and **Format** property is `\\w{4}` for this field, only the string whose length is 4 will be parsed.

Thus, when a **Format** property is specified for a string, **Data policy** may cause fail of the graph (if **Data policy** is `Strict`).

If **Data policy** is set to `Controlled` or `Lenient`, the records in which this string value matches the specified **Format** property are read, the others are skipped (either sent to **Console** or to the rejected port).

Locale and Locale Sensitivity

Various data types (date and time, any numeric values, strings) can be displayed, parsed, or formatted in different ways according to the **Locale** property. See Locale (p. 112) for more information.

Strings can also be influenced by **Locale sensitivity**. See Locale Sensitivity (p. 116).

Locale

Locale represents a specific geographical, political, or cultural region. An operation that requires a **locale** to perform its task is called locale-sensitive and uses the **locale** to tailor information for the user. For example, displaying a number is a locale-sensitive operation as the number should be formatted according to the customs/conventions of the native country, region, or culture of the user.

Each locale code consists of the language code and country arguments.

The language argument is a valid `ISO Language Code`. These codes are the lower-case, two-letter codes as defined by `ISO-639`.

The country argument is a valid `ISO Country Code`. These codes are the upper-case, two-letter codes as defined by `ISO-3166`.

Instead of specifying the format parameter (or together with it), you can specify the locale parameter.

- In strings, instead of setting a format for the whole date field, specify e.g. the German locale. Clover will then automatically choose the proper date format used in Germany. If the locale is not specified at all, Clover will choose the default one which is given by your system. In order to learn how to change the default locale, refer to Changing Default CloverETL Settings (p. 76)
- In numbers, on the other hand, there are cases when both the format and locale parameters are meaningful. In case of specifying the format of decimal numbers, you define the format/pattern with a decimal separator and the locale determines whether the separator is a comma or a dot. If neither the locale or format is specified, the number is converted to string using a universal technique (without checking defaultProperties). If only the format parameter is given, the default locale is used.

Example 19.2. Examples of Locale

en.US or en.GB

To get more examples of other formatting that is affected when the locale is changed see Locale-Sensitive Formatting (p. 107).

Dates, too, can have different formats in different locales (even with different countries of the same language). For instance, `March 2, 2009` (in the USA) vs. `2 March 2009` (in the UK).

List of all Locale

A complete list of the locale supported by CloverETL can be found in a separate table below. The locale format as described above is always "language.COUNTRY".

Table 19.14. List of all Locale

Locale code	Meaning
[system default]	Locale determined by your OS
ar	Arabic language
ar.AE	Arabic - United Arab Emirates
ar.BH	Arabic - Bahrain
ar.DZ	Arabic - Algeria

Locale code	Meaning
ar.EG	Arabic - Egypt
ar.IQ	Arabic - Iraq
ar.JO	Arabic - Jordan
ar.KW	Arabic - Kuwait
ar.LB	Arabic - Lebanon
ar.LY	Arabic - Lybia
ar.MA	Arabic - Morocco
ar.OM	Arabic - Oman
ar.QA	Arabic - Qatar
ar.SA	Arabic - Saudi Arabia
ar.SD	Arabic - Sudan
ar.SY	Arabic - Syrian Arab Republic
ar.TN	Arabic - Tunisia
ar.YE	Arabic - Yemen
be	Byelorussian language
be.BY	Byelorussian - Belarus
bg	Bulgarian language
bg.BG	Bulgarian - Bulgaria
ca	Catalan language
ca.ES	Catalan - Spain
cs	Czech language
cs.CZ	Czech - Czech Republic
da	Danish language
da.DK	Danish - Denmark
de	German language
de.AT	German - Austria
de.CH	German - Switzerland
de.DE	German - Germany
de.LU	German - Luxembourg
el	Greek language
el.CY	Greek - Cyprus
el.GR	Greek - Greece
en	English language
en.AU	English - Australia
en.CA	English - Canada
en.GB	English - Great Britain
en.IE	English - Ireland
en.IN	English - India
en.MT	English - Malta
en.NZ	English - New Zealand

Locale code	Meaning
en.PH	English - Philippines
en.SG	English - Singapore
en.US	English - United States
en.ZA	English - South Africa
es	Spanish language
es.AR	Spanish - Argentina
es.BO	Spanish - Bolivia
es.CL	Spanish - Chile
es.CO	Spanish - Colombia
es.CR	Spanish - Costa Rica
es.DO	Spanish - Dominican Republic
es.EC	Spanish - Ecuador
es.ES	Spanish - Spain
es.GT	Spanish - Guatemala
es.HN	Spanish - Honduras
es.MX	Spanish - Mexico
es.NI	Spanish - Nicaragua
es.PA	Spanish - Panama
es.PR	Spanish - Puerto Rico
es.PY	Spanish - Paraguay
es.US	Spanish - United States
es.UY	Spanish - Uruguay
es.VE	Spanish - Venezuela
et	Estonian language
et.EE	Estonian - Estonia
fi	Finnish language
fi.FI	Finnish - Finland
fr	French language
fr.BE	French - Belgium
fr.CA	French - Canada
fr.CH	French - Switzerland
fr.FR	French - France
fr.LU	French - Luxembourg
ga	Irish language
ga.IE	Irish - Ireland
he	Hebrew language
he.IL	Hebrew - Israel
hi.IN	Hindi - India
hr	Croatian language
hr.HR	Croatian - Croatia

Locale code	Meaning
id	Indonesian language
id.ID	Indonesian - Indonesia
is	Icelandic language
is.IS	Icelandic - Iceland
it	Italian language
it.CH	Italian - Switzerland
it.IT	Italian - Italy
iw	Hebrew language
iw.IL	Hebrew - Israel
ja	Japanese language
ja.JP	Japanese - Japan
ko	Korean languate
ko.KR	Korean - Republic of Korea
lt	Lithuanian language
lt.LT	Lithuanian language - Lithuania
lv	Latvian language
lv.LV	Latvian language - Latvia
mk	Macedonian language
mk.MK	Macedonian - The Former Yugoslav Republic of Macedonia
ms	Malay language
ms.MY	Malay - Burmese
mt	Maltese language
mt.MT	Maltese - Malta
nl	Dutch language
nl.BE	Dutch - Belgium
nl.NL	Dutch - Netherlands
no	Norwegian language
no.NO	Norwegian - Norway
pl	Polish language
pl.PL	Polish - Poland
pt	Portuguese language
pt.BR	Portuguese - Brazil
pt.PT	Portuguese - Portugal
ro	Romanian language
ro.RO	Romanian - Romany
ru	Russian language
ru.RU	Russian - Russian Federation
sk	Slovak language
sk.SK	Slovak - Slovakia
sl	Slovenian language

Locale code	Meaning
sl.SI	Slovenian - Slovenia
sq	Albanian language
sq.AL	Albanian - Albania
sr	Serbian language
sr.BA	Serbian - Bosnia and Herzegowina
sr.CS	Serbian - Serbia and Montenegro
sr.ME	Serbian - Serbia (Cyrillic, Montenegro)
sr.RS	Serbian - Serbia (Latin, Serbia)
sv	Swedish language
sv.SE	Swedish - Sweden
th	Thai language
th.TH	Thai - Thailand
tr	Turkish language
tr.TR	Turkish - Turkey
uk	Ukrainian language
uk.UA	Ukrainian - Ukraine
vi.VN	Vietnamese - Vietnam
zh	Chinese language
zh.CN	Chinese - China
zh.HK	Chinese - Hong Kong
zh.SG	Chinese - Singapore
zh.TW	Chinese - Taiwan

Locale Sensitivity

Locale sensitivity can be applied to the `string` data type only. What is more, the **Locale** has to be specified either for the field or the whole record.

Field settings override the **Locale sensitivity** specified for the whole record.

Values of **Locale sensitivity** are the following:

- `base_letter_sensitivity`

  Does not distinguish different cases of letters nor letters with diacritical marks.

- `accent_sensitivity`

  Does not distinguish different cases of letters. It distinguishes letters with diacritical marks.

- `case_sensitivity`

  Distinguishes different cases of letters and letters with diacritical marks. It does not distinguish the letter encoding ("\u00C0" equals to "A\u0300")

- `identical_sensitivity`

  Distinguishes the letter encoding ("\u00C0" equals to "A\u0300")

Autofilling Functions

The following functions are supported by most **Reader**s, except **ParallelReader**, **QuickBaseRecordReader**, and **QuickBaseQueryReader**.

The `ErrCode` and `ErrText` functions can be used only in the following components: **DBExecute**, **DBOutputTable**, **XMLExtract**.

Note a special case of `true` autofilling value in **MultiLevelReader** component.

- `default_value` - value of corresponding data type specified as the **Default** property is set if no value is read by the **Reader**.
- `global_row_count`. This function counts the records of all sources that are read by one **Reader**. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The numbering starts at 0. However, if data records are read from more data sources, the numbering goes continuously throughout all data sources. If some edge does not include such field (in **XMLExtract**, e.g.), corresponding numbers are skipped. And the numbering continues.
- `source_row_count`. This function counts the records of each source, read by one **Reader**, separately. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The records of each source file are numbered independently on the other sources. The numbering starts at 0 for each data source. If some edge does not include such field (in **XMLExtract**, e.g.), corresponding numbers are skipped. And the numbering continues.
- `metadata_row_count`. This function counts the records of all sources that are both read by one **Reader** and sent to edges with the same metadata assigned. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The numbering starts at 0. However, if data records are read from more data sources, the numbering goes continuously throughout all data sources.
- `metadata_source_row_count`. This function counts the records of each source that are both read by one **Reader** and sent to edges with the same metadata assigned. It fills the specified field of any numeric data type in the edge(s) with integer numbers sequentially. The records are numbered in the same order they are sent out through the output port(s). The records of each source file are numbered independently on the other sources. The numbering starts at 0 for each data source.
- `source_name`. This function fills the specified record fields of string data type with the name of data source from which records are read.
- `source_timestamp`. This function fills the specified record fields of date data type with the timestamp corresponding to the data source from which records are read. This function cannot be used in **DBInputTable**.
- `source_size`. This function fills the specified record fields of any numeric data type with the size of data source from which records are read. This function cannot be used in **DBInputTable**.
- `row_timestamp`. This function fills the specified record fields of date data type with the time when individual records are read.
- `reader_timestamp`. This function fills the specified record fields of date data type with the time when the reader starts reading. The value is the same for all records read by the reader.
- `ErrCode`. This function fills the specified record fields of integer data type with error codes returned by component. It can be used by **DBOutputTable** and **DBExecute** components only.
- `ErrText`. This function fills the specified record fields of string data type with error messages returned by component. It can be used by **DBOutputTable** and **DBExecute** components only.

- `sheet_name`. This function fills the specified record fields of string data type with name of the sheet of input XLS file from which data records are read. It can be used by **XLSDataReader** component only.

Internal Metadata

As mentioned above, internal metadata are part of a graph, they are contained in it and can be seen in its source tab.

Creating Internal Metadata

If you want to create internal metadata, you can do it in two ways:

- You can do it in the **Outline** pane.

  In the **Outline** pane, you can select the **Metadata** item and open the context menu by right-clicking and select the **New metadata** item there.

- You can do it in the **Graph Editor**.

  In the **Graph Editor**, you must open the context menu by right-clicking any of the edges. There you can see the **New metadata** item.

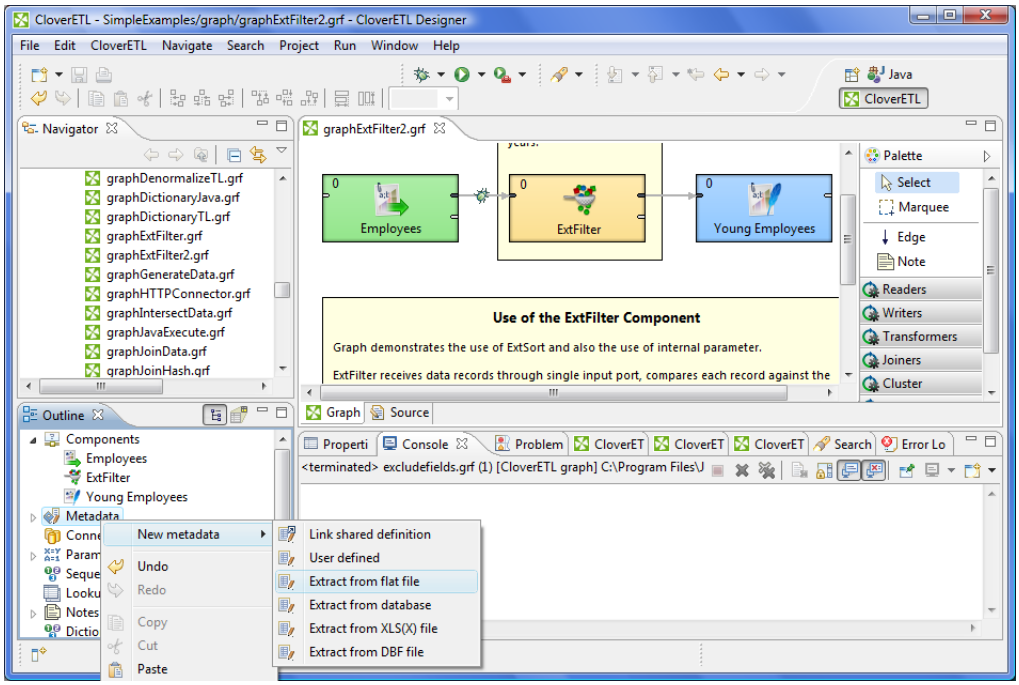


Figure 19.1. Creating Internal Metadata in the Outline Pane

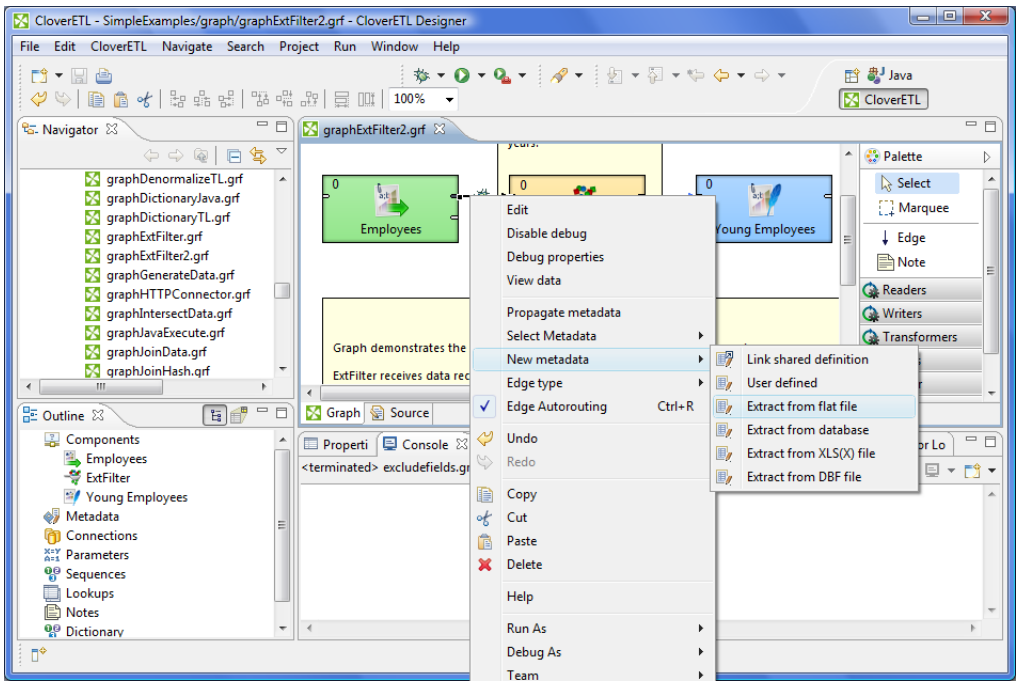


Figure 19.2. Creating Internal Metadata in the Graph Editor

In both cases, after selecting the **New metadata** item, a new submenu appears. There you can select the way how to define metadata.

Now you have three possibilities for either case mentioned above: If you want to define metadata yourself, you must select the **User defined** item or, if you want to extract metadata from a file, you must select the **Extract from flat file** or **Extract from xls(x) file** items, if you want to extract metadata from a database, you must select the **Extract from database** item. This way, you can only create internal metadata.

If you define metadata using the context menu, they are assigned to the edge as soon as they have been created.

Externalizing Internal Metadata

After you have created internal metadata as a part of a graph, you may want to convert them to external (shared) metadata. In such a case, you would be able to use the same metadata in other graphs (other graphs would share them).

You can externalize any internal metadata item into external (shared) file by right-clicking an internal metadata item in the **Outline** pane and selecting **Externalize metadata** from the context menu. After doing that, a new wizard will open in which the `meta` folder of your project is offered as the location for this new external (shared) metadata file and then you can click **OK**. If you want you can rename the offered metadata filename.

After that, the internal metadata item disappears from the **Outline** pane **Metadata** group, but, at the same location, already linked, the newly created external (shared) metadata file appears. The same metadata file appears in the `meta` subfolder of the project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal metadata items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize metadata** from the context menu. After doing that, a new wizard will open in which the `meta` folder of your project will be offered as the location for the first of the selected internal metadata items and then you can click **OK**. The same wizard will open for each the selected metadata items until they are all externalized. If you want (a file with the same name may already exist), you can change the offered metadata filename.

You can choose adjacent metadata items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired metadata items instead.

Exporting Internal Metadata

This case is somewhat similar to that of externalizing metadata. Now you create a metadata file that is outside the graph in the same way as that of externalized file, but such a file is not linked to the original graph. Only a metadata file is being created. Subsequently you can use such a file for more graphs as an external (shared) metadata file as mentioned in the previous sections.

You can export internal metadata into external (shared) one by right-clicking some of the internal metadata items in the **Outline** pane, clicking **Export metadata** from the context menu, selecting the project you want to add metadata into, expanding that project, selecting the `meta` folder, renaming the metadata file, if necessary, and clicking **Finish**.

After that, the **Outline** pane metadata folder remains the same, but in the `meta` folder in the **Navigator** pane the newly created metadata file appears.

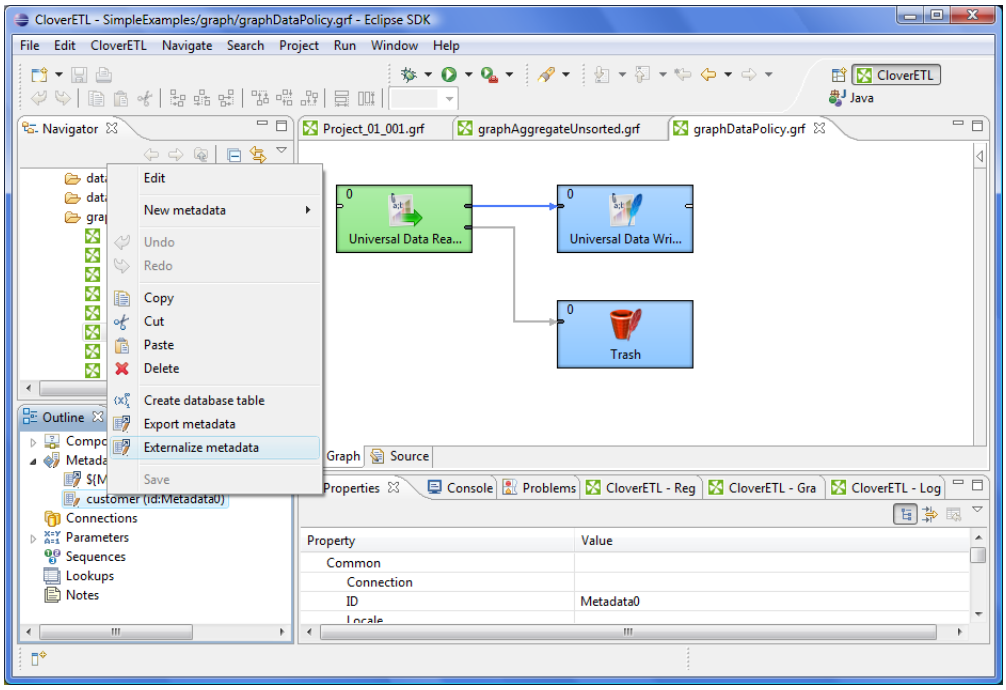


Figure 19.3. Externalizing and/or Exporting Internal Metadata

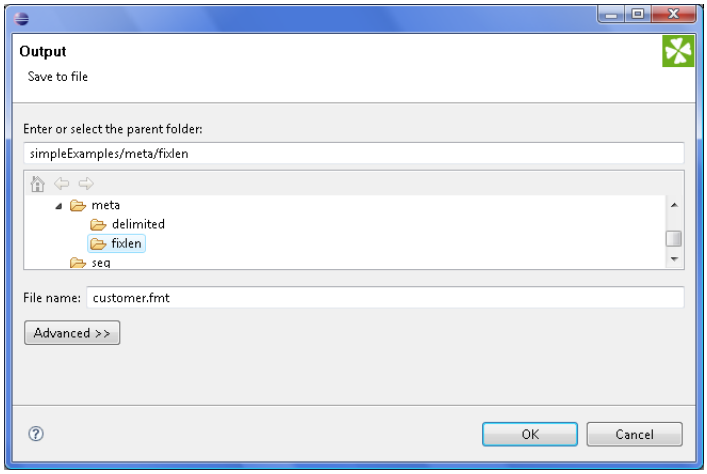


Figure 19.4. Selecting a Location for a New Externalized and/or Exported Internal Metadata

External (Shared) Metadata

As mentioned above, external (shared) metadata are metadata that serve for more graphs than only one. They are located outside the graph and can be shared across multiple graphs.

Creating External (Shared) Metadata

If you want to create shared metadata, you can do it in two ways:

- You can do it by selecting **File →New →Other** in the main menu.

  To create external (shared) metadata, after clicking the **Other** item, you must select the **CloverETL** item, expand it and decide whether you want to define metadata yourself (**User defined**), extract them from a file (**Extract from flat file** or **Extract from XLS file**), or extract them from a database (**Extract from database**).

- You can do it in the **Navigator** pane.

  To create external (shared) metadata, you can open the context menu by right-clicking, select **New →Others** from it, and after opening the list of wizards you must select the **CloverETL** item, expand it and decide whether you want to define metadata yourself (**User defined**), extract them from a file (**Extract from flat file** or **Extract from XLS file**), or extract them from a database (**Extract from database**).

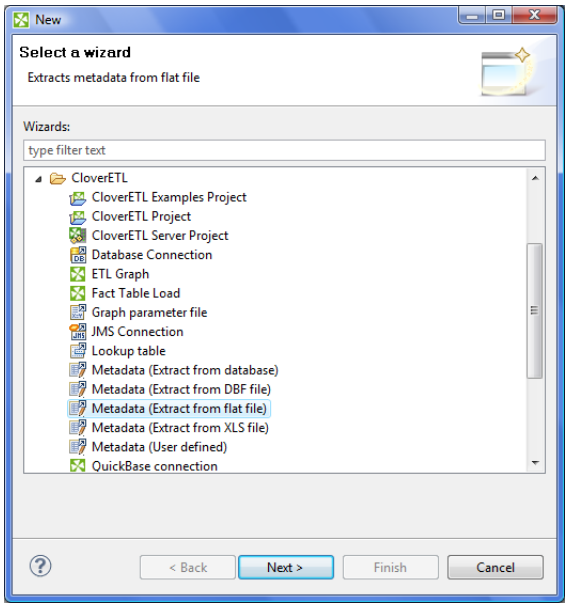


Figure 19.5. Creating External (Shared) Metadata in the Main Menu and/or in the Navigator Pane

Linking External (Shared) Metadata

After their creation (see previous sections), external (shared) metadata must be linked to each graph in which they are to be used. You need to right-click either the **Metadata** group or any of its items and select **New metadata →Link shared definition** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `meta` folder in this wizard and select the desired metadata file from all the files contained in this wizard.

You can even link multiple external (shared) metadata files at once. To do this, right-click either the **Metadata** group or any of its items and select **New metadata →Link shared definition** from the context menu. After that,

a **File selection** wizard displaying the project content will open. You must expand the `meta` folder in this wizard and select the desired metadata files from all the files contained in this wizard. You can select adjacent file items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Metadata

Once you have created and linked external (shared) metadata, in case you want to put them into the graph, you need to convert them to internal metadata. In such a case you would see their structure in the graph itself.

You can internalize any linked external (shared) metadata file by right-clicking the linked external (shared) metadata item in the **Outline** pane and clicking **Internalize metadata** from the context menu.

You can even internalize multiple linked external (shared) metadata files at once. To do this, select the desired external (shared) metadata items in the **Outline** pane. You can select adjacent items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the selected linked external (shared) metadata items disappear from the **Outline** pane **Metadata** group, but, at the same location, newly created internal metadata items appear.

The original external (shared) metadata files still exist in the `meta` subfolder and can be seen in the **Navigator** pane.

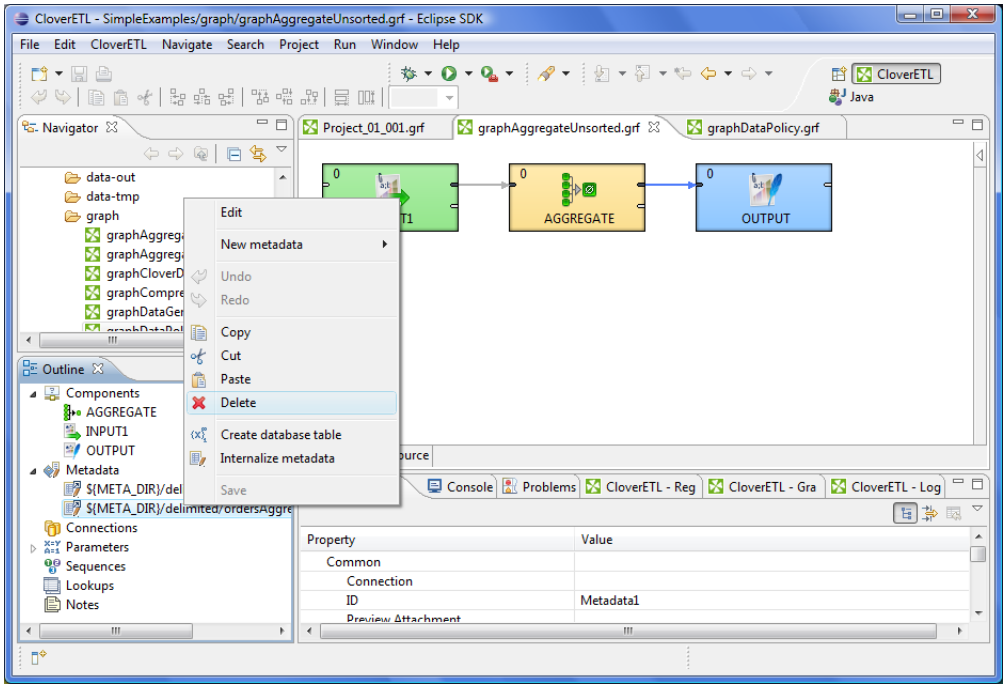


Figure 19.6. Internalizing External (Shared) Metadata

Creating Metadata

As mentioned above, metadata describe the structure of data.

Data itself can be contained in flat files, XLS files, DBF files, XML files, or database tables. You need to extract or create metadata in a different way for each of these data sources. You can also create metadata by hand.

Each description below is valid for both internal and external (shared) metadata.

Extracting Metadata from a Flat File

When you want to create metadata by extracting them from a flat file, start by clicking **Extract from flat file**. After that the **Flat file** wizard opens.

In the wizard, type the file name or locate it with the help of the **Browse...** button. Once you have selected the file, you can specify the **Encoding** and **Record type** options as well. The default **Encoding** is `ISO-8859-1` and the default **Record type** is `delimited`.

If the fields of records are separated from each other by some delimiters, you may agree with the default **Delimited** as the **Record type** option. If the fields are of some defined sizes, you need to switch to the **Fixed Length** option.

After selecting the file, its contents will be displayed in the **Input file** pane. See below:

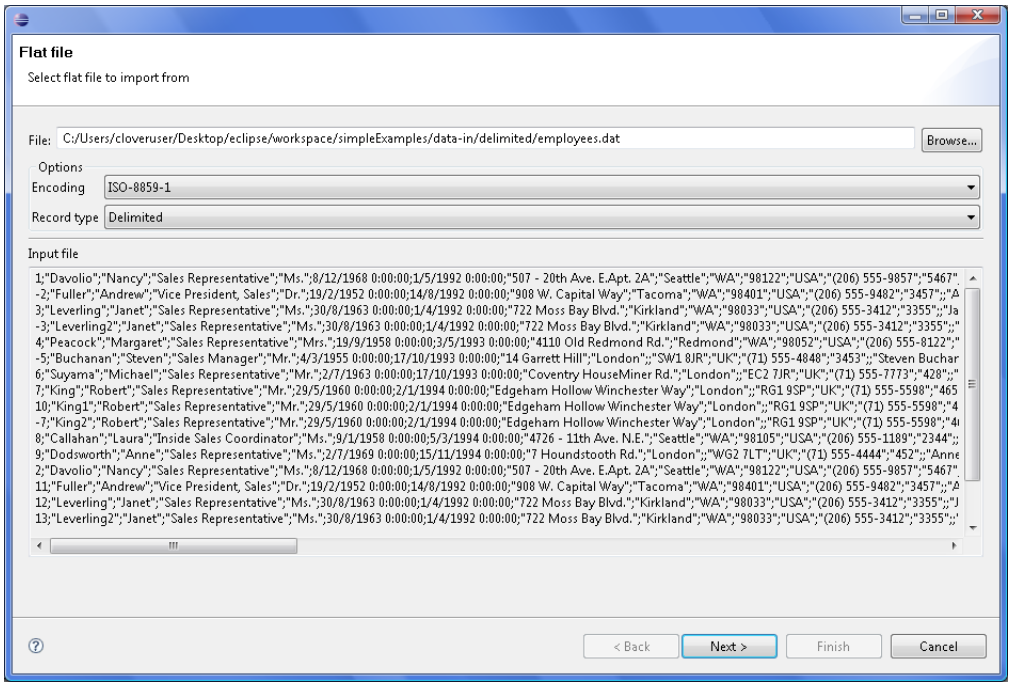


Figure 19.7. Extracting Metadata from Delimited Flat File

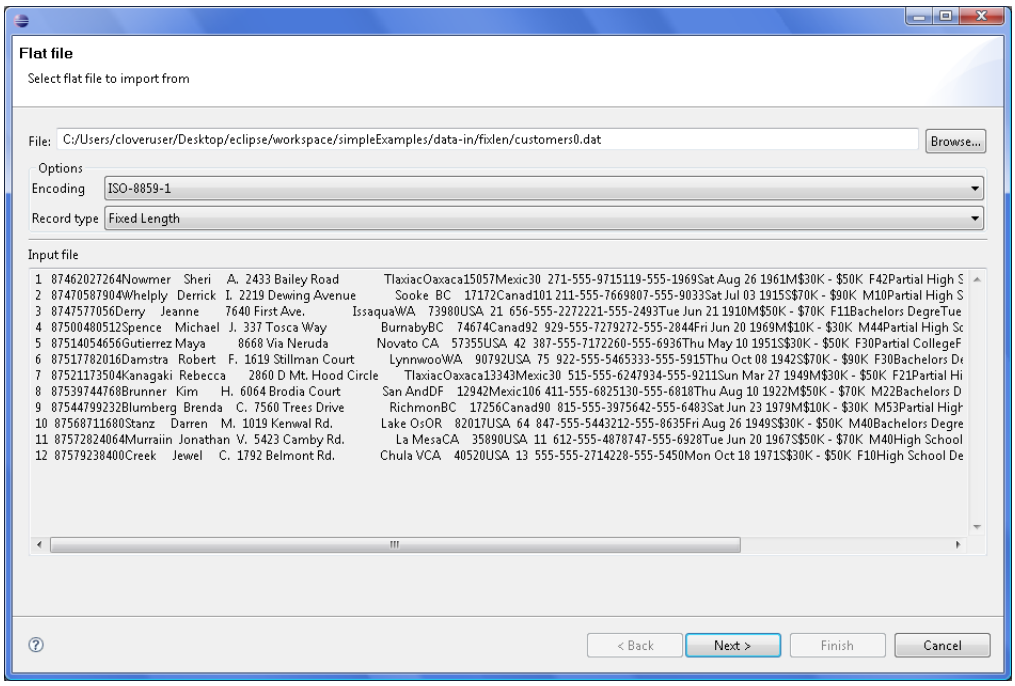


Figure 19.8. Extracting Metadata from Fixed Length Flat File

After clicking **Next**, you can see more detailed information about the content of the input file and the delimiters in the **Metadata** dialog. It consists of four panes. The first two are at the upper part of the window, the third is at the middle, the fourth is at the bottom. Each pane can be expanded to the whole window by clicking the corresponding symbol in its upper right corner.

The first two panes at the top are the panes described in Metadata Editor (p. 140). If you want to set up the metadata, you can do it in the way explained in more details in the mentioned section. You can click the symbol in the upper right corner of the pane after which the two panes expand to the whole window. The left and the right panes can be called the **Record** and the **Details** panes, respectively. In the **Record** pane, there are displayed either **Delimiters** (for delimited metadata), or **Sizes** (for fixed length metadata) of the fields or both (for mixed metadata only).

After clicking any of the fields in the **Record** pane, detailed information about the selected field or the whole record will be displayed in the **Details** pane.

Some **Properties** have default values, whereas others have not.

In this pane, you can see **Basic** properties (**Name** of the field, **Type** of the field, **Delimiter** after the field, **Size** of the field, **Nullable**, **Default** value of the field, **Skip source rows**, **Description**) and **Advanced** properties (**Format**, **Locale**, **Autofilling**, **Shift**, **EOF as delimiter**). For more details on how you can change the metadata structure see Metadata Editor (p. 140).

You can change some metadata settings in the third pane. You can specify whether the first line of the file contains the names of the record fields. If so, you need to check the **Extract names** checkbox. If you want, you can also click some column header and decide whether you want to change the name of the field (**Rename**) or the data type of the field (**Retype**). If there are no field names in the file, **CloverETL Designer** gives them the names **Field#** as the default names of the fields. By default, the type of all record fields is set to `string`. You can change this data type for any other type by selecting the right option from the presented list. These options are as follows: `boolean`, `byte`, `cbyte`, `date`, `decimal`, `integer`, `long`, `number`, `string`. For more detailed description see Data Types and Record Types (p. 97).

This third pane is different between **Delimited** and **Fixed Length** files. See:

- Extracting Metadata from Delimited Files (p. 126)
- Extracting Metadata from Fixed Length Files (p. 128)

At the bottom of the wizard, the fourth pane displays the contents of the file.

In case you are creating internal metadata, you only need to click the **Finish** button. If you are creating external (shared) metadata, you must click the offered **Next** button, then select the folder (`meta`) and name of metadata and click **Finish**. The extension `.fmt` will be added to the metadata file automatically.

Extracting Metadata from Delimited Files

If you expand the pane in the middle to the whole wizard window, you will see the following:

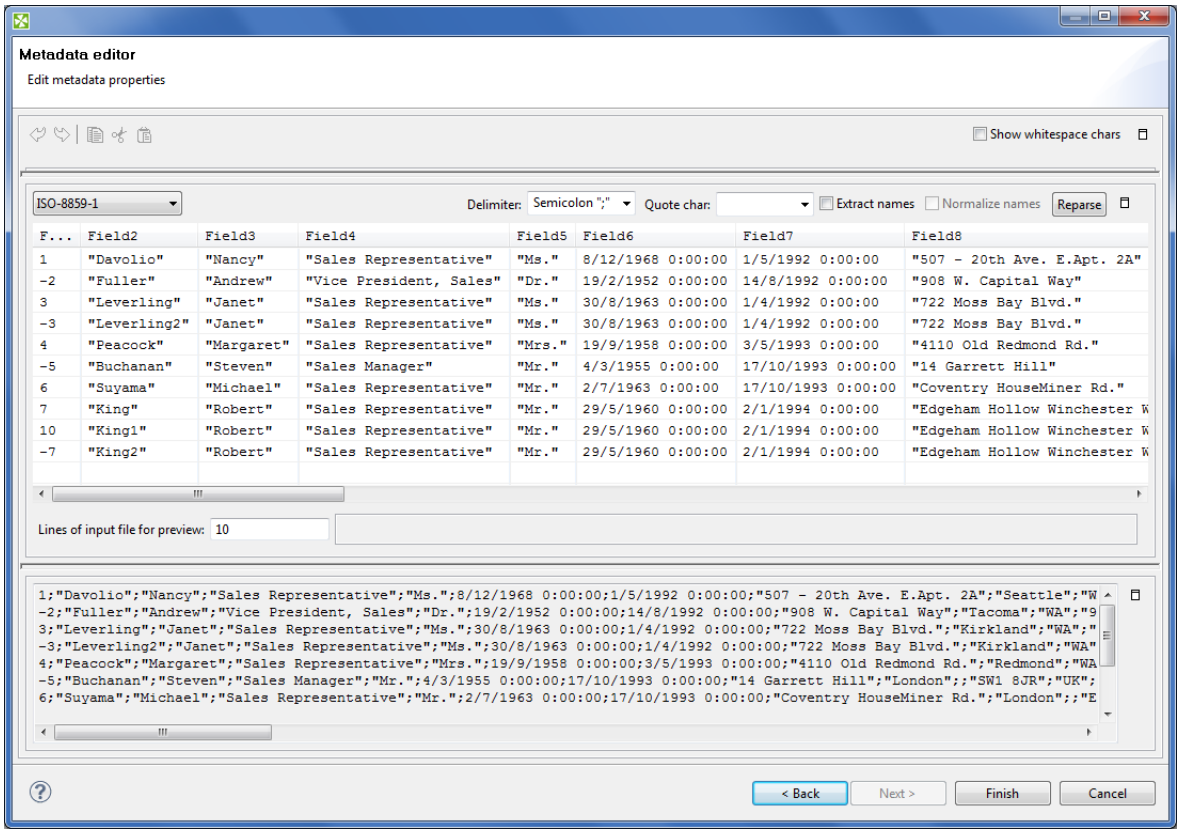


Figure 19.9. Setting Up Delimited Metadata

You may need to specify which delimiter is used in the file (**Delimiter**). The delimiter can be a comma, colon, semicolon, space, tabulator, or a sequence of characters. You need to select the right option.

Finally, click the **Reparse** button after which you will see the file as it has been parsed in the pane below.

The **Normalize names** option allows you to ged rid of invalid characters in fields. They will be replaced with the underscore character, i.e. _. This is available only with **Extract names** checked.

Alternatively, use the **Quote char** combo box to select which kind of quotation marks should be removed from string fields. Do not forget to click **Reparse** after you select one of the options: **"** or **'** or **Both " and '**. Quotation marks have to form a pair and selecting one kind of **Quote char** results in ignoring the other one (e.g. if you select " then they will be removed from each field while all ' characters are treated as common strings). If you need to retain the actual quote character in the field, it has to be escaped, e.g. "" - this will be extracted as a single ". Delimiters (selected in **Delimiter**) surrounded by quotes are ignored. What is more, you can enter your own delimiter into the combo box as a single character, e.g. the pipe - type only | (no quotes around).

Examples:

`"person"` - will be extracted as `person` (**Quote char** set to " or **Both** " **and** ')

`"address"1` - will not be extracted and the field will show an error; the reason is the delimiter is expected right after the quotes (`"address";` would be fine with ; as the delimiter)

`first"Name"` - will be extracted as `first"Name"` - if there is no quotation mark at the beginning of the field, the whole field is regarded as a common string

`"'doubleQuotes'"` (**Quote char** set to " or **Both** " **and** ') - will be extracted as `'doubleQuotes'` as only the outer quotation marks are always removed and the rest of the field is left untouched

`"unpaired` - will not be extracted as quotation marks have be in pair; this would be an error

`'delimiter;'` (with **Quote char** set to ' or **Both** " **and** ' and **Delimiter** set to ;) - will be extracted as `delimiter;` as the delimiter inside quotation marks is ignored

Extracting Metadata from Fixed Length Files

If you expand the pane in the middle to the whole wizard window, you will see the following:

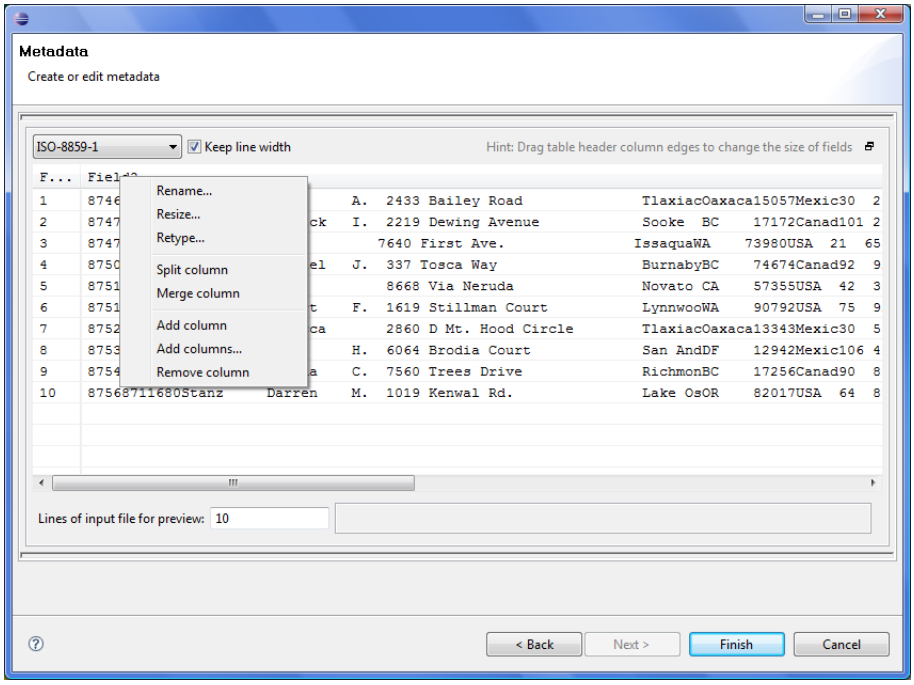


Figure 19.10. Setting Up Fixed Length Metadata

You must specify the sizes of each field (**Resize**). You may also want to split any column, merge columsn, add one or more columns, remove columns. You can change the sizes by moving the borders of the columns.

Extracting Metadata from an XLS(X) File

When you want to extract metadata from an XLS(X) file, you must select the **Extract from xls(x) file** option.

In the **Sheet properties** wizard that appears after selecting this option, you must browse and locate the desired XLS file and click the **Open** button.

After that, some properties that you can see in the wizard appear filled with some values. They are **Sheet name**, **Metadata row**, **Sample data row**, **Charset**. If they do not appear filled, you can do it yourself. Also, at the bottom, you can see data sample from the selected XLS file.

You can select the **Sheet name**. You may want to choose the charset as well.

As regards **Metadata row** and **Sample data row**: **Metadata row** is set to 1 and **Sample data row** is set to 2 by default. (**Sample data row** means the row from which data types are extracted. **Metadata row** is the row which contains the names of the fields. Together they give rise to metadata description of the file.)

If the XSL file does not contain any row with field names, you should set **Metadata row** to 0. In such a case, headers or codes of columns (letters starting from A, etc.) will serve as the names of the fields.

In case of XSL files, data types are set to their right types thanks to the **Sample data row**. Also the formats are set to the right format types.

You can also select the **Number of lines in preview**. By default it is 100.

As the last step, click either the **OK** button (when creating internal metadata), or the **Next** button, select the location (`meta`, by default) and choose some name (when creating external (shared) metadata file). The extension `.fmt` will be added to the name of metadata file automatically.

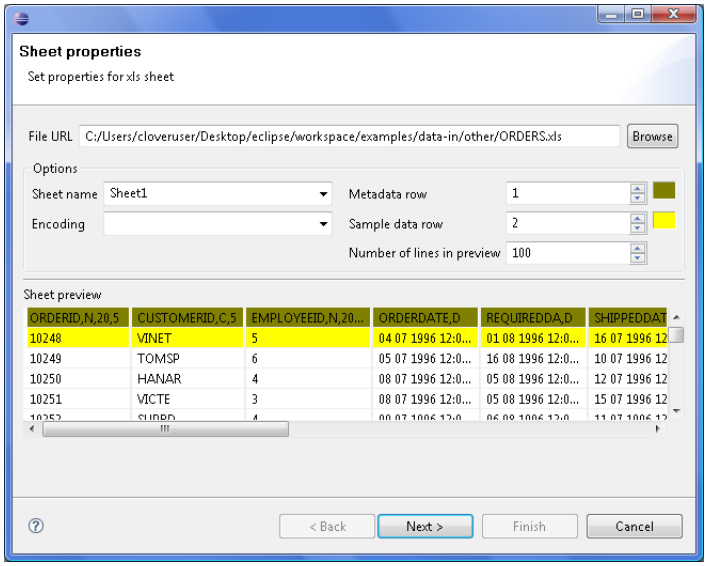


Figure 19.11. Extracting Metadata from XLS File

Extracting Metadata from a Database

If you want to extract metadata from a database (when you select the **Extract from database** option), you must have some database connection defined prior to extracting metadata.

In addition to this, if you want to extract internal metadata from a database, you can also right-click any connection item in the **Outline** pane and select **New metadata** →**Extract from database**.

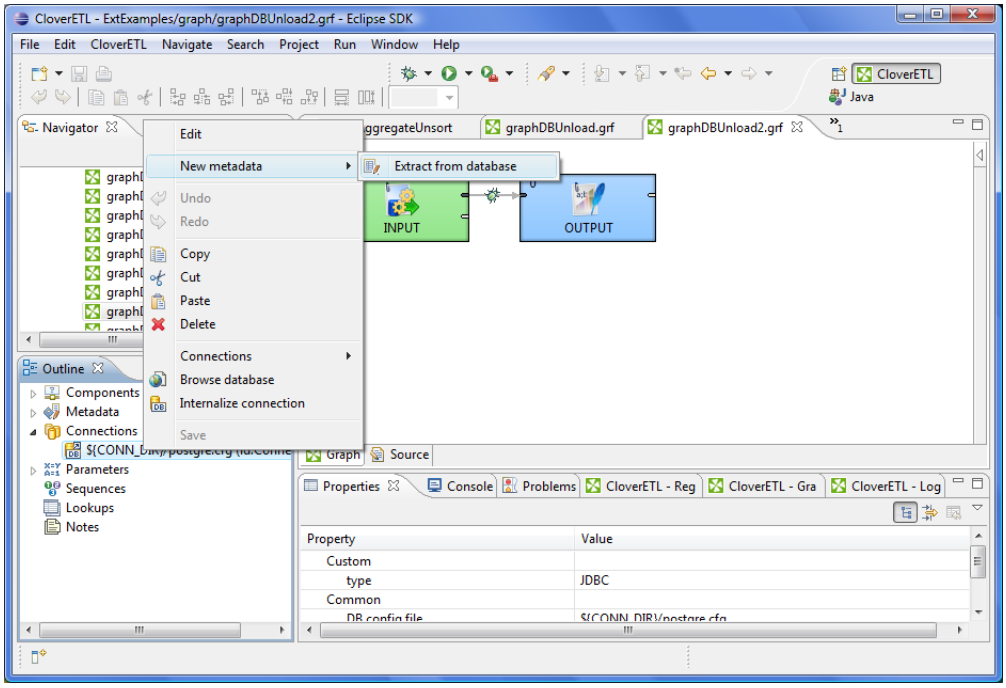


Figure 19.12. Extracting Internal Metadata from a Database

After each of these three options, a **Database Connection** wizard opens.

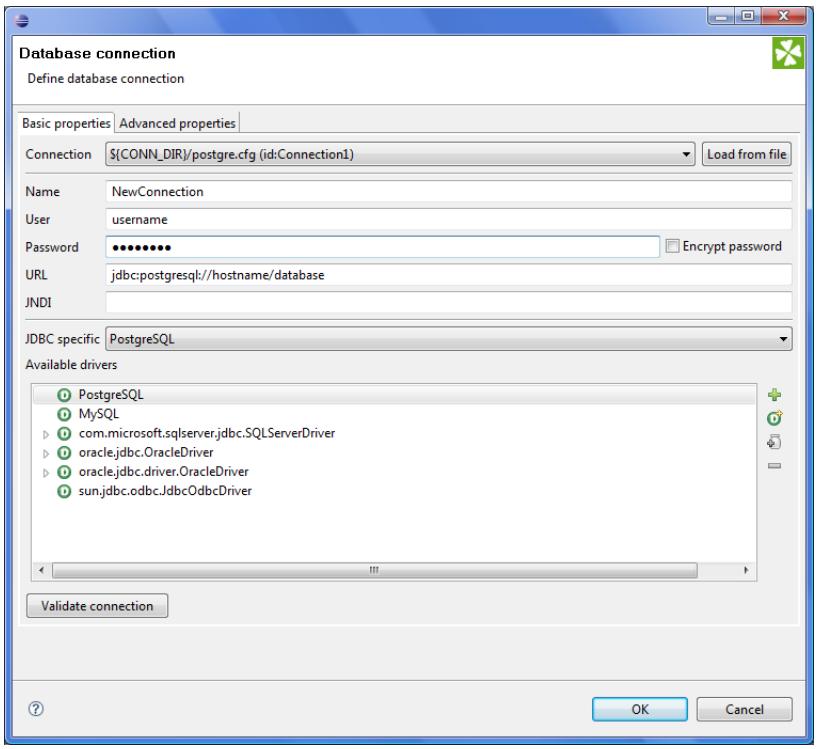


Figure 19.13. Database Connection Wizard

In order to extract metadata, you must first select database connection from the existing ones (using the **Connection** menu) or load a database connection using the **Load from file** button or create a new connection as shown in corresponding section. Once it has been defined, **Name**, **User**, **Password**, **URL** and/or **JNDI** fields become filled in the **Database Connection** wizard.

Then you must click **Next**. After that, you can see a database schema.

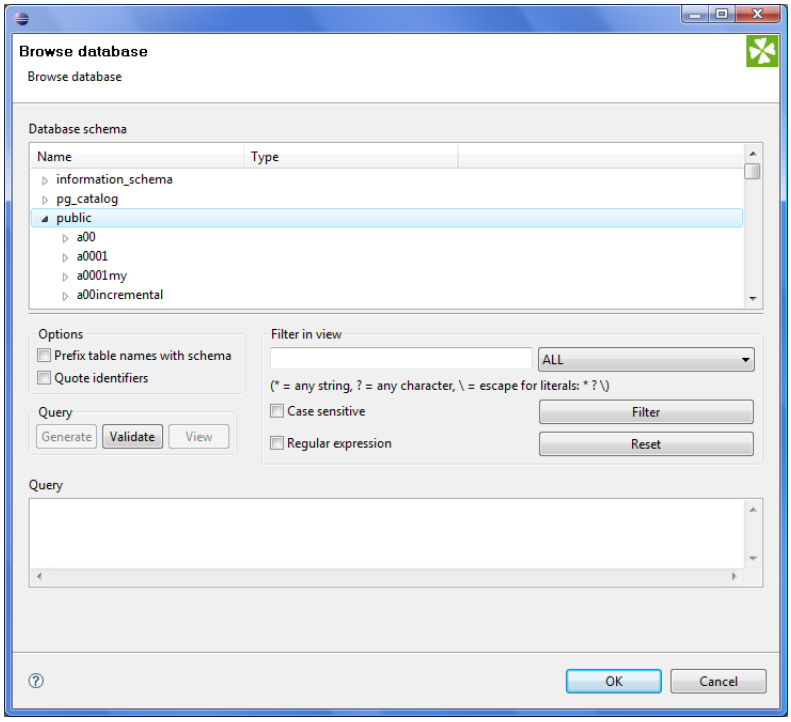


Figure 19.14. Selecting Columns for Metadata

Now you have two possibilities:

Either you write a query directly, or you generate the query by selecting individual columns of database tables.

If you want to generate the query, hold **Ctrl** on the keyboard, highlight individual columns from individual tables by clicking the mouse button and click the **Generate** button. The query will be generated automatically.

See following window:

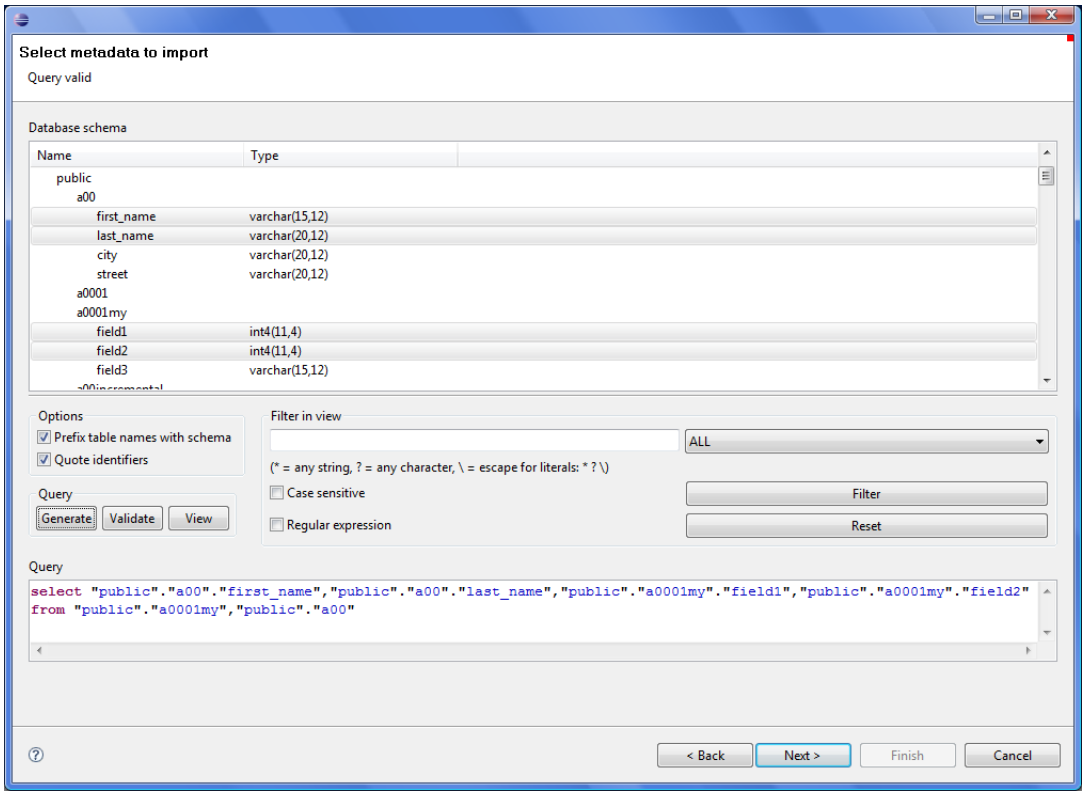


Figure 19.15. Generating a Query

If you check the **Prefix table names with schema** checkbox, it will have the following form: `schema.table.column`. If you check the **Quote identifiers** checkbox, it will look like one of this: `"schema"."table"."column"` (**Prefix table names with schema** is checked) or `"table"."column"` only (the mentioned checkbox is not checked). This query is also generated using the default (**Generic**) JDBC specific. Only it does not include quotes.

Remember that **Sybase** has another type of query which is prefixed by schema. It looks like this:

```
"schema"."dbowner"."table"."column"
```

Important

Remember that quoted identifiers may differ for different databases. They are:

- **double quotes**

  DB2, **Informix** (for **Informix**, the `DELIMIDENT` variable must be set to `yes` otherwise no quoted identifiers will be used), **Oracle**, **PostgreSQL**, **SQLite**, **Sybase**

- **back quotes**

  Infobright

- **backslash with back quotes**

MySQL (backquote is used as inline CTL special character)

- **square brackets**

  MSSQL 2008, **MSSQL 2000-2005**

- **without quotes**

  When the default (**Generic**) JDBC specific or **Derby** specific are selected for corresponding database, the generated query will not be quoted at all.

Once you have written or generated the query, you can check its validity by clicking the **Validate** button.

Then you must click **Next**. After that, **Metadata Editor** opens. In it, you must finish the extraction of metadata. If you wish to store the original database field length constraints (especially for `strings`/`varchars`), choose the **fixed** length or **mixed** record type. Such metadata provide the exact database field definition when used for creating (generating) table in a database, see Create Database Table from Metadata (p. 138)

- By clicking the **Finish** button (in case of internal metadata), you will get internal metadata in the **Outline** pane.
- On the other hand, if you wanted to extract external (shared) metadata, you must click the **Next** button first, after which you will be prompted to decide which project and which subfolder should contain your future metadata file. After expanding the project, selecting the `meta` subfolder, specifying the name of the metadata file and clicking **Finish**, it is saved into the selected location.

Extracting Metadata from a DBase File

When you want to extract metadata from a DBase file, you must select the **Extract from DBF file** option.

Locate the file from which you want to extract metadata. The file will open in the following editor:

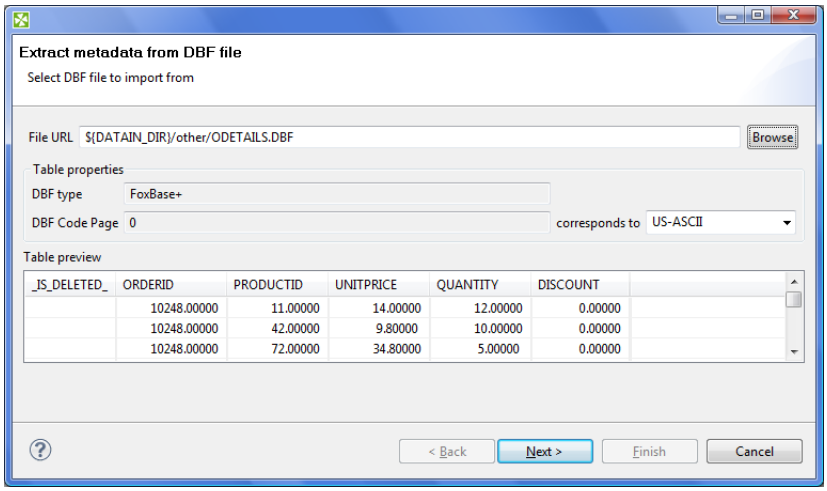


Figure 19.16. DBF Metadata Editor

DBF type, **DBF Code Page** will be selected automatically. If they do not correspond to what you want, change their values.

When you click **Next**, the **Metadata Editor** with extracted metadata will open. You can keep the default metadata values and types and click **Finish**.

Creating Metadata by User

If you want to create metadata yourself (**User defined**), you must do it in the following manner:

After opening the **Metadata Editor**, you must add a desired number of fields by clicking the plus sign, set up their names, their data types, their delimiters, their sizes, formats and all that has been described above.

For more detailed information see Metadata Editor (p. 140).

Once you have done all of that, you must click either **OK** for internal metadata, or **Next** for external (shared) metadata. In the last case, you only need to select the location (`meta`, by default) and a name for metadata file. When you click **OK**, your metadata file will be saved and the extension `.fmt` will be added to the file automatically.

Extracting Metadata from Lotus Notes

For Lotus Notes components (see LotusReader (p. 320) LotusWriter (p. 391) for further info) it is required to provide metadata for Lotus data you will be working with. The **LotusReader** component needs metadata to properly read data from Lotus views. Metadata describes how many columns there is in a view and assigns names and types to the columns. The **LotusWriter** component uses metadata to determine the types of written data fields.

Metadata can be obtained from Lotus views either as internal or external metadata. See sections Internal Metadata (p. 119) and External (Shared) Metadata (p. 122) to learn how to create internal and external metadata.

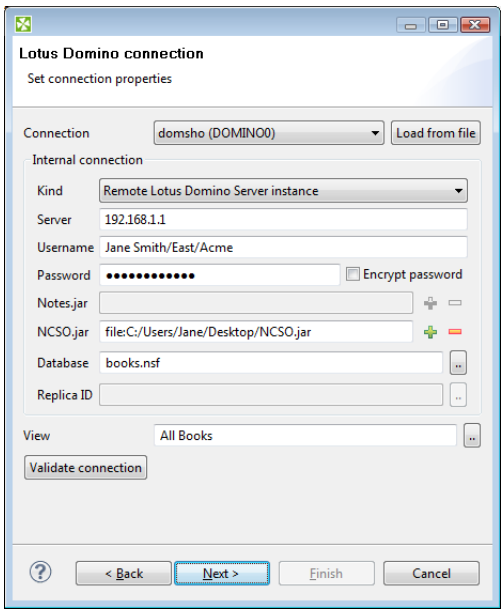


Figure 19.17. Specifying Lotus Notes connection for metadata extraction

On the first page of Lotus Notes metadata extraction Wizard, you are asked to provide details of connection to Lotus Notes or Lotus Domino server. You can either select an existing Lotus connection, load external connection by using the **Load from file** button, or define new connection by selecting **<custom>** from the **connection** menu.

See Chapter 23, Lotus Connections (p. 165) for description of connection details.

Finally, to be able to extract metadata, you need to specify the **View** from which the metadata will be extracted.

The extraction process prepares metadata with the same amount of fields as is the amount of columns in the selected View. It will also assign names to the fields based on the names of the View columns. All columns in Lotus views have internal (programmatic) names. Some columns can have user-defined names for better readability. The extraction wizard will use user-defined names where possible, in the latter case it will use the internal programmatic name of the column.

The metadata extraction process will set types of all fields to **String**. This is because Lotus View columns do not have types assigned to them. The value in a column can contain arbitrary type, for example based on certain condition or result of complex calculation.

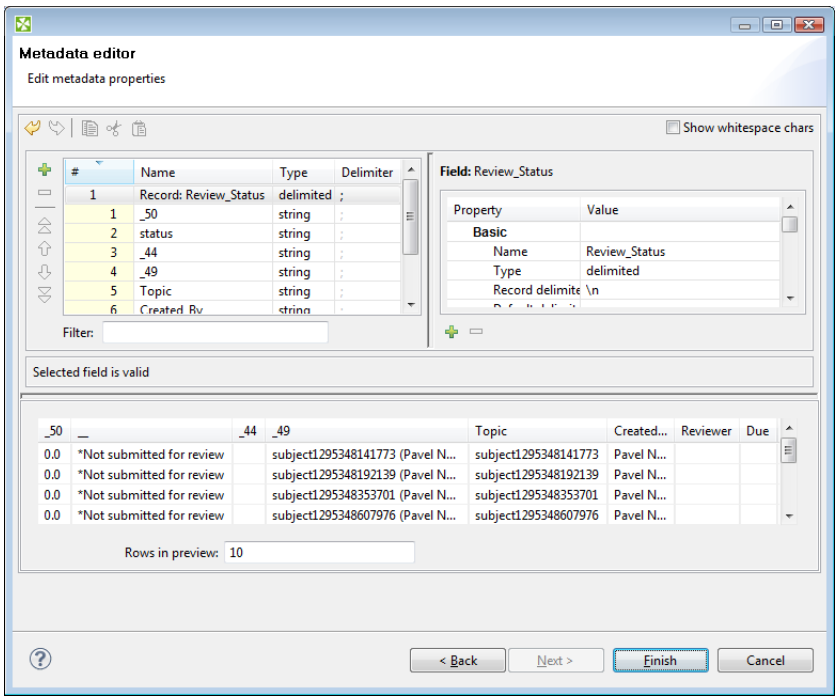


Figure 19.18. Lotus Notes metadata extraction wizard, page 2

The second page of Lotus Notes metadata extraction Wizard is separated into two parts. In the upper part there is standard Metadata editor available to customize the result of metadata extraction. In the lower part there is a preview of the data contained in the View.

On this page you can for example change the names of fields or change the types of fields from default String type to something specific. **In such case, you must be sure you can guarantee the incoming data will be convertible to selected data type.** The **LotusReader** component will always succeed converting Lotus data to strings. However, it may fail if invalid conversion is attempted. For example attemt to convert Integer to Date data type would result in a data conversion exception and the whole reading process would fail.

If you are extracting internal metadata, this was the last page of the Lotus Notes metadata extraction wizard. Clicking **Finish** will add internal metadata to the currently opened graph. In case you were extracting external metadata, on the following page you will be asked to specify the location to store the extracted metadata.

Dynamic Metadata

In addition to the metadata created or extracted using **CloverETL Designer**, you can also write metadata definition in the **Source** tab of the **Graph Editor** pane. Unlike the metadata defined in **CloverETL Designer**, such metadata written in the **Source** tab cannot be edited in **CloverETL Designer**.

To define the metadata in the **Source** tab, open this tab and write there the following:

```
<Metadata id="YourMetadataId" connection="YourConnectionToDB"
sqlQuery="YourQuery"/>
```

Specify a unique expression for `YourMetadataId` (e.g. DynamicMetadata1) and an id of a previously created DB connection that should be used to connect to DB as `YourConnectionToDB`. Type the query that will be used to extract meta data from DB as `YourQuery` (e.g. select * from myTable).

In order to speed up the metadata extraction, add the clause `"where 1=0"` or `"and 1=0"` to the query. The former one should be added to a query with no where condition and the latter clause should be added to the query which already contains `"where ..."` expression. This way only metadata are extracted and no data will be read.

Remember that such metadata are generated dynamically at runtime only. Its fields cannot be viewed or modified in metadata editor in **CloverETL Designer**.

Reading Metadata from Special Sources

In the similar way like the dynamic metadata mentioned in the previous section, another metadata definitions can also be used in the **Source** tab of the **Graph Editor** pane.

Remember that neither these metadata can be edited in **CloverETL Designer**.

In addition to the simplest form that defines external (shared) metadata (`fileURL="${META_DIR}/metadatafile.fmt"`) in the source code of the graph, you can use more complicated URLs which also define paths to other external (shared) metadata in the **Source** tab.

For example:

```
<Metadata fileURL="zip:(${META_DIR}\delimited.zip)#delimited/employees.fmt"
id="Metadata0"/>
```

or:

```
<Metadata fileURL="ftp://guest:guest@localhost:21/employees.fmt"
id="Metadata0"/>
```

Such expressions can specify the sources from which the external (shared) metadata should be loaded and linked to the graph.

Creating Database Table from Metadata and Database Connection

As the last option, you can also create a database table on the basis of metadata (both internal and external).

When you select the **Create database table** item from each of the two context menus (called out from the **Outline** pane and/or **Graph Editor**), a wizard with a SQL query that can create database table opens.

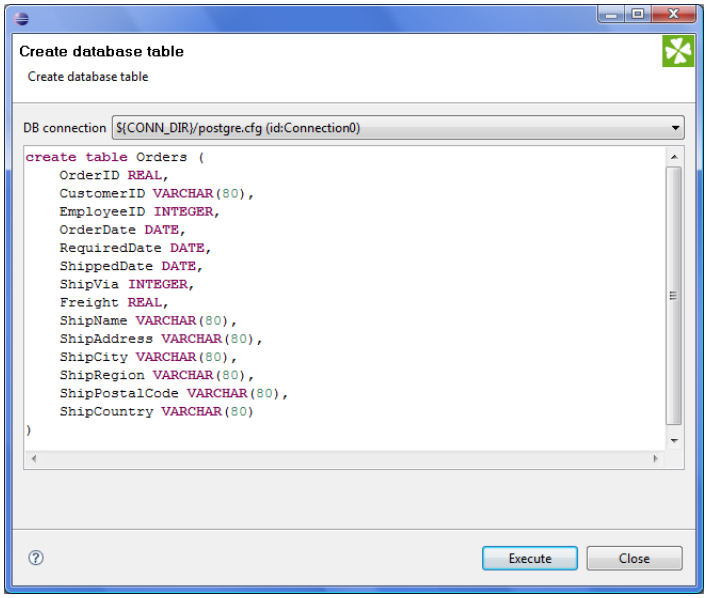


Figure 19.19. Creating Database Table from Metadata and Database Connection

You can edit the contents of this window if you want.

When you select some connection to a database. For more details see Chapter 20, Database Connections (p. 150). Such database table will be created.

On the next page we present two tables with an overview of conversion from Clover data types to SQL (database) data types.

Note

If multiple SQL types are listed, actual syntax depends on particular metadata (size for fixed-length field, length, scale, etc.).

Table 19.15. CloverETL-to-SQL Data Types Transformation Table (Part I)

DB type / Clover type	DB2 & Derby	Informix	MySQL	MSSQL 2000-2005	MSSQL 2008
boolean	SMALLINT	BOOLEAN	TINYINT(1)	BIT	BIT
byte	VARCHAR(80) FOR BIT DATA CHAR(n) FOR BIT DATA	BYTE	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)
cbyte	VARCHAR(80) FOR BIT DATA CHAR(n) FOR BIT DATA	BYTE	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)
date	TIMESTAMP DATE TIME	DATETIME YEAR TO SECOND DATE DATETIME HOUR TO SECOND	DATETIME YEAR DATE TIME	DATETIME	DATETIME DATE TIME
decimal	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)
integer	INTEGER	INTEGER	INT	INT	INT
long	BIGINT	INT8	BIGINT	BIGINT	BIGINT
number	DOUBLE	FLOAT	DOUBLE	FLOAT	FLOAT
string	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)

Table 19.16. CloverETL-to-SQL Data Types Transformation Table (Part II)

DB type / Clover type	Oracle	PostgreSQL	SQLite	Sybase	Generic
boolean	SMALLINT	BOOLEAN	BOOLEAN	BIT	BOOLEAN
byte	RAW(80) RAW(n)	BYTEA	VARBINARY(80) VARBINARY(80)	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)
cbyte	RAW(80) RAW(n)	BYTEA	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)	VARBINARY(80) BINARY(n)
date	TIMESTAMP DATE	TIMESTAMP DATE TIME	TIMESTAMP DATE TIME	DATETIME DATE TIME	TIMESTAMP DATE TIME
decimal	DECIMAL DECIMAL(p) DECIMAL(p,s)	NUMERIC NUMERIC(p) NUMERIC(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)	DECIMAL DECIMAL(p) DECIMAL(p,s)

DB type / Clover type	Oracle	PostgreSQL	SQLite	Sybase	Generic
integer	INTEGER	INTEGER	INTEGER	INT	INTEGER
long	NUMBER(11,0)	BIGINT	BIGINT	BIGINT	BIGINT
number	FLOAT	REAL	NUMERIC	FLOAT	FLOAT
string	VARCHAR2(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)	VARCHAR(80) CHAR(n)

Revised: 2010-07-30

Metadata Editor

Metadata editor is a visual tool for editing metadata.

Opening Metadata Editor

Metadata Editor opens during creation of metadata from flat file (the two upper panes), database or when you create metadata by hand.

You can also open **Metadata Editor** to edit any existing metadata.

- If you want to edit any metadata assigned to an edge (both internal and external), you can do it in the **Graph Editor** pane in one of the following ways:
  - Double-click the edge.
  - Select the edge and press **Enter**.
  - Right-click the edge and select **Edit** from the context menu.
- If you want to edit any metadata (both internal and external), you can do it after expanding the **Metadata** category in the **Outline** pane:
  - Double-click the metadata item.
  - Select the metadata item and press **Enter**.
  - Right-click the metadata item and select **Edit** from the context menu.
- If you want to edit any external (shared) metadata from any project, you can do it after expanding the **meta** subfolder in the **Navigator** pane:
  - Double-click the metadata file.
  - Select the metadata file and press **Enter**.
  - Right-click the metadata file and select **Open With →CloverETL Metadata Editor** from the context menu.

Basics of Metadata Editor

We assume that you already know how to open **Metadata Editor**. For information you can see Opening Metadata Editor (p. 140).

Here we will describe the appearance of **Metadata Editor**.

In this editor you can see buttons on the left, two panes and one filter text area:

- On the left side of the dialog, there are six buttons (down from the top) - for adding or removing fields, for moving one or more fields to top, up, down or bottom. Above these buttons, there are two arrows (for undoing and redoing, from left to right).

- The pane on the left will be called the **Record** pane.

  See Record Pane (p. 142) for more detailed information.

- That on the right will be called the **Details** pane.

  See Details Pane (p. 143) for more detailed information.

- In the **Filter** text area, you can type any expression you want to search among the fields of the **Record** pane. Note that this is case sensitive.

In the **Record** pane, you can see an overview of information about the record as a whole and also the list of its fields with delimiters, sizes or both.

The contents of the **Details** pane changes in accordance with the row selected in the **Record** pane:

- If the first row is selected, details about the record are displayed in the **Details** pane.

  See Record Details (p. 143) for more detailed information.

- If other row is selected, details about selected field are displayed in the **Details** pane.

  See Field Details (p. 144) for more detailed information.

Below you can see an example of delimited metadata and another one of fixed length metadata. Mixed metadata would be a combination of both cases. For some field names delimiter would be defined and no size would be specified, whereas for others size would be defined and no delimiter would be specified or both would be defined. To create such a metadata, you must do it by hand.

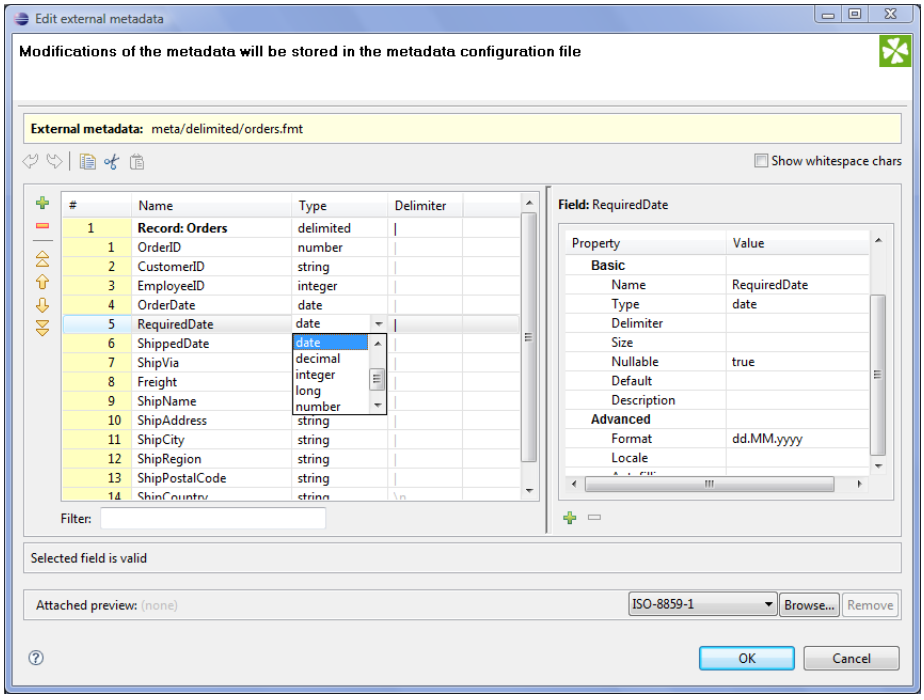


Figure 19.20. Metadata Editor for a Delimited File

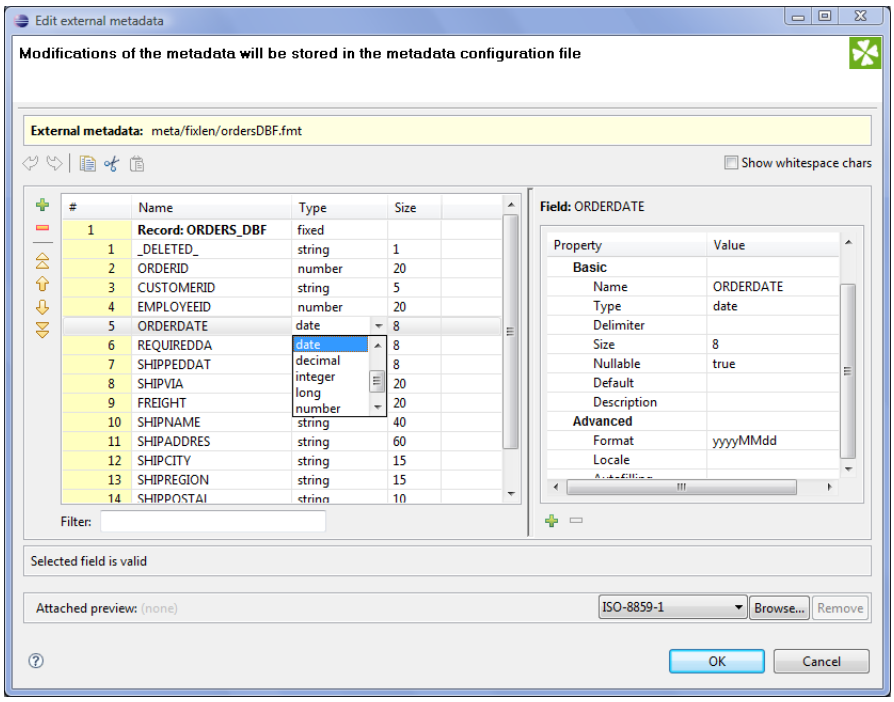


Figure 19.21. Metadata Editor for a Fixed Length File

Record Pane

This pane displays an overview of the record as a whole and all its fields:

- The first row presents an overview of the whole record:

  It consists of the following columns:

  - The name of the record is displayed in the second column and can be changed there.
  - The type of the record is displayed in the third column and can be selected as delimited, fixed or mixed.
  - The other columns display default delimiter which is used as field delimiter that follows each field (except the last one) in which no other non-default delimiter is defined, the size of the whole record or both. The delimiter is displayed greyish.

- The other rows except the last one present the list of the record fields:

  - The first column displays the number of the field. Fields are numbered starting from `1`.
  - The second column displays the name of the field. It can be changed there. We suggest you only use the following characters for the field names: `[a-zA-Z0-9_]`.
  - The third column displays the data type of the field. One of the data types for metadata can be selected. See Data Types and Record Types (p. 97) for more information.
  - The other columns display the delimiter which follows the field displayed in the row, the size of the field or both the delimiter and size. If the delimiter is displayed greyish, it is the default delimiter, if it is black, it is non-default delimiter.

- The last row presents the last field:

  - The first three columns are the same as those in other field rows.

- The other columns display record delimiter which follows the last field (if it is displayed greyish) or the non-default delimiter which follows the last field and precedes the record delimiter (if it is displayed black), the size of the field or both the delimiter and size.

For detailed information about delimiters see Changing and Defining Delimiters (p. 145).

Details Pane

The contents of the **Details** pane changes in accordance with the row selected in the **Record** pane.

- If you select the first row, details about the whole record are displayed.

  See Record Details (p. 143).

- If you select other row, details about the selected field are displayed.

  See Field Details (p. 144).

Record Details

When the **Details** pane presents information about the record as a whole, there are displayed its properties.

Basic properties are the following:

- **Name**. This is the name of the record. It can be changed there.
- **Type**. This is the type of the record. One of the following three can be selected: `delimited`, `fixed`, `mixed`. See Record Types (p. 98) for more information.
- **Record delimiter**. This is the delimiter following the last field meaning the end of the record. It can be changed there. If the delimiter in the last row of the **Record** pane in its **Delimiter** column is displayed greyish, it is this record delimiter. If it is black, it is other, non-default delimiter defined for the last field which follows it and precedes the record delimiter.

  See Changing and Defining Delimiters (p. 145) for more detailed information.

- **Record size**. Displayed for `fixed` or `mixed` record type only. This is the length of the record counted in number of characters. It can be changed there.
- **Default delimiter**. Displayed for `delimited` or `mixed` record type only. This is the delimiter following by default each field of the record except the last one. It can be changed there. This delimiter is displayed in each other row (except the last one) of the **Record** pane in its **Delimiter** column if it is greyish. If it is black, it is other, non-default delimiter defined for such a field which overrides the default one and is used instead of it.

  See Changing and Defining Delimiters (p. 145) for more detailed information.

- **Skip source rows**. This is the number of records that will be skipped for each input file. If an edge with this attribute is connected to a **Reader**, this value overrides the default value of the **Number of skipped records per source** attribute, which is `0`. If the **Number of skipped records per source** attribute is not specified, this number of records are skipped from each input file. If the attribute in the **Reader** is set to any value, it overrides this property value. Remember that these two values are not summed.
- **Description**. This property describes the meaning of the record.

Advanced properties are the following:

- **Locale**. This is the locale that is used for the whole record. This property can be useful for date formats or for decimal separator, for example. It can be overriden by the **Locale** specified for individual field.

  See Locale (p. 112) for detailed information.

- **Locale sensitivity**. Applied for the whole record. It can be overriden by the **Locale sensitivity** specified for individual field (of `string` data type).

  See Locale Sensitivity (p. 116) for detailed information.

- **Null value**. This property is set for the whole record. It is used to specify what values of fields should be processed as `null`. By default, empty field or empty string (`""`) are processed as `null`. You can set this property value to any string of characters that should be interpreted as `null`. All of the other string values remain unchanged. If you set this property to any non-empty string, empty string or empty field value will remain to be empty string (`""`).

  It can be overriden by the value of **Null value** property of individual field.

- **Preview attachment**. This is the file URL of the file attached to the metadata. It can be changed there or located using the **Browse...** button.

- **Preview Charset**. This is the charset of the file attached to the metadata. It can be changed there or by selecting from the combobox.

- **Preview Attachment Metadata Row**. This is the number of the row of the attached file where record field names are located.

- **Preview Attachment Sample Data Row**. This is the number of the row of the attached file from where field data types are guessed.

Also **Custom** properties can be defined by clicking the **Plus** sign button. For example, these properties can be the following:

- **charset**. This is the charset of the record. For example, when metadata are extracted from dBase files, these properties may be displayed.

- **dataOffset**. Displayed for `fixed` or `mixed` record type only.

Field Details

When the **Details** pane presents information about a field, there are displayed its properties.

Basic properties are the following:

- **Name**. This is the same field name as in the **Record** pane.

- **Type**. This is the same data type as in the **Record** pane.

  See Data Types and Record Types (p. 97) for more detailed information.

- **Delimiter**. This is the non-default field delimiter as in the **Record** pane. If it is empty, default delimiter is used instead.

  See Changing and Defining Delimiters (p. 145) for more detailed information.

- **Size**. This is the same size as in the **Record** pane.

- **Nullable**. This can be `true` or `false`. The default value is `true`. In such a case, the field value can be null. Otherwise, null values are prohibited and graph fails if null is met.

- **Default**. This is the default value of the field. It is used if you set the **Autofilling** property to `default_value`.

  See Autofilling Functions (p. 117) for more detailed information.

- **Length**. Displayed for `decimal` data type only. For `decimal` data types you can optionally define its length. It is the maximum number of digits in this number. The default value is `8`.

See Data Types and Record Types (p. 97) for more detailed information.

- **Scale**. Displayed for `decimal` data type only. For `decimal` data types you can optionally define scale. It is the maximum number of digits following the decimal dot. The default value is `2`.

  See Data Types and Record Types (p. 97) for more detailed information.

- **Description**. This property describes the meaning of the selected field.

Advanced properties are the following:

- **Format**. Format defining the parsing and/or the formatting of a `boolean`, `date`, `decimal`, `integer`, `long`, `number`, and `string` data field.

  See Data Formats (p. 99) for more information.

- **Locale**. This property can be useful for date formats or for decimal separator, for example. It overrides the **Locale** specified for the whole record.

  See Locale (p. 112) for detailed information.

- **Locale sensitivity**. Displayed for `string` data type only. Is applied only if **Locale** is specified for the field or the whole record. It overrides the **Locale sensitivity** specified for the whole record.

  See Locale Sensitivity (p. 116) for detailed information.

- **Null value**. This property can be set up to specify what values of fields should be processed as `null`. By default, empty field or empty string (`""`) are processed as `null`. You can set this property value to any string of characters that should be interpreted as `null`. All of the other string values remain unchanged. If you set this property to any non-empty string, empty string or empty field value will remain to be empty string (`""`).

  It overrides the value of **Null value** property of the whole record.

- **Autofilling**. If defined, field marked as `autofilling` is filled with a value by one of the functions listed in the Autofilling Functions (p. 117) section.

- **Shift**. This is the gap between the end of one field and the start of the next one when the fields are part of fixed or mixed record and their sizes are set to some value.

- **EOF as delimiter**. This can be set to true or false according to whether EOF character is used as delimiter. It can be useful when your file does not end with any other delimiter. If you did not set this property to true, run of the graph with such data file would fail (by default it is false). Displayed in delimited or mixed data records only.

Changing and Defining Delimiters

You can see the numbers in the first column of the **Record** pane of the **Metadata Editor**. These are the numbers of individual record fields. The field names corresponding to these numbers are displayed in the second column (**Name** column). The delimiters corresponding to these fields are displayed in the fourth column (**Delimiter** column) of the **Record** pane.

If the delimiter in this **Delimiter** column of the **Record** pane is greyish, this means that the default delimiter is used. If you look at the **Delimiter** row in the **Details** pane on the right side from the **Record** pane, you will see that this row is empty.

Note

Remember that the first row of the **Record** pane displays the information about the record as a whole instead of about its fields. Field numbers, field names, their types, delimiters and/or sizes are displayed starting from the second row. For this reason, if you click the first row of the **Record**

pane, information about the whole record instead of any individual field will be displayed in the **Details** pane.

You can do the following:

- **change record delimiter**

  See Changing Record Delimiter (p. 147) for more information.

- **change default delimiter**

  See Changing Default Delimiter (p. 148) for more information.

- **define other, non-default delimiter**

  See Defining Non-Default Delimiter for a Field (p. 148) for more information.

Important

- **Multiple delimiters**

  If you have records with multiple delimiters (for example: `John;Smith\30000,London|Baker Street`), you can specify default delimiter as follows:

  Type all these delimiters as a sequence separated by `\\|`. The sequence does not contain white spaces.

  For the example above there would be `,\\|;\\||\\|\\` as the default delimiter. Note that double backslashes stand for single backslash as delimiter.

  The same can be used for any other delimiter, also for record delimiter and/or non-default delimiter.

  For example, record delimiter can be the following:

  ```
  \n\\|\r\n
  ```

  Remember also that you can have delimiter as a part of field value of flat files if you set the **Quoted string** attribute of **UniversalDataReader** to `true` and surround the field containing such delimiter by quotes. For example, if you have records with comma as field delimiter, you can process the following as one field:

  ```
  "John,Smith"
  ```

- **CTL expression delimiters**

  If you need to use any non-printable delimiter, you can write it down as a CTL expression. For example, you can type the following sequence as the delimiter in your metadata:

  ```
  \u0014
  ```

  Such expressions consist of the unicode `\uxxxx` code with no quotation marks around. Please note that each backslash character '\' contained in the input data will actually be doubled when viewed. Thus, you will see "\\" in your metadata.

Important

Java-style Unicode expressions

Remember that (since version 3.0 of **CloverETL**) you can also use the Java-style Unicode expressions anyway in **CloverETL** (except in URL attributes).

You may use one or more Java-style Unicode expressions (for example, like this one): `\u0014`.

Such expressions consist of series of the `\uxxxx` codes of characters.

They may also serve as delimiter (like CTL expression shown above, without any quotes):

```
\u0014
```

Changing Record Delimiter

If you want to change the record delimiter for any other value, you can do it in the following way:

- Click the first row in the **Record** pane of the **Metadata Editor**.

After that, there will appear record properties in the **Details** pane. Among them, there will be the **Record delimiter** property. Change this delimiter for any other value.

Such new value of record delimiter will appear in the last row of the **Record** pane instead of the previous value of record delimiter. It will again be displayed greyish.

Important

Remember that if you tried to change the record delimiter by changing the value displayed in the last row of the **Record** pane, you would not change the record delimiter. This way, you would only define other delimiter following the last field and preceding the record delimiter!

Changing Default Delimiter

If you want to change the default delimiter for any other value, you can do it in one of the following two ways:

- Click any column of the first row in the **Record** pane of the **Metadata Editor**. After that, there will appear record properties in the **Details** pane. Among them, there will be the **Default delimiter** property. Change this delimiter for any other value.

  Such new value of default delimiter will appear in the rows of the **Record** pane where default delimiter has been used instead of the previous value of default delimiter. These values will again be displayed greyish.

- Click the **Delimiter** column of the first row in the **Record** pane of the **Metadata Editor**. After that, you only need to replace the value of this cell by any other value.

  Change this delimiter for any other value.

  Such new value will appear both in the **Default delimiter** row of the **Details** pane and in the rows of the **Record** pane where default delimiter has been used instead of the previous value of such default delimiter. These values will again be displayed greyish.

Defining Non-Default Delimiter for a Field

If you want to replace the default delimiter value by any other value for any of the record fields, you can do it in one of the following two ways:

- Click any column of the row of such field in the **Record** pane of the **Metadata Editor**. After that, there will appear the properties of such field in the **Details** pane. Among them, there will be the **Delimiter** property. It will be empty if default delimiter has been used. Type there any value of this property.

  Such new character(s) will override the default delimiter and will be used as the delimiter between the field in the same row and the field in the following row.

- Click the **Delimiter** column of the row of such field in the **Record** pane and replace it by any other character(s).

  Such new character(s) will override the default delimiter and will be used as the delimiter between the field in the same row and the field in the following row. Such non-default delimiter will also be displayed in the **Delimiter** row of the **Details** pane, which was empty if default delimiter had been used.

Important

Remember that if you defined any other delimiter for the last field in any of the two ways described now, such non-default delimiter would not override the record delimiter. It would only append its value to the last field of the record and would be located between the last field and before the record delimiter.

Editing Metadata in the Source Code

You can also edit metadata in the source code:

- If you want to edit **internal** metadata, their definition can be displayed in the **Source** tab of the **Graph Editor** pane.
- If you want to edit **external** metadata, right-click the metadata file item in the **Navigator** pane and select **Open With →Text Editor** from the context menu. The file contents will open in the **Graph Editor** pane.

Chapter 20. Database Connections

If you want to parse data, you need to have some sources of data. Sometimes you get data from files, in other cases from databases or other data sources.

Now we will describe how you can work with the resources that are not files. In order to work with them, you need to make a connection to such data sources. By now we will describe only how to work with databases, some of the more advanced data sources using connections will be described later.

When you want to work with databases, you can do it in two following ways: Either you have a client on your computer that connects with a database located on some server by means of some client utility . The other way is to use a JDBC driver. Now we will describe the database connections that use some JDBC drivers. The other way (client-server architecture) will be described later when we are talking about components.

As in the case of metadata, database connections can be internal or external (shared). You can create them in two ways.

Each database connection can be created as:

- **Internal**: See Internal Database Connections (p. 150).

  Internal database connection can be:

  - **Externalized**: See Externalizing Internal Database Connections (p. 151).
  - **Exported**: See Exporting Internal Database Connections (p. 152).
- **External (shared)**: See External (Shared) Database Connections (p. 153).

  External (shared) database connection can be:

  - **Linked to the graph**: See Linking External (Shared) Database Connections (p. 153).
  - **Internalized**: See Internalizing External (Shared) Database Connections (p. 153).

Database Connection Wizard is described in Database Connection Wizard (p. 154).

Access password can be encrypted. See Encrypting the Access Password (p. 157).

Database connection can serve as resource for creating metadata. See Browsing Database and Extracting Metadata from Database Tables (p. 158).

Remember that you can also create database table directly from metadata. See Create Database Table from Metadata (p. 138).

Internal Database Connections

As mentioned above about metadata, also internal database connections are part of a graph, they are contained in it and can be seen in its source tab. This property is common for all internal structures.

Creating Internal Database Connections

If you want to create an internal database connection, you must do it in the **Outline** pane by selecting the **Connections** item, right-clicking this item, selecting **Connections →Create DB connection**.

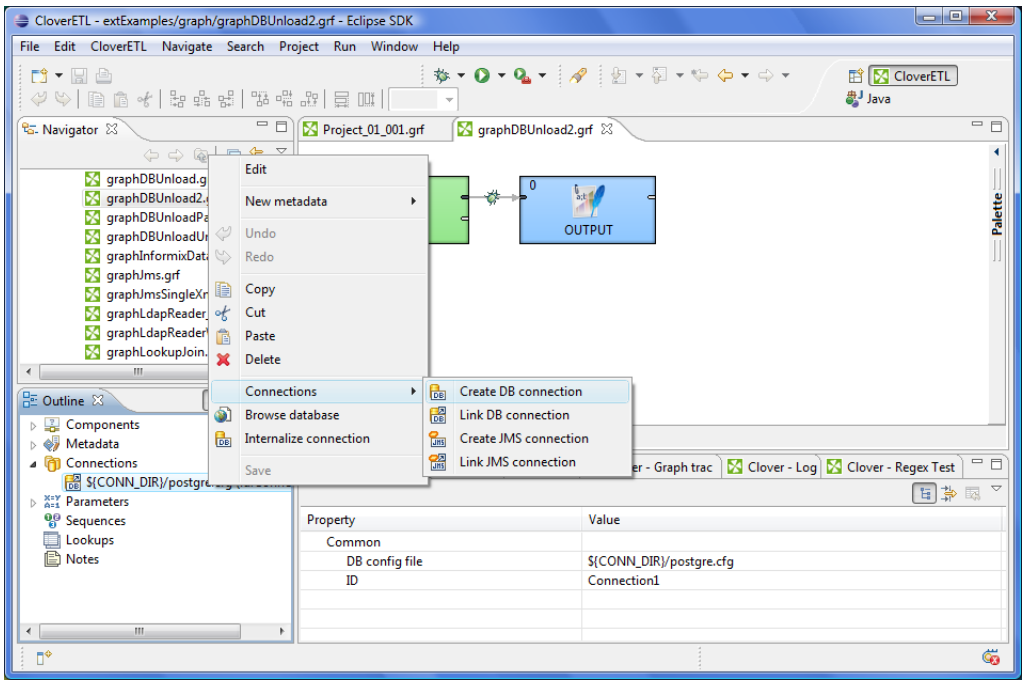


Figure 20.1. Creating Internal Database Connection

A **Database connection** wizard opens. (You can also open this wizard when selecting some DB connection item in the **Outline** pane and pressing **Enter**.)

See Database Connection Wizard (p. 154) for detailed information about how database connection should be created.

When all attributes of the connection has been set, you can validate your connection by clicking the **Validate connection** button.

After clicking **Finish**, your internal database connection has been created.

Externalizing Internal Database Connections

After you have created internal database connection as a part of a graph, you have it in your graph. Once it is contained and visible in the graph, you may want to convert it into external (shared) database connection. Thus, you would be able to use the same database connection for more graphs (more graphs would share the connection).

You can externalize any internal connection item into external (shared) file by right-clicking an internal connection item in the **Outline** pane and selecting **Externalize connection** from the context menu. After doing that, a new wizard will open in which the `conn` folder of your project is offered as the location for this new external (shared) connection configuration file and then you can click **OK**. If you want (the file with the same name may already exist), you can change the offered name of the connection configuration file.

After that, the internal connection item disappears from the **Outline** pane **Connections** group, but, at the same location, there appears already linked the newly created external (shared) connection configuration file. The same configuration file appears in the `conn` subfolder of the project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal connection items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize connection** from the context menu. After doing that, a new wizard will open in which the `conn` folder of your project will be offered as the location for the first of the selected internal connection items and then you can click **OK**. The same wizard will open for each the selected connection items until they are all externalized. If you want (the file with the same name may already exist), you can change the offered name of any connection configuration file.

You can choose adjacent connection items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

The same is valid for both database and JMS connections.

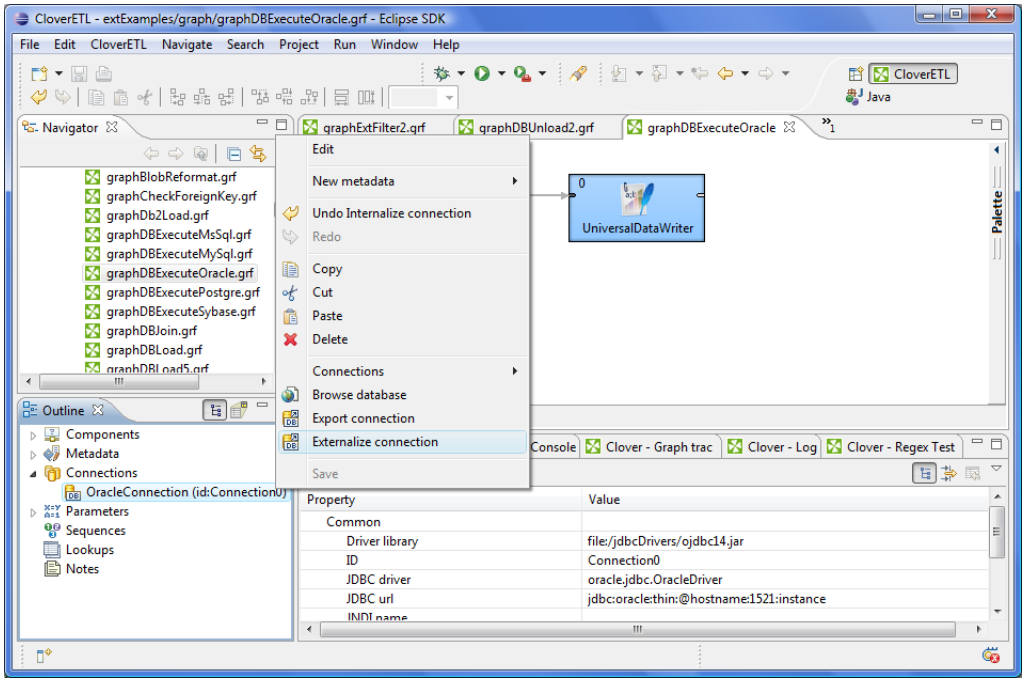


Figure 20.2. Externalizing Internal Database Connection

After that, the internal file disappears from the **Outline** pane connections folder, but, at the same location, a newly created configuration file appears.

The same configuration file appears in the `conn` subfolder in the **Navigator** pane.

Exporting Internal Database Connections

This case is somewhat similar to that of externalizing internal database connection. You create a connection configuration file that is outside the graph in the same way as an externalized connection, but such a file is no longer linked to the original graph. Subsequently you can use such a file in other graphs as an external (shared) connection configuration file as mentioned in the previous sections.

You can export an internal database connection into an external (shared) one by right-clicking one of the internal database connection items in the **Outline** pane and selecting **Export connection** from the context menu. The `conn` folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the offered and you create the file by clicking **Finish**.

After that, the **Outline** pane connection folder remains the same, but in the `conn` folder in the **Navigator** pane the newly created connection configuration file appears.

You can even export more selected internal database connections in a similar way as it is described in the previous section about externalizing.

External (Shared) Database Connections

As mentioned above, external (shared) database connections are connections that can be used in multiple graphs. They are stored outside the graphs and that is why graphs can share them.

Creating External (Shared) Database Connections

If you want to create an external (shared) database connection, you must select **File →New →Other...** from the main menu, expand the **CloverETL** category and either click the **Database Connection** item and then **Next**, or double-click the **Database Connection** item. The **Database Connection Wizard** will then open.

Then you must specify the properties of the external (shared) database connection in the same way as in the case of internal one. See Database Connection Wizard (p. 154) for detailed information about how database connections should be created.

When all properties of the connection has been set, you can validate your connection by clicking the **Validate connection** button.

After clicking **Next**, you will select the project, its `conn` subfolder, choose the name for your external database connection file, and click **Finish**.

Linking External (Shared) Database Connections

After their creation (see previous section and Database Connection Wizard (p. 154)) external (shared) database connections can be linked to each graph in which they should be used. You need to right-click either the **Connections** group or any of its items and select **Connections →Link DB connection** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `conn` folder in this wizard and select the desired connection configuration file from all the files contained in this wizard.

You can even link multiple external (shared) connection configuration files at once. To do this, right-click either the **Connections** group or any of its items and select **Connections →Link DB connection** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `conn` folder in this wizard and select the desired connection configuration files from all the files contained in this wizard. You can select adjacent file items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

The same is valid for both database and JMS connections.

Internalizing External (Shared) Database Connections

Once you have created and linked an external (shared) connection, if you want to put it into the graph, you need to convert it to an internal connection. In such a case you would see the connection structure in the graph itself.

You can convert any external (shared) connection configuration file into internal connection by right-clicking the linked external (shared) connection item in the **Outline** pane and clicking **Internalize connection** from the context menu.

You can even internalize multiple linked external (shared) connection configuration files at once. To do this, select the desired linked external (shared) connection items in the **Outline** pane. You can select adjacent items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the selected linked external (shared) connection items disappear from the **Outline** pane **Connections** group, but, at the same location, newly created internal connection items appear.

However, the original external (shared) connection configuration files still remain in the `conn` subfolder what can be seen in the **Navigator** pane.

The same is valid for both database and JMS connections.

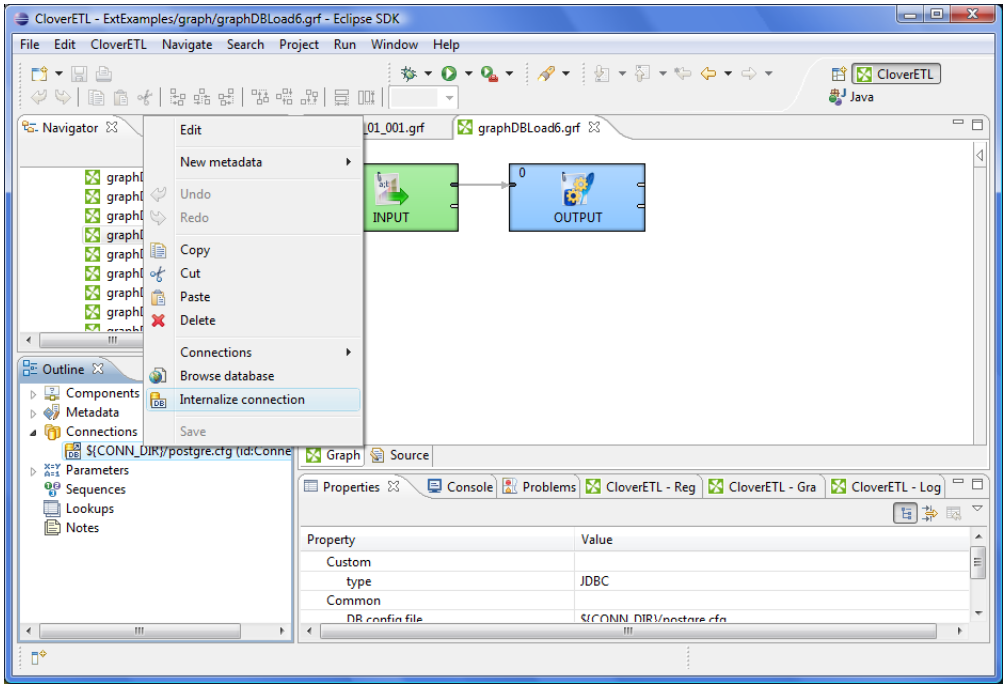


Figure 20.3. Internalizing External (Shared) Database Connection

Database Connection Wizard

This wizard consists of two tabs: **Basic properties** and **Advanced properties**

In the **Basic properties** tab of the **Database connection** wizard, you must specify the name of the connection, type your **User** name, your access **Password** and **URL** of the database connection (**hostname**, **database** name or other properties) or **JNDI**. You can also decide whether you want to encrypt the access password by checking the checkbox. You need to set the **JDBC specific** property; you can use the default one, however, it may not do all that you want. By setting **JDBC specific** you can slightly change the behaviors of the connection such as different data type conversion, getting auto-generated keys, etc.

Database connection is optimized due to this attribute. **JDBC specific** adjusts the connection for the best co-operation with the given type of database.

You can also select some built-in connections. Now the following connections are built in **CloverETL**: **Derby**, **Firebird**, **Microsoft SQL Server** (for **Microsoft SQL Server 2008** or **Microsoft SQL Server 2000-2005** specific), **MySQL**, **Oracle**, **PostgreSQL**, **Sybase**, and **SQLite**. After selecting one of them, you can see in the connection code one of the following expressions: `database="DERBY"`, `database="FIREBIRD"`, `database="MSSQL"`, `database="MYSQL"`, `database="ORACLE"`, `database="POSTGRE"`, `database="SYBASE"`, or `database="SQLITE"`, respectively.

Important

The `sun.jdbc.odbc.JdbcOdbcDriver` driver should only be used as the last resort. It is not a standard way of connecting to a database. Choose it only if other direct JDBC drivers do not work. Moreover, mind using a proper ODBC version which suits your Clover - either 32 or 64 bit.

When creating a new database connection, you can choose to use an existing one (either internal and external) that is already linked to the graph by selecting it from the **Connection** list menu. You can also load some external (non-linked) connection from connection configuration file by clicking the **Load from file** button.

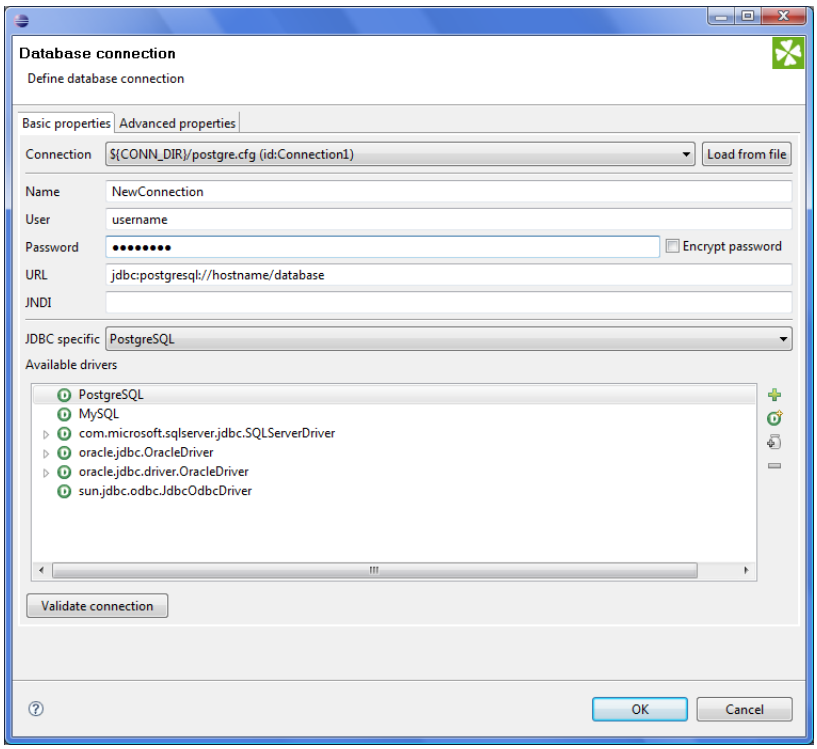


Figure 20.4. Database Connection Wizard

All attributes will be changed in a corresponding way.

If you want to use some other driver (that is not built-in), you can use one of the **Available drivers**. If the desired JDBC driver is not in the list, you can add it by clicking the **Plus** sign located on the right side of the wizard ("Load driver from JAR"). Then you can to locate the driver and confirm its selection. The result can look as follows:

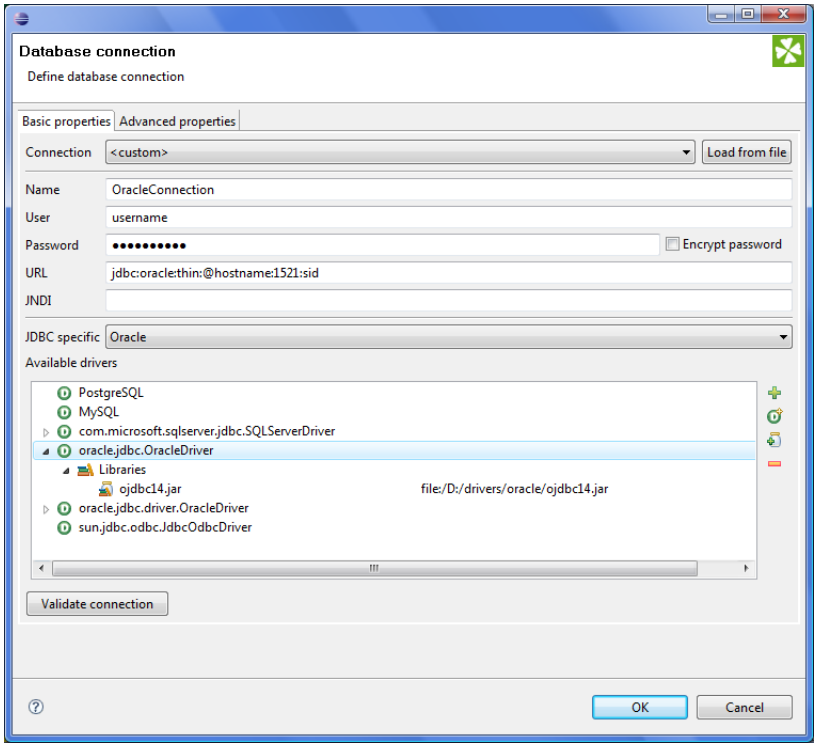


Figure 20.5. Adding a New JDBC Driver into the List of Available Drivers

If necessary, you can also add another JAR to the driver classpath (**Add JAR to driver classpath**). For example, some databases may need their license be added as well as the driver.

You can also add some property (**Add user-defined property**).

Note that you can also remove a driver from the list (**Remove selected**) by clicking the **Minus** sign.

As was mentioned already, **CloverETL** already provides following built-in **JDBC** drivers that are displayed in the list of available drivers. They are the **JDBC** drivers for **Derby**, **Firebird**, **Microsoft SQL Server 2008**, **MySQL**, **Oracle**, **PostgreSQL**, **SQLite**, and **Sybase** databases.

You can choose any **JDBC** driver from the list of available drivers. By clicking any of them, a connection string hint appears in the **URL** text area. You only need to modify the connection. You can also specify **JNDI**.

Important

Remember that **CloverETL** supports JDBC 3 drivers and higher.

Once you have selected the driver from the list, you only need to type your username and password for connecting to the database. You also need to change the "hostname" to its correct name. You must also type the right database name instead of the "database" filler word. Some other drivers provide different URLs that must be changed in a different way. You can also load an existing connection from one of the existing configuration files. You can set up the **JDBC specific** property, or use the default one, however, it may not do all that you want. By setting **JDBC specific** you can slightly change the selected connection behavior such as different data type conversion, getting auto-generated keys, etc.

Database connections are optimized based on this attribute. **JDBC specific** adjusts the connection for the best co-operation with the given type of database.

In addition to the **Basic properties** tab desribed above, the **Database connection** wizard also offers the **Advanced properties** tab. If you switch to this tab, you can specify some other properties of the selected connection:

- **threadSafeConnection**

  By default, it is set to `true`. In this default setting, each thread gets its own connection so as to prevent problems when more components converse with DB through the same connection object which is not thread safe.

- **transactionIsolation**

  Allows to specify certain transaction isolation level. More details can be found here: http://java.sun.com/j2se/1.6.0/docs/api/java/sql/Connection.html. Possible values of this attribute are the following numbers:

  - `0` (`TRANSACTION_NONE`).

    A constant indicating that transactions are not supported.

  - `1` (`TRANSACTION_READ_UNCOMMITTED`).

    A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur. This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

    This is the default value for **DB2**, **Derby**, **Informix**, **MySQL**, **MS SQL Server 2008**, **MS SQL Server 2000-2005**, **PostgreSQL**, and **SQLite** specifics.

    This value is also used as default when **JDBC specific** called **Generic** is used.

  - `2` (`TRANSACTION_READ_COMMITTED`).

    A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur. This level only prohibits a transaction from reading a row with uncommitted changes in it.

This is the default value for **Oracle** and **Sybase** specifics.

- `4` (`TRANSACTION_REPEATABLE_READ`).

  A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur. This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

- `8` (`TRANSACTION_SERIALIZABLE`).

  A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented. This level includes the prohibitions in `TRANSACTION_REPEATABLE_READ` and further prohibits the situation where one transaction reads all rows that satisfy a "`where`" condition, a second transaction inserts a row that satisfies that "`where`" condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

- **holdability**

  Allows to specify holdability of `ResultSet` objects created using the `Connection`. More details can be found here: http://java.sun.com/j2se/1.6.0/docs/api/java/sql/ResultSet.html. Possible options are the following:

  - `1` (`HOLD_CURSORS_OVER_COMMIT`).

    The constant indicating that `ResultSet` objects should not be closed when the method `Connection.commit` is called

    This is the default value for **Informix** and **MS SQL Server 2008** specifics.

  - `2` (`CLOSE_CURSORS_AT_COMMIT`).

    The constant indicating that `ResultSet` objects should be closed when the method `Connection.commit` is called.

    This is the default value for **DB2**, **Derby**, **MS SQL Server 2000-2005**, **MySQL**, **Oracle**, **PostgreSQL**, **SQLite**, and **Sybase** specifics.

    This value is also used as default when **JDBC specific** called **Generic** is used.

Encrypting the Access Password

If you do not encrypt your access password, it remains stored and visible in the configuration file (shared connection) or in the graph itself (internal connection). Thus, the access password can be visable in either of these two locations.

Of course, this would not present any problem if you were the only one who had access to your graph and computer. But if this is not the case then it would be wise to encrypt it, since the password allows access to the database in question.

So, in case you want and need to give someone any of your graphs, you need not give him or her the access password to the whole database. This is why it is possible to encrypt your access password. Without this option, you would be at great risk of some intrusion into your database or of some other damage from whoever who could get this access password.

Thus, it is important and possible that you give him or her the graph with the access password encrypted. This way, they would not be able to simply extract your password.

In order to hide your access password, you must select the **Encrypt password** checkbox in the **Database connection** wizard, typing a new (encrypting) password to encrypt the original (now encrypted) access password and finally clicking the **Finish** button.

This setting will prevent you from running the graph by choosing **Run as →CloverETL graph**. To run the graph, you must use the **Run Configurations** wizard. There, in the **Main** tab, you must type or find by browsing, the name of the project, the graph name, and parameter file. Then, type in the **Password** text area the encrypting password. The access password cannot be read now, it has been already encrypted and cannot be seen either in the configuration file or the graph.

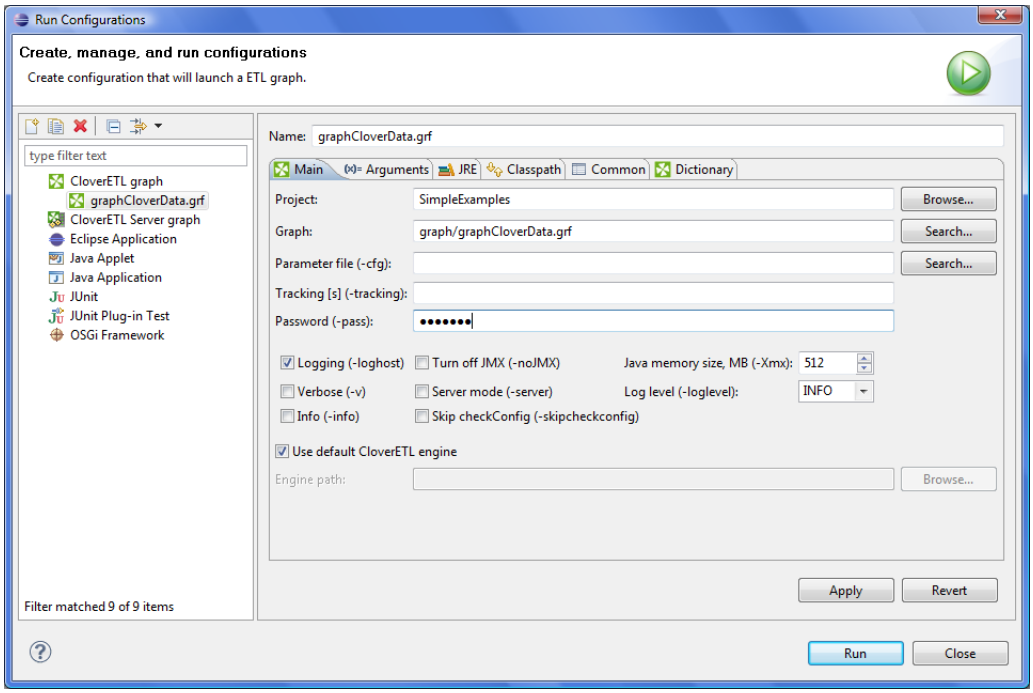


Figure 20.6. Running a Graph with the Password Encrypted

If you should want to return to your access password, you can do it by typing the encrypting password into the **Database connection** wizard and clicking **Finish**.

Browsing Database and Extracting Metadata from Database Tables

As you could see above (see Externalizing Internal Database Connections (p. 151) and Internalizing External (Shared) Database Connections (p. 153)), in both of these cases the context menu contains two interesting items: the **Browse database** and **New metadata** items. These give you the opportunity to browse a database (if your connection is valid) and/or extract metadata from some selected database table. Such metadata will be internal only, but you can later externalize and/or export them.

Important

Remember that you can also create a database table directly from metadata. See Create Database Table from Metadata (p. 138).

Chapter 21. JMS Connections

For receiving JMS messages you need JMS connections. Like metadata, parameters and database connections, these can also be internal or external (shared).

Each JMS connection can be created as:

- **Internal**: See Internal JMS Connections (p. 159).

  Internal JMS connection can be:

  - **Externalized**: See Externalizing Internal JMS Connections (p. 159).
  - **Exported**: See Exporting Internal JMS Connections (p. 160).
- **External (shared)**: See External (Shared) JMS Connections (p. 161).

  External (shared) JMS connection can be:

  - **Linked to the graph**: See Linking External (Shared) JMS Connection (p. 161).
  - **Internalized**: See Internalizing External (Shared) JMS Connections (p. 161).

Edit JMS Connection Wizard is described in Edit JMS Connection Wizard (p. 162).

Authentication password can be encrypted. See Encrypting the Authentication Password (p. 163).

Internal JMS Connections

As mentioned above in case for other tools (metadata, database connections and parameters), also internal JMS connections are part of a graph, they are contained in it and can be seen in its source tab. This property is common for all internal structures.

Creating Internal JMS Connections

If you want to create an internal JMS connection, you must do it in the **Outline** pane by selecting the **Connections** item, right-clicking this item, selecting **Connections →Create JMS connection**. An **Edit JMS connection** wizard opens. You can define the JMS connection in this wizard. Its appearance and the way how you must set up the connection are described in Edit JMS Connection Wizard (p. 162).

Externalizing Internal JMS Connections

Once you have created internal JMS connection as a part of a graph, you may want to convert it into external (shared) JMS connection. This gives you the ability to use the same JMS connection across multiple graphs.

You can externalize any internal connection item into an external (shared) file by right-clicking an internal connection item in the **Outline** pane and selecting **Externalize connection** from the context menu. After doing that, a new wizard will open in which the `conn` folder of your project is offered as the location for this new external (shared) connection configuration file and then you can click **OK**. If you want (a file with the same name may already exist), you can change the suggested name of the connection configuration file.

After that, the internal connection item disappears from the **Outline** pane **Connections** group, but, at the same location, there appears, already linked, the newly created external (shared) connection. The same configuration file appears in the `conn` subfolder of the project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal connection items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize connection** from the context menu. After doing that, a new wizard will

open in which the conn folder of your project will be offered as the location for the first of the selected internal connection items and then you can click **OK**. The same wizard will open for each of the selected connection items until they are all externalized. If you want (a file with the same name may already exist), you can change the suggested name of any connection configuration file.

You can choose adjacent connection items when you press **Shift** and move the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

The same is valid for both database and JMS connections.

Exporting Internal JMS Connections

This case is somewhat similar to that of externalizing internal JMS connection. But, while you create a connection configuration file that is outside the graph in the same way as externalizing, the file is not linked to the original graph. Only the connection configuration file is being created. Subsequently you can use such a file for more graphs as an external (shared) connection configuration file as mentioned in the previous sections.

You can export internal JMS connection into external (shared) one by right-clicking one of the internal JMS connection items in the **Outline** pane and clicking **Export connection** from the context menu. The conn folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the offered and you create the file by clicking **Finish**.

After that, the **Outline** pane connection folder remains the same, but in the conn folder in the **Navigator** pane the newly created connection configuration file appears.

You can export multiple selected internal JMS connections in a similar way to how it is described in the previous section about externalizing.

External (Shared) JMS Connections

As mentioned above, external (shared) JMS connections are connections that are usable across multiple graphs. They are stored outside the graph and that is why they can be shared.

Creating External (Shared) JMS Connections

If you want to create an external (shared) JMS connection, you must select **File** →**New** →**Other...**, expand the **CloverETL** item and either click the **JMS connection** item and then **Next**, or double-click the **JMS Connection** item. An **Edit JMS connection** wizard opens. See Edit JMS Connection Wizard (p. 162).

When all properties of the connection has been set, you can validate your connection by clicking the **Validate connection** button.

After clicking **Next**, you will select the project, its `conn` subfolder, choose the name for your external JMS connection file, and click **Finish**.

Linking External (Shared) JMS Connection

After their creation (see previous section and Edit JMS Connection Wizard (p. 162)), external (shared) connections can be linked to any graph that you want them to be used in. You simply need to right-click either the **Connections** group or any of its items and select **Connections** →**Link JMS connection** from the context menu. After that, a **File selection** wizard, displaying the project content, will open. You must expand the `conn` folder in this wizard and select the desired connection configuration file.

You can link multiple external (shared) connection configuration files at once. To do this, right-click either the **Connections** group or any of its items and select **Connections** →**Link JMS connection** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `conn` folder in this wizard and select the desired connection configuration files. You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

The same is valid for both database and JMS connections.

Internalizing External (Shared) JMS Connections

Once you have created and linked external (shared) connection, in case you want to put it into the graph, you need to convert it to an internal connection. In such a case you would see the connection structure in the graph itself.

You can internalize any external (shared) connection configuration file into internal connection by right-clicking such linked external (shared) connection item in the **Outline** pane and clicking **Internalize connection** from the context menu.

You can even internalize multiple linked external (shared) connection configuration files at once. To do this, select the desired linked external (shared) connection items in the **Outline** pane. You can select adjacent items when you press **Shift** and then the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the selected linked external (shared) connection items disappear from the **Outline** pane **Connections** group, but, at the same location, newly created internal connection items appear.

However, the original external (shared) connection configuration files still remain to exist in the `conn` subfolder what can be seen in the **Navigator** pane.

The same is valid for both database and JMS connections.

Edit JMS Connection Wizard

As you can see, the **Edit JMS connection** wizard contains eight text areas that must be filled by: **Name**, **Initial context factory class** (fully qualified name of the factory class creating the initial context), **Libraries**, **URL**, **Connection factory JNDI name** (implements `javax.jms.ConnectionFactory` interface), **Destination JNDI** (implements `javax.jms.Destination` interface), **User**, **Password** (password to receive and/or produce the messages).

(You can also open this wizard when selecting some JMS connection item in the **Outline** pane and pressing **Enter**.)

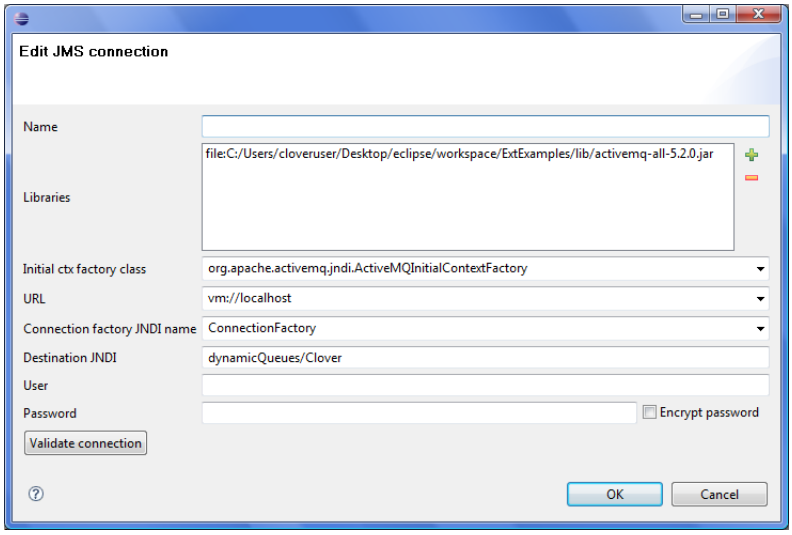


Figure 21.1. Edit JMS Connection Wizard

In the **Edit JMS connection** wizard, you must specify the **name** of the connection, select necessary **libraries** (you can add them by clicking the **plus** button), specify **Initial context factory class** (fully qualified name of the factory class creating the initial context), **URL** of the connection, **Connection factory JNDI name** (implements `javax.jms.ConnectionFactory` interface), **Destination JNDI name** (implements `javax.jms.Destination` interface), your authentication username (**User**) and your authentication password (**Password**). You can also decide whether you want to encrypt this authentication password. This can be done by checking the **Encrypt password** checkbox. If you are creating the external (shared) JMS connection, you must select a filename for this external (shared) JMS connection and its location.

Encrypting the Authentication Password

If you do not encrypt your authentication password, it remains stored and visible in the configuration file (shared connection) or in the graph itself (internal connection). Thus, the authentication password could be seen in one of these two locations.

Of course, this would not present any problem if you were the only one who had access to your graph or computer. But if this is not the case then you would be wise to encrypt your password since it provides access to your database.

So, in case you want or need to give someone any of your graphs, you likely rather not give him or her the authentication password. This is the reason why it is important to encrypt your authentication password. Without doing so, you would be at great risk of some intrusion actions or other damage from whoever who could get this authentication password.

Thus, it is important and possible that you give him or her the graph with the authentication password encrypted. This way, no person would be able to receive and/or produce the messages without your permission.

In order to hide your authentication password, you must select **Encrypt password** by checking the checkbox in the **Edit JMS connection** wizard, typing a new (encrypting) password to encrypt the original (now encrypted) authentication password and clicking the **Finish** button.

You will no longer be able to run the graph by choosing **Run as →CloverETL graph** if you encrypt the password. Instead, to run the graph, you must use the **Run Configurations** wizard. There, in the **Main** tab, you must type or find by browsing the name of the project, its graph name, its parameter file and, most importantly, type the encrypting password in the **Password** text area. The authentication password cannot be read now, it has been already encrypted and cannot be seen either in the configuration file or the graph.

If you should want to return to your authentication password, you can do it by typing the encrypting password into the **JMS connection** wizard and clicking **Finish**.

Chapter 22. QuickBase Connections

To work with a QuickBase database, use the QuickBase connection wizard to define connection parameters first.

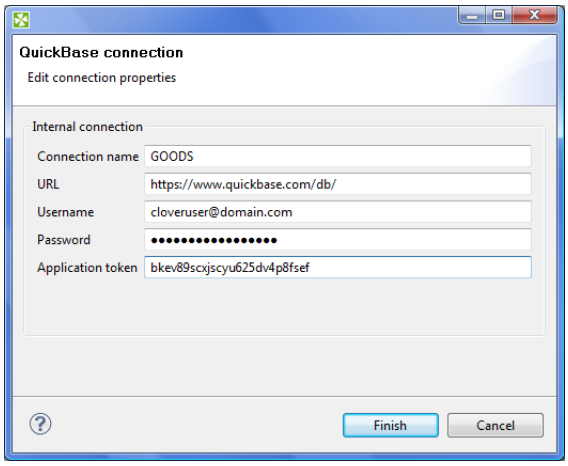


Figure 22.1. QuickBase Connection Dialog

Give a name to the connection (**Connection name**) and select the proper URL. Defaultly, your QuickBase database allows only SSL access via API.

As the **Username** fill in the *Email Address* or the *Screen Name* of your QuickBase User Profile. The required **Password** relates to the user account.

Application token is a string of characters that can be created and assigned to the database. Tokens make it all but impossible for an unauthorized person to connect to your database.

Chapter 23. Lotus Connections

To work with Lotus databases a Lotus Domino connection needs to be specified first. Lotus Domino connections can be created as both internal and external. See sections Creating Internal Database Connections (p. 150) and Creating External (Shared) Database Connections (p. 153) to learn how to create them. The process for Lotus Domino connections is very similar to other Database connections.

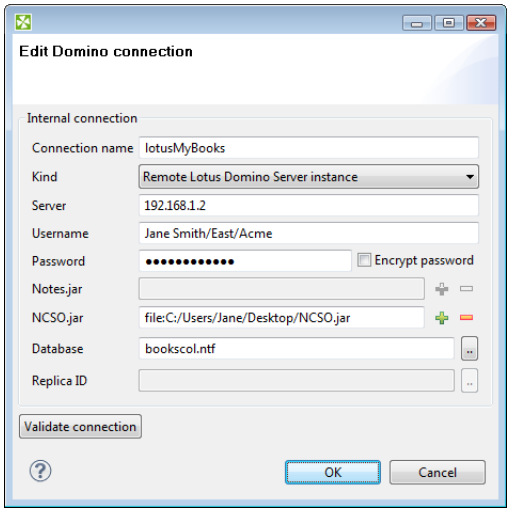


Figure 23.1. Lotus Notes Connection Dialog

Give a name to the connection (**Connection name**) and select the connection **Kind**. Currently the only **Kind** supported is to a **Remote Lotus Domino Server**.

When you are connecting to a remote Lotus server, you need to specify its location in the **server** field. This can be either an IP address or the network name of the server.

Connections to any kind of server require a **username** to be specified. The user name must match a Person document in the Domino Directory for the server.

You also have to fill in the **password** for the selected user. Access password can be encrypted. See Encrypting the Access Password (p. 157).

For a connection to be established, you are required to provide Lotus libraries for connecting to Lotus Notes.

To connect to remote Lotus Domino server, the **Notes.jar** library can be used. It can be found in the program directory of any Notes/Domino installation. For example: **c:\lotus\domino\Notes.jar** A light-weight version of **Notes.jar** can be provided instead. This version contains only support for remote connections and is stored in a file called **NCSO.jar**. This file can be found in the Lotus Domino server installation. For example: **c:\lotus\domino\data\domino\java\NCSO.jar**

To select a database to read/write data from/to, you can enter the file name of the database in the **database** field. Another option is to enter the **Replica ID** number of the desired database.

Chapter 24. Lookup Tables

When you are working with **CloverETL Designer**, you can also create and use **Lookup Tables**. These tables are data structures that allow fast access to data stored using a known key or SQL query. This way you can reduce the need to browse database or data files.

Warning

Remember that you should not use lookup tables in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template and the same methods of Java interfaces.

All data records stored in any lookup table are kept in files, in databases or cached in memory.

As in the case of metadata an database connections, also lookup tables can be internal or external (shared). You can create them in two ways.

Each lookup table can be created as:

- **Internal**: See Internal Lookup Tables (p. 167).

  Internal lookup tables can be:

  - **Externalized**: See Externalizing Internal Lookup Tables (p. 167).
  - **Exported**: See Exporting Internal Lookup Tables (p. 169).
- **External (shared)**: See External (Shared) Lookup Tables (p. 170).

  External (shared) lookup tables can be:

  - **Linked to the graph**: See Linking External (Shared) Lookup Tables (p. 170).
  - **Internalized**: See Internalizing External (Shared) Lookup Tables (p. 171).

Types of lookup tables are the following:

- **Simple lookup table**: See Simple Lookup Table (p. 172).
- **Database lookup table**: See Database Lookup Table (p. 175).
- **Range lookup table**: See Range Lookup Table (p. 176).
- **Persistent lookup table**: See Persistent Lookup Table (p. 178).
- **Aspell lookup table**: See Aspell Lookup Table (p. 179).

Internal Lookup Tables

Internal lookup tables are part of a graph, they are contained in the graph and can be seen in its source tab.

Creating Internal Lookup Tables

If you want to create an internal lookup table, you must do it in the **Outline** pane by selecting the **Lookups** item, right-clicking this item, selecting **Lookup tables** →**Create lookup table**. A **Lookup table** wizard opens. See Types of Lookup Tables (p. 172). After selecting the lookup table type and clicking **Next**, you can specify the properties of the selected lookup table. More details about lookup tables and types of lookup tables can be found in corresponding sections below.

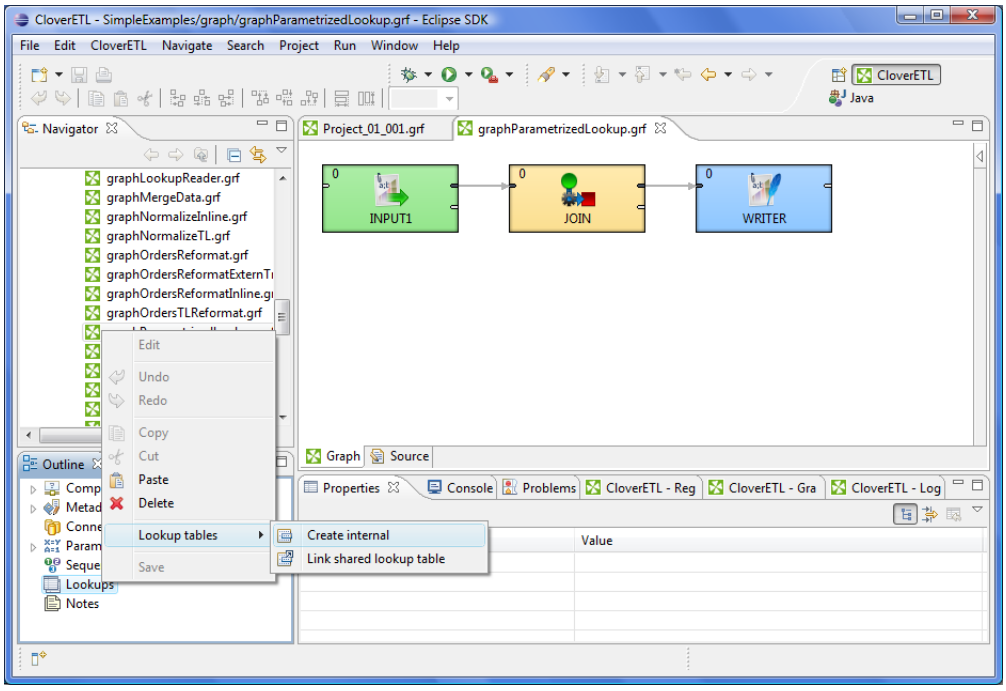


Figure 24.1. Creating Internal Lookup Table

Externalizing Internal Lookup Tables

After you have created an internal lookup table as a part of a graph, you may want to convert it to an external (shared) lookup table. So that you would be able to use the same lookup table for other graphs.

If you want to externalize internal lookup table into external (shared) file, do the following: Right-click the desired internal lookup table item in the **Outline** pane within **Lookups** group, then click **Externalize lookup table** from the context menu. If your lookup table contains internal metadata, you will see the following wizard.

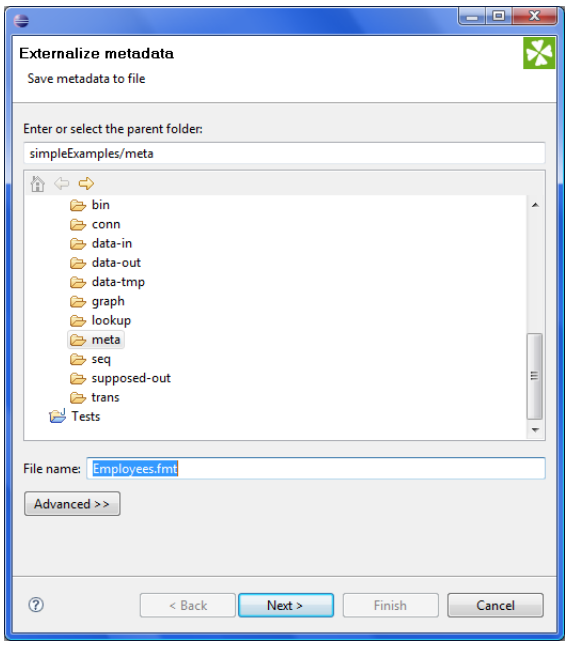


Figure 24.2. Externalizing Wizard

In this wizard, you will be offered the `meta` subfolder of your project as well as a filename of the new external (shared) metadata file to which the internal metadata assigned to the selected lookup table should be externalized. If you want (a file with the same name may already exist), you can change the suggested name of the external (shared) metadata file. After clicking **Next**, a similar wizard for externalizing database connection will be open. Do the same as for metadata. Finally, the wizard for lookup tables will open. In it, you will be presented with the `lookup` folder of your project as the location for this new external (shared) lookup table file and then you can click **OK**. If you want (a file with the same name may already exist), you can change the suggested name of the lookup table file.

After that, the internal metadata (and internal connection) and lookup table items disappear from the **Outline** pane **Metadata** (and **Connections**) and **Lookups** group, respectively, but, at the same location, new entries appear, already linked the newly created external (shared) metadata (and connection configuration file) and lookup table files within the corresponding groups. The same files appear in the `meta`, `conn`, and `lookup` subfolders of the project, respectively, and can be seen in the **Navigator** pane.

If your lookup table contains only external (shared) metadata (and external database connection), only the last wizard (for externalizing lookup tables) will open. In it, you will be presented with the `lookup` folder of your project as the location for this new external (shared) lookup table file and then you will click **OK**. If you want (the file with the same name may already exist), you can rename the offered name of the lookup table file.

After the internal lookup table has been externalized, the internal item disappears from the **Outline** pane **Lookups** group, but, at the same location, there appears, already linked, the new lookup table file item. The same file appears in the `lookup` subfolder of the project and can be seen in the **Navigator** pane.

You can even externalize multiple internal lookup table items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize lookup table** from the context menu. The process described above will be repeated again and again until all the selected lookup tables (along with the metadata and/or connection assigned to them, if needed) are externalized.

You can choose adjacent lookup table items when you press **Shift** and then press the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired connection items instead.

Exporting Internal Lookup Tables

This case is somewhat similar to that of externalizing internal lookup tables, except while you create a lookup table file that is outside the graph in the same way as that of an externalized file, the file is not linked to the original graph. Only an external lookup table file (maybe also metadata and/or connection) is created. Subsequently you can use such a file in other graphs as an external (shared) lookup table file as mentioned in the previous sections.

You can export internal lookup table into external (shared) one by right-clicking some of the internal lookup tables items in the **Outline** pane and clicking **Export lookup table** from the context menu. The `lookup` folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the suggested and you create the file by clicking **Finish**.

After that, the **Outline** pane lookups folder remains the same, but in the `lookup` folder in the **Navigator** pane the newly created lookup table file appears.

You can export multiple selected internal lookup tables in a similar way as it is described in the previous section about externalizing.

External (Shared) Lookup Tables

As mentioned previously, external (shared) lookup tables are able to be shared across multiple graphs. This allows easy access, but removes them from a graph's source

Creating External (Shared) Lookup Tables

In order to create an external (shared) lookup table, select **File →New →Other...**

Then you must expand the **CloverETL** item and either click the **Lookup table** item and **Next**, or double-click the **Lookup table** item.

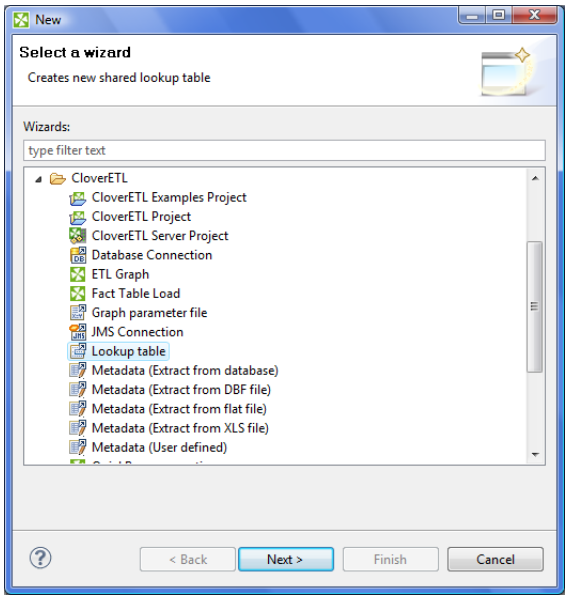


Figure 24.3. Selecting Lookup Table Item

After that, the **New lookup table** wizard opens. See Types of Lookup Tables(p. 172). In this wizard, you need to select the desired lookup table type, define it and confirm. You also need to select the file name of the lookup table within the `lookup` folder. After clicking **Finish**, your external (shared) database connection has been created.

Linking External (Shared) Lookup Tables

After their creation (see previous sections), external (shared) lookup tables can be linked to multiple graphs. You need to right-click either the **Lookups** group or any of its items and select **Lookup tables →Link shared lookup table** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `lookup` folder in this wizard and select the desired lookup table file from all the files contained in this wizard.

You can even link multiple external (shared) lookup table files at once. To do this, right-click either the **Lookups** group or any of its items and select **Lookup tables →Link shared lookup table** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must expand the `lookup` folder in this wizard and select the desired lookup table files from all the files contained in this wizard. You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Lookup Tables

Once you have created and linked external (shared) lookup table file, in case you want to put this lookup table into the graph, you need to convert it into internal lookup table. Thus, you could see its structure in the graph itself.

You can internalize any linked external (shared) lookup table file into internal lookup table by right-clicking such external (shared) lookup table items in the **Outline** pane and clicking **Internalize connection** from the context menu.

After doing that, the following wizard opens that allows you to decide whether you also want to internalize metadata assigned to the lookup table and/or its DB connection (in case of **Database lookup table**).

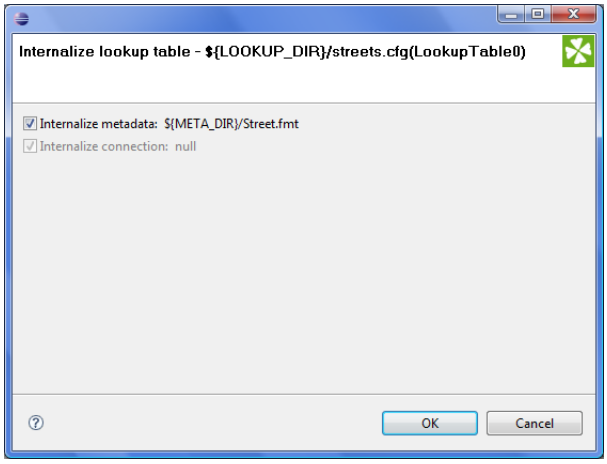


Figure 24.4. Lookup Table Internalization Wizard

When you check the checkboxes or leave them unchecked, click **OK**.

After that, the selected linked external (shared) lookup table items disappear from the **Outline** pane **Lookups** group, but, at the same location, newly created internal lookup table items appear. If you have also decided to internalize the linked external (shared) metadata assigned to the lookup table, their item is converted to internal metadata item what can be seen in the **Metadata** group of the **Outline** pane.

However, the original external (shared) lookup table file still remains to exist in the `lookup` subfolder. You can see it in this folder in the **Navigator** pane.

You can even internalize multiple linked external (shared) lookup table files at once. To do this, select the desired linked external (shared) lookup table items in the **Outline** pane. After that, you only need to repeat the process described above for each selected lookup table. You can select adjacent items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

Types of Lookup Tables

After opening the **New lookup table** wizard, you need to select the desired lookup table type. After selecting the radio button and clicking **Next**, the corresponding wizard opens.

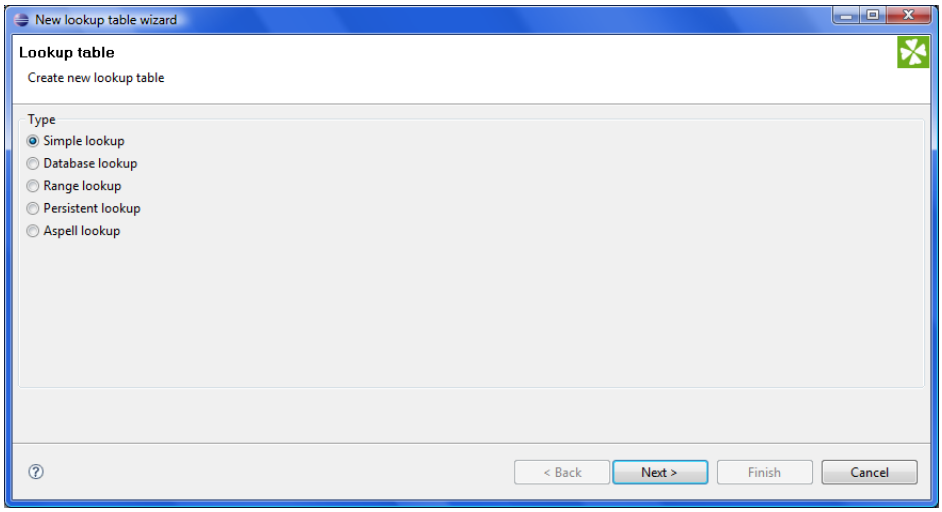


Figure 24.5. Lookup Table Wizard

Simple Lookup Table

All data records stored in this lookup table are kept in memory. For this reason, to store all data records from the lookup table, sufficient memory must be available. If data records are loaded to a simple lookup table from a data file, the size of the available memory should be approximately at least 6 times bigger than that of the data file. However, this multiplier is different for different types of data records stored in the data file.

In the **Simple lookup table** wizard, you must set up the demanded properties:

In the **Table definition** tab, you must give a **Name** to the lookup table, select the corresponding **Metadata** and the **Key** that should be used to look up data records from the table. You can select a **Charset** and the **Initial size** of the lookup table (512 by default) as wll. You can change the default value by changing the `Lookup.LOOKUP_INITIAL_CAPACITY` value in `defaultProperties`.

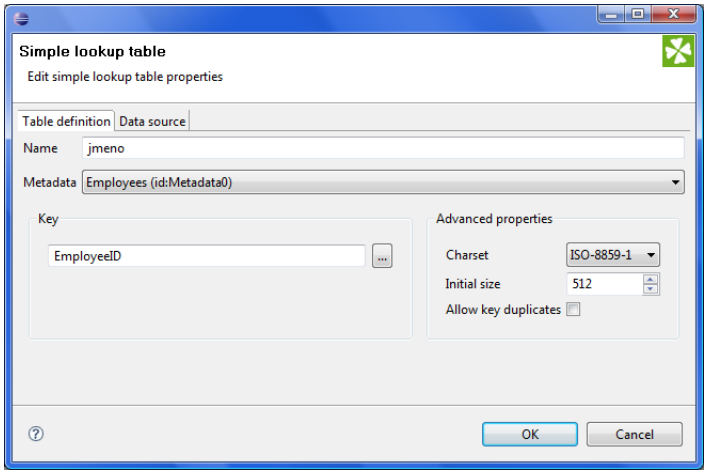


Figure 24.6. Simple Lookup Table Wizard

After clicking the button on the right side from the **Key** area, you will be presented with the **Edit key** wizard which helps you select the **Key**.

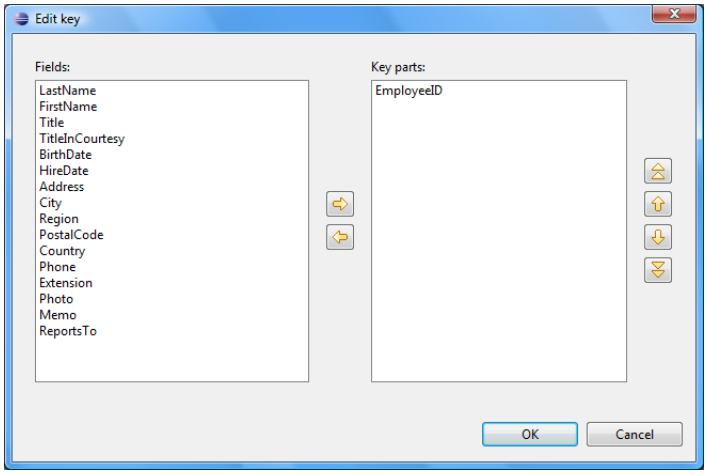


Figure 24.7. Edit Key Wizard

By highlighting some of the field names in the **Field** pane and clicking the **Right arrow** button you can move the field name into the **Key parts** pane. You can keep moving more fields into the **Key parts** pane. You can also change the position of any of them in the list of the Key parts by clicking the **Up** or **Down** buttons. The key parts that are higher in the list have higher priority. When you have finished, you only need to click **OK**. (You can also remove any of them by highlighting it and clicking the **Left arrow** button.)

In the **Data source** tab, you can either locate the file **URL** or fill in the grid after clicking the **Edit data** button. After clicking **OK**, the data will appear in the **Data** text area.

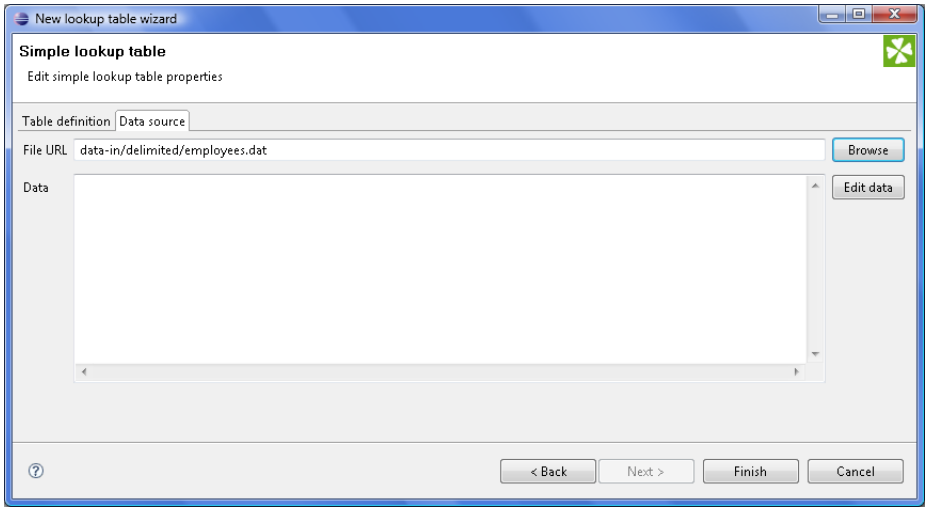


Figure 24.8. Simple Lookup Table Wizard with File URL

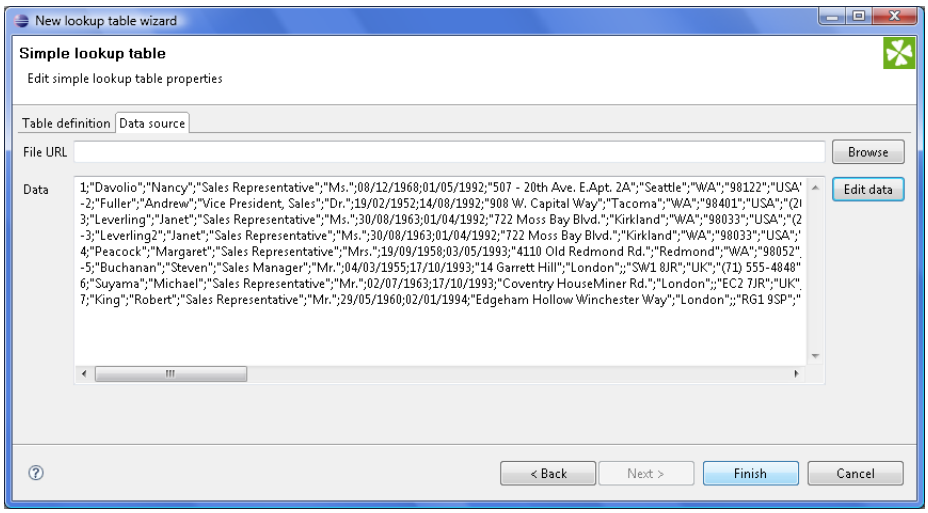


Figure 24.9. Simple Lookup Table Wizard with Data

You can set or edit the data after clicking the **Edit data** button.

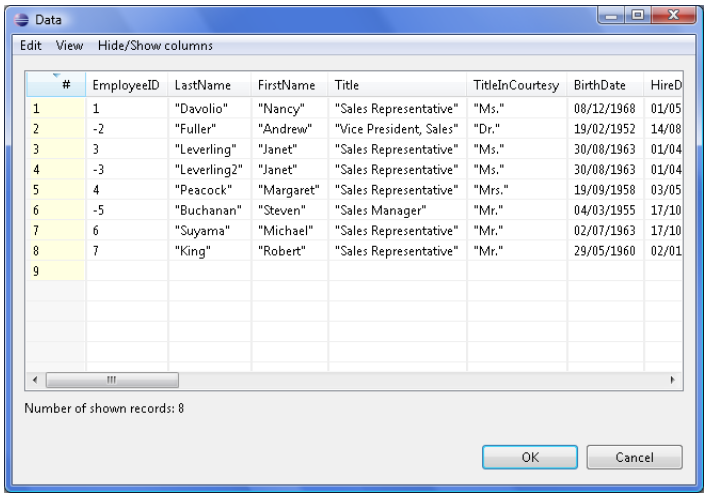


Figure 24.10. Changing Data

After all has been done, you can click **OK** and then **Finish**.

Simple lookup table are allowed to contain data specified directly in the grid, data in the file or data that can be read using **LookupTableReaderWriter**.

Important

Remember that you can also check the **Allow key duplicates** checkbox. This way, you are allowing multiple data records with the same key value (duplicate records).

If you want that only one record per each key value is contained in **Simple lookup table**, leave the mentioned checkbox unchecked (the default setting). If only one record is selected, new records overwrite the older ones. In such a case, the last record is the only one that is included in **Simple lookup table**.

Database Lookup Table

This type of lookup table works with databases and unloads data from them by using SQL query. Database lookup table reads data from the specified database table. The key which serves to search records from this lookup tables is the "`where = ? [and ...]`" part of the query. Data records unloaded from database can be cached in memory keeping the LRU order (the least recently used items are discarded first). To cache them, you must specify the number of such records (**Max cached records**). In case no record can be found in database under some key value, this response can be saved if you check the **Store negative key response** checkbox. Then, lookup table will not search through the database table when the same key value is given again. Remember that **Database lookup table** allows to work with duplicate records (multiple records with the same key value).

When creating or editing a **Database lookup table**, you must check the **Database lookup** radio button and click **Next**. (See Figure 24.5, Lookup Table Wizard (p. 172).)

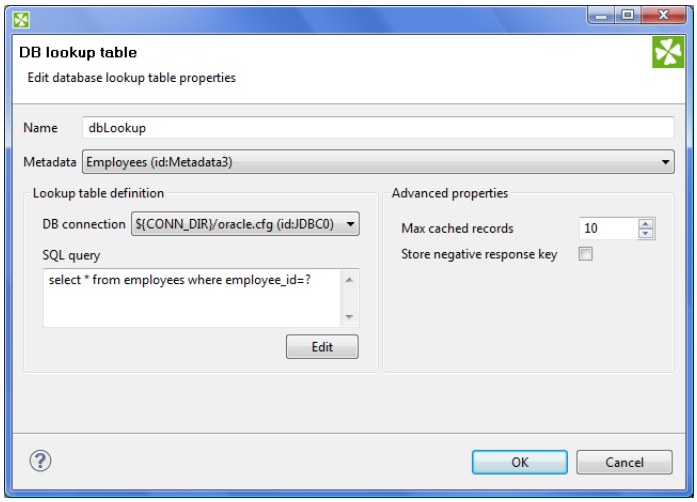


Figure 24.11. Database Lookup Table Wizard

Then, in the **Database lookup table** wizard, you must give a **Name** to the selected lookup table, specify some **Metadata** and **DB connection**.

Remember that **Metadata** definition is not required for transformations written in Java. In them, you can simply select the **no metadata** option. However, with CTL it is indispensable to specify **Metadata**.

You must also type or edit some SQL query that serves to look up data records from database. When you want to edit the query, you must click the **Edit** button and, if your database connection is valid and working, you will be presented with the **Query** wizard, where you can browse the database, generate some query, validate it and view the resulting data. To specify some lookup table key, add a "`where = ? [and ...]`" part to the end of the query.

Then, you can click **OK** and then **Finish**. See Extracting Metadata from a Database (p. 130) for more details about extracting metadata from a database.

Range Lookup Table

You can create a **Range lookup table** only if some fields of the records create ranges. That means the fields are of the same data type and they can be assigned both start and end. You can see this in the following example:

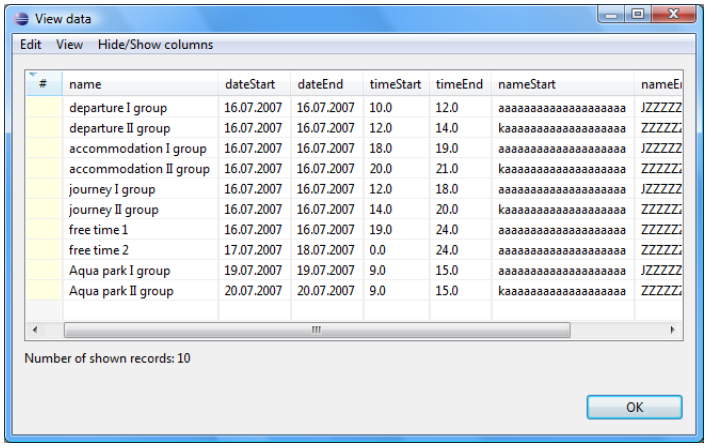


Figure 24.12. Appropriate Data for Range Lookup Table

When you create a **Range lookup table**, you must check the **Range lookup** radio button and click **Next**. (See Figure 24.5, Lookup Table Wizard (p. 172).)

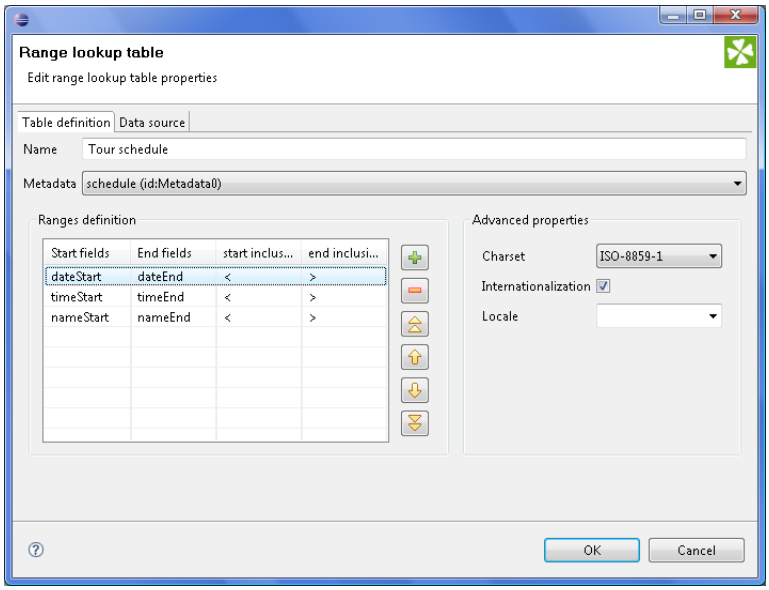


Figure 24.13. Range Lookup Table Wizard

Then, in the **Range lookup table** wizard, you must give a **Name** to the selected lookup table, and specify **Metadata**.

You can select **Charset** and decide whether **Internationalization** and what **Locale** should be used.

By clicking the buttons at the right side, you can add either of the items, or move them up or down.

You must also select whether any start or end field should be included in these ranges or not. You can do it by selecting any of them in the corresponding column of the wizard and clicking.

When you switch to the **Data source** tab, you can specify the file or directly type the data in the grid. You can also write data to lookup table using **LookupTableReaderWriter**.

By clicking the **Edit** button, you can edit the data contained in the lookup table. At the end, you only need to click **OK** and then **Finish**.

Important

Remember that **Range lookup table** includes only the first record with identical key value.

Persistent Lookup Table

Commercial Lookup Table

This lookup table is commercial and can only be used with the commercial license of **CloverETL Designer**.

This type of lookup table serves a greater number of data records. Unlike the **Simple lookup table**, data is stored in a file specified in the wizard of **Persistent lookup table**. These files are in `jdbm` format (http://jdbm.sourceforge.net).

In the **Persistent lookup table** wizard, you set up the demanded properties. You must give a **Name** to the lookup table, select the corresponding **Metadata**, specify the **File** where the data records of the lookup table will be stored and the **Key** that should be used to look up data records from the table.

Remember that this file has some internal format which should be create first and then used. When you specify some file, two files will be created and filled with data (with `db` and `lg` extensions). Upon each writing to this table, new records with the same key values may follow into this file. If you want older records to be overwritten by newer ones, you need to check the **Replace** checkbox.

You can also decide whether transactions should be disabled (**Disable transactions**). If you want to increase graph performance this can be desirable, however, it can cause data loss.

You can select some **Advanced properties**: **Commit interval**, **Page size** and **Cache size**. **Commit interval** defines the number of records that are committed at once. By specifying **Page size**, you are defining the number of entries per node. **Cache size** specifies the maximum number of records in cache.

Important

Remember that **Persistent lookup table** does not contain multiple records with identical value of the key. Such duplicates are not allowed.

If the **Replace** checkbox is checked, the last record from all those with the same key value is the only one that is included in the lookup table. On the other hand, if the checkbox is left unchecked, the first record is the only one that is included in it.

At the end, you only need to click **OK** and then **Finish**.

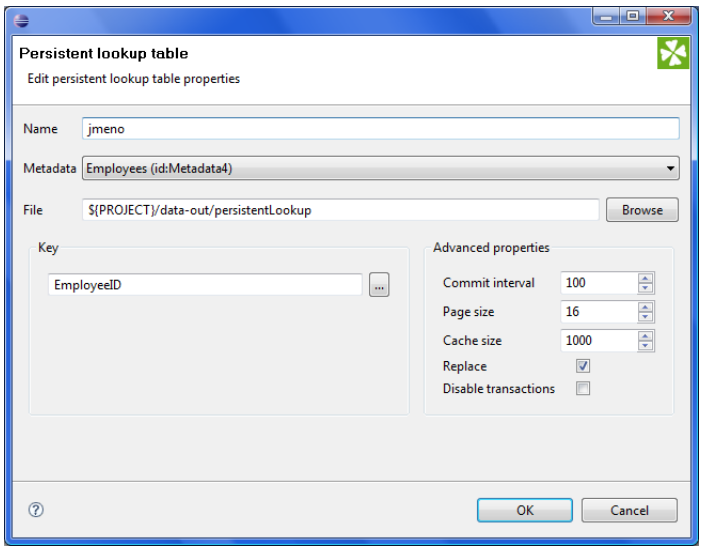


Figure 24.14. Persistent Lookup Table Wizard

Aspell Lookup Table

Commercial Lookup Table

This lookup table is commercial and can only be used with the commercial license of **CloverETL Designer**.

All data records stored in this lookup table are kept in memory. For this reason, to store all data records from the lookup table, sufficient memory must be available. If data records are loaded to aspell lookup table from a data file, the size of available memory should be approximately at least 7 times bigger than that of the data file. However, this multiplier is different for different types of data records stored in the data file.

If you are working with data records that are similar but not fully identical, you should use this type of lookup table. For example, you can use **Aspell lookup table** for addresses.

In the **Aspell lookup table** wizard, you set up the required properties. You must give a **Name** to the lookup table, select the corresponding **Metadata**, select the **Lookup key field** that should be used to look up data records from the table (must be of string data type).

You can also specify the **Data file URL** where the data records of the lookup table will be stored and the charset of data file (**Data file charset**) The default charset is `ISO-8859-1`.

You can set the threshold that should be used by the lookup table (**Spelling threshold**). It must be higher than 0. The higher the threshold, the more tolerant is the component to spelling errors. Its default value is `230`. It is the `edit_distance` value from the query to the results. Words with this value higher that the specified limit are not included in the results.

You can also change the default costs of individual operations (**Edit costs**):

- **Case cost**

  Used when the case of one character is changed.

- **Transpose cost**

  Used when one character is transposed with another in the string.

- **Delete cost**

  Used when one character is deleted from the string.

- **Insert cost**

  Used when one character is inserted to the string.

- **Replace cost**

  Used when one character is replaced by another one.

You need to decide whether the letters with diacritical marks are considered identical with those without these marks. To do that, you need to set the value of **Remove diacritical marks** attribute. If you want diacritical marks to be removed before computing the `edit_distance` value, you need to set this value to `true`. This way, letters with diacritical marks are considered equal to their latin equivalents. (Default value is `false`. By default, letters with diacritical marks are considered different from those without.)

If you want best guesses to be included in the results, set the **Include best guesses** to `true`. Default value is `false`. Best guesses are the words whose `edit_distance` value is higher than the **Spelling threshold**, for which there is no other better counterpart.

At the end, you only need to click **OK** and then **Finish**.

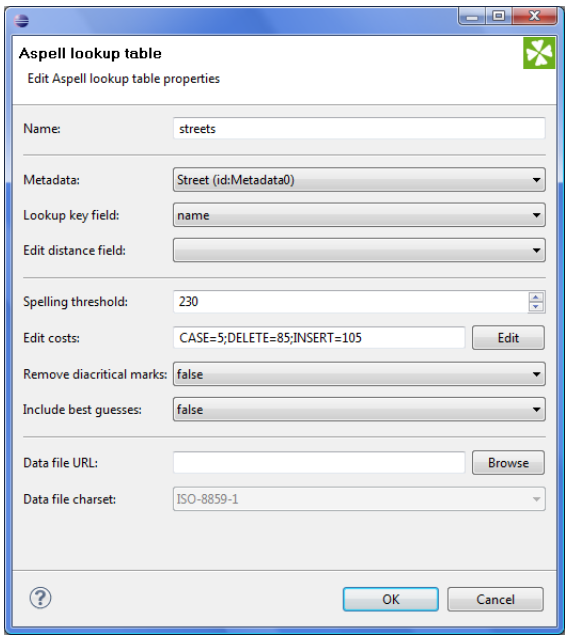


Figure 24.15. Aspell Lookup Table Wizard

Important

If you want to know what is the distance between lookup table and edge values, you must add another field of numeric type to lookup table metadata. Set this field to **Autofilling** (`default_value`).

Select this field in the **Edit distance field** combo.

When you are using **Aspell lookup table** in **LookupJoin**, you can map this lookup table field to corresponding field on the output port 0.

This way, values that will be stored in the specified **Edit distance field** of lookup table will be sent to the output to another specified field.

Chapter 25. Sequences

CloverETL Designer contains a tool designed to create sequences of numbers that can be used, for example, for numbering records. In records, a new field is created and filled by numbers taken from the sequence.

Warning

Remember that you should not use sequences in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template and the same methods of Java interfaces.

Each sequence can be created as:

- **Internal**: See Internal Sequences (p. 182).

  Internal sequences can be:

  - **Externalized**: See Externalizing Internal Sequences (p. 182).
  - **Exported**: See Exporting Internal Sequences (p. 183).

- **External (shared)**: See External (Shared) Sequences (p. 184).

  External (shared) sequences can be:

  - **Linked to the graph**: See Linking External (Shared) Sequences (p. 184).
  - **Internalized**: See Internalizing External (Shared) Sequences (p. 184).

Editing Sequence Wizard is described in Editing a Sequence (p. 185).

Internal Sequences

Internal sequences are stored in the graph (except the file in which its data are stored), they can be seen there. If you want to use one sequence for multiple graphs, it is better to use an external (shared) sequence. If you want to give someone your graph, it is better to have internal sequences. It is the same as with metadata, connections and parameters.

Creating Internal Sequences

If you want to create an internal sequence, you must right-click the **Sequence** item in the **Outline** pane and choose **Sequence** →**Create sequence** from the context menu. After that, a **Sequence** wizard appears. See Editing a Sequence (p. 185).

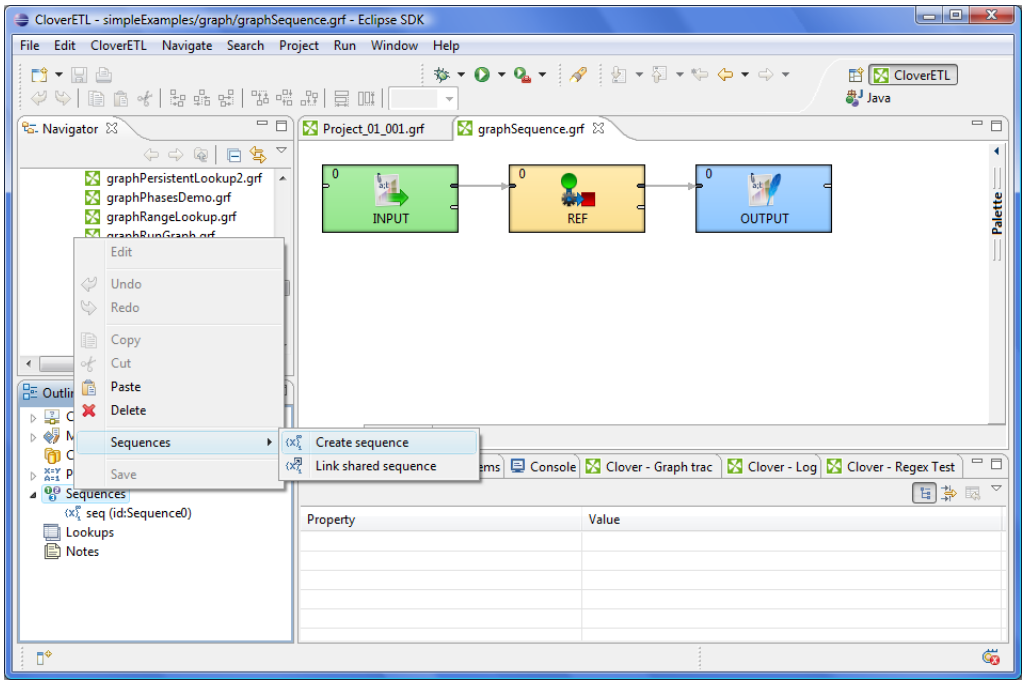


Figure 25.1. Creating a Sequence

Externalizing Internal Sequences

Once you have created an internal sequence as a part of a graph, you have it in your graph, but you may want to convert it into external (shared) sequence. Thus, you would be able to use the same sequence for more graphs (when more graphs share it).

You can externalize any internal sequence item into an external (shared) file by right-clicking an internal sequence item in the **Outline** pane and selecting **Externalize sequence** from the context menu. After doing that, a new wizard will open in which a list of projects of your workspace can be seen and the `seq` folder of the corresponding project will be offered as the location for this new external (shared) sequence file. If you want (a file with the same name may already exist), you can change the suggested name of the sequence file. Then you can click **OK**.

After that, the internal sequence item disappears from the **Outline** pane **Sequences** group, but, at the same location, there appears, already linked, the newly created external (shared) sequence file. The same sequence file appears in the `seq` folder of the selected project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal sequence items at once. To do this, select them in the **Outline** pane and, after right-click, select **Externalize sequence** from the context menu. After doing that, a new wizard will

open in which a `seq` folder of the corresponding projects of your workspace can be seen and it will be offered as the location for this new external (shared) sequence file. If you want (a file with the same name may already exist), you can change the suggested name of the sequence file. Then you can to click **OK**.

After that, the selected internal sequence items disappear from the **Outline** pane's' **Sequences** group, but, at the same location, there appears, already linked, the newly created external (shared) sequence file. The same sequence file appears in the selected project and it can be seen in the **Navigator** pane.

You can choose adjacent sequence items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired sequence items instead.

Exporting Internal Sequences

This case is somewhat similar to that of externalizing internal sequences. But, while you create a sequence file that is outside the graph in the same way as that of externalized file, the a file is not linked to the original graph. Only an external sequence file is being created. Subsequently you can use such a file in other graphs as an external (shared) sequence file as mentioned in the previous sections.

You can export an internal sequence into an external (shared) one by right-clicking one of the internal sequence items in the **Outline** pane and then clicking **Export sequence** from the context menu. The `seq` folder of the corresponding project will be offered for the newly created external file. You can also give the file any other name than the offered and then create the file by clicking **Finish**.

After that, the **Outline** pane's sequences folder remains the same, but in the **Navigator** pane the newly created sequence file appears.

You can even export multiple selected internal sequences in a similar way to how it is described in the previous section about externalizing.

External (Shared) Sequences

External (shared) sequences are stored outside the graph, they are stored in a separate file within the project folder. If you want to share the sequence among more graphs, it is better to have external (shared) sequence. But, if you want to give someone your graph, it is better to have internal sequence. It is the same as with metadata, connections, lookup tables and parameters.

Creating External (Shared) Sequences

If you want to create external (shared) sequences, you must select **File** →**New** →**Other** from the main menu and expand the **CloverETL** category and either click the **Sequence** item and the **Next** button or double-click the **Sequence** item. **Sequence** wizard will open. See Editing a Sequence (p. 185).

You will create the external (shared) sequence and save the created sequence definition file to the selected project.

Linking External (Shared) Sequences

After their creation (see previous section and Editing a Sequence (p. 185)), external (shared) sequences must be linked to each graph in which they would be used. You need to right-click either the **Sequences** group or any of its items and select **Sequences** →**Link shared sequence** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must locate the desired sequence file from all the files contained in the project (sequence files have the `.cfg` extension).

You can even link multiple external (shared) sequence files at once. To do this, right-click either the **Sequences** group or any of its items and select **Sequences** →**Link shared sequence** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must locate the desired sequence files from all the files contained in the project. You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Sequences

Once you have created and linked external (shared) sequence file, in case you want to put it into the graph, you need to convert it into internal sequence. In such a case you would see it in the graph itself.

You can internalize any linked external (shared) sequence file into internal sequence by right-clicking some of the external (shared) sequence items in the **Outline** pane and clicking **Internalize sequence** from the context menu.

You can even internalize multiple linked external (shared) sequence files at once. To do this, select the desired linked external (shared) sequence items in the **Outline** pane. You can select adjacent items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the linked external (shared) sequence items disappear from the **Outline** pane **Sequences** group, but, at the same location, the newly created internal sequence items appear.

However, the original external (shared) sequence files still remain to exist in the `seq` folder of the corresponding project what can be seen in the **Navigator** pane (sequence files have the `.cfg` extensions).

Editing a Sequence

In this wizard, you must type the name of the sequence, select the value of its first number, the incrementing step (in other words, the difference between every pair of adjacent numbers), the number of precomputed values that you want to be cached and, optionally, the name of the sequence file where the numbers should be stored. If no sequence file is specified, the sequence will not be persistent and the value will be reset with every run of the graph. The name can be, for example, `${SEQ_DIR}/sequencefile.seq` or `${SEQ_DIR}/anyothername`. Note that we are using here the `SEQ_DIR` parameter defined in the `workspace.prm` file, whose value is `${PROJECT}/seq`. And `PROJECT` is another parameter defining the path to your project located in workspace.

When you want to edit some of the existing sequences, you must select the sequence name in the **Outline** pane, open the context menu by right-clicking this name and select the **Edit** item. A **Sequence** wizard appears. (You can also open this wizard when selecting some sequence item in the **Outline** pane and pressing **Enter**.)

Now it differs from that mentioned above by a new text area with the current value of the sequence number. The value has been taken from a file. If you want, you can change all of the sequence properties and you can reset the current value to its original value by clicking the button.

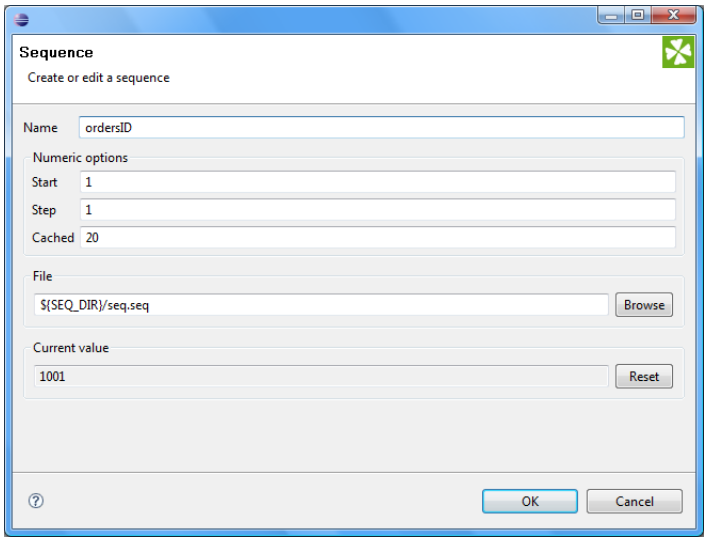


Figure 25.2. Editing a Sequence

And when the graph has been run once again, the same sequence started from 1001:

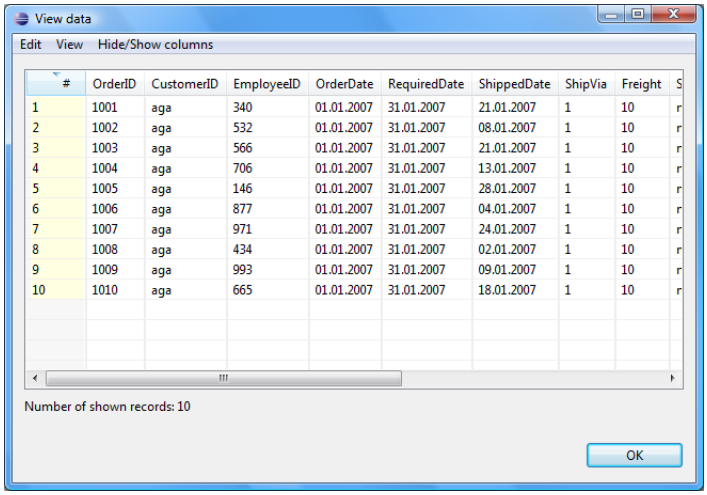


Figure 25.3. A New Run of the Graph with the Previous Start Value of the Sequence

You can also see how the sequence numbers fill one of the record fields.

Chapter 26. Parameters

When working with graphs, it may be necessary to create parameters. Like metadata and connections, parameters can be both internal and external (shared). The reason for creating parameters is the following: when using parameters, you can simplify graph management. Every value, number, path, filename, attribute, etc. can be set up or changed with the help of parameters. Parameters are similar to named constants. They are stored in one place and after the value of any of them is changed, this new value is used in the program.

Priorities

- These parameters have less priority than those specified in the **Main** tab or **Arguments** tab of **Run Configurations...**. In other words, both the internal and the external parameters can be overwritten by those specified in **Run Configurations...**. However, both the external and the internal parameters have higher priority than all environment variables and can overwrite them. Remember also that the external parameters can overwrite the internal ones.

If you use parameters in CTL, you should type them as `'${MyParameter}'`. Be careful when working with them, you can also use escape sequences for specifying some characters.

Each parameter can be created as:

- **Internal**: See Internal Parameters (p. 187).

  Internal parameters can be:

  - **Externalized**: See Externalizing Internal Parameters (p. 188).
  - **Exported**: See Exporting Internal Parameters (p. 189).

- **External (shared)**: See External (Shared) Parameters (p. 190).

  External (shared) parameters can be:

  - **Linked to the graph**: See Linking External (Shared) Parameters (p. 190).
  - **Internalized**: See Internalizing External (Shared) Parameters (p. 190).

Parameters Wizard is described in Parameters Wizard (p. 192).

Internal Parameters

Internal parameters are stored in the graph, and thus are present in the source. If you want to change the value of some parameter, it is better to have external (shared) parameters. If you want to give someone your graph, it is better to have internal parameters. It is the same as with metadata and connections.

Creating Internal Parameters

If you want to create internal parameters, you must do it in the **Outline** pane by selecting the **Parameters** item, right-clicking this item, selecting **Parameters →Create internal parameter**. A **Graph parameters** wizard appears. See Parameters Wizard (p. 192).

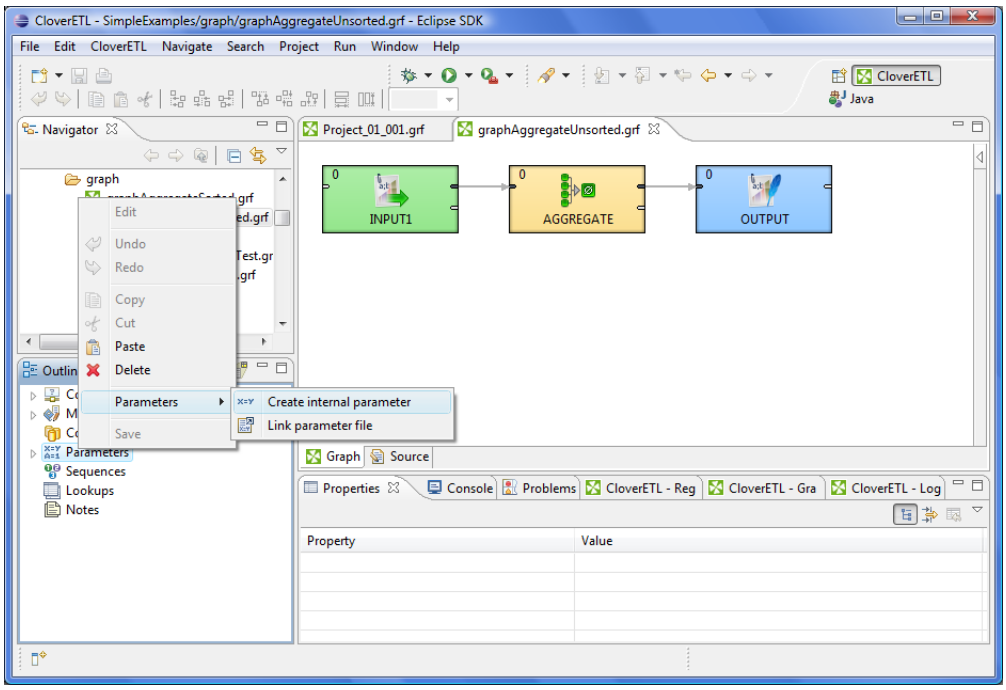


Figure 26.1. Creating Internal Parameters

Externalizing Internal Parameters

Once you have created internal parameters as a part of a graph, you have them in your graph, but you may want to convert them into external (shared) parameters. Thus, you would be able to use the same parameters for multiple graphs.

You can externalize any internal parameter item into external (shared) file by right-clicking an internal parameter item in the **Outline** pane and selecting **Externalize parameters** from the context menu. After doing that, a new wizard will open in which a list of projects of your workspace can be seen and the corresponding project will be offered as the location for this new external (shared) parameter file. If you want (the file with the same name may already exist), you can change the suggested name of the parameter file. Then you click **OK**.

After that, the internal parameter item disappears from the **Outline** pane **Parameters** group, but, at the same location, there appears, already linked, the newly created external (shared) parameter file. The same parameter file appears in the selected project and it can be seen in the **Navigator** pane.

You can even externalize multiple internal parameter items at once. This way, they will be externalized into one external (shared) parameter file. To do this, select them in the **Outline** pane and, after right-click, select **Externalize parameters** from the context menu. After doing that, a new wizard will open in which a list of projects of your workspace can be seen and the corresponding project will be offered as the location for this new external (shared) parameter file. If you want (a file with the same name may already exist), you can change the suggested name of the parameter file. Then you click **OK**.

After that, the selected internal parameter items disappear from the **Outline** pane **Parameters** group, but, at the same location, there appears already linked the newly created external (shared) parameter file. The same parameter file appears in the selected project and it can be seen in the **Navigator** pane. This file contain the definition of all selected parameters.

You can choose adjacent parameter items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to choose non-adjacent items, use **Ctrl**+**Click** at each of the desired parameter items instead.

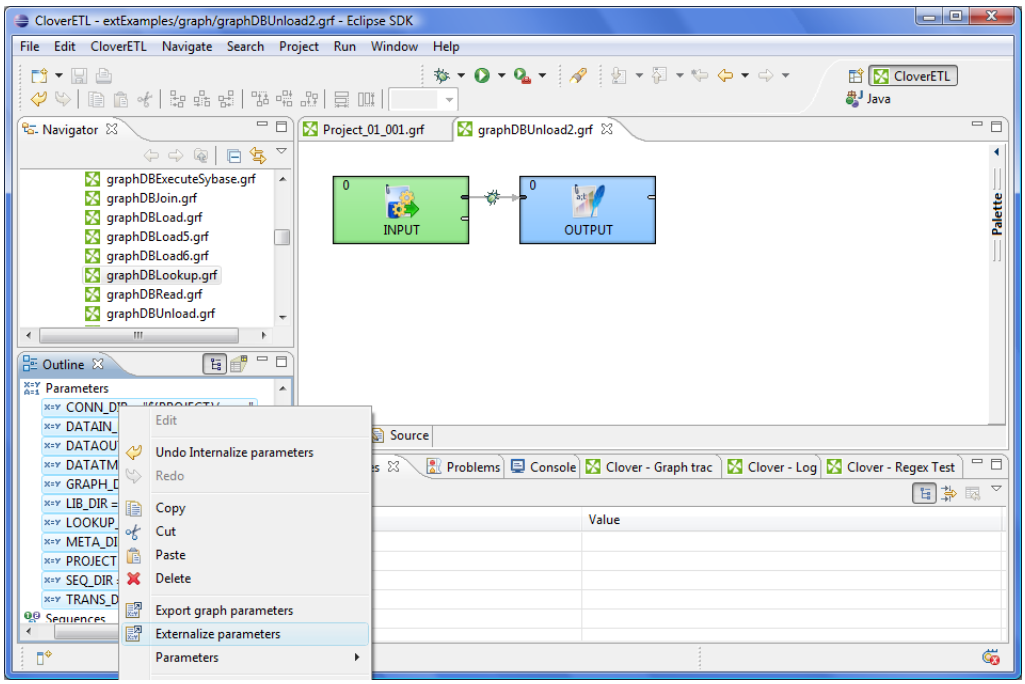


Figure 26.2. Externalizing Internal Parameters

Exporting Internal Parameters

This case is somewhat similar to that of externalizing internal parameters. While you create a parameter file that is outside the graph in the same way as that of externalized file, the file is not linked to the original graph. Only an external parameter file is being created. Subsequently you can use such a file in multiple graphs as an external (shared) parameter file as mentioned in the previous sections.

You can export internal parameter into external (shared) one by right-clicking some of the internal parameter items in the **Outline** pane and clicking **Export parameter** from the context menu. The corresponding project will be offered for the newly created external file. You can also give the file any other name than the offered and you create the file by clicking **Finish**.

After that, the **Outline** pane parameters folder remains the same, but in the **Navigator** pane the newly created parameters file appears.

You can even export multiple selected internal parameters in a similar way as it is described in the previous section about externalizing.

External (Shared) Parameters

External (shared) parameters are stored outside the graph, they are stored in a separate file within the project folder. If you want to change the value of some of the parameters, it is better to have external (shared) parameters. But, if you want to give someone your graph, it is better to have internal parameters. It is the same as with metadata and connections.

Creating External (Shared) Parameters

If you want to create external (shared) parameters, you must select **File** →**New** →**Other** from the main menu, expand the **CloverETL** category and either click the **Graph parameter file** item and the **Next** button or double-click the **Graph parameter file** item. **Graph parameters** wizard opens. See Parameters Wizard (p. 192). In this wizard you will create names and values of parameters.

Linking External (Shared) Parameters

After their creation (see previous section and Parameters Wizard (p. 192)), external (shared) parameters can be linked to each graph in which they should be used. You need to right-click either the **Parameters** group or any of its items and select **Parameters** →**Link parameter file** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must locate the desired parameter file from all the files contained in the project (parameter files have the `.prm` extension).

You can even link more external (shared) parameter files at once. To do this, right-click either the **Parameters** group or any of its items and select **Parameters** →**Link parameter file** from the context menu. After that, a **File selection** wizard displaying the project content will open. You must locate the desired parameter files from all the files contained in the project. You can select adjacent file items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired file items instead.

Internalizing External (Shared) Parameters

Once you have created and linked external (shared) parameter file, in case you want to put it into the graph, you need to convert it into internal parameters. In such a case you would see them in the graph itself. Remember that one parameter file with more parameters will create more internal parameters.

You can internalize any linked external (shared) parameter file into internal parameters by right-clicking some of the external (shared) parameters items in the **Outline** pane and clicking **Internalize parameters** from the context menu.

You can even internalize multiple linked external (shared) parameter files at once. To do this, select the desired linked external (shared) parameter items in the **Outline** pane. You can select adjacent items when you press **Shift** and press the **Down Cursor** or the **Up Cursor** key. If you want to select non-adjacent items, use **Ctrl**+**Click** at each of the desired items instead.

After that, the linked external (shared) parameters items disappear from the **Outline** pane **Parameters** group, but, at the same location, the newly created internal parameter items appear.

However, the original external (shared) parameter files still remain to exist in the project what can be seen in the **Navigator** pane (parameter files have the `.prm` extensions).

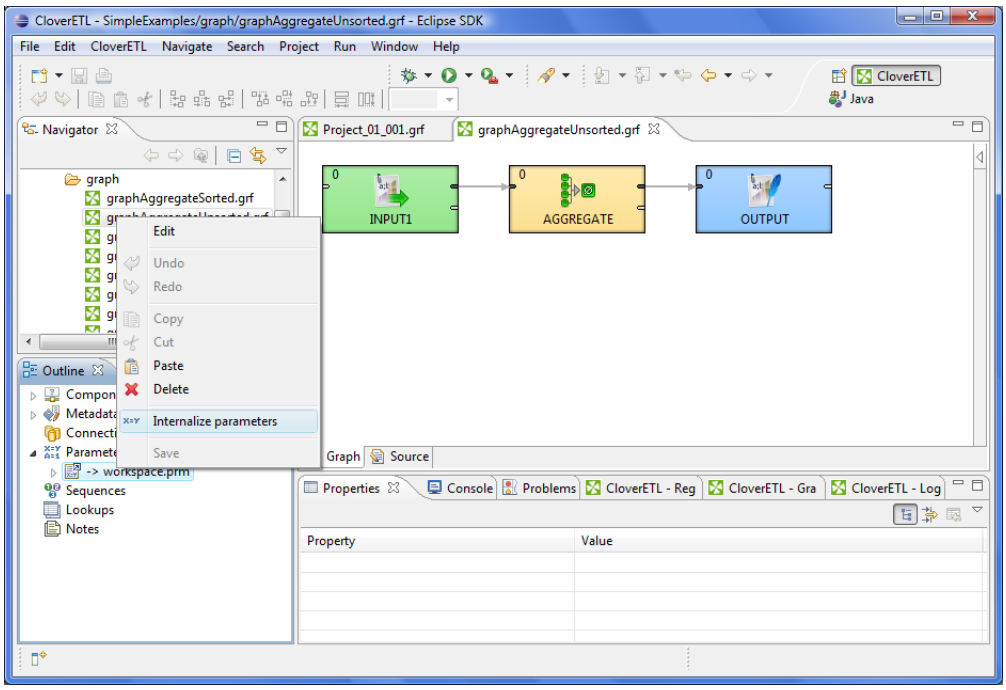


Figure 26.3. Internalizing External (Shared) Parameter

Parameters Wizard

(You can also open this wizard when selecting some parameters item in the **Outline** pane and pressing **Enter**.)

By clicking the **plus** button on the right side, a pair of words "**name**" and "**value**" appear in the wizard. After each clicking the **Plus** button, a new line with **name** and **value** labels appears and you must set up both names and values. You can do it when highlight any of them by clicking and change it to whatever you want and need. When you select all names and set up all values you want, you can click the **Finish** button (for internal parameters) or the **Next** button and type the name of the parameter file. The extension `.prm` will be added to the file automatically.

You also need to select the location for the parameter file in the project folder. Then you can click the **Finish** button. After that, the file will be saved.

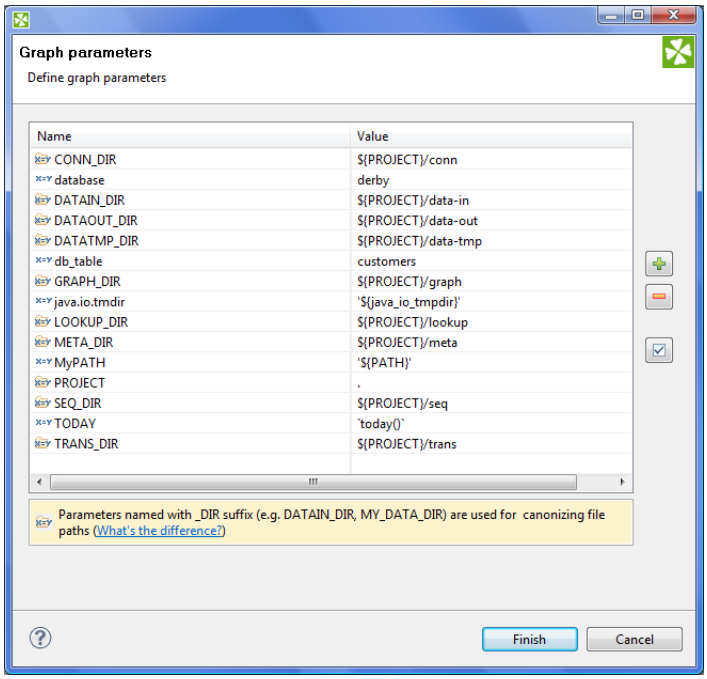


Figure 26.4. Example of a Parameter-Value Pairs

Note

Note two kinds of icons next to parameter names. One of them mark parameters that are able to canonicalize paths (those with `_DIR` suffix and the `PROJECT` parameter), the others do not canonicalize paths. See Canonizing File Paths (p. 193) for detailed information.

Note

Moreover, note the usage of following parameters:

1. `TODAY`

   This parameter uses a CTL1 expression. See Parameters with CTL Expressions (p. 193) for details.

2. `java.io.tmdir`, `MyPATH`

   These parameters resolve to environment variables. See Environment Variables (p. 193) for details.

3. `database`, `db_table`

These are standard parametrs.

If you want to see the value to which parameter resolves, click the button that is the most below on the right side of the dialog.

Parameters with CTL Expressions

Since the version 2.8.0 of **CloverETL**, you can also use CTL expressions in parameters and other places of **CloverETL**. Such CTL expressions can use any possibilities of CTL language. However, these CTL expressions must be surrounded by back quotes.

For example, if you define a parameter `TODAY="`today()`"` and use it in your CTL codes, such `${TODAY}` expression will be resolved to the current day.

If you want to display a back quote as is, you must use this back quote preceded by back slash as follows: `\``.

Important

CTL1 version is used in such expressions.

Environment Variables

Environment variables are parameters that are not defined in **CloverETL**. They are defined in Operation system.

You can get the values of these environment variables using the same expression that can be used for all other parameters.

- To get the value of environment variable called `PATH`, use the following expression:

  '${PATH}'

- To get the value of a variable whose name contains dots (e.g, `java.io.tmpdir`), replace each dot with underscore character and type:

  ```
  '${java_io_tmpdir} '
  ```

  Note that the terminal single quote must be preceded by a white space since `java.io.tmpdir` itself ends with a backslash and we do not want to get an escape sequence (`\'`). With this white space we will get `\ '` at the end.

Important

Use single quotes to avoid escape sequences in Windows paths!

Canonizing File Paths

All parameters can be divided into two groups:

1. The `PROJECT` parameter and any other parameter with `_DIR` used as suffix (`DATAIN_DIR`, `CONN_DIR`, `MY_PARAM_DIR`, etc.).

2. All the other parameters.

Either group is distinguished with corresponding icon in the **Parameter Wizard**.

The parameters of the first group serve to automatically canonicalize file paths displayed in the **URL File dialog** and in the **Outline** pane in the following way:

1. If any of these parameters matches the beginning of a path, corresponding part of the beginning of the path is replaced with this parameter.

2. If multiple parameters match different parts of the beginning of the same path, parameter expressing the longest part of the path is selected.

Example 26.1. Canonizing File Paths

- If you have two parameters:

  `MY_PARAM1_DIR` and `MY_PARAM2_DIR`

  Their values are:

  `MY_PARAM1_DIR = "mypath/to"` and `MY_PARAM2_DIR = "mypath/to/some"`

  If the path is:

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  The path is displayed as follows:

  ```
  ${MY_PARAM2_DIR}/directory/with/the/file.txt
  ```

- If you had two parameters:

  `MY_PARAM1_DIR` and `MY_PARAM3_DIR`

  With the values:

  `MY_PARAM1_DIR = "mypath/to"` and `MY_PARAM3_DIR = "some"`

  With the same path as above:

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  The path would be displayed as follows:

  ```
  ${MY_PARAM1_DIR}/some/directory/with/the/file.txt
  ```

- If you had a parameter:

  `MY_PARAM1`

  With the value:

  `MY_PARAM1 = "mypath/to"`

  With the same path as above:

  ```
  mypath/to/some/directory/with/the/file.txt
  ```

  The path would not be canonicalized at all!

  Although the same string `mypath/to` at the beginning of the path can be expressed using the parameter called `MY_PARAM1`, such parameter does not belong to the group of parameters that are able to canonicalize the paths. For this reason, the path would not be canonicalized with this parameter and the full path would be displayed as is.

Important

Remember that the following paths would not be displayed in **URL File dialog** and **Outline** pane:

```
${MY_PARAM1_DIR}/${MY_PARAM3_DIR}/directory/with/the/file.txt

${MY_PARAM1}/some/directory/with/the/file.txt

mypath/to/${MY_PARAM2_DIR}/directory/with/the/file.txt
```

Using Parameters

When you have defined, for example, a `db_table` (parameter) which means a database table named `employee` (its value) (as above), you can only use `${db_table}` instead of `employee` wherever you are using this database table.

Note

Remember that since the version 2.8.0 of **CloverETL**, you can also use CTL expressions in parameters. Such CTL expressions can use any possibilities of CTL language. However, these CTL expressions must be surrounded by back quotes.

For example, if you define a parameter `TODAY="`today()`"` and use it in your CTL codes, such `${TODAY}` expression will be resolved to the date of this day.

If you want to display a back quote as is, you must use this back quote preceded by back slash as follows: `\``.

Important

CTL1 version is used in such expressions.

As was mentioned above, all can be expressed using a parameter.

Chapter 27. Internal/External Graph Elements

This chapter applies for Metadata (p. 96), Database Connections (p. 150), JMS Connections (p. 159), QuickBase Connections (p. 164), Lookup Tables (p. 166), Sequences (p. 181), and Parameters (p. 187).

There are some properties which are common for all of the mentioned graph elements.

They all can be internal or external (shared).

Internal Graph Elements

If they are internal, they are part of the graph. They are contained in the graph and you can see them when you look at the **Source** tab in the **Graph Editor**.

External (Shared) Graph Elements

If they are external (shared), they are located outside the graph in some external file (in the `meta`, `conn`, `lookup`, `seq` subfolders, or in the `project` itself, by default).

If you look at the **Source** tab, you can only see a link to such external file. It is in that file these elements are described.

Working with Graph Elements

Let us suppose that you have multiple graphs that use the same data files or the same database tables or any other data resource. For each such graph you can have the same metadata, connection, lookup tables, sequences, or parameters. These can be defined either in each of these graphs separately, or all of the graphs can share them.

In addition to metadata, the same is valid for connections (database connections, JMS connections, and QuickBase connections), lookup tables, sequences, and parameters. Also connections, sequences and parameters can be internal and external (shared).

Advantages of External (Shared) Graph Elements

It is more convenient and simple to have one external (shared) definition for multiple graphs in one location, i.e. to have one external file (shared by all of these graphs) that is linked to these various graphs that use the same resources.

It would be very difficult if you worked with these shared elements across multiple graphs separately in case you wanted to make some changes to all of them. In such a case you should have to change the same characteristics in each of the graphs. As you can see, it is much better to be able to change the desired property in only one location - in an external (shared) definition file.

You can create external (shared) graph elements directly, or you can also export or externalize those internal.

Advantages of Internal Graph Elements

On the other hand, if you want to give someone any of your graphs, you must give them not only the graph, but also all linked information. In this case, it is much simpler to have these elements contained in your graph.

You can create internal graph elements directly, or you can internalized those external (shared) elements after they have been linked to the graph.

Changes of the Form of Graph Elements

CloverETL Designer helps you to solve this problem of when to have internal or external (shared) elements:

- **Linking External Graph Elements to the Graph**

  If you have some elements defined in some file or multiple files outside a graph, you can link them to it. You can see these links in the **Source** tab of the **Graph Editor** pane.

- **Internalizing External Graph Elements into the Graph**

  If you have some elements defined in some file or multiple files outside the graph but linked to the graph, you can internalize them. The files still exist, but new internal graph elements appear in the graph.

- **Externalizing Internal Graph Elements in the Graph**

  If you have some elements defined in the graph, you can externalize them. They will be converted to the files in corresponding subdirectories and only links to these files will appear in the graph instead of the original internal graph elements.

- **Exporting Internal Graph Elements outside the Graph**

  If you have some elements defined in the graph, you can export them. New files outside the graph will be created (non-linked to the graph) and the original internal graph elements will remain in the graph.

Chapter 28. Dictionary

Dictionary is a data storage object associated with each run of a graph in **CloverETL**. Its purpose is to provide simple and type-safe storage of the various parameters required by the graph.

It is not limited to storing only input or output parameters but can also be used as a way of sharing data between various components of a single graph.

When a graph is loaded from its XML definition file, the dictionary is initialized based on its definition in the graph specification. Each value is initialized to its default value (if any default value is set) or it must be set by an external source (e.g., **Launch Service**, etc.).

Important

Two versions of Clover Transformation Language differ on whether dictionary must be defined before it is used, or not.

- **CTL1**

  CTL1 allows the user to create dictionary entries without their previous definitions using a set of dictionary functions.

  See Dictionary Functions (p. 661)

- **CTL2**

  Unlike in CTL1, in CTL2 dictionary entries must always be defined first before they are used. The user needs to use standard CTL2 syntax for working with dictionaries. No dictionary functions are available in CTL2.

  See Dictionary in CTL2 (p. 675)

Between two subsequent runs of any graph, the dictionary is reset to the initial or default settings so that all dictionary runtime changes are destroyed. For this reason, dictionary cannot be used to pass values between different runs of the same graph.

In this chapter we will describe how a dictionary should be created and how it should be used:

- Creating a Dictionary (p. 198)
- Using the Dictionary in a Graph (p. 200)

Creating a Dictionary

The dictionary specification provides so called "interface" of the graph and is always required, even, for example, when the graph is used with **Launch Service**.

In the source code, the entries of the dictionary are specified inside the `<Dictionary>` element.

To create a dictionary, right-click the **Dictionary** item in the **Outline** pane and choose **Edit** from the context menu. The **Dictionary** editor will open.

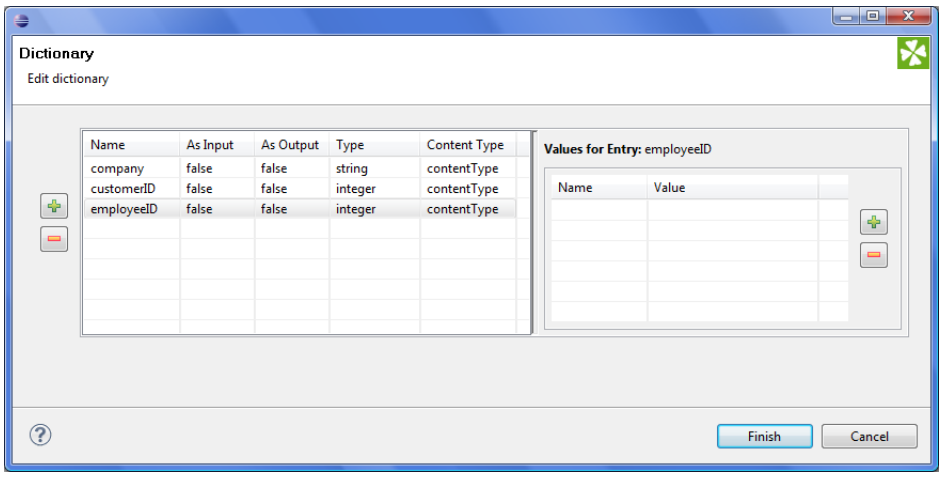


Figure 28.1. Dictionary Dialog with Defined Entries

Click the **Plus sign** button on the left to create a new dictionary entry.

After that, specify its **Name** (Names are case-sensitive, must be unique within the dictionary and should be legal java identifiers.). You must also specify other properties of the entry:

1. `Name`

   Specifies the name of the dictionary entry. Names are case-sensitive, must be unique within the dictionary and should be legal java identifiers.

2. `As Input`

   Specifies whether the dictionary entry can be used as input or not. Its value can be `true` or `false`.

3. `As Output`

   Specifies whether the dictionary entry can be used as output or not. Its value can be `true` or `false`.

4. `Type`

   Specifies the type of the dictionary entry.

   Dictionary types are the following primitive Clover data types:

   - `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, and `string`.

   Any of these can also be accessed in CTL2. See Dictionary in CTL2 (p. 675) for detailed information.

   There are three other data types of dictionary entry (available in Java):

   - `object` - **CloverETL** data type available with **CloverETL Engine**.
   - `readable.channel` - the input will be read directly from the entry by the **Reader** according to the configuration of the **Reader**. Therefore, the entry must contain data in valid format.
   - `writable.channel` - the output will be written directly to this entry in the format given by the output **Writer** (e.g., text file, XLS file, etc.)

5. `contentType`

   You may need to select `contentType`.

   This specifies the content type of the output entry. This content type will used, for example, when the graph is launched via **Launch Service** to send the results back to user.

Each entry can have some properties (name and value). To specify them, click corresponding button on the right and specify the following two properties:

- `Name`

  Specifies the name of the value of corresponding entry.

- `Value`

  Specifies the value of the name corresponding to an entry.

Using the Dictionary in a Graph

The dictionary can be accessed in multiple ways by various components in the graph. It can be accessed from:

Readers and **Writers**. Both of them support dictionaries as their data source or data target via their **File URL** attribute.

The dictionary can also be accessed with CTL or Java source code in any component that defines a transformation (all **Joiners**, **Reformat**, **Normalizer**, etc).

Accessing the Dictionary from Readers and Writers

To reference the dictionary parameter in the **File URL** attribute of a graph component, this attribute must have the following form: `dict:<Parameter name>[:processingType]`. Depending on the type of the parameter in the dictionary and the `processingType`, the value can be used either as a name of the input or output file or it can be used directly as data source or data target (in other words, the data will be read from or written to the parameter directly).

Processing types are the following:

1. **For Readers**

   - `discrete`

     This is the default processing type, needs not be specified.

   - `source`

   See also Reading from Dictionary (p. 254) for information about URL in **Readers**.

2. **For Writers**

   - `stream`

     This is the default processing type, needs not be specified.

   - `discrete`

   See also Writing to Dictionary (p. 266) for information about URL in **Writers**.

For example, `dict:mountains.csv` can be used as either input or output in a **Reader** or a **Writer**, respectively (in this case, the property type is `writable.channel`).

Accessing the Dictionary with Java

To access the values from the Java code embedded in the components of a graph, methods of the `org.jetel.graph.Dictionary` class must be used.

For example, to get the value of the `heightMin` property, you can use a code similar to the following snippet:

```
getGraph().getDictionary().getValue("heightMin")
```

In the snippet above, you can see that we need an instance of `TransformationGraph`, which is usually available via the `getGraph()` method in any place where you can put your own code. The current dictionary is then retrieved via the `getDictionary()` method and finally the value of the property is read by calling the `getValue(String)` method.

Note

For further information check out the **JavaDoc** documentation.

Accessing the Dictionary with CTL2

If the dictionary entries should be used in CTL2, they must be defined in the graph. Working with the entries uses standard CTL2 syntax. No dictionary functions are available in CTL2.

For more information see Dictionary in CTL2 (p. 675).

Accessing the Dictionary with CTL1

Dictionary can be accessed from CTL1 using a set of functions for entries of `string` data type.

Even if the dictionary entries should be used in CTL1, they do not need to be defined in the graph.

For more information see Dictionary Functions (p. 661).

Chapter 29. Notes in the Graphs

The mentioned **Palette of Components** contains also a **Note** icon. When you are creating any graph, you can paste one or more notes in the **Graph Editor** pane. To do that, click this **Note** icon in the **Palette**, move to the **Graph Editor** and click again. After that, a new **Note** will appear there. It bears a **New note** label on it.

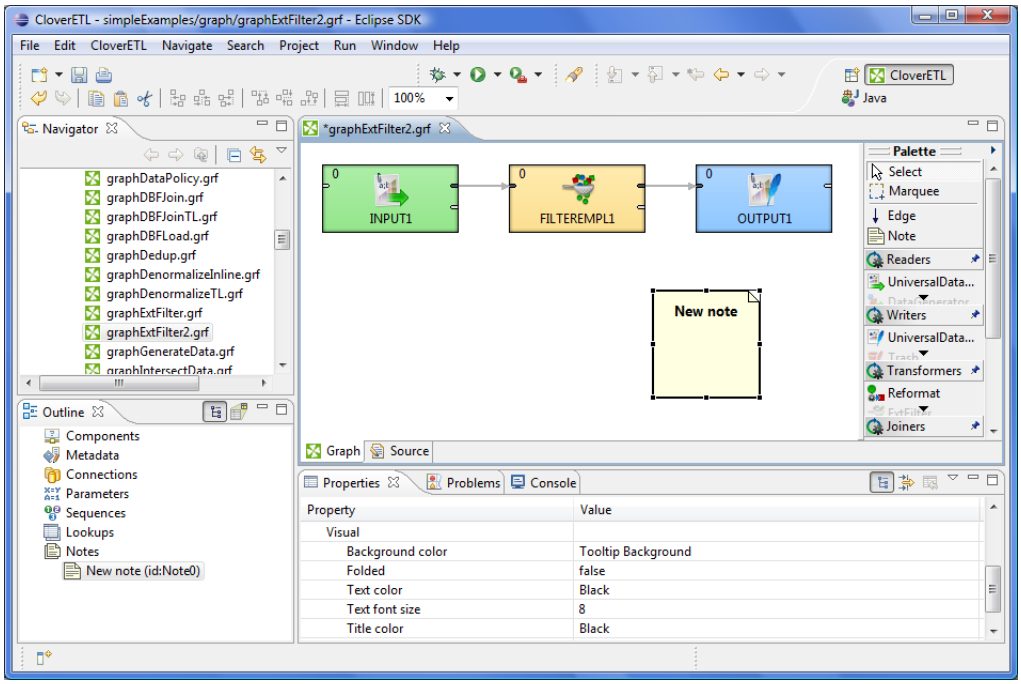


Figure 29.1. Pasting a Note to the Graph Editor Pane

You can also enlarge the **Note** by clicking it and by dragging any of its margins that have been highlighted after this click.

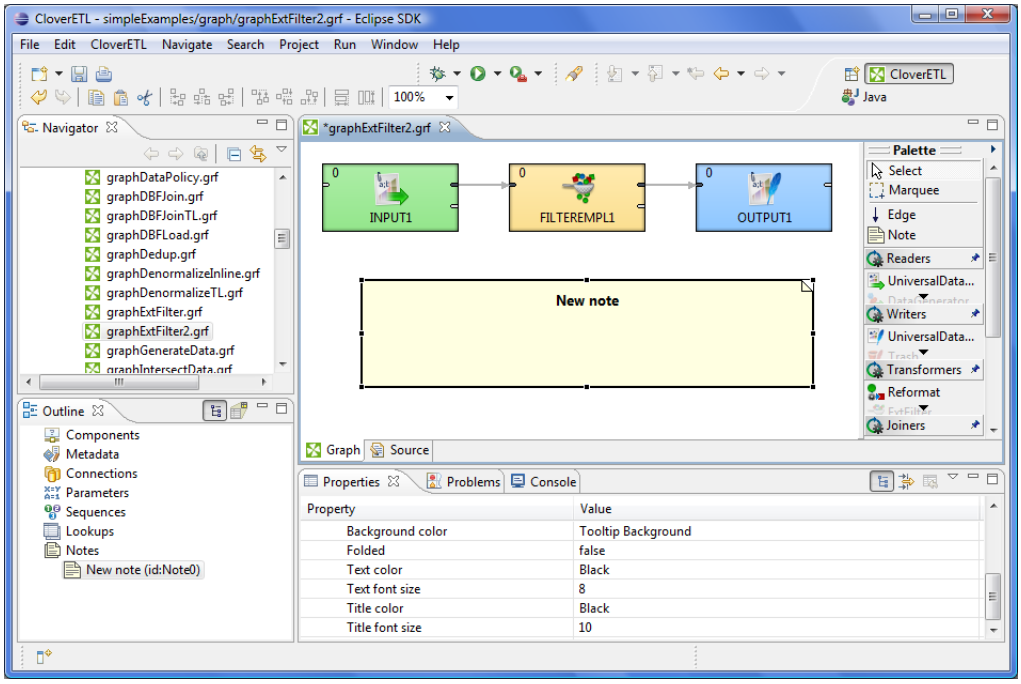


Figure 29.2. Enlarging the Note

At the end, click anywhere outside the **Note** so that the highlighting disappears.

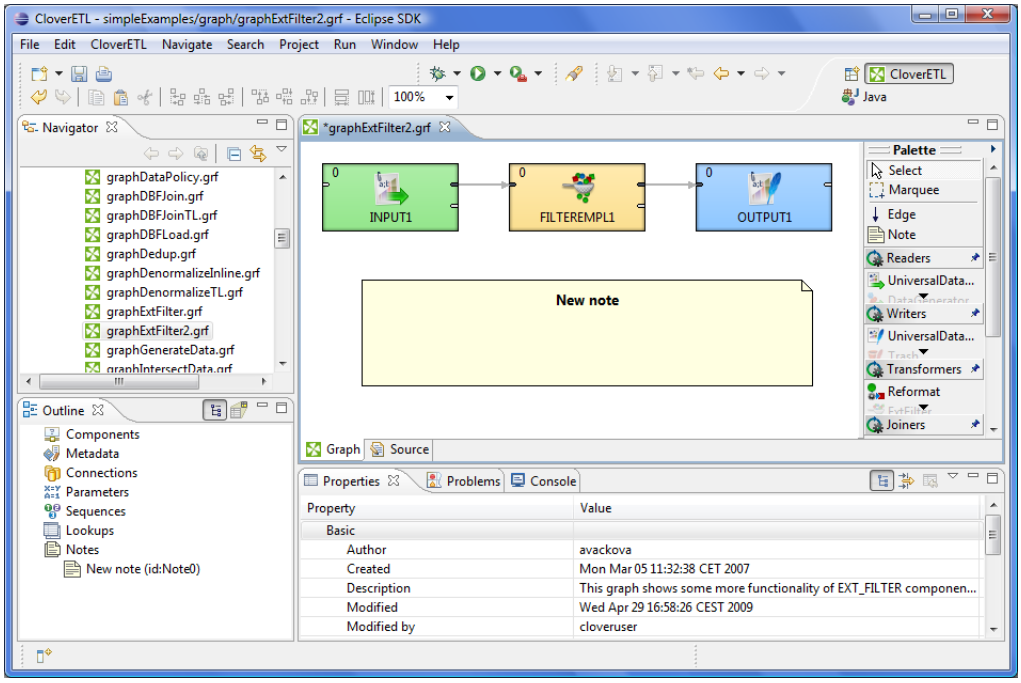


Figure 29.3. Highlighted Margins of the Note Have Disappeared

When you want to write some description in the **Note** of what the graph should do, click inside the **Note** two times. After the first click, the margins are highlighted, after the second one, a white rectangular space appears in the **Note**. If you make this click on the **New note** label, this label appears in the rectangle and you can change it.

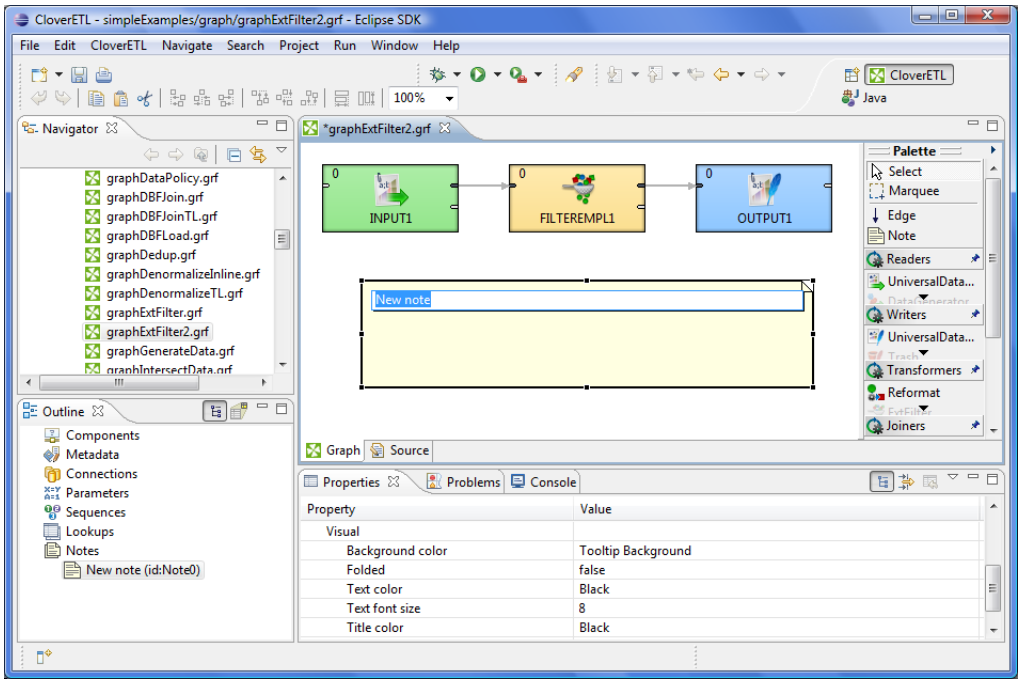


Figure 29.4. Changing the Note Label

If you make this click outside the **New note** label, a new rectangle space appears and you can write any description of the graph in it. You can also enlarge the space and write more of the text in it.

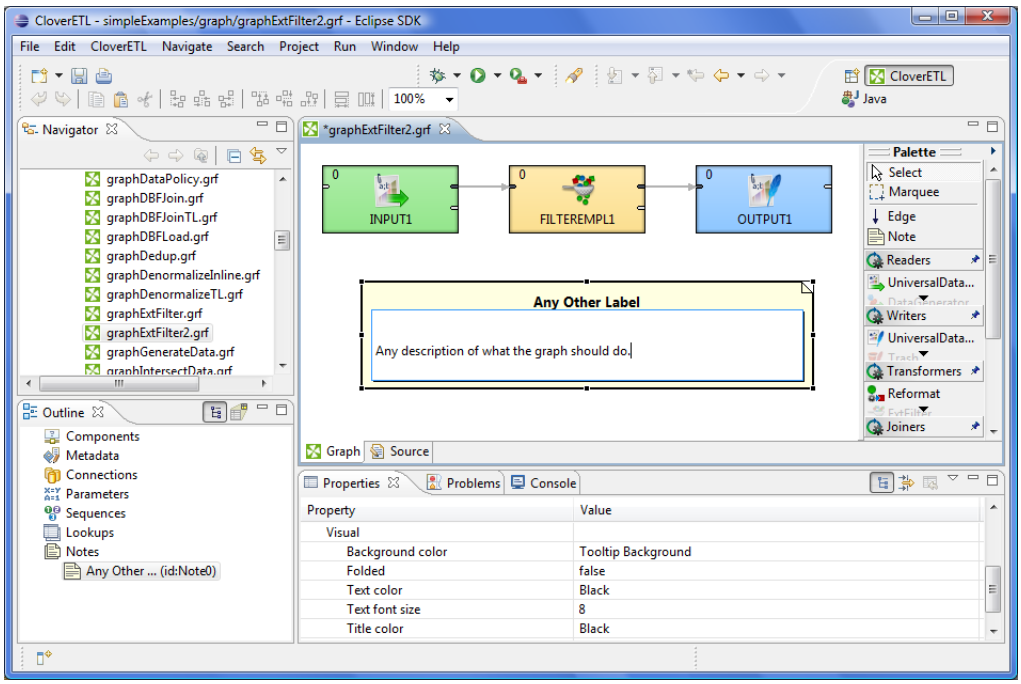


Figure 29.5. Writing a New Description in the Note

The resulting **Note** can be as follows:

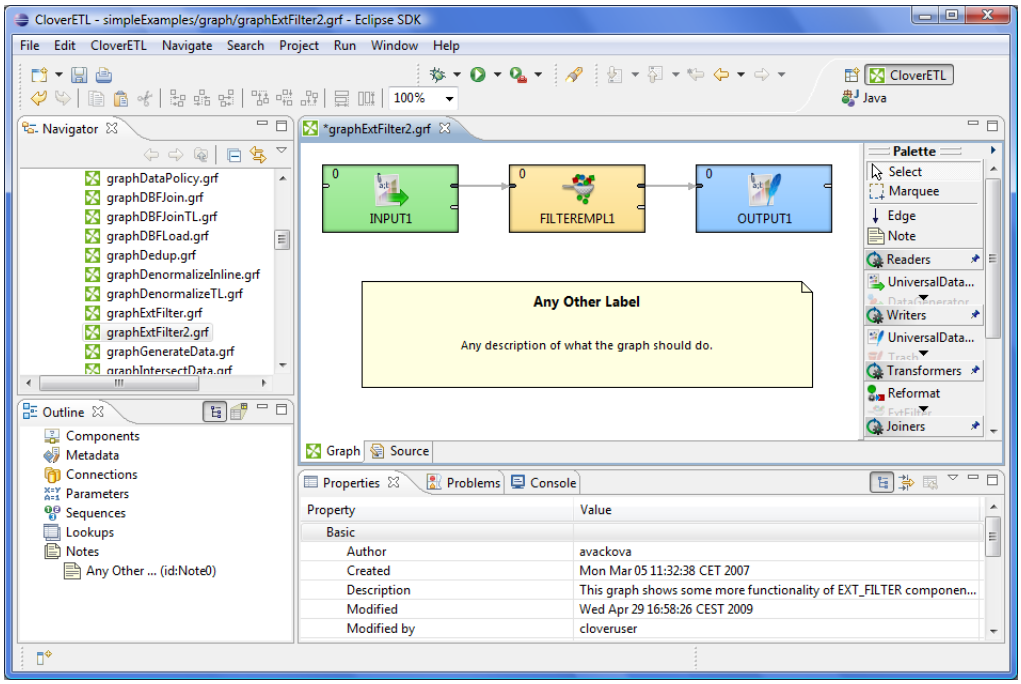


Figure 29.6. A New Note with a New Description

This way, you can paste more **Notes** in one graph.

Remember that if you type any parameter in a **Note**, not the parameter, but its value will be displayed!

Also must be mentioned that each component lying in a **Note** will be moved together with the **Note** if you move the **Note**.

You can also fold any **Note** by selecting the **Fold** item from the context menu. From the resulting **Note** only the label will be visible and it will look like this:

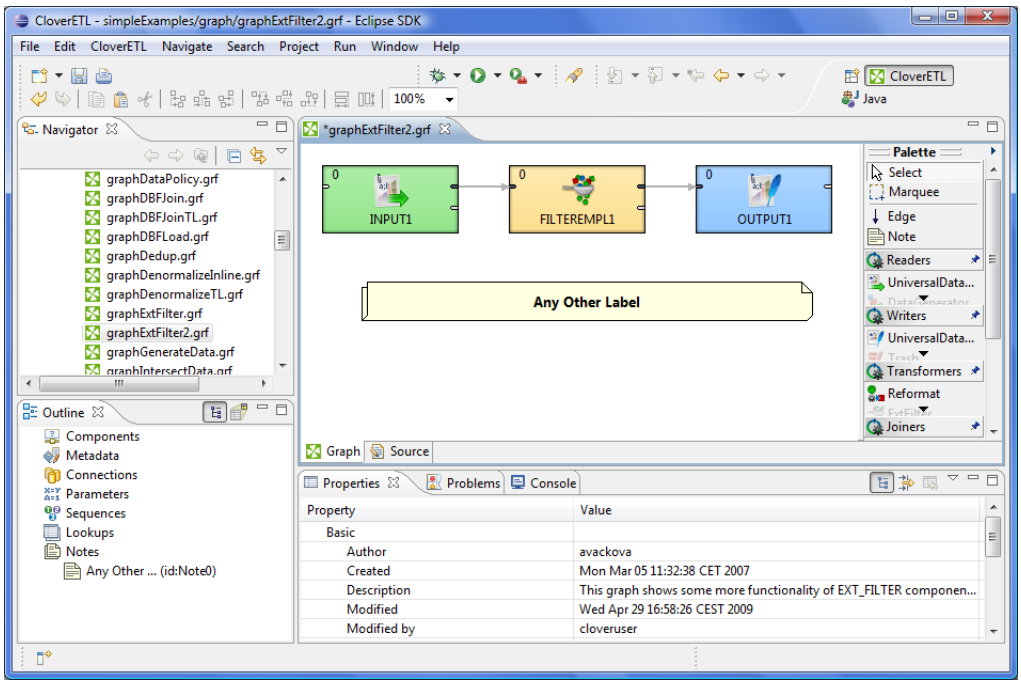


Figure 29.7. Folding the Note

You can also set up many properties of any **Note** when you click the **Note** and switch to the **Properties** tab. You can see both **Text** and **Title** of the **Note** there. Each of them can be changed in this tab. You can also decide whether the **Text** should be aligned to the **Left**, **Center** or **Right**. **Title** is aligned to the center by default. Both **Text** and **Title** can have some specified colors, you can select one from the combo list. They both are black and the background has the color of tooltip background by default. Any of these properties can be set here. The default font sizes are also displayed in this tab and can be changed as well. If you want to fold the **Note**, set the **Folded** attribute to true. Each **Note** has an **ID** like any other graph component.

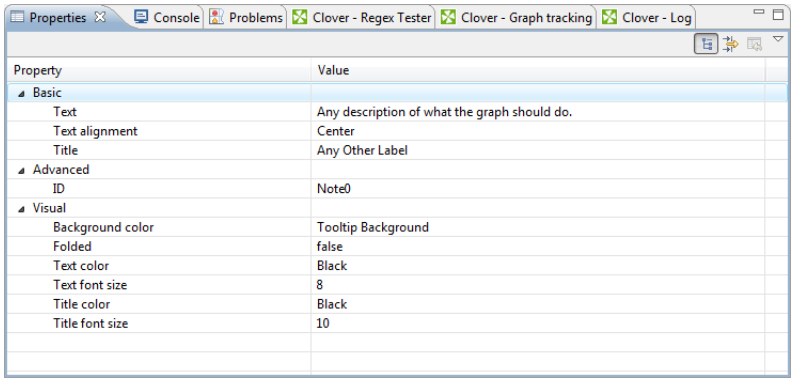


Figure 29.8. Properties of a Note

Chapter 30. Search Functionality

If you select **Search** →**Search...** from the main menu of **CloverETL Designer**, a window with following tabs opens:

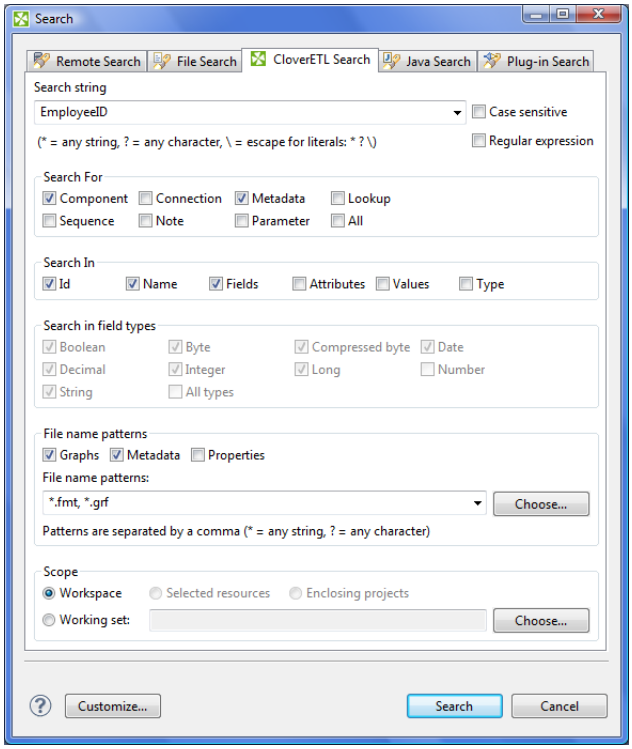


Figure 30.1. CloverETL Search Tab

In the **CloverETL search** tab, you need to specify what you wish to find.

First, you can specify whether searching should be case sensitive, or not. And whether the string typed in the **Search string** text area should be considered to be a regular expression or not.

Second, you need to specify what should be searched: **Components**, **Connections**, **Lookups**, **Metadata**, **Sequences**, **Notes**, **Parameters** or **All**.

Third, you should decide in which characteristics of objects mentioned above searching should be done: **Id**, **Names**, **Fields**, **Attributes**, **Values** or **Type**. (If you check the **Type** checkbox, you will be able to select from all available data types.)

Fourth, decide in which files searching should be done: **Graphs** (`*.grf`), **Metadata** (`*.fmt`) or **Properties** (files defining parameters: `*.prm`). You can even choose your own files by typing or by clicking the button and choosing from the list of file extensions.

Remember that, for example, if you search metadata in graphs, both internal and external metadata are searched, including fields, attributes and values. The same is valid for internal and external connections and parameters.

As the last step, you need to decide whether searching should regard whole **workspace**, only **selected resources** (selection should be done using **Ctrl**+**Click**), or the projects enclosing the selected resources (**enclosing projects**). You can also define your **working set** and **customize** your searching options.

When you click the **Search** button, a new tab containing the results of search appears in the **Tabs** pane. If you expand the categories and double-click any inner item, it opens in text editor, metadata wizard, etc.

If you expand the **Search** tab, you can see the search results:

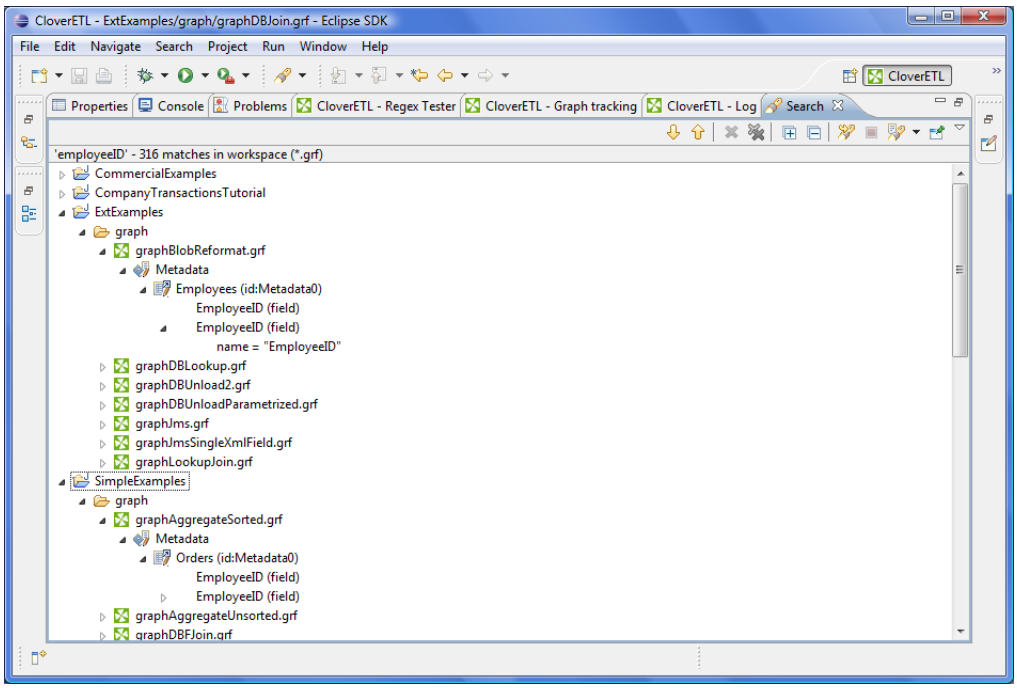


Figure 30.2. Search Results

Chapter 31. Transformations

Each transformation graph consists of components. All components process data during the graph run. Some of all components process data using so called transformation.

Transformation is a piece of code that defines how data on the input is transformed into that on the output on its way through the component.

Note

Remember that transformation graph and transformation itself are different notions. Transformation graph consists of components, edges, metadata, connections, lookup tables, sequences, parameters, and notes whereas transformation is defined as an attribute of a component and is used by the component. Unlike transformation graph, transformation is a piece of code that is executed during graph execution.

Any transformation can be defined by defining one of the following three attributes:

- Each transformation is defined using one of the three attributes of a component:
  - **Transform**, **Denormalize**, **Normalize**, etc.
  - **Transform URL**, **Denormalize URL**, **Normalize URL**, etc.
    - When any of these attributes is defined, you can also specify its encoding: **Transform source charset**, **Denormalize source charset**, **Normalize source charset**, etc.
  - **Transform class**, **Denormalize class**, **Normalize class**, etc.
- In some transforming components, transformation is required, in others, it is only optional.

  For a table overview of components that allow or require a transformation see Transformations Overview (p. 237).
- Each transformation can always be written in Java, mostly transformation can also be written in Clover Transformation Language.

  Since version 3.0 of **CloverETL**, Clover Transformation Language (CTL) exists in two versions: CTL1 and CTL2.

  See Part VIII, CTL - CloverETL Transformation Language (p. 588) for details about Clover Transformation Language and each of its versions.

See Defining Transformations (p. 234) for more detailed information about transformations.

Chapter 32. Fact table loader

The Fact Table Loader (FTL) is designed to reduce time, when a user needs to create a new data transformation for creating and inserting a new fact into the fact table of the data warehouse.

Example 32.1. Example of usage

Let's assume, that the user has some input data from a production database saved in a text file. From this data he needs to create a fact which is then inserted into a fact table of the data warehouse. In the process of creating mentioned fact, user has to use some of the dimensional tables of the same data warehouse to find a dimension key for the row where the field from the text file equals the field of the selected column. Using **CloverETL** this can be done by using the **ExtHashJoin** and **DBLookup** components. More, let's assume the user has to process 4 or more dimensional tables. To create a data transformation (graph) which has this functionality the user needs to use 10 or more components and set all the required options. It takes some time. For this there is the FTL tool.

FTL is a wizard integrated into **CloverETL Designer**. In the wizard the user inputs relevant information and the wizard creates a new **CloverETL** graph with the requested data transformation.

In the following sections we will show the following:

1. How **Fact Table Loader** wizard should be launched. See Launching Fact Table Loader Wizard (p. 209).

2. How should be worked with **Fact Table Loader** wizard. See Working with Fact Table Loader Wizard (p. 211).

3. How a created graph looks like. See Created graph (p. 217).

Launching Fact Table Loader Wizard

In this section two ways will be shown how to launch the FTL wizard and difference between them will be discussed. The difference is in the first way, the wizard is enabled to use a graph parameter file and in the second it is disabled. Project parameters are used to parameterize paths to files.

See:

- Wizard with project parameters file enabled (p. 209)
- Wizard with the project parameter file disabled (p. 211)

Wizard with project parameters file enabled

To enable the wizard to use the graph parameter file the user needs to select an existing **CloverETL Project** or any subdirectory or file in it.

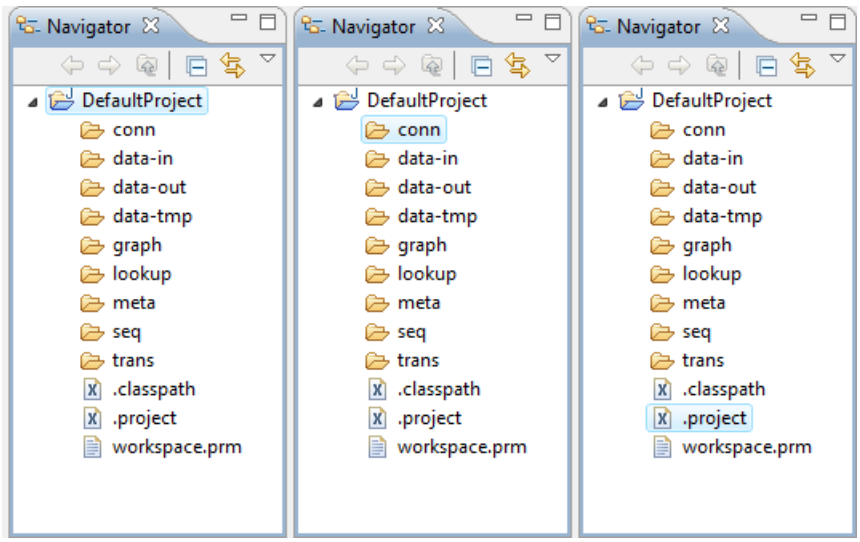


Figure 32.1. Valid selections

When the selection is chosen using right-click and selecting **New →Other...** from the context menu or **File →New →Other...** from the main menu or simply by **Ctrl+N**.

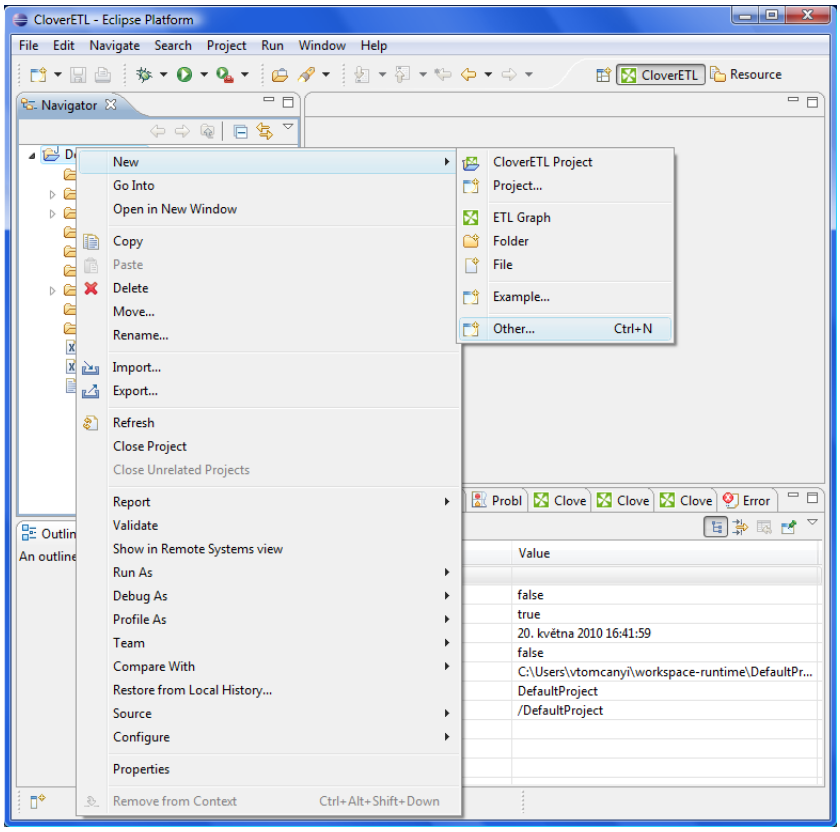


Figure 32.2. How to select the FTL wizard

From options in the displayed window select Fact Table Load.

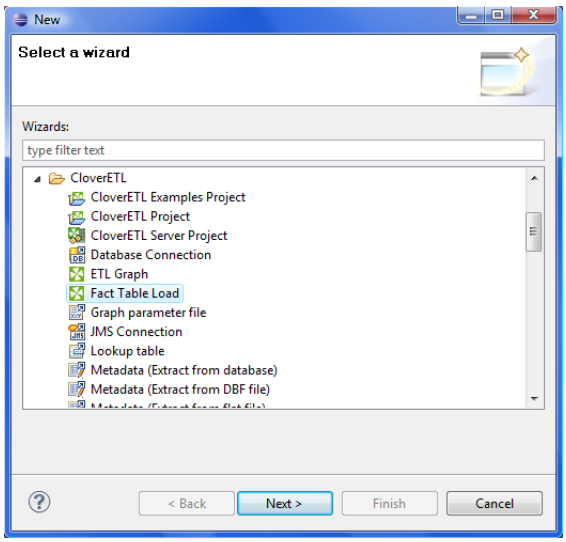


Figure 32.3. Select a wizard (new wizard selection window)

Wizard with the project parameter file disabled

To make the wizard not use the project parameters the user needs to uncheck all of the selected directories, subdirectories or files in the project. This can be done by holding the **Crtl** button and clicking on the selected items.

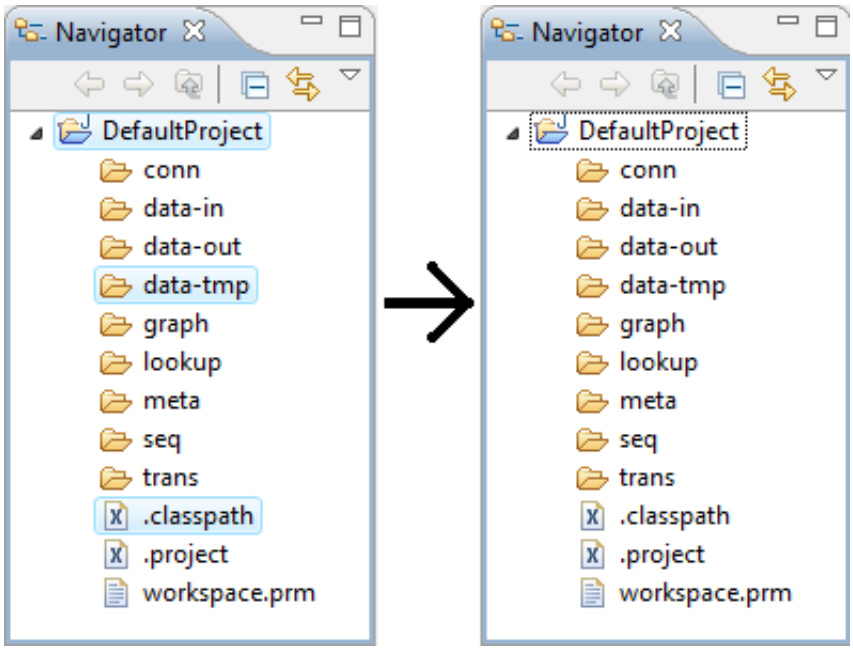


Figure 32.4. Deselection

When nothing is selected: **File** →**New** →**Other...** or simply press **Ctrl**+**N** and follow the steps in Wizard with project parameters file enabled (p. 209).

Working with Fact Table Loader Wizard

Right after the **Next** button, in the new wizard selection window, is pressed a new FTL wizard is launched. This case covers the wizard launched with an associated project.

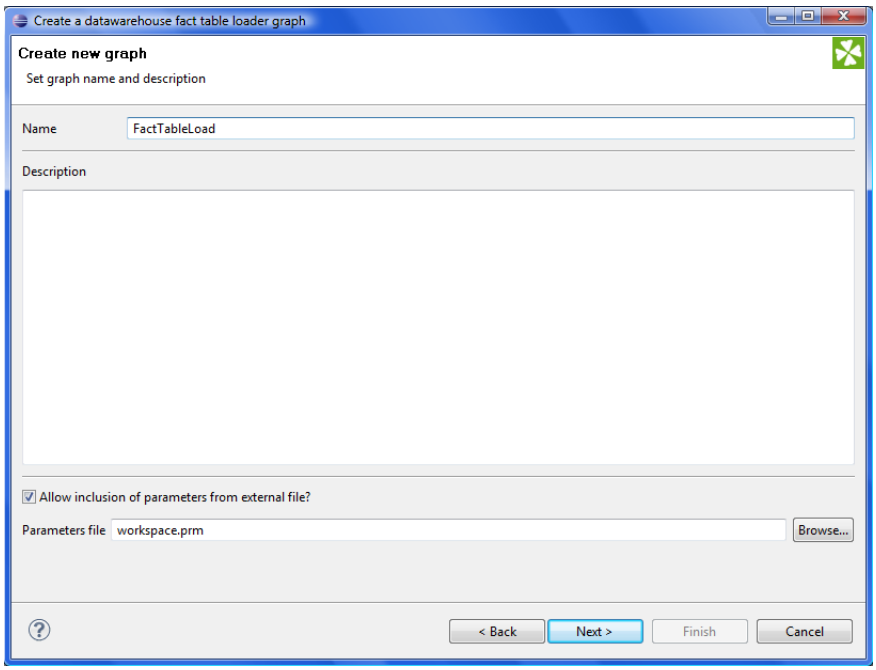


Figure 32.5. New Graph Name Page

In this page the user has to enter the graph name and can check or uncheck the checkbox which determines whether the wizard is able to use the project parameters.

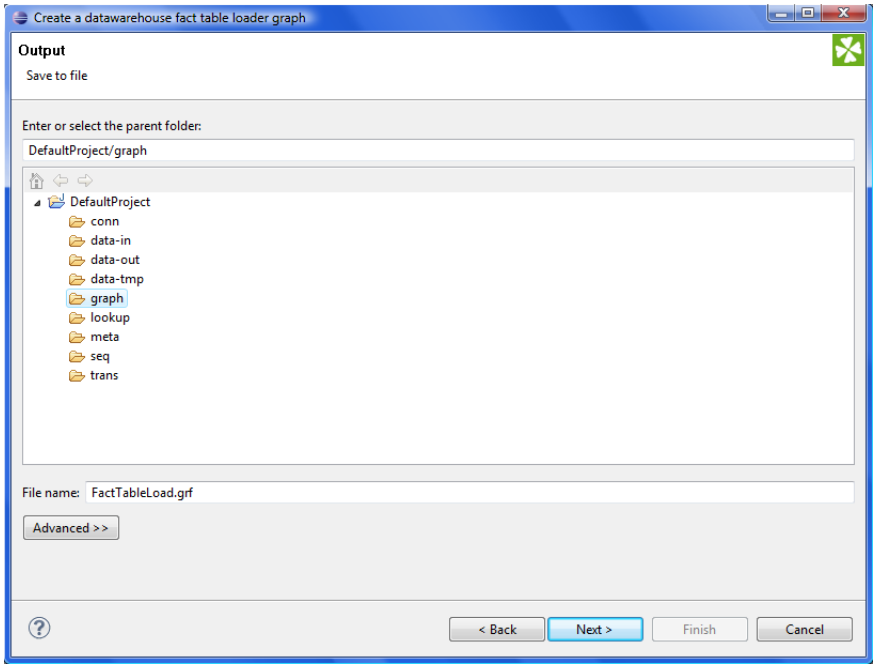


Figure 32.6. Output Page

After clicking on the **Next** button the wizard displays the **Output Page**. In this page the user can choose the directory in which the created graph will be saved.

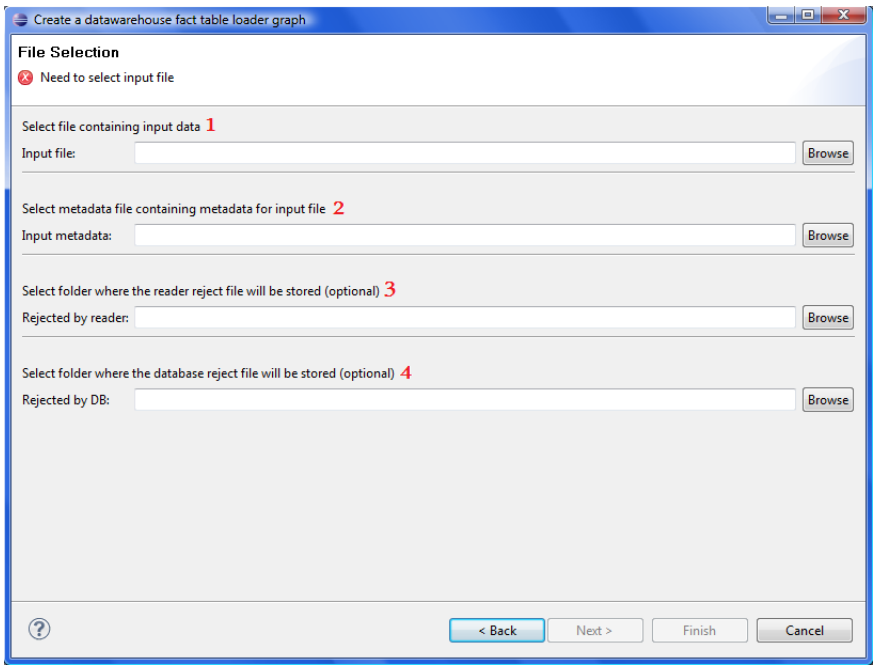


Figure 32.7. File Selection Page

In this window the user has to enter paths to the needed files.

1. The first row contains the path to the file with the input data.
2. The second row contains the path to the file with the metadata definition for the input file (`.fmt` file)
3. The third row contains the path to the directory where the reject file with the data rejected by the reader will be stored. This is optional - if this row is empty no reject file will be created.
4. The forth row contains the path to the directory where the reject file with the data rejected by the database writer will be stored, this also optional.

If the user clicks on the **Browse** button they can browse the project using the special dialog.

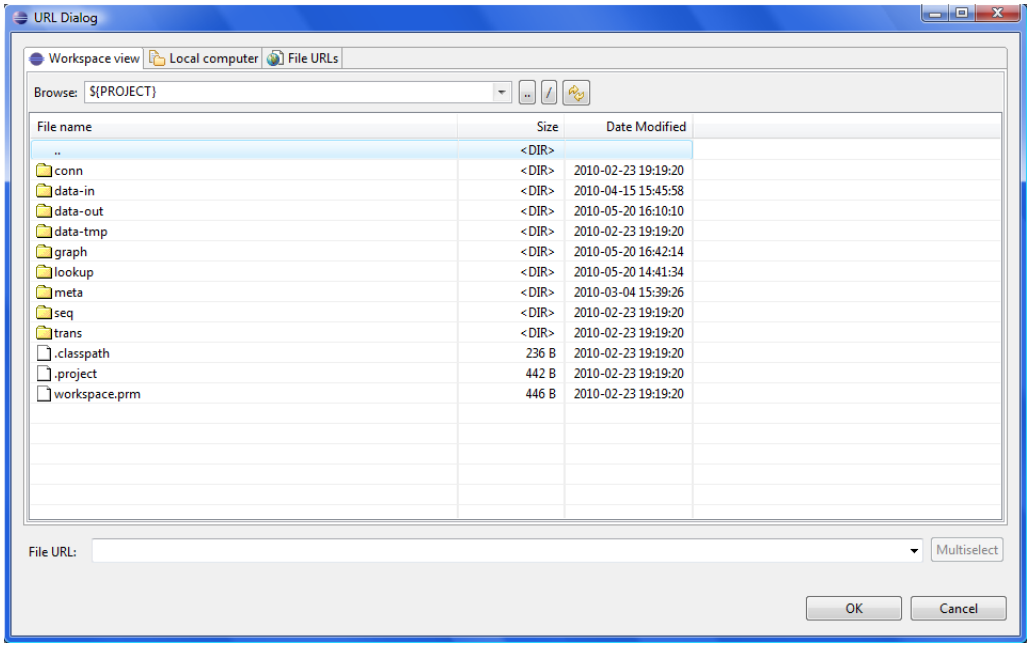


Figure 32.8. URL Dialog

In this dialog the user can browse the project or the whole filesystem. The dialog returns the path to the file in parameter form, after the user clicks the **OK** button (if the project is associated).

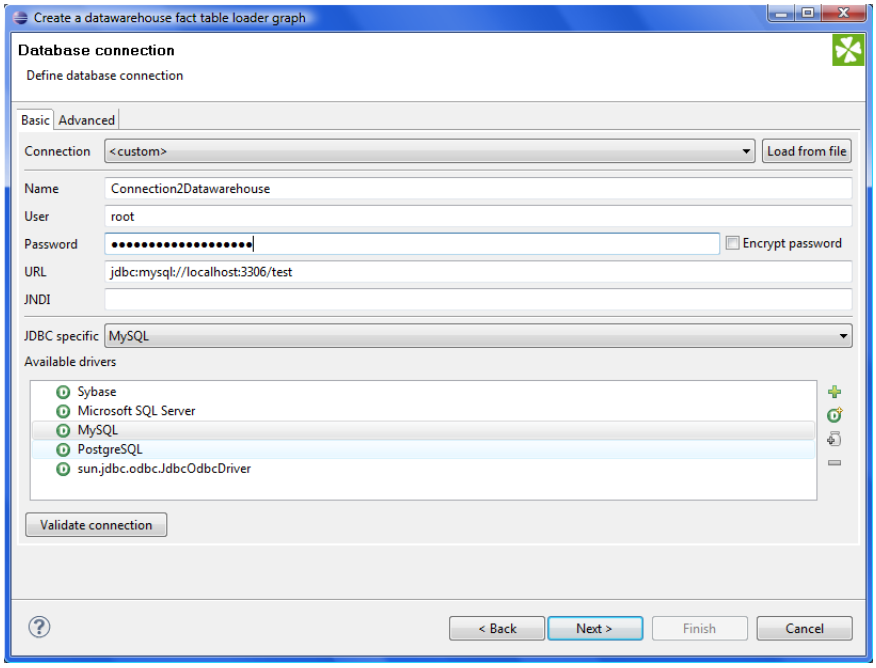


Figure 32.9. Database Connection Page

The database connection page is displayed, after the user clicks on the **Next** button in the **File selection page**. In this page, the user provides the requested information needed to establish a connection to a database. Only one connection is created, so the fact table and the dimension tables of the data warehouse have to be in the same database.

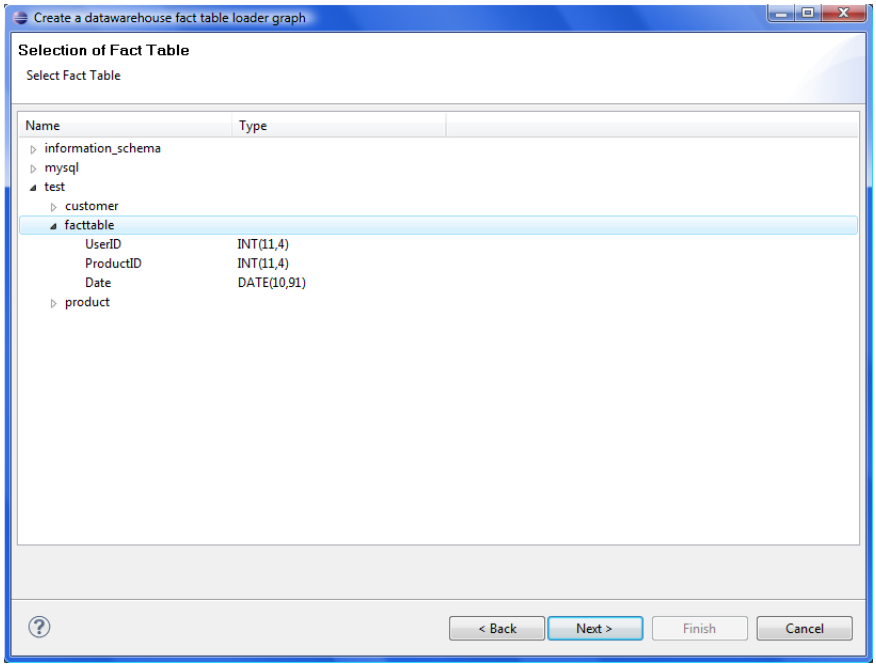


Figure 32.10. Fact Table Selection Page

The user has to select one table from the database which will be used as the fact table. Only one table can be selected. The created fact will be inserted into this table.

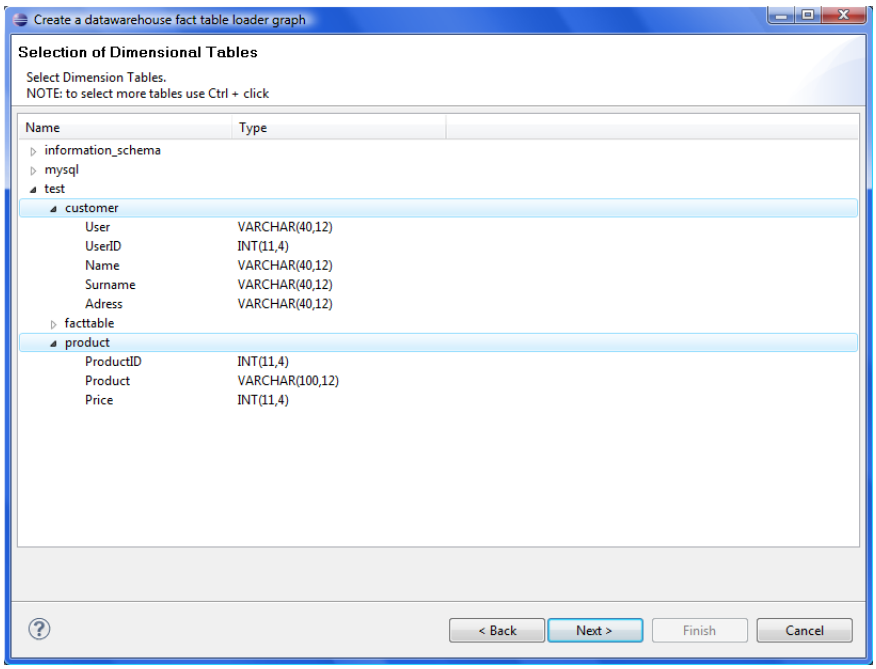


Figure 32.11. Dimension Table Selection Page

The user has to select one or more tables from the database in this window. These tables will be considered the dimension tables of the data warehouse.

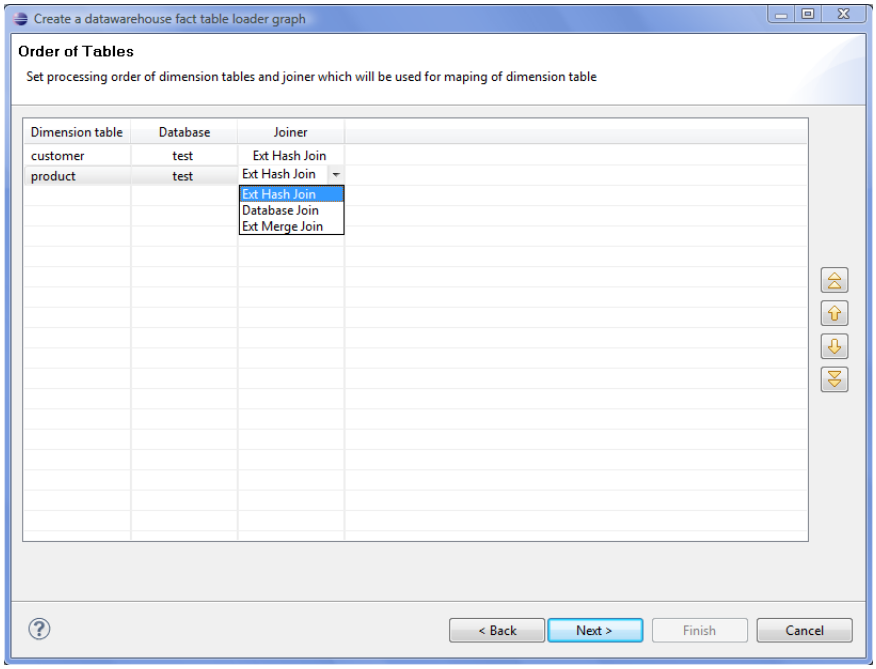


Figure 32.12. Order Table Page

In this window the user is able to see selected dimension tables and databes, select joiner which will be used for mapping and change the order of the dimension tables shown in this window. This is useful if the dimension key of one of the dimension tables is needed for searching within another dimension table.

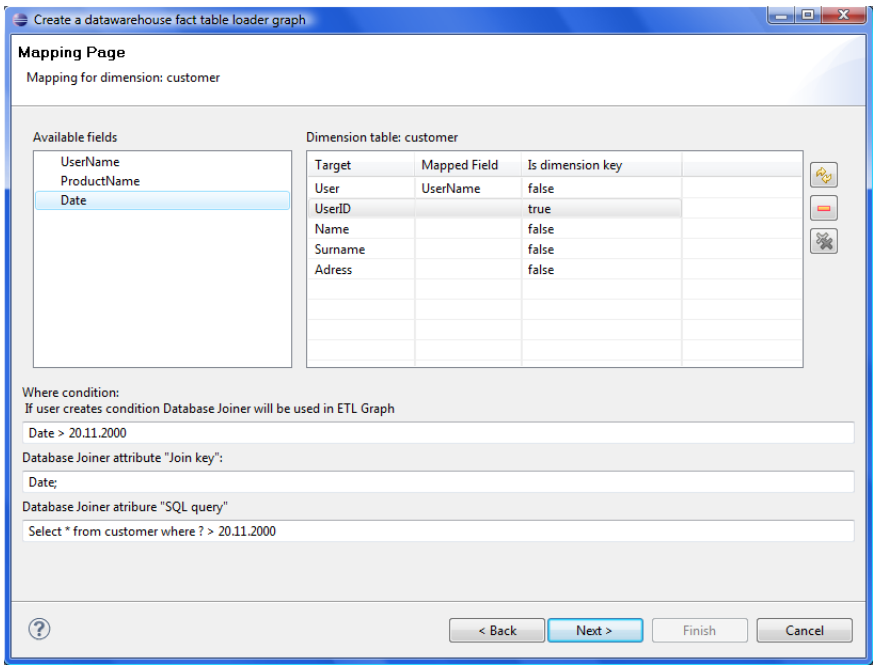


Figure 32.13. Mapping Page

The user maps the available fields onto fields of the dimension table using drag-and-drop. The user can also choose which field of the dimension table will be considered the dimension key and will be added to the fact. This can be done by setting the boolean value of the column **Is dimension key** to `true`. In this window the SQL condition can be also created. The user can create a `where condition` from which are the other **DBJoin** attributes generated. If the user creates `where condition` for the dimension table and he selected the **ExtHashJoin** or **ExtMergeJoin**, **DBJoin** will be used in a graph instead. A **Mapping Page** is shown for every dimension table.

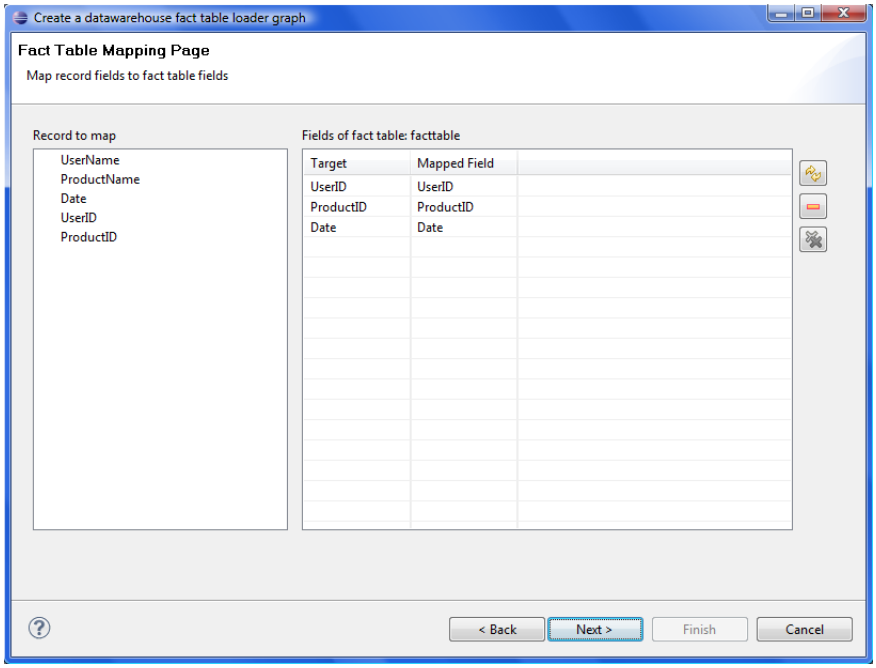


Figure 32.14. Fact Mapping Page

The user maps the created fact to the fact table in this window. The editor uses an auto-rule which matches fields by name if the names are equal.

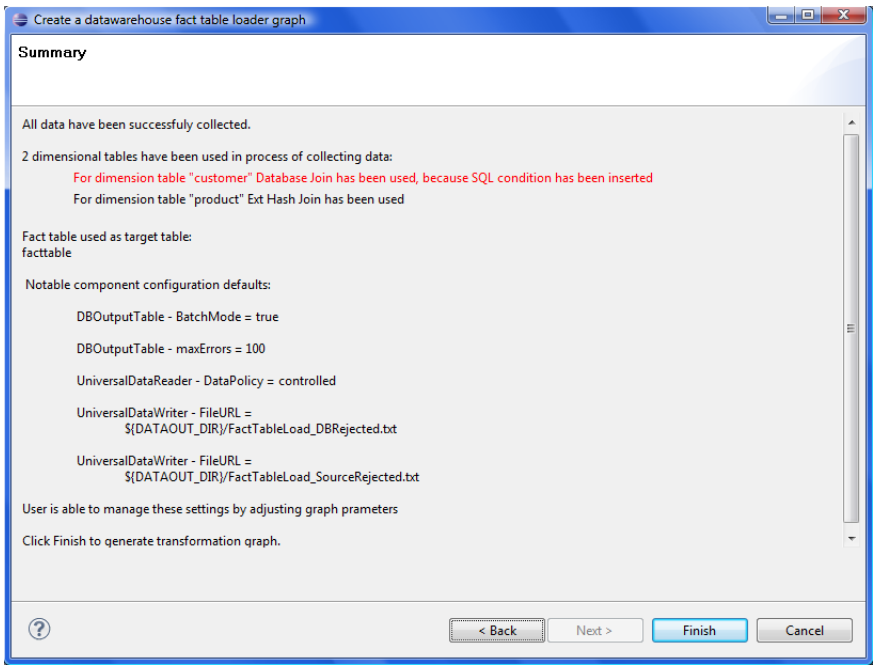


Figure 32.15. Summary Page

The last page of the wizard gives the user the information from the wizard. It also informs the user about new parameters which will be created in the **CloverETL** graph. The last three parameters are not created if the user did not enter a path for an output directory for rejected files. The user is also warned if **DBJoin** is used insted of user selected joiner.

Created graph

After clicking on the **Finish** button the wizard generates a new **CloverETL** graph. The user just has to save this graph and then it is ready to run.

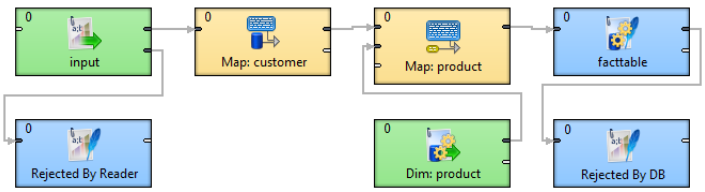


Figure 32.16. Created Graph

This graph also contains created parameters.

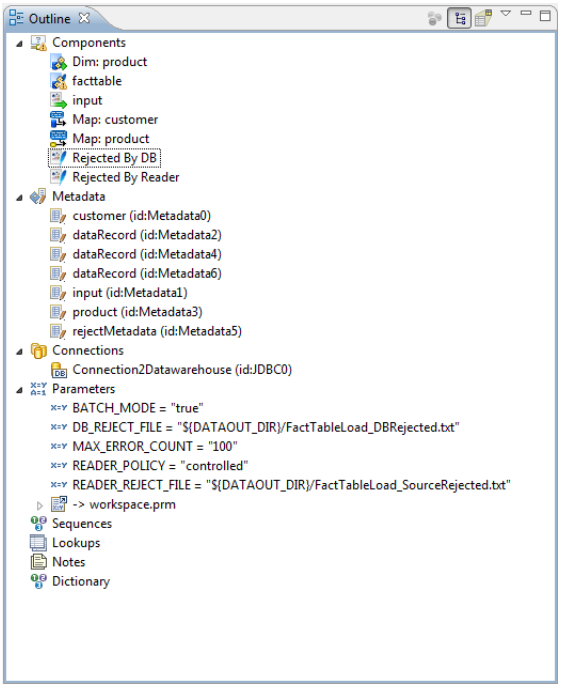


Figure 32.17. Graph Parameters

Part VI. Components Overview

Chapter 33. Introduction to Components

For basic information about components see Chapter 17, Components (p. 84).

In the palette of components of the **Graph Editor**, all components are divided into following groups: Readers (p. 287) Writers (p. 360) Transformers (p. 439) Joiners (p. 523) Cluster Components (p. 553), and Others (p. 559). We will describe each group step by step.

One more category is called **Deprecated**. It should not be used any more and we will not describe them.

So far we have talked about how to paste components to graphs. We will now discuss the properties of components and the manner of configuring them. You can configure the properties of any graph component in the following way:

- You can simply double-click the component in the **Graph Editor**.
- You can do it by clicking the component and/or its item in the **Outline** pane and editing the items in the **Properties** tab.
- You can select the component item in the **Outline** pane and press **Enter**.
- You can also open the context menu by right-clicking the component in the **Graph Editor** and/or in the **Outline** pane. Then you can select the **Edit** item from the context menu and edit the items in the **Edit component** wizard.

Chapter 34. Palette of Components

CloverETL Designer provides all components in the **Palette of Components**. However, you can choose which should be included in the **Palette** and which not. If you want to choose only some components, select **Window →Preferences...** from the main menu.

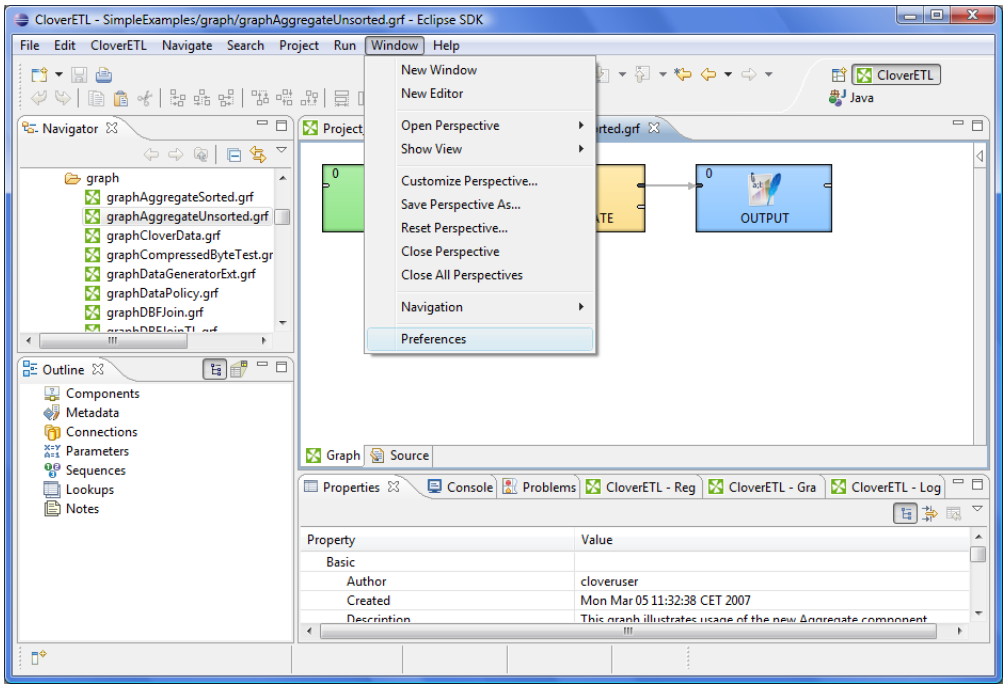


Figure 34.1. Selecting Components

After that, you must expand the **CloverETL** item and choose **Components in Palette**.

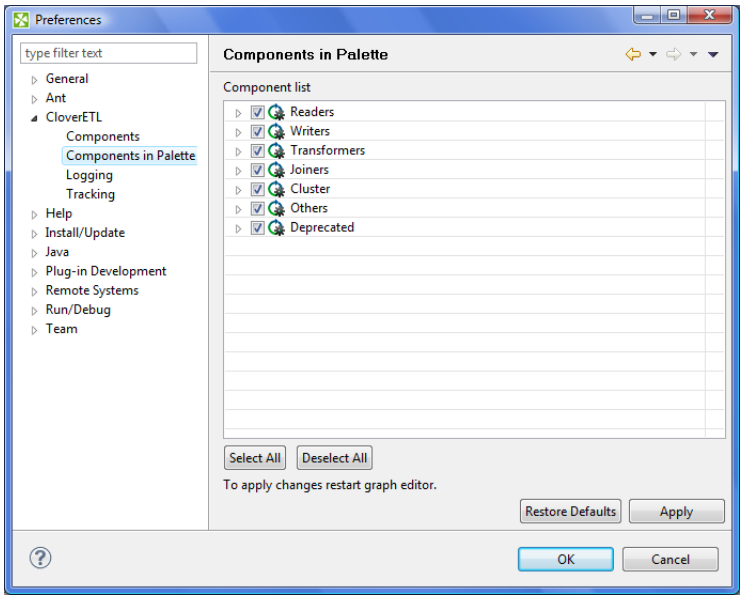


Figure 34.2. Components in Palette

In the window, you can see the categories of components. Expand the category you want and uncheck the checkboxes of the components you want to remove from the palette.

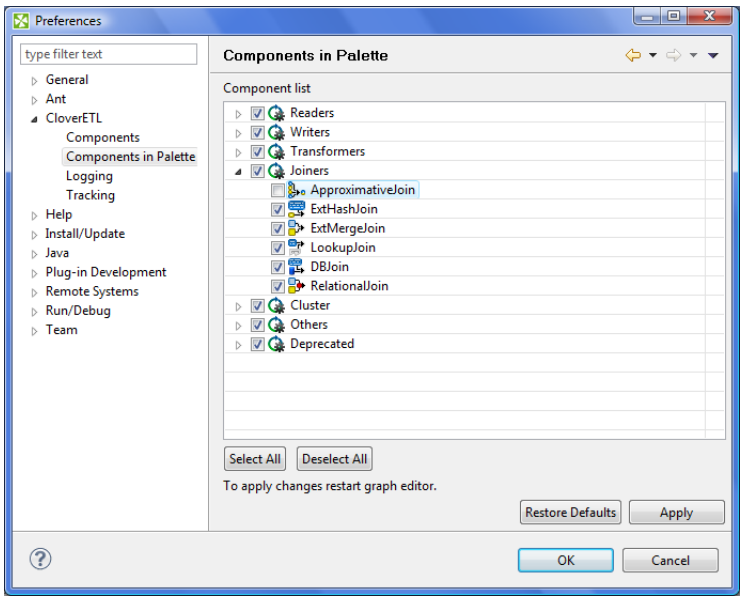


Figure 34.3. Removing Components from the Palette

Then you only need to close and open graph and the components will be removed from the **Palette**.

Chapter 35. Common Properties of All Components

Some properties are common for all components. They are the following:

- Any time, you can choose which components should be displayed in the **Palette of Components** and which should be removed from there (Chapter 34, Palette of Components (p. 221)).
- Each component can be set up using **Edit Component Dialog** (Edit Component Dialog (p. 224)).

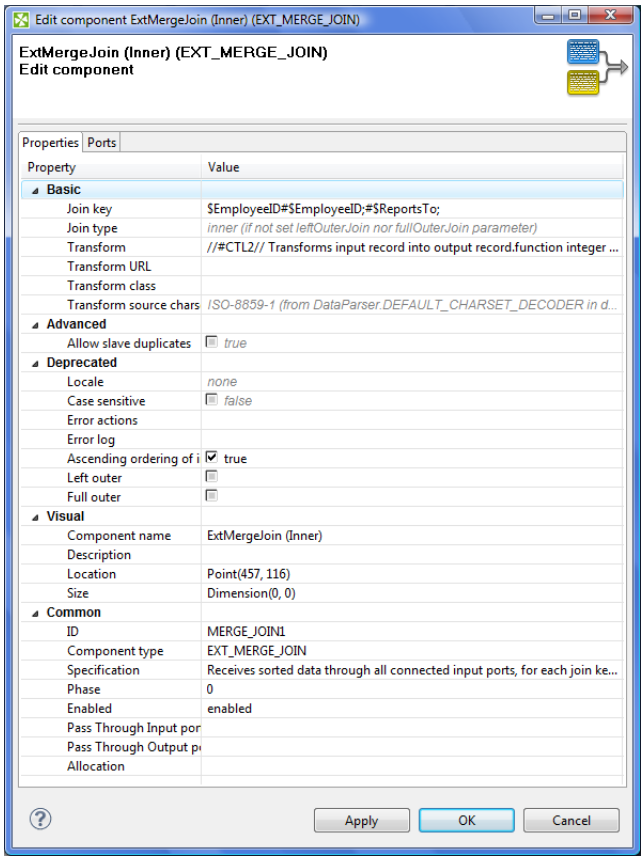


Figure 35.1. Edit Component Dialog (Properties Tab)

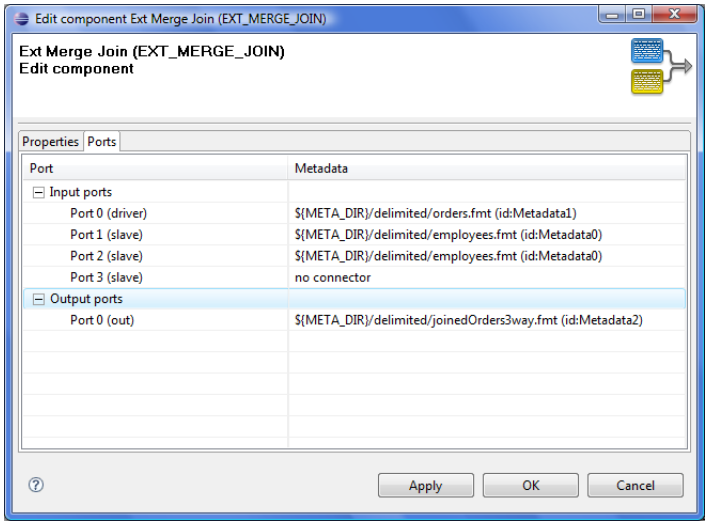


Figure 35.2. Edit Component Dialog (Ports Tab)

Among the properties that can be set in this **Edit Component Dialog**, the following are described in more detail:

- Each component bears a label with **Component name** (Component Name (p. 227)).
- Each graph can be processed in phases (Phases (p. 228)).
- Components can be disabled (Enable/Disable Component (p. 229)).
- Components can be switched to **PassThrough mode** (PassThrough Mode (p. 230)).

Edit Component Dialog

The **Edit component** dialog allows you to edit component attributes in each component. You can access the dialog by double-clicking the component that has been pasted in the **Graph Editor** pane.

This dialog consists of two tabs: **Properties** tab and **Ports** tab.

- **Properties** tab presents an overview of component attributes that can be set.
- **Ports** tab presents an overview of both input and output ports and their metadata.

Properties Tab

In the **Properties** dialog, all attributes of the components are divided into 5 groups: **Basic**, **Advanced**, **Deprecated**, **Visual** and **Common**.

Only the last two groups (**Visual** and **Common**) can be set in all of them.

The others groups (**Basic**, **Advanced**, and **Deprecated**) differ in different components.

However, some of them may be common for most of them or, at least, for some category of components (**Readers**, **Writers**, **Transformers**, **Joiners**, or **Others**).

- **Basic**. These are the basic attributes of the components. The components differ by them. They can be either required, or optional.

  May be specific for an individual component, for a category of components, or for most of the components.

  - **Required**. Required attributes are marked by warning sign. Some of them can be expressed in two or more ways, two or more attributes can serve to the same purpose.
  - **Optional**. They are displayed without any warning sign.

- **Advanced**. These attributes serve to more complicated (advanced) settings of the components. They differ in different components.

  May be specific for an individual component, for a category of components, or for most of the components.

- **Deprecated**. These attributes were used in older releases of **CloverETL Designer** and they still remain here and can be used even now. However, we suggest you do not use them unless necessary.

  May be specific for an individual component, for a category of components, or for most of the components.

- **Visual**. These are the attributes that can be seen in the graph.

  These attributes are common for all components.

  - **Component name**. Component name is a label visible on each component. It should signify what the component should do. You can set it in the **Edit component** dialog or by double-clicking the component and replacing the default component name.

    See Component Name (p. 227) for more detailed information.

- **Description**. You can write some description in this attribute. It will be displayed as a hint when the cursor appears on the component. It can describe what this instance of the component will do.

- **Common**.

  Also these attributes are common for all components.

  - **ID**. The ID identifies the component among all of the other components within the same component type. We do not describe it in more detail.

  - **Component type**. This describes the type of the component. By adding a number to this component type, you can get a component ID. We will not describe it in more detail.

  - **Specification**. This is the description of what this component type can do. It cannot be changed. We will not describe it in more detail.

  - **Phase**. This is an integer number of the phase to which the component belongs. All components with the same phase number run in parallel. And all phase numbers follow each other. Each phase starts after the previous one has terminated successfully, otherwise, data parsing stops.

    See Phases (p. 228) for more detailed description.

  - **Enabled**. This attribute can serve to specify whether the component should be **enabled**, **disabled** or whether it should run in a **passThrough** mode. This can also be set in the **Properties** tab or in the context menu (except the **passThrough** mode).

    See Enable/Disable Component (p. 229) for a more detailed description.

  - **Pass Through Input port**. If the component runs in the **passTrough** mode, you should specify which input port should receive the data records and which output port should send the data records out. This attribute serves to select the input port from the combo list of all input ports.

    See PassThrough Mode (p. 230) for more detailed description.

  - **Pass Through Output port**. If the component runs in the **passTrough** mode, you should specify which input port should receive the data records and which output port should send the data records out. This attribute serves to select the output port from the combo list of all output ports.

    See PassThrough Mode (p. 230) for more detailed description.

  - **Allocation**. If the graph is executed by a Cluster of **CloverETL Servers**, this attribute must be specified in the graph.

    Allocation must be defined in at least one component within a phase after **ClusterPartitioner** and before **ClusterGather**, including either of these two components.

    Its value is inhereted by all other components in the same phase that are connected by edges and that lie between **ClusterPartitioner**, on one side, and before **ClusterGather**, on the other side.

    If components are not connected by edges, if they belong to different, separate branches of the same phase, **Allocation** must be defined in each branch of the phase.

    Its form is as follows:

    ```
    sandbox:<sanboxID>
    ```

    Here sandbox is a partitioned sanbox.

    It is also sufficient to define this **Allocation** by specifying the partitioned sandbox in the **File URL** of a **Reader** or **Writer**component in the same phase.

Such partitioned sanbox URL is: `sandbox://<sanboxID>`.

Only different phases can use different **Allocation**. If there are different values of **Allocation** within the same phase, graph will fail.

Ports Tab

In this tab, you can see the list of all ports of the component. You can expand any of the two items (**Input ports**, **Output ports**) and view the metadata assigned to each of these ports.

This tab is common for all components.

Important

Java-style Unicode expressions

Remember that (since version 3.0 of **CloverETL**) you can also use the Java-style Unicode expressions anyway in **CloverETL** (except in URL attributes).

You may use one or more Java-style Unicode expressions (for example, like this one): `\u0014`.

Such expressions consist of series of the `\uxxxx` codes of characters.

They may also serve as delimiter (like CTL expression shown above, without any quotes):

```
\u0014
```

Component Name

Each component has a label on it which can be changed for another one. As you may have many components in your graph and they may have some specified functions, you can give them names according to what they do. Otherwise you would have many different components with identical names in your graph.

You can rename any component in one of the following four ways:

- You can rename the component in the **Edit component** dialog by specifying the **Component name** attribute.
- You can rename the component in the **Properties** tab by specifying the **Component name** attribute.
- You can rename the component by highlighting and clicking it.

  If you highlight any component (by clicking the component itself or by clicking its item in the **Outline** pane), a hint appears showing the name of the component. After that, when you click the highlighted component, a rectangle appears below the component, showing the **Component name** on a blue background. You can change the name shown in this rectangle and then you only need to press **Enter**. The **Component name** has been changed and it can be seen on the component.

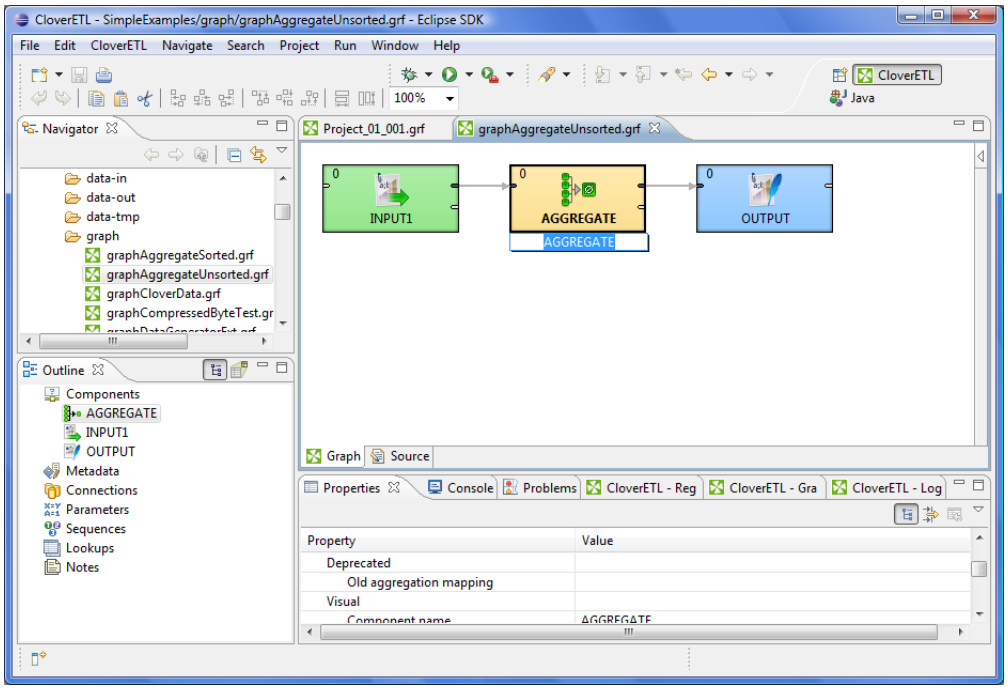


Figure 35.3. Simple Renaming Components

- You can right-click the component and select **Rename** from the context menu. After that, the same rectangle as mentioned above appears below the component. You can rename the component in the way described above.

Phases

Each graph can be divided into some amount of phases by setting the phase numbers on components. You can see this phase number in the upper left corner of every component.

The meaning of a phase is that each graph runs in parallel within the same phase number. That means that each component and each edge that have the same phase number run simultaneously. If the process stops within some phase, higher phases do not start. Only after all processes within one phase terminate successfully, will the next phase start.

That is why the phases must remain the same while the graph is running. They cannot descend.

So, when you increase some phase number on any of the graph components, all components with the same phase number (unlike those with higher phase numbers) lying further along the graph change their phase to this new value automatically.

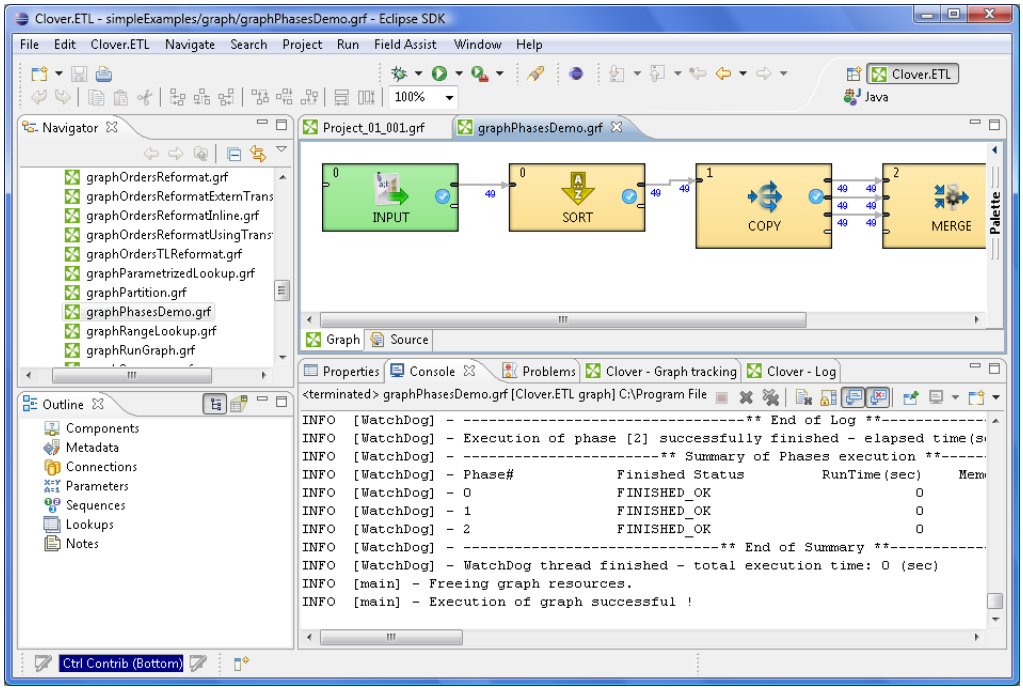


Figure 35.4. Running a Graph with Various Phases

You can select more components and set their phase number(s). Either you set the same phase number for all selected components or you can choose the step by which the phase number of each individual component should be incremented or decremented.

To do that, use the following **Phase setting** wizard:

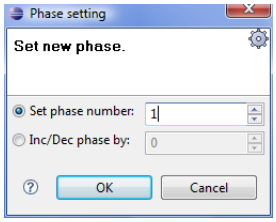


Figure 35.5. Setting the Phases for More Components

Enable/Disable Component

By default all components are enabled. Once configured, they can parse data. However, you can turn off any group of components of any graph. Each component can be disabled. When you disable some component, it becomes greyish and does not parse data when the process starts. Also, neither the components that lie further along the graph parse data. Only if there is another enabled component that enter the branch further along the graph, data can flow into the branch through that enabled component. But, if some component from which data flows to the disabled component or to which data flows from the disabled component cannot parse data without the disabled component, graph terminates with error. Data that are parsed by some component must be sent to other components and if it is not possible, parsing is impossible as well. Disabling can be done in the context menu or **Properties** tab. You can see the following example of when parsing is possible even with some component disabled:

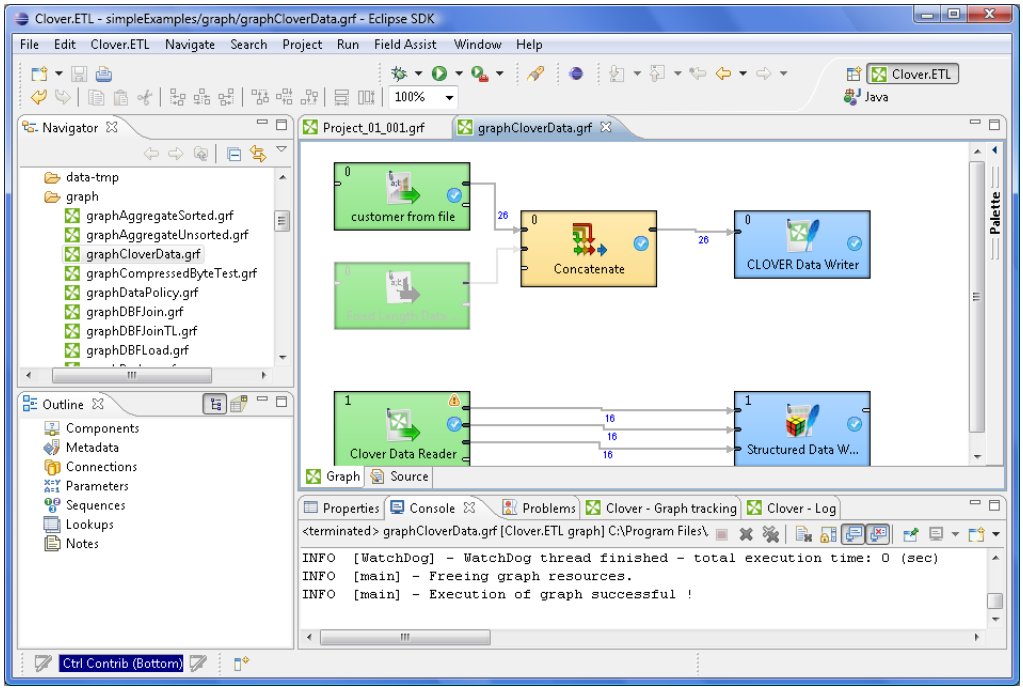


Figure 35.6. Running a Graph with Disabled Component

You can see that data records from the disabled component are not necessary for the **Concatenate** component and for this reason parsing is possible. Nevertheless, if you disable the **Concatenate** component, readers before this component would not have at their disposal any component to which they could send their data records and graph would terminate with error.

PassThrough Mode

As described in the previous section (Enable/Disable Component (p. 229)), if you want to process the graph with some component turned off (as if it did not exist in the graph), you can achieve it by setting the component to the **passThrough** mode. Thus, data records will pass through the component from input to output ports and the component will not change them. This mode can also be selected from the context menu or the **Properties** tab.

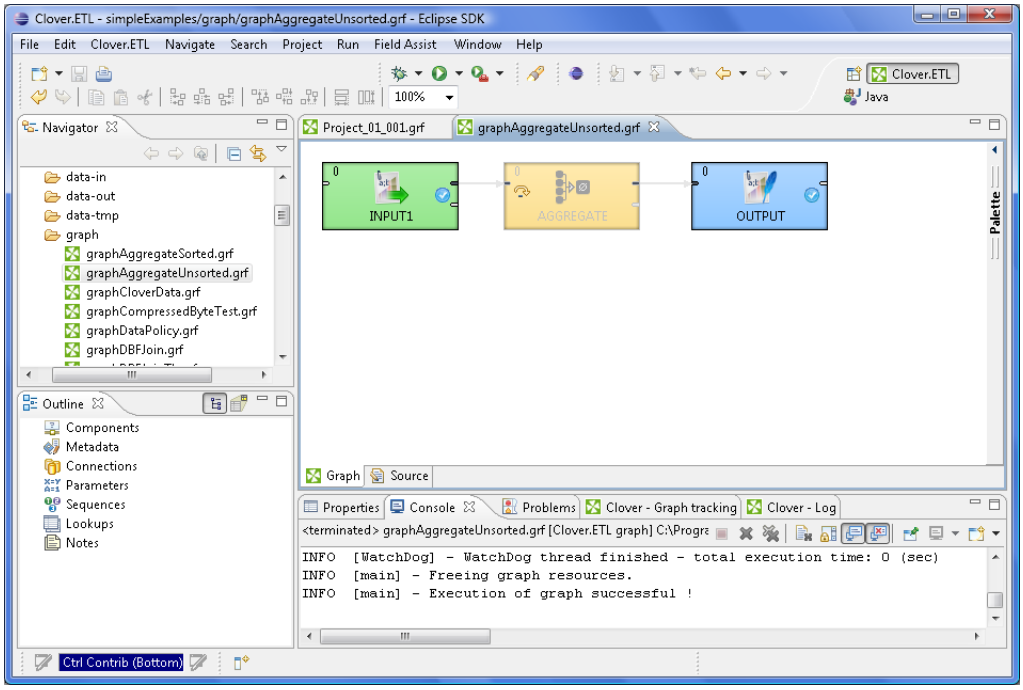


Figure 35.7. Running a Graph with Component in PassThrough Mode

Note

Remember that in some components (with more input and/or output ports) you must specify which of them should be used for input and/or output data, respectively. In them, you must set up the **Pass Through Input port** and/or **Pass Through Output port** as described in Properties Tab (p. 224).

Chapter 36. Common Properties of Most Components

Here we present a brief overview of properties that are common for various groups of components.

Links to corresponding sections follow:

- When you need to specify some file in a component, you need to use URL File Dialog (p. 59).
- In some of the components, records must be grouped according the values of a group key. In this key, neither the order of key fields nor the sort order are of importance. See Group Key (p. 231).
- In some of the components, records must be grouped and sorted according the values of a sort key. In this key, both the order of key fields and the sort order are of importance. See Sort Key (p. 232).
- In many components from different groups of components, a transformation can be or must be defined. See Defining Transformations (p. 234).

Group Key

Sometimes you need to select fields that will create a grouping key. This can be done in the **Edit key** dialog. After opening the dialog, you need to select the fields that should create the group key.

Select the fields you want and drag and drop each of the selected key fields to the **Key parts** pane on the right. (You can also use the **Arrow** buttons.)

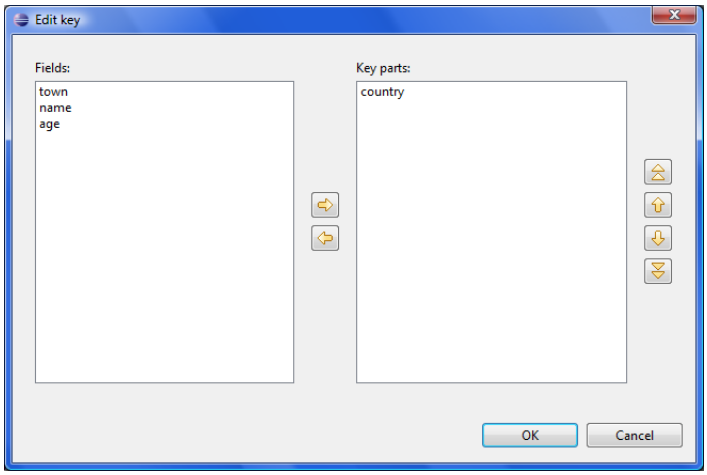


Figure 36.1. Defining Group Key

After selecting the fields, you can click the **OK** button and the selected field names will turn to a sequence of the same field names separated by semicolon. This sequence can be seen in the corresponding attribute row.

The resulting group key is a sequence of field names separated by semicolon. It looks like this: `FieldM;...FieldN`.

In this kind of key, no sort order is shown unlike in **Sort key**. By default, the order is ascending for all fields and priority of these fiels descends down from top in the dialog pane and to the right from the left in the attribute row. See Sort Key (p. 232) for more detailed information.

When a key is defined and used in a component, input records are gathered together into a group of the records with equal key values.

Group key is used in the following components:

- **Group key** in SortWithinGroups (p. 519)
- **Merge key** in Merge (p. 479)
- **Partition key** in Partition (p. 491), and ClusterPartitioner (p. 556)
- **Aggregate key** in Aggregate (p. 445)
- **Key** in Denormalizer (p. 456)
- **Group key** in Rollup (p. 505)
- **Matching key** in ApproximativeJoin (p. 524)
- Also **Partition key** that serves for distributing data records among different output ports (or Cluster nodes in case of **ClusterPartitioner**) is of this type. See Partitioning Output into Different Output Files (p. 270)

Sort Key

In some of the components you need to define a sort key. Like a group key, this sort key can also be defined by selecting key fields using the **Edit key** dialog. There you can also choose what sort order should be used for each of the selected fields.

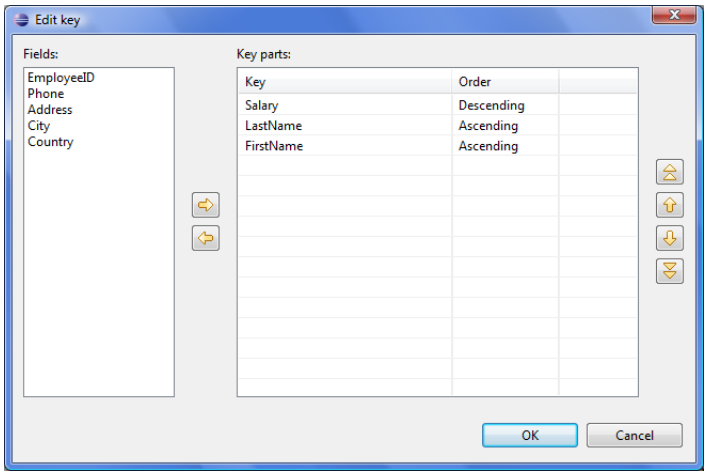


Figure 36.2. Defining Sort Key and Sort Order

In the **Edit key** dialog, select the fields you want and drag and drop each of the selected key fields to the **Key** column of the **Key parts** pane on the right. (You can also use the **Arrow** buttons.)

Unlike in the case of a group key, in any sort key the order in which the fields are selected is of importance.

In every sort key, the field at the top has the highest sorting priority. Then the sorting priority descends down from top. The field at the bottom has the lowest sorting priority.

When you click the **OK** button, the selected fields will turn to a sequence of the same field names and an `a` or a `d` letter in parentheses (with the meaning: `ascending` or `descending`, respectively) separated by semicolon.

It can look like this: `FieldM(a);...FieldN(d)`.

This sequence can be seen in the corresponding attribute row. (The highest sorting priority has the first field in the sequence. The priority descends towards the end of the sequence.)

As you can see, in this kind of key, the sort order is expressed separately for each key field (either **Ascending** or **Descending**). Default sort order is **Ascending**. The default sort order can also be changed in the **Order** column of the **Key parts** pane.

Important

ASCIIbetical vs. alphabetical order

Remember that `string` data fields are sorted in ASCII order (0,1,11,12,2,21,22 ... A,B,C,D ... a,b,c,d,e,f ...) while the other data type fields in the alphabetical order (0,1,2,11,12,21,22 ... A,a,B,b,C,c,D,d ...)

Example 36.1. Sorting

If your sort key is the following: `Salary(d);LastName(a);FirstName(a)`. The records will be sorted according to the `Salary` values in descending order, then the records will be sorted according to `LastName` within the same `Salary` value and they will be sorted according to `FirstName` within the same `LastName` and the same `Salary` (both in ascending order) value.

Thus, any person with `Salary` of `25000` will be processed after any other person with salary of `28000`. And, within the same `Salary`, any `Brown` will be processed before any `Smith`. And again, within the same salary, any `John Smith` will be processed before any `Peter Smith`. The highest priority is `Salary`, the lowest is `FirstName`.

Sort key is used in the following cases:

- **Sort key** in ExtSort (p. 472)
- **Sort key** in FastSort (p. 474)
- **Sort key** in SortWithinGroups (p. 519)
- **Dedup key** in Dedup (p. 454)
- **Sort key** in SequenceChecker (p. 579)

Defining Transformations

For basic information about transformations see Chapter 31, Transformations (p. 208).

Here we will explain how you should create transformations that change the data flowing through some components.

For brief table overview of transformations see Transformations Overview (p. 237).

Below we can learn the following:

1. What components allow transformations.

   Components Allowing Transformation (p. 234)

2. What language can be used to write transformations.

   Java or CTL (p. 235)

3. Whether definition can be internal or external.

   Internal or External Definition (p. 235)

4. What the return values of transformations are.

   Return Values of Transformations (p. 238)

5. What can be done when error occurs.

   Error Actions and Error Log (deprecated since 3.0) (p. 240)

6. The **Transform editor** and how to work with it.

   Transform Editor (p. 241)

7. What interfaces are common for many of the transformation-allowing components.

   Common Java Interfaces (p. 249)

Components Allowing Transformation

The transformations can be defined in the following components:

- **DataGenerator**, **Reformat**, and **Rollup**

  These components require a transformation.

  You can define the transformation in Java or Clover transformation language.

  In these components, different data records can be sent out through different output ports using return values of the transformation.

  In order to send different records to different output ports, you must both create some mapping of the record to the corresponding output port and return the corresponding integer value.

- **Partition**, or **ClusterPartitioner**

  In the **Partition**, or **ClusterPartitioner** component, transformation is optional. It is required only if neither the **Ranges** nor the **Partition key** attributes are defined.

You can define the transformation in Java or Clover transformation language.

In **Partition**, different data records can be sent out through different output ports using return values of the transformation.

In **ClusterPartitioner**, different data records can also be sent out to different Cluster nodes (through virtual output ports) using return values of the transformation.

In order to send different records to different output ports or Cluster nodes, you must return corresponding integer value. But no mapping need to be written in this component since all of the records are sent out automatically.

- **DataIntersection**, **Denormalizer**, **Normalizer**, **Pivot**, **ApproximativeJoin**, **ExtHashJoin**, **ExtMergeJoin**, **LookupJoin**, **DBJoin**, and **RelationalJoin**

  These components require a transformation.

  You can define the transformation in Java or Clover transformation language.

  In **Pivot**, transformation can be defined setting one of the Key or Group size attributes. Writing it in Java or CTL is still possible.

- **MultiLevelReader** and **JavaExecute**

  These components require a transformation.

  You can only write it in Java.

- **JMSReader** and **JMSWriter**

  In these components, transformation is optional.

  If any is defined, it must be written in Java.

Java or CTL

Transformations can be written in Java or Clover transformation language (CTL):

- Java can be used in all components.

  Transformations executed in Java are faster than those written in CTL. Transformation can always be written in Java.

- CTL cannot be used in **JMSReader**, **JMSWriter**, **JavaExecute**, and **MultiLevelReader**.

  Nevertheless, CTL is very simple scripting language that can be used in most of the transforming components. Even people who do not know Java are able to use CTL. CTL does not require any Java knowledge.

Internal or External Definition

Each transformation can be defined as internal or external:

- **Internal transformation:**

  An attribute like **Transform**, **Denormalize**, etc. must be defined.

  In such a case, the piece of code is written directly in the graph and can be seen in it.

- **External transformation:**

One of the following two kinds of attributes may be defined:

- **Transform URL**, **Denormalize URL**, etc., for both Java and CTL

  The code is written in an external file. Also charset of such external file can be specified (**Transform source charset**, **Denormalize source charset**, etc.).

  For transformations written in Java, folder with transformation source code need to be specified as source for Java compiler so that the transformation may be executed successfully.

- **Transform class**, **Denormalize class**, etc.

  It is a compiled Java class.

  The class must be in classpath so that the transformation may be executed successfully.

Here we provide a brief overview:

- **Transform, Denormalize, etc.**

  To define a transformation in the graph itself, you must use the **Transform editor** (or the **Edit value** dialog in case of **JMSReader**, **JMSWriter** and **JavaExecute** components). In them you can define a transformation located and visible in the graph itself. The languages which can be used for writing transformation have been mentioned above (Java or CTL).

  For more detailed information about the editor or the dialog see Transform Editor (p. 241) or Edit Value Dialog (p. 60).

- **Transform URL, Denormalize URL, etc.**

  You can also use a transformation defined in some source file outside the graph. To locate the transformation source file, use the URL File Dialog (p. 59). Each of the mentioned components can use this transformation definition. This file must contain the definition of the transformation written in either Java or CTL. In this case, transformation is located outside the graph.

  For more detailed information see URL File Dialog (p. 59).

- **Transform class, Denormalize class, etc.**

  In all transforming components, you can use some compiled transformation class. To do that, use the **Open Type** wizard. In this case, transformation is located outside the graph.

  See Open Type Dialog (p. 61) for more detailed information.

More details about how you should define the transformations can be found in the sections concerning corresponding components. Both transformation functions (required and optional) of CTL templates and Java interfaces are described there.

Here we present a brief table with an overview of transformation-allowing components:

Table 36.1. Transformations Overview

Component	Transformation required	Java	CTL	Each to all outputs[1)]	Different to different outputs[2)]	CTL template	Java interface
Readers							
DataGenerator (p. 298)	✔	✔	✔	✖	✔	(p. 301)	(p. 304)
JMSReader (p. 314)	✖	✔	✖	✔	✖	-	(p. 316)
MultiLevelReader (p. 322)	✔	✔	✖	✖	✔	-	(p. 324)
Writers							
JMSWriter (p. 386)	✖	✔	✖	-	-	-	(p. 388)
Transformers							
Partition (p. 491)	✖	✔	✔	✖	✔	(p. 493)	(p. 497)
DataIntersection (p. 449)	✔	✔	✔	-	-	(p. 451)	(p. 451)
Reformat (p. 502)	✔	✔	✔	✖	✔	(p. 503)	(p. 504)
Denormalizer (p. 456)	✔	✔	✔	-	-	(p. 458)	(p. 464)
Pivot (p. 498)	✔	✔	✔	-	-	(p. 501)	(p. 501)
Normalizer (p. 484)	✔	✔	✔	-	-	(p. 485)	(p. 490)
MetaPivot (p. 481)	✖	✖	✖	-	-	-	-
Rollup (p. 505)	✔	✔	✔	✖	✔	(p. 507)	(p. 515)
DataSampler (p. 452)	✔	✖	✖	-	-	-	-
Joiners							
ApproximativeJoin (p. 524)	✔	✔	✔	-	-	(p. 278)	(p. 281)
ExtHashJoin (p. 535)	✔	✔	✔	-	-	(p. 278)	(p. 281)
ExtMergeJoin (p. 541)	✔	✔	✔	-	-	(p. 278)	(p. 281)
LookupJoin (p. 546)	✔	✔	✔	-	-	(p. 278)	(p. 281)
DBJoin (p. 532)	✔	✔	✔	-	-	(p. 278)	(p. 281)
RelationalJoin (p. 549)	✔	✔	✔	-	-	(p. 278)	(p. 281)
Cluster Components							
ClusterPartitioner (p. 556)	✖	✔	✔	✖	✔	(p. 558)	(p. 558)
Others							
JavaExecute (p. 571)	✔	✔	✖	-	-	-	(p. 572)

Legend

1): If this is `yes`, each data record is always sent out through all connected output ports.

2): If this is `yes`, each data record can be sent out through the connected output port whose number is returned by the transformation. See Return Values of Transformations (p. 238) for more information.

Return Values of Transformations

In those components in which a transformations are defined, some return values can also be defined. They are integer numbers greater than, equal to or less than 0.

Note

Remember that **DBExecute** can also return integer values less than 0 in form of SQLExceptions.

- **Positive or zero return values**
  - **ALL = Integer.MAX_VALUE**

    In this case, the record is sent out through all output ports. Remember that this variable does not need to be declared before it is used. In CTL, `ALL` equals to `2147483647`, in other words, it is `Integer.MAX_VALUE`. Both `ALL` and `2147483647` can be used.
  - **OK = 0**

    In this case, the record is sent out through single output port or output port 0 (if component may have multiple output ports, e.g. **Reformat**, **Rollup**, etc. Remember that this variable does not need to be declared before it is used.
  - **Any other integer number greater than or equal to 0**

    In this case, the record is sent out through the output port whose number equals to this return value. These values can be called **Mapping codes**.
- **Negative return values**
  - **SKIP = - 1**

    This value serves to define that error has occurred but the incorrect record would be skipped and process would continue. Remember that this variable does not need to be declared before it is used. Both `SKIP` and `-1` can be used.

    This return value has the same meaning as setting of `CONTINUE` in the **Error actions** attribute (which is deprecated release 3.0 of **CloverETL**).
  - **STOP = - 2**

    This value serves to define that error has occurred but the processing should be stopped. Remember that this variable does not need to be declared before it is used. Both `STOP` and `-2` can be used.

    This return value has the same meaning as setting of `STOP` in the **Error actions** attribute (which is deprecated since release 3.0 of **CloverETL**).

    ### Important

    The same return value is `ERROR` in CTL1. `STOP` can be used in CTL2.
  - **Any integer number less than or equal to -1**

    These values should be defined by user as described below. Their meaning is fatal error. These values can be called **Error codes**. They can be used for defining Error actions (p. 240) in some components (This attribute along with **Error log** is deprecated since release 3.0 of **CloverETL**).

Important

1. **Values greater than or equal to 0**

   Remember that all return value that are greater than or equal to 0 allow to send the same data record to the specified output ports only in case of **DataGenerator**, **Partition**, **Reformat**, and **Rollup**. Do not forget to define the mapping for each such connected output port in **DataGenerator**, **Reformat**, and **Rollup**. In **Partition** (and **ClusterPartitioner**), mapping is performed automatically. In the other components, this has no meaning. They have either unique output port or their output ports are strictly defined for explicit outputs. On the other hand, **CloverDataReader**, **XLSDataReader**, and **DBFDataReader** always send each data record to all of the connected output ports.

2. **Values less than -1**

   Remember that you do not call corresponding optional `OnError()` function of CTL template using these return values. To call any optional `<required function>OnError()`, you may use, for example, the following function:

   ```
   raiseError(string Arg)
   ```

   It throws an exception which is able to call such `<required function>OnError()`, e.g. `transformOnError()`, etc. Any other exception thrown by any `<required function>()` function calls corresponding `<required function>OnError()`, if this is defined.

3. **Values less than or equal to -2**

   Remember that if any of the functions that return `integer` values, returns any value less than or equal to -2 (including `STOP`), the `getMessage()` function is called (if it is defined).

   Thus, to allow calling this function, you must add `return` statement(s) with values less than or equal to -2 to the functions that return `integer`. For example, if any of the functions like `transform()`, `append()`, or `count()`, etc. returns -2, `getMessage()` is called and the message is written to Console.

Important

You should also remember that if graph fails with an exception or with returning any negative value *less then* -1, no record will be written to the output file.

If you want that previously processed records are written to the output, you need to return SKIP (-1). This way, such records will be skipped, graph will not fail and at least some records will be written to the output.

Error Actions and Error Log (deprecated since 3.0)

Important

Since release 3.0 of **CloverETL**, these attributes are deprecated. They should be replaced with either `SKIP`, or `STOP` return values, if processing should either continue, or stop, respectively.

The **Error codes** can be used in some components to define the following two attributes:

- **Error actions**

  Any of these values means that a fatal error occurred and the user decides if the process should stop or continue. To define what should be done with the record, click the **Error actions** attribute row, click the button that appears and specify the actions in the following dialog. By clicking the **Plus sign** button, you add rows to this dialog pane. Select STOP or CONTINUE in the **Error action** column. Type an integer number to the **Error code** column. Leaving `MIN_INT` value in the left column means that the action will be applied to all other integer values that have not been specified explicitly in this dialog.

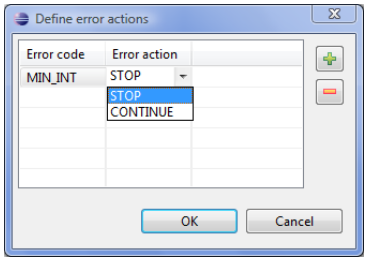


Figure 36.3. Define Error Actions Dialog

  The **Error actions** attribute has the form of a sequence of assignments (`errorCode=someAction`) separated by semicolon from each other.

  - The left side can be `MIN_INT` or any integer number less than 0 specified as some return value in the transformation definition.

    If `errorCode` is `MIN_INT`, this means that the specified action will be performed for all values that have not been specified in the sequence.

  - The right side of assignments can be `STOP` and/or `CONTINUE`.

    If `someAction` is `STOP`, when its corresponding `errorCode` is returned, `TransformExceptions` is thrown and graph stops.

    If `someAction` is `CONTINUE`, when its corresponding `errorCode` is returned, error message is written to **Console** or to the file specified by the **Error log** attribute and graph continues with the next record.

  #### Example 36.2. Example of the Error Actions Attribute

  `-1=CONTINUE;-3=CONTINUE;MIN_INT=STOP`. In this case, if the transformation returns `-1` or `-3`, process continues, if it returns any other negative value (including `-2`), process stops.

- **Error log**

  In this attribute, you can specify whether the error messages should be written on **Console** or in a specified file. The file should be defined using URL File Dialog (p. 59).

Transform Editor

Some of the components provide the **Transform editor** in which you can define the transformation.

When you open the **Transform editor**, you can see the following tabs: the **Transformations** and **Source** tabs.

The **Transformations** tab can look like this:

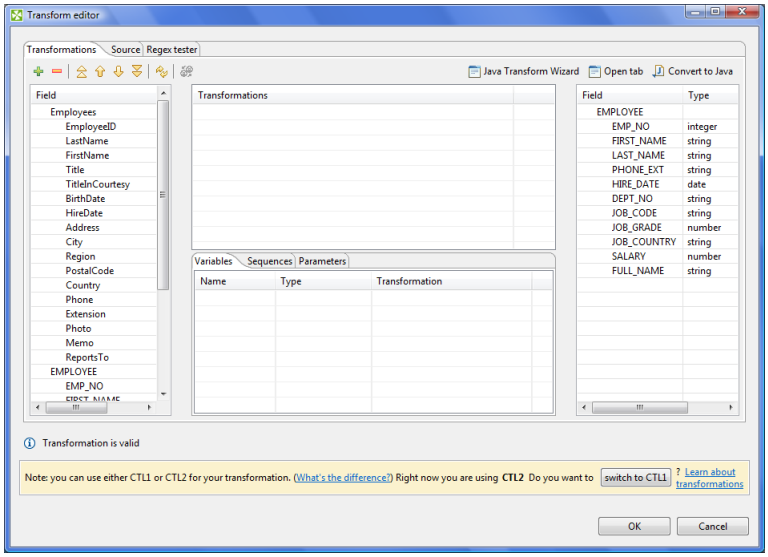


Figure 36.4. Transformations Tab of the Transform Editor

In the **Transformations** tab, you can define the transformation using a simple mapping of inputs to outputs. First, you must have both input and output metadata defined and assigned. Only after that, can you define the desired mapping.

After opening the **Transform editor**, you can see some panes and tabs in it. You can see the input fields of all input ports and their data types in the left pane and the output fields of all output ports and their data types in the right pane. You can see the following three tabs in the middle bottom: **Variables**, **Sequences**, **Parameters**.

If you want to define the mapping, you must select some of the input fields, push down the left mouse button on it, hold the button, drag to the **Transformations** pane in the middle and release the button. After that, the selected field name appears in the **Transformations** pane.

The following will be the resulting form of the expression: `$portnumber.fieldname`.

After that, you can do the same with some of the other input fields. If you want to concatenate the values of various fields (even from different input ports, in case of **Joiners** and the **DataIntersection** component), you can transfer all of the selected fields to the same row in the **Transformations** pane after which there will appear the expression that can look like this: `$portnumber1.fieldnameA+$portnumber2.fieldnameB`.

The port numbers can be the same or different. The `portnumber1` and `portnumber2` can be 0 or 1 or any other integer number. (In all components both input and output ports are numbered starting from 0.) This way you have defined some part of the transformation. You only need to assign these expressions to the output fields.

In order to assign these expressions to the output, you must select any item in the **Transformations** pane in the middle, push the left mouse button on it, hold the button, drag to the desired field in right pane and release the button. The output field in the right pane becomes bold.

Another point, you can see empty little circles on the left from each of these expressions (still in the **Transformations** pane). Whenever some mapping is made, the corresponding circle is filled in with blue. This way

you must map all of the expressions in the **Transformations** pane to the output fields until all of the expressions in the **Transformations** pane becomes blue. At that moment, the transformation has been defined.

You can also copy any input field to the output by right-clicking the input item in the left pane and selecting **Copy fields to...** and the name of the output metadata:

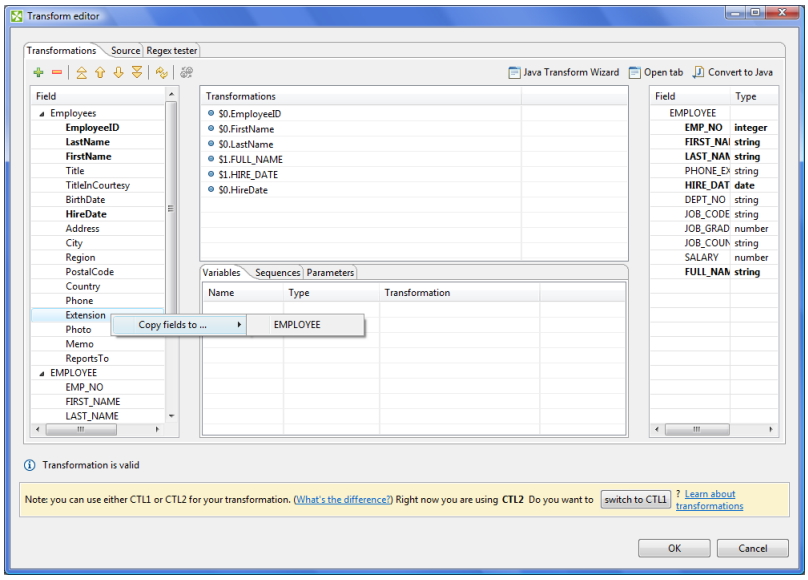


Figure 36.5. Copying the Input Field to the Output

Remember that if you have not defined the output metadata before defining the transformation, you can define them even here, by copying and renaming the output fields using right-click, however, it is much more simple to define new metadata prior to defining the transformation. If you defined the output metadata using this **Transform editor**, you would be informed that output records are not known and you would have to confirm the transformation with this error and (after that) specify the delimiters in metadata editor.

The resulting simple mapping can look like this:

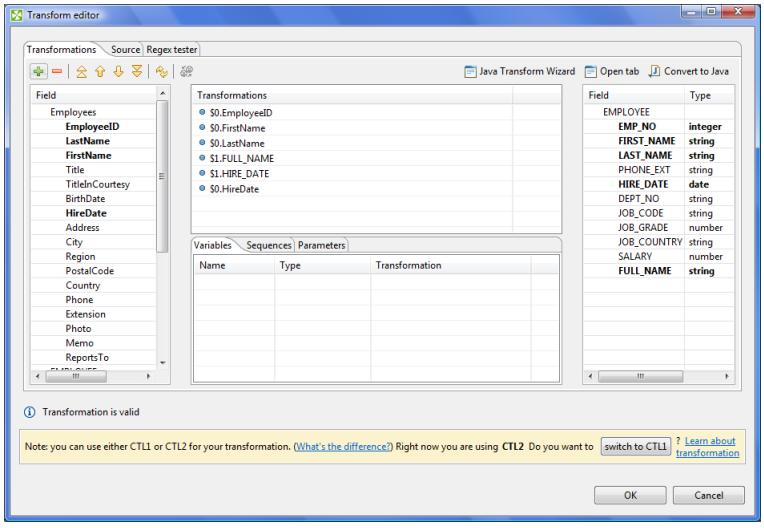


Figure 36.6. Transformation Definition in CTL (Transformations Tab)

If you select any item in the left, middle or right pane, corresponding items will be connected by lines. See example below:

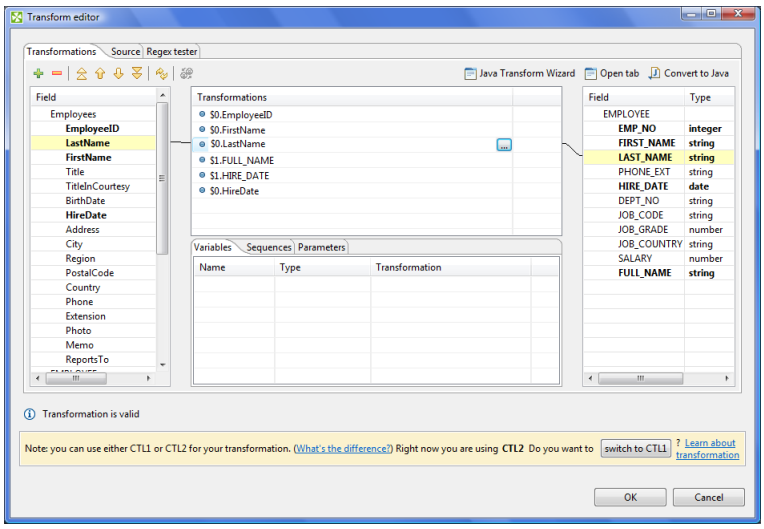


Figure 36.7. Mapping of Inputs to Outputs (Connecting Lines)

By clicking the button that appears after selecting the row of the **Transformations** pane, you can also open the editor for defining the transformation of each individual field. It contains a list of fields, functions and operators and it also provides the hints. See below:

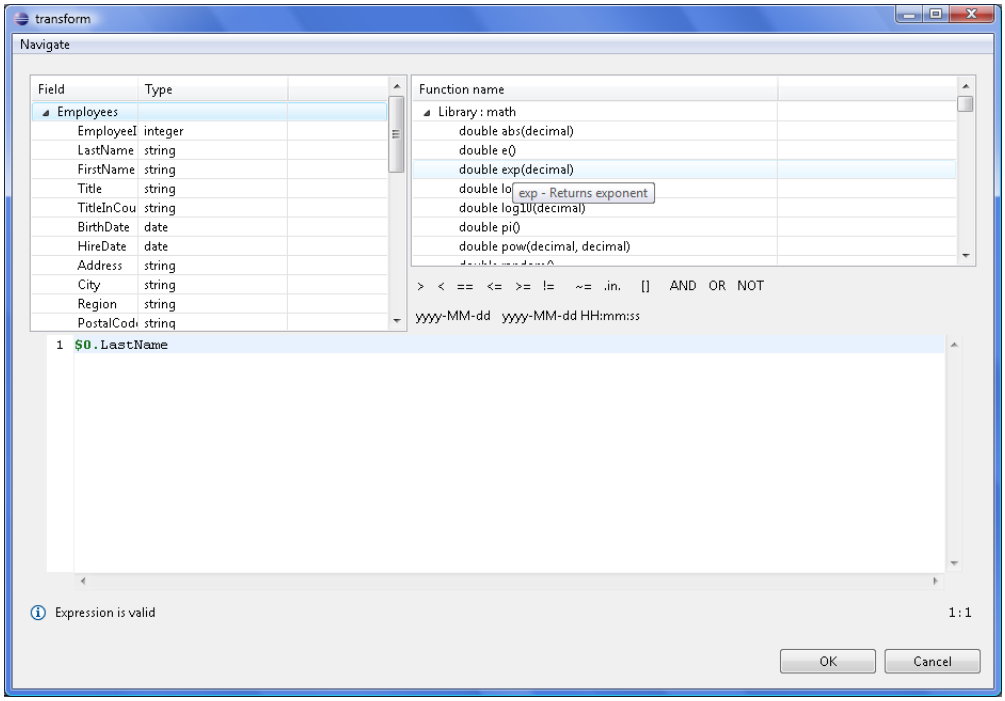


Figure 36.8. Editor with Fields and Functions

Some of your transformations may be complicated and it is difficult to define them in the **Transformations** tab. You can use the **Source** tab instead.

(**Source** tabs for individual components are displayed in corresponding sections concerning these components.)

Below you can see the **Source** tab with the transformation defined above. It is written in Clover transformation language.

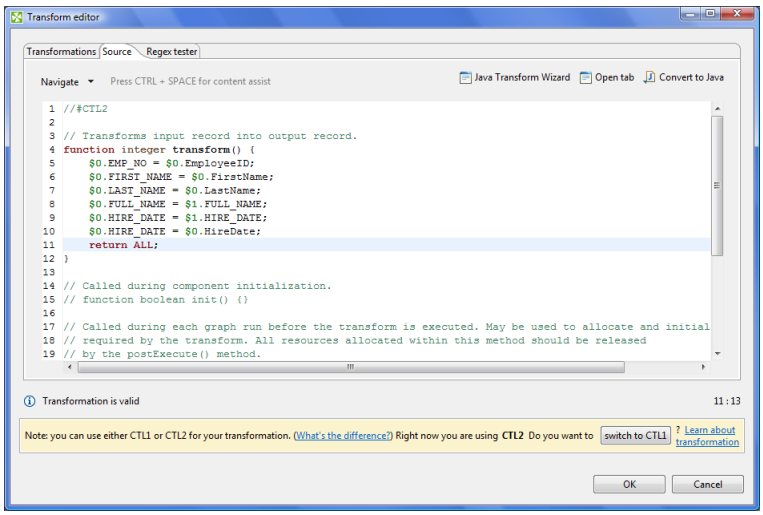


Figure 36.9. Transformation Definition in CTL (Source Tab)

In the upper right corner of either tab, there are three buttons: for launching a wizard to create a new Java transform class (**Java Transform Wizard** button), for creating a new tab in **Graph Editor** (**Open tab** button), and for converting the defined transformation to Java (**Convert to Java** button).

If you want to create a new Java transform class, press the **Java Transform Wizard** button. The following dialog will open:

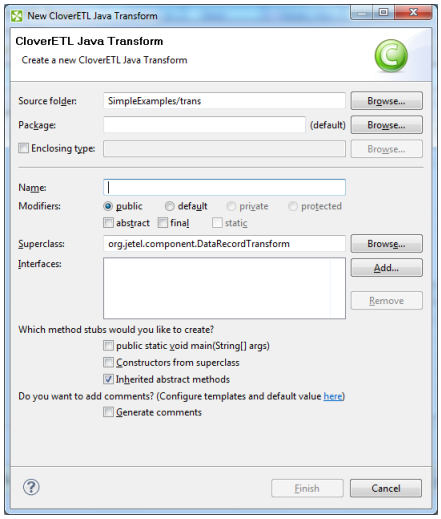


Figure 36.10. Java Transform Wizard Dialog

The **Source folder** field will be mapped to the project `${TRANS_DIR}`, for example `SimpleExamples/trans`. The value of the **Superclass** field depends on the target component. It will be set to a suitable abstract class implementing the required interface. For additional information, see Transformations Overview (p. 237). A new transform class can be created by entering the **Name** of the class and, optionally, the containing **Package** and pressing the **Finish** button. The newly created class will be located in the **Source folder**.

If you click the second button in the upper right corner of the **Transform editor**, the **Open tab** button, a new tab with the CTL source code of the transformation will be opened in the **Graph Editor**. It will be confirmed by the following message:

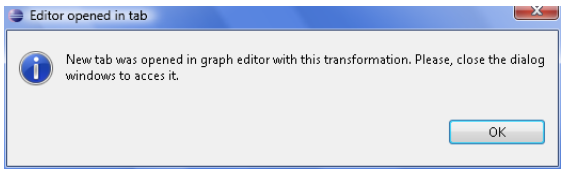


Figure 36.11. Confirmation Message

The tab can look like this:

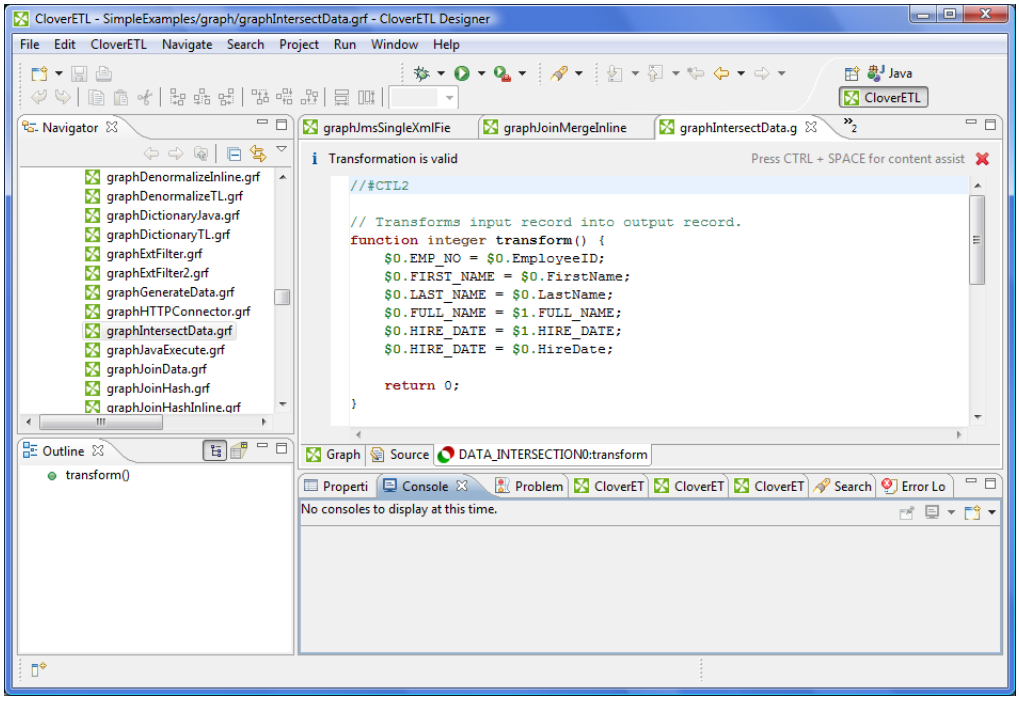


Figure 36.12. Transformation Definition in CTL (Transform Tab of the Graph Editor)

If you switch to this tab, you can view the declared variables and functions in the **Outline** pane. (The tab can be closed by clicking the red cross in the upper right corner of the tab.)

The **Outline** pane can look like this:

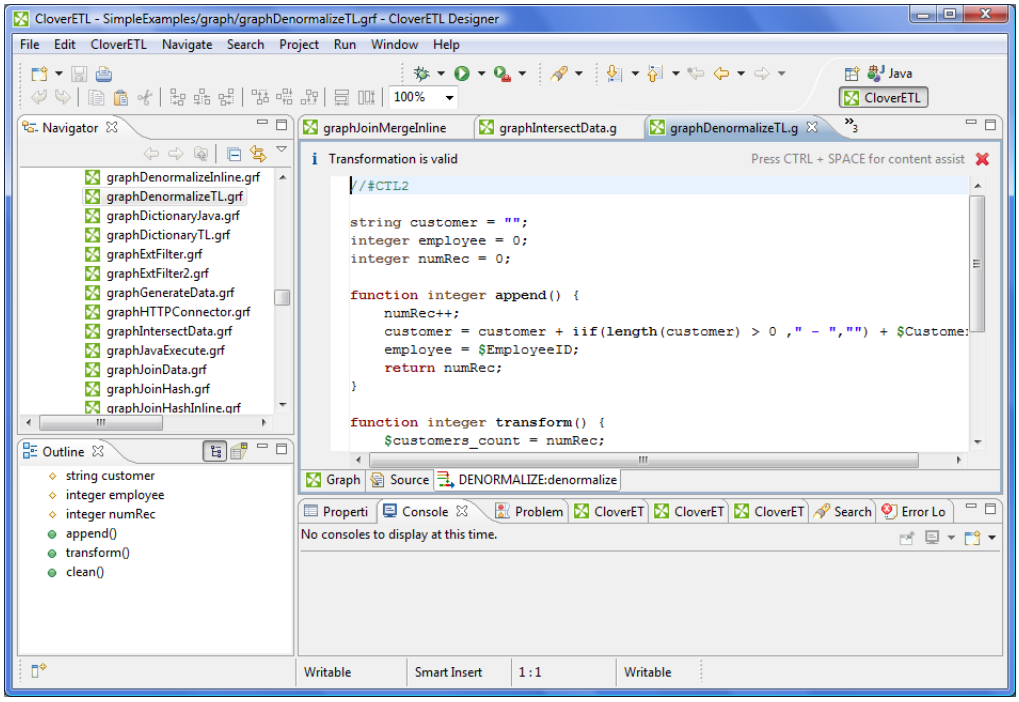


Figure 36.13. Outline Pane Displaying Variables and Functions

Note that you can also use some content assist by pressing **Ctrl**+**Space**.

If you press these two keys inside any of the expressions, the help advises what should be written to define the transformation.

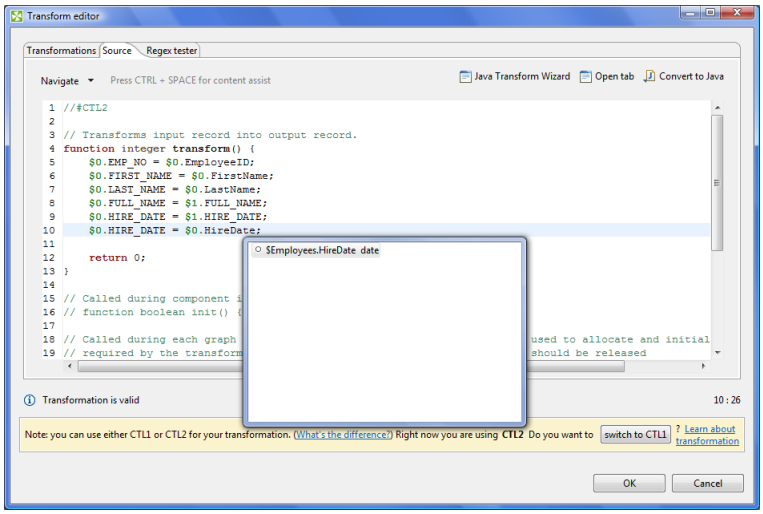


Figure 36.14. Content Assist (Record and Field Names)

If you press these two keys outside any of the expressions, the help gives a list of functions that can be used to define the transformation.

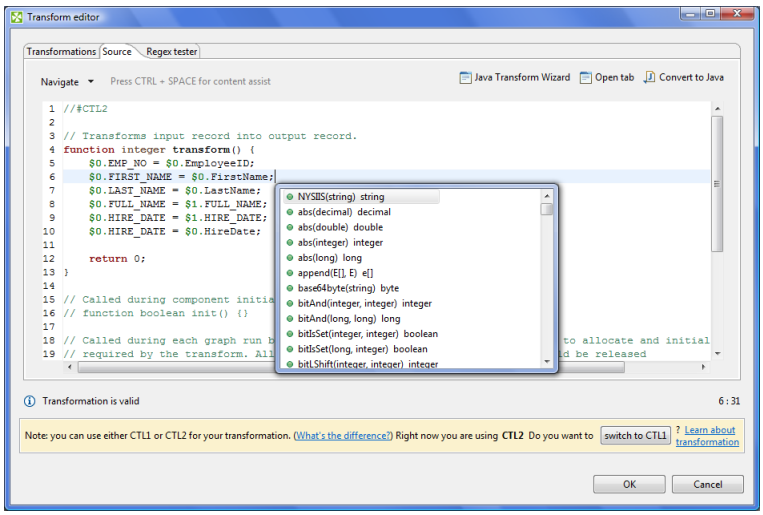


Figure 36.15. Content Assist (List of CTL Functions)

If you have some error in your definition, the line will be highlighted by red circle with a white cross in it and at the lower left corner there will be a more detailed information about it.

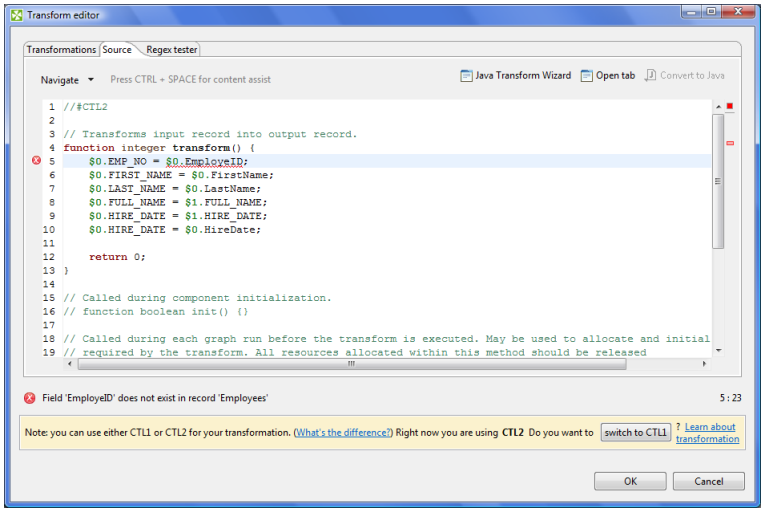


Figure 36.16. Error in Transformation

If you want to convert the transformation code into the Java language, click the **Convert to Java** button and select whether you want to use clover preprocessor macros or not.

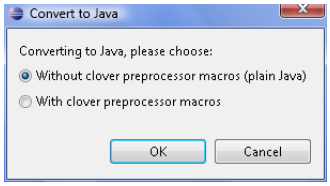


Figure 36.17. Converting Transformation to Java

After selecting and clicking **OK**, the transformation converts into the following form:

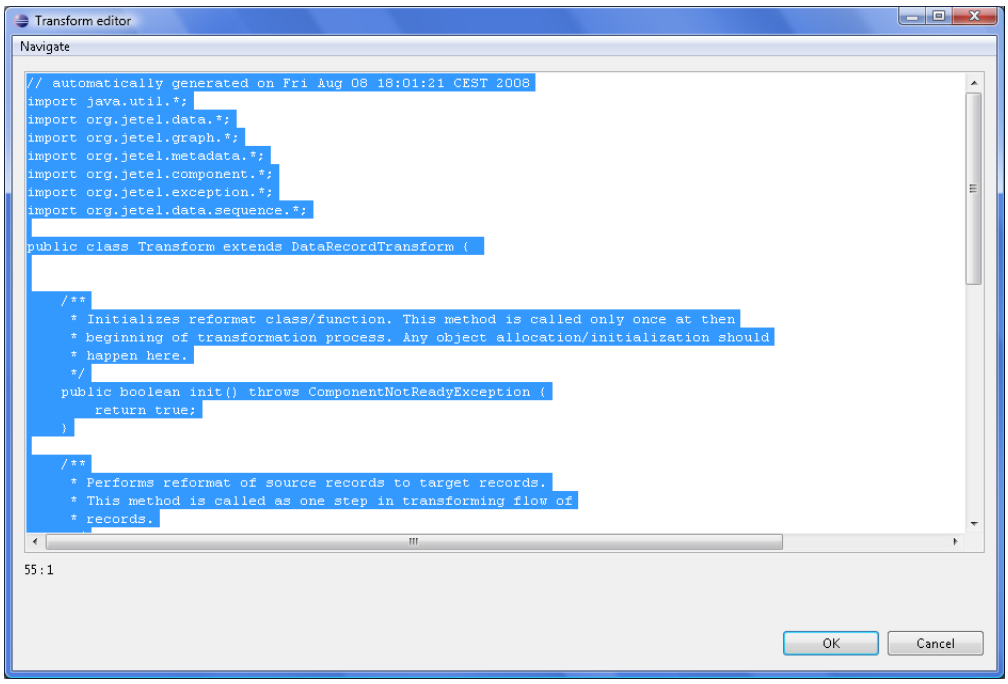


Figure 36.18. Transformation Definition in Java

Remember also that you can define your own error messages by defining the last function: `getMessage()`. It returns strings that are written to console. More details about transformations in each component can be found in the sections in which these components are desribed.

Important

Remember that the `getMessage()` function is only called from within functions that return `integer` data type.

To allow calling this function, you must add `return` statement(s) with values less than or equal to -2 to the functions that return `integer`. For example, if any of the functions like `transform()`, `append()`, or `count()`, etc. returns -2, `getMessage()` is called and the message is written to Console.

Common Java Interfaces

Following are the methods of the common `Transform` interface:

- `void setNode(Node node)`

  Associates a graph `Node` with this transform.

- `Node getNode()`

  return a graph `Node` associated with this transform, or `null` if no graph node is associated

- `TransformationGraph getGraph()`

  Returns a `TransformationGraph` associated with this transform, or `null` if no graph is associated.

- `void preExecute()`

  Called during each graph run before the transform is executed. May be used to allocate and initialize resources required by the transform. All resources allocated within this method should be released by the `postExecute()` method.

- `void postExecute(TransactionMethod transactionMethod)`

  Called during each graph run after the entire transform was executed. Should be used to free any resources allocated within the `preExecute()` method.

- `String getMessage()`

  Called to report any user-defined error message if an error occurred during the transform and the transform returned value less than or equal to -2. It is called when either `append()`, `count()`, `generate()`, `getOutputPort()`, `transform()`, or `updateTansform()` or any of their `OnError()` counterparts returns value less than or equal to -2.

- `void finished()` (deprecated)

  Called at the end of the transform after all input data records were processed.

- `void reset()` (deprecated)

  Resets the transform to the initial state (for another execution). This method may be called only if the transform was successfully initialized before.

Chapter 37. Common Properties of Readers

Readers are the initial components of graphs. They read data from data sources and send it to other graph components. This is the reason why each reader must have at least one output port through which the data flows out. **Readers** can read data from files or databases located on disk. They can also receive data through some connection using FTP, LDAP, or JMS. Some **Readers** can log the information about errors. Among the readers, there is also the **Data Generator** component that generates data according to some specified pattern. And, some **Readers** have an optional input port through which they can also receive data. They can also read data from dictionary.

Remember that you can see some part of input data when you right-click a reader and select the **View data** option. After that, you will be prompted with the same **View data** dialog as when debugging the edges. For more details see Viewing Debug Data (p. 93). This dialog allows you to view the read data (it can even be used before graph has been run).

Here we present a brief overview of links to these options:

- Some examples of the **File URL** attribute for reading from local and remote files, through proxy, from console, input port and dictionary:

  Supported File URL Formats for Readers (p. 251)

- Viewing Data on Readers (p. 255)
- Input Port Reading (p. 257)
- Incremental Reading (p. 258)
- Selecting Input Records (p. 258)
- Data Policy (p. 260)
- XML Features (p. 260)
- As has been shown in Defining Transformations (p. 234), some **Readers** allow that a transformation can be or must be defined in them. We also provide some examples of attributes for reading from local and remote files, through proxy, from console, input port and dictionary. For information about transformation templates for transformations written in CTL see:

  CTL Templates for Readers (p. 261)

- As has been shown in Defining Transformations (p. 234), some **Readers** allow that a transformation can be or must be defined in them. We also provide some examples of attribute for reading from local and remote files, through proxy, from console, input port and dictionary. For information about transformation interfaces that must be implemented in transformations written in Java see:

  Java Interfaces for Readers (p. 261)

Here we present an overview of all **Readers**:

Table 37.1. Readers Comparison

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
DataGenerator (p. 298)	none	0	1-n	✖	✔	✔	yes[3)]	✔	✔
UniversalDataReader (p. 333)	flat file	0-1	1-2	✖	✖	✖	✖	✖	✖
ParallelReader (p. 326)	flat file	0	1	✖	✖	✖	✖	✖	✖
CloverDataReader (p. 289)	clover binary file	0	1-n	✔	✖	✖	✖	✖	✖
XLSDataReader (p. 338)	XLS(X) file	0-1	1-n	✔	✖	✖	✖	✖	✖
DBFDataReader (p. 305)	dBase file	0-1	1-n	✔	✖	✖	✖	✖	✖
DBInputTable (p. 307)	database	0	1-n	✔	✖	✖	✖	✖	✖
XMLExtract (p. 342)	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖
XMLXPathReader (p. 354)	XML file	0-1	1-n	✖	✔	✖	✖	✖	✖
JMSReader (p. 314)	jms messages	0	1	-	-	✔	✖	✔	✖
EmailReader (p. 311)	email messages	0	1	-	-	✔	✖	✔	✖
LDAPReader (p. 317)	LDAP directory tree	0	1-n	✖	✖	✖	✖	✖	✖
MultiLevelReader (p. 322)	flat file	1	1-n	✖	✔	✔	✔	✔	✖
ComplexDataReader (p. 291)	flat file	1	1-n	✖	✔	✔	✔	✔	✔
QuickBaseRecordReader (p. 329)	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖
QuickBaseQueryReader (p. 331)	QuickBase	0	1	✖	✖	✖	✖	✖	✖
LotusReader (p. 320)	Lotus Notes	0	1	✖	✖	✖	✖	✖	✖

Legend

1) Component sends each data record to all of the connected output ports.

2) Component sends different data records to different output ports using return values of the transformation (**DataGenerator** and **MultiLevelReader**). See Return Values of Transformations (p. 238) for more information. **XMLExtract** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Supported File URL Formats for Readers

The **File URL** attribute may be defined using the URL File Dialog (p. 59).

Important

To ensure graph portability, forward slashes must be used when defining the path in URLs (even on Microsoft Windows).

Here we present some examples of possible URL for **Readers**:

Reading of Local Files

- `/path/filename.txt`

  Reads specified file.

- `/path1/filename1.txt;/path2/filename2.txt`

  Reads two specified files.

- `/path/filename?.txt`

  Reads all files satisfying the mask.

- `/path/*`

  Reads all files in specified directory.

- `zip:(/path/file.zip)`

  Reads the first file compressed in the `file.zip` file.

- `zip:(/path/file.zip)#innerfolder/filename.txt`

  Reads specified file compressed in the `file.zip` file.

- `gzip:(/path/file.gz)`

  Reads the first file compressed in the `file.gz` file.

- `tar:(/path/file.tar)#innerfolder/filename.txt`

  Reads specified file archived in the `file.tar` file.

- `zip:(/path/file??.zip)#innerfolder?/filename.*`

  Reads all files from the compressed zipped file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `tar:(/path/file????.tar)#innerfolder??/filename*.txt`

  Reads all files from the archive file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `gzip:(/path/file*.gz)`

  Reads all files each of them has been gzipped into the file that satisfy the specified mask. Wild cards may be used in the compressed file names.

- `tar:(gzip:/path/file.tar.gz)#innerfolder/filename.txt`

  Reads specified file compressed in the `file.tar.gz` file. Note that although **CloverETL** can read data from `.tar` file, writing to `.tar` files is not supported.

- `tar:(gzip:/path/file??.tar.gz)#innerfolder?/filename*.txt`

  Reads all files from the gzipped `tar` archive file(s) that satisfy the specified mask. Wild cards (`?` and `*`) may be used in the compressed file names, inner folder and inner file names.

- `zip:(zip:(/path/name?.zip)#innerfolder/file.zip)#innermostfolder?/filename*.txt`

Reads all files satisfying the file mask from all paths satisfying the path mask from all compressed files satisfying the specified zip mask. Wild cards (`?` and `*`) may be used in the outer compressed files, innermost folder and innermost file names. They cannot be used in the inner folder and inner zip file names.

Reading of Remote Files

- `ftp://username:password@server/path/filename.txt`

  Reads specified `filename.txt` file on remote server connected via ftp protocol using username and password.

- `sftp://username:password@server/path/filename.txt`

  Reads specified `filename.txt` file on remote server connected via ftp protocol using username and password.

- `http://server/path/filename.txt`

  Reads specified `filename.txt` file on remote server connected via http protocol.

- `https://server/path/filename.txt`

  Reads specified `filename.txt` file on remote server connected via https protocol.

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Reads specified `filename.txt` file compressed in the `file.zip` file on remote server connected via ftp protocol using username and password.

- `zip:(http://server/path/file.zip)#innerfolder/filename.txt`

  Reads specified `filename.txt` file compressed in the `file.zip` file on remote server connected via http protocol.

- `tar:(ftp://username:password@server/path/file.tar)#innerfolder/filename.txt`

  Reads specified `filename.txt` file archived in the `file.tar` file on remote server connected via ftp protocol using username and password.

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  Reads specified `filename.txt` file compressed in the `file.zip` file that is also compressed in the `name.zip` file on remote server connected via ftp protocol using username and password.

- `gzip:(http://server/path/file.gz)`

  Reads the first file compressed in the `file.gz` file on remote server connected via http protocol.

- `http://server/filename*.dat`

  Reads all files from WebDAV server which satisfy specified mask (only * is supported.)

- `http://access_key_id:secret_access_key@bucketname.s3.amazonaws.com/filename*.out`

  Reads all files which satisfy specified mask (only * is supported) from Amazon S3 web storage service from given bucket using access key ID and secret access key.

Reading from Input Port

- `port:$0.FieldName:discrete`

  Data from each record field selected for input port reading are read as a single input file.

- `port:$0.FieldName:source`

  URL addresses, i.e., values of field selected for input port reading, are loaded in and parsed.

- `port:$0.FieldName:stream`

  Input port field values are concatenated and processed as an input file(s); `null` values are replaced by the `eof`.

Reading from Console

- `-`

  Reads data from `stdin` after start of the graph. When you want to stop reading, press **Ctrl**+**Z**.

Using Proxy in Readers

- `http:(direct:)//seznam.cz`

  Without proxy.

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  Proxy setting for http protocol.

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  Proxy setting for ftp protocol.

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  Proxy setting for sftp protocol.

Reading from Dictionary

- `dict:keyName:discrete`[1)]

  Reads data from dictionary.

- `dict:keyName:source`[1)]

  Reads data from dictionary in the same way like the `discrete` processing type, but expects that the dictionary values are input file URLs. The data from this input passes to the **Reader**.

Legend:

1): **Reader** finds out the type of source value from the dictionary and creates readable channel for the parser. **Reader** supports following type of sources: `InputStream`, `byte[]`, `ReadableByteChannel`, `CharSequence`, `CharSequence[]`, `List<CharSequence>`, `List<byte[]>`, `ByteArrayOutputStream`.

Sandbox Resource as Data Source

A sandbox resource, whether it is a shared, local or partitioned sandbox, is specified in the graph under the fileURL attributes as a so called sandbox URL like this:

```
sandbox://data/path/to/file/file.dat
```

where "data" is code for sandbox and "path/to/file/file.dat" is the path to the resource from the sandbox root. URL is evaluated by CloverETL Server during graph execution and a component (reader or writer) obtains the opened stream from the server. This may be a stream to a local file or to some other remote resource. Thus, a graph does not have to run on the node which has local access to the resource. There may be more sandbox resources used in the graph and each of them may be on a different node. In such cases, CloverETL Server would choose the node with the most local resources to minimalize remote streams.

The sandbox URL has a specific use for parallel data processing. When the sandbox URL with the resource in a *partitioned sandbox* is used, that part of the graph/phase runs in parallel, according to the node allocation specified by the list of partitioned sandbox locations. Thus, each worker has its own local sandbox resource. CloverETL Server evaluates the sandbox URL on each worker and provides an open stream to a local resource to the component.

Viewing Data on Readers

You can view data on **Readers** using the context menu. To do that, right-click the desired component and select **View data** from the context menu.

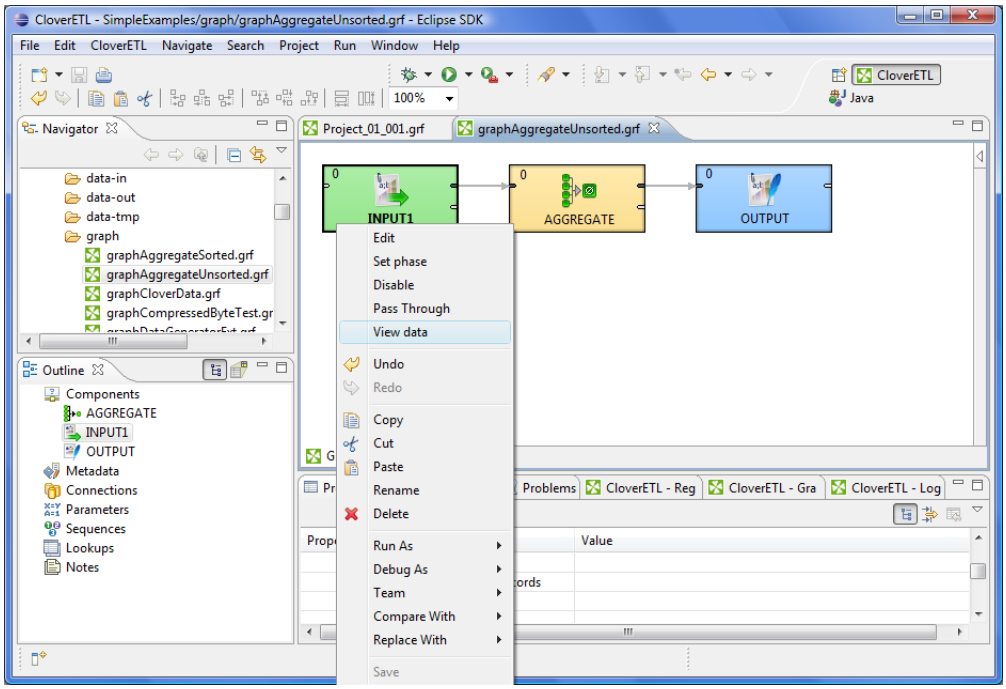


Figure 37.1. Viewing Data in Components

After that, you can choose whether you want to see data as a plain text or grid (a preview of parsed data). If you select the **Plain text** option, you can select **Charset**, but you cannot select any filter expression. Windows with results are modal, so that user can view data from components at the same time. To differ between results window title provides info about viewing edge in format GRAPH.name:COMPONENT.name.

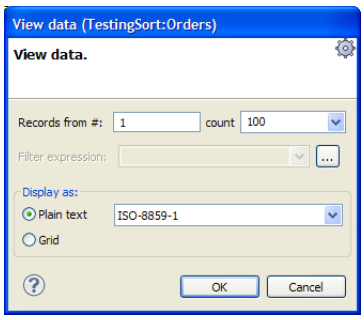


Figure 37.2. Viewing Data as Plain Text

On the other hand, if you select the **Grid** option, you can select **Filter expression**, but no **Charset**.

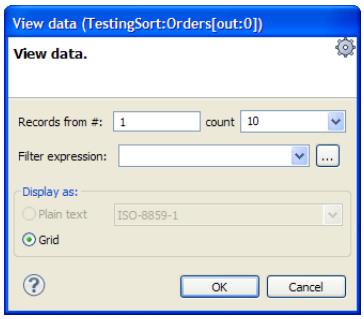


Figure 37.3. Viewing Data as Grid

The result can be seen as follows in **Plain text** mode:

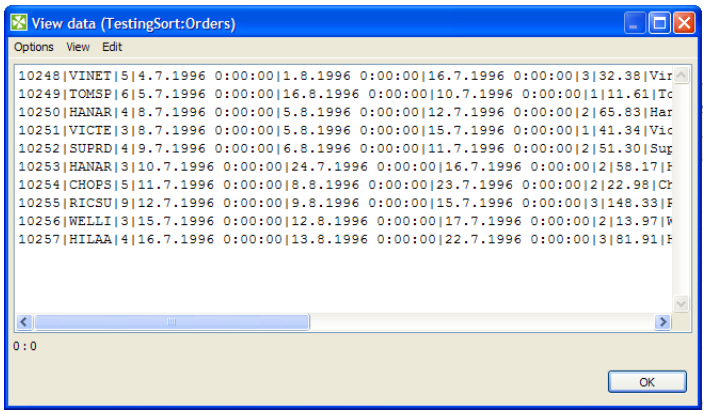


Figure 37.4. Plain Text Data Viewing

Or in the **Grid** mode, it can be like the following:

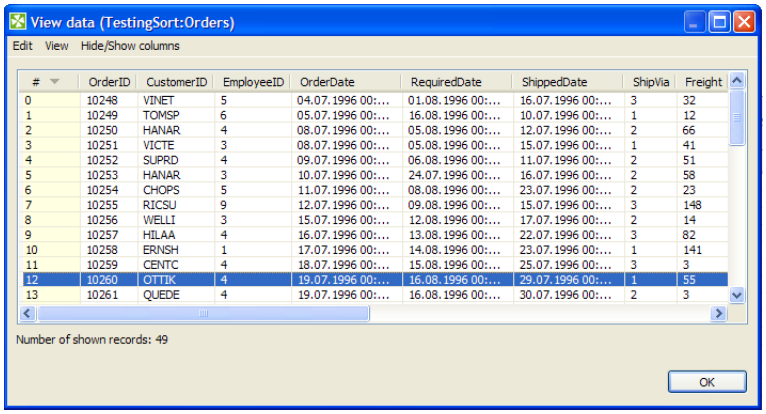


Figure 37.5. Grid Data Viewing

The same can be done in some of the **Writers**. See Viewing Data on Writers (p. 266). However, only after the output file has been created.

Input Port Reading

Some **Readers** allow to read data from the optional input port.

Input port reading is supported by the following **Reader**s:

- **UniversalDataReader**
- **XLSDataReader**
- **DBFDataReader**
- **XMLExtract**
- **XMLXPathReader**
- **MultiLevelReader (Commercial Component)**

Important

Remember that port reading can also be used by **DBExecute** for receiving SQL commands. **Query URL** will be as follows: port:$0.fieldName:discrete. Also SQL command can be read from a file. Its name, including path, is then passed to DBExecute from input port and the **Query URL** attribute should be the following: port:$0.fieldName:source.

If you connect the optional input port of any of these **Reader**s to an edge, you must also connect the other side of this edge to some data source. To define the protocol for field mapping, a field from where you want to read data must be set in the **File URL** attribute of the **Reader**. The type of the `FieldName` input field can only be `string`, `byte`, or `cbyte` as defined in input edge metadata.

The protocol has the syntax `port:$0.FieldName[:processingType]`.

Here `processingType` is optional and defines if the data is processed as plain data or url addresses. It can be `source`, `discrete`, or `stream`. If not set explicitly, `discrete` is applied by default.

To define the attributes of input port reading, URL File Dialog (p. 59) can be used.

When graph runs, data is read from the original data source (according to the metadata of the edge connected to the optional input port of the **Readers**) and received by the **Reader** through its optional input port. Each record is read independently of the other records. The specified field of each one is processed by the **Reader** according to the output metadata.

- `discrete`

  Each data record field from input port represents one particular data source.

- `source`

  Each data record field from input port represents an URL to be load in and parsed.

- `stream`

  All data fields from input port are concatenated and processed as one input file. If the `null` value of this field is met, it is replaced by the `eof`. Following data record fields are parsed as another input file in the same way, i.e., until the `null` value is met. See Output Port Writing (p. 268) for more information about writing with `stream` processing type.

Incremental Reading

Some **Readers** allow to use so called incremental reading. If the graph reads the same input file or a collection of files several times, it may be interested only in those records or files, that have been added since the last graph run.

In the following four **Readers**, you can set the **Incremental file** and **Incremental key** attributes. The **Incremental key** is a string that holds the information about read records/files. This key is stored in the **Incremental file**. This way, the component reads only those records or files that have not been marked in the **Incremental file**.

The **Readers** allowing incremental reading are as follows:

- **UniversalDataReader**
- **XLSDataReader**
- **DBFDataReader**

The component which reads data from databases performs this incremental reading in a different way.

- **DBInputTable**

Unlike the other incremental readers, in this database component, more database columns can be evaluated and used as key fields. **Incremental key** is a sequence of the following individual expression separated by semicolon: `keyname=FUNCTIONNAME(db_field)[!InitialValue]`. For example, you can have the following **Incremental key**: `key01=MAX(EmployeeID);key02=FIRST(CustomerID)!20`. The functions that can be selected are the following four: `FIRST`, `LAST`, `MIN`, `MAX`. At the same time, when you define an **Incremental key**, you also need to add these key parts to the **Query**. In the query, a part of the `"where"` sentence will appear, for example, something like this: `where db_field1 > #key01 and db_field2 < #key02` . This way, you can limit which records will be read next time. It depends on the values of their `db_field1` and `db_field2` fields. Only the records that satisfy the condition specified by the query will be read. These key fields values are stored in the **Incremental file**. To define **Incremental key**, click this attribute row and, by clicking the **Plus** or **Minus** buttons in the **Define incremental key** dialog, add or remove key names, and select db field names and function names. Each one of the last two is to be selected from combo list of possible values.

Note

Since the version 2.8.1 of **CloverETL Designer**, you can also define `Initial value` for each key. This way, non existing **Incremental file** can be created with the specified values.

Selecting Input Records

When you set up **Readers**, you may want to limit the records that should be read.

Some **Readers** allow to read more files at the same time. In these **Readers**, you can define the records that should be read for each input file separately and for all of the input files in total.

In these **Readers**, you can define the **Number of skipped records** and/or **Max number of records** attributes. The former attribute specifies how many records should be skipped, the latter defines how many records should be read. Remember that these records are skipped and/or read continuously throughout all input files. These records are skipped and/or read independently on the values of the two similar attributes mentioned below.

In these components you can also specify how many records should be skipped and/or read from each input file. To do this, set up the following two attributes: **Number of skipped records per source** and/or **Max number of records per source**.

Thus, total number of records that are skipped equals to **Number of skipped records per source** multiplicated by the number of source files plus **Number of skipped records**.

And total number of records that are read equals to **Max number of records per source** multiplicated by the number of source files plus **Max number of records**.

The **Readers** that allow limiting the records for both individual input file and all input files in total are the following:

- **UniversalDataReader**
- **XLSDataReader**
- **DBFDataReader**
- **MultiLevelReader (Commercial Component)**

Unlike the components mentioned above, **CloverDataReader** only allows you to limit the total number of records from all input files:

- **CloverDataReader** only allows you to limit the total number of records by using the **Number of skipped records** and/or **Max number of records** attributes as shown in previous components.

The following two **Readers** allow you to limit the total number of records by using the **Number of skipped mappings** and/or **Max number of mappings** attributes. What is called **mapping** here, is a subtree which should be mapped and sent out through the output ports.

- **XMLExtract**. In addition to the mentioned above, this component also allows to use the `skipRows` and/or the `numRecords` attributes of individual XML elements.
- **XMLXPathReader**. In addition to the mentioned above, this component allows to use XPath language to limit the number of mapped XML structures.

The following **Readers** allow limiting the numbers in a different way:

- **JMSReader** allows you to limit the number of messages that are received and processed by using the **Max msg count** attribute and/or the `false` return value of `endOfInput()` method of the component interface.
- **QuickBaseRecordReader (Commercial Component)**. This component uses the **Records list** attribute to specify the records that should be read.
- **QuickBaseQueryReader (Commercial Component)**. This component can use the **Query** or the **Options** attributes to limit the number of records.
- **DBInputTable**. Also this component can use the **SQL query** or the **Query URL** attribute to limit the number of records.

The following **Readers** do not allow limiting the number of records that should be read (they read them all):

- **LDAPReader**
- **ParallelReader (Commercial Component)**

Data Policy

Data policy can be set in some **Readers**. Here we provide their list:

- **UniversalDataReader**
- **ParallelReader (Commercial Component)**
- **XLSDataReader**
- **DBFDataReader**
- **DBInputTable**
- **XMLXPathReader**
- **MultiLevelReader (Commercial Component)**

When you want to configure these components, you must first decide what should be done when incorrect or incomplete records are parsed. This can be specified with the help of this **Data Policy** attribute. You have three options according to what data policy you want to select:

- **Strict**. This data policy is set by default. It means that data parsing stops if a record field with an incorrect value or format is read. Next processing is aborted.
- **Controlled**. This data policy means that every error is logged, but incorrect records are skipped and data parsing continues. Generally, incorrect records with error information are logged into `stdout`. Only **UniversalDataReader** enables to sent them out through the optional second port.

Important

If you set the **Data policy** attribute to `controlled` in **UniversalDataReader**, you need to select the components that should process the information or maybe you only want to write it. You must select an edge and connect the error port of the **UniversalDataReader** (in which the data policy attribute is set to controlled) with the input port of the selected writer if you only want to write it or with the input port other processing component. And you must assign metadata to this edge. The metadata must be created by hand. They consist of 4 fields: `number of incorrect record`, `number of incorrect field`, `incorrect record`, `error message`. The first two fields are of `integer` data type, the other two are `strings`. See Creating Metadata by User (p. 134) for detailed information about how metadata should be created by user.

- **Lenient**. This data policy means that incorrect records are only skipped and data parsing continues.

XML Features

In XMLExtract (p. 342)and XMLXPathReader (p. 354)you can validate your input XML files by specifying the **Xml features** attribute. It is expressed as a sequence of individual expressions of one of the following form: `nameM:=true` or `nameN:=false`, where each `nameM` is an XML feature that should be validated. These expressions are separated from each other by semicolon.

The options for validation are the following:

- **Custom parser setting**
- **Default parser setting**
- **No validations**
- **All validations**

You can define this attribute using the following dialog:

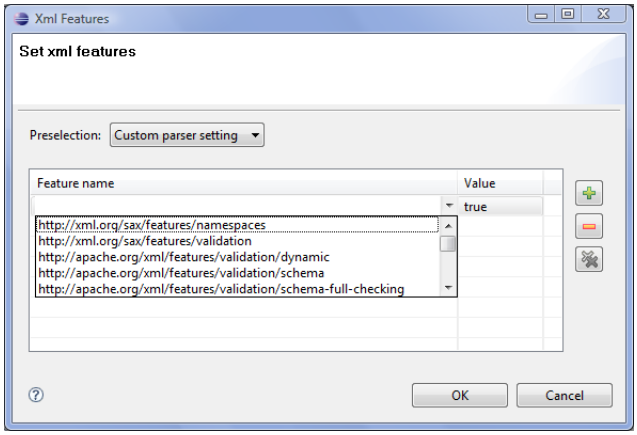


Figure 37.6. XML Features Dialog

In this dialog, you can add features with the help of **Plus** button, select their `true` or `false` values, etc.

CTL Templates for Readers

- DataGenerator (p. 298) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for DataGenerator (p. 301) for more information about the transformation template.

  Remember that this component allows to send each record through the connected output port whose number equals the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

Java Interfaces for Readers

- DataGenerator (p. 298) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for DataGenerator (p. 304) for more information about the interface.

  Remember that this component allows sending of each record through the connected output port whose number equals the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

- JMSReader (p. 314) allows optionally a transformation which can be written in Java only.

  See Java Interfaces for JMSReader (p. 316) for more information about the interface.

  Remember that this component sends each record through all of the connected output ports. Mapping does not need to be defined.

- MultiLevelReader (p. 322) requires a transformation which can only be written in Java.

  See Java Interfaces for MultiLevelReader (p. 324) for more information.

Chapter 38. Common Properties of Writers

Writers are the final components of the transformation graph. Each writer must have at least one input port through which the data flows to this graph component from some of the others. The writers serve to write data to files or database tables located on disk or to send data using some FTP, LDAP or JMS connection. Among the writers, there is also the **Trash** component which discards all of the records it receives (unless it is set to store them in a debug file).

In all writers it is important to decide whether you want either to append data to the existing file or sheet or database table (**Append** attribute for files, for example), or to replace the existing file or sheet or database table by a new one. The **Append** attribute is set to false by default. That means "do not append data, replace it".

It is important to know that you can also write data to one file or one database table by more writers of the same graph, but in such a case you should write data by different writers in different phases.

Remember that (in case of most writers) you can see some part of resulting data when you right-click a writer and select the **View data** option. After that, you will be prompted with the same View data dialog as when debugging the edges. For more details see Viewing Debug Data (p. 93). This dialog allows you to view the written data (it can only be used after graph has already been run).

Here we present a brief overview of links to these options:

- Some examples of **File URL** attribute for writing to local and remote files, through proxy, to console, output port and dictionary.

  Supported File URL Formats for Writers (p. 263)

- Viewing Data on Writers (p. 266)
- Output Port Writing (p. 268)
- How and Where Data Should Be Written (p. 269)
- Selecting Output Records (p. 270)
- Partitioning Output into Different Output Files (p. 270)
- As has been shown in Defining Transformations (p. 234), some **Writers** allow that a transformation may be or must be defined in them. For information about transformation interfaces that must be implemented in transformations written in Java see:

  Java Interfaces for Writers (p. 272)

Here we present an overview of all **Writers**:

Table 38.1. Writers Comparison

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
Trash (p. 414)	none	1	0	✖	✖	✖	✖
UniversalDataWriter (p. 416)	flat file	1	0-1	✖	✖	✖	✖
CloverDataWriter (p. 362)	Clover binary file	1	0	✖	✖	✖	✖
XLSDataWriter (p. 419)	XLS(X) file	1	0-1	✖	✖	✖	✖
StructuredDataWriter (p. 410)	structured flat file	1-3	0-1	✖	✖	✖	✖
EmailSender (p. 377)	e-mails	1	0-2	✖	✖	✖	✖
DBOutputTable (p. 369)	database	1	0-2	✖	✖	✖	✖
DB2DataWriter (p. 364)	database	0-1	0-1	✖	✖	✖	✖
InfobrightDataWriter (p. 381)	database	1	0-1	✖	✖	✖	✖
InformixDataWriter (p. 383)	database	0-1	0-1	✖	✖	✖	✖
MSSQLDataWriter (p. 393)	database	0-1	0-1	✖	✖	✖	✖
MySQLDataWriter (p. 396)	database	0-1	0-1	✖	✖	✖	✖
OracleDataWriter (p. 399)	database	0-1	0-1	✖	✖	✖	✖
PostgreSQLDataWriter (p. 403)	database	0-1	0	✖	✖	✖	✖
XMLWriter (p. 422)	XML file	1-n	0-1	✖	✖	✖	✖
JMSWriter (p. 386)	jms messages	1	0	✔	✖	✔	✖
LDAPWriter (p. 389)	LDAP directory tree	1	0-1	✖	✖	✖	✖
QuickBaseRecordWriter (p. 408)	QuickBase	1	0-1	✖	✖	✖	✖
QuickBaseImportCSV (p. 406)	QuickBase	1	0-2	✖	✖	✖	✖
LotusWriter (p. 391)	Lotus Notes	1	0-1	✖	✖	✖	✖

Supported File URL Formats for Writers

The **File URL** attribute may be defined using the URL File Dialog (p. 59).

The URL shown below can also contain placeholders - dollar sign or hash sign.

Important

You need to differentiate between dollar sign and hash sign usage.

- **Dollar sign** should be used when each of multiple output files contains only a specified number of records based on the **Records per file** attribute.
- **Hash sign** should be used when each of multiple output files only contains records correspoding to the value of specified **Partition key**.

Note

Hash signs in URL examples in this section serve to separate a compressed file (`zip`, `gz`) from its contents. These are not placeholders!

Important

To ensure graph portability, forward slashes must be used when defining the path in URLs (even on Microsoft Windows).

Here we present some examples of possible URL for **Writers**:

Writing to Local Files

- `/path/filename.out`

  Writes specified file on disk.

- `/path1/filename1.out;/path2/filename2.out`

  Writes two specified files on disk.

- `/path/filename$.out`

  Writes some number of files on disk. The dollar sign represents one digit. Thus, the output files can have the names from `filename0.out` to `filename9.out`. The dollar sign is used when **Records per file** is set.

- `/path/filename$$.out`

  Writes some number of files on disk. Two dollar signs represent two digits. Thus, the output files can have the names from `filename00.out` to `filename99.out`. The dollar sign is used when **Records per file** is set.

- `zip:(/path/file$.zip)`

  Writes some number of compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files can have the names from `file0.zip` to `file9.zip`. The dollar sign is used when **Records per file** is set.

- `zip:(/path/file$.zip)#innerfolder/filename.out`

  Writes specified file inside the compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files containing the specified filename.out file can have the names from `file0.zip` to `file9.zip`. The dollar sign is used when **Records per file** is set.

- `gzip:(/path/file$.gz)`

  Writes some number of compressed files on disk. The dollar sign represents one digit. Thus, the compressed output files can have the names from `file0.gz` to `file9.gz`. The dollar sign is used when **Records per file** is set.

Note

Although **CloverETL** can read data from a `.tar` file, writing to a `.tar` file is not supported.

Writing to Remote Files

- `ftp://user:password@server/path/filename.out`

  Writes specified `filename.out` file on remote server connected via ftp protocol using username and password.

- `sftp://user:password@server/path/filename.out`

  Writes specified `filename.out` file on remote server connected via sftp protocol using username and password.

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Writes specified `filename.txt` file compressed in the `file.zip` file on remote server connected via ftp protocol using username and password.

- `zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt`

  Writes specified `filename.txt` file compressed in the `file.zip` file on remote server connected via ftp protocol.

- `zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt`

  Writes specified `filename.txt` file compressed in the `file.zip` file that is also compressed in the `name.zip` file on remote server connected via ftp protocol using username and password.

- `gzip:(ftp://username:password@server/path/file.gz)`

  Writes the first file compressed in the `file.gz` file on remote server connected via ftp protocol.

- `http://username:password@server/filename.out`

  Writes specified `filename.out` file on remote server connected via WebDAV protocol using username and password.

- `http://access_key_id:secret_access_key@bucketname.s3.amazonaws.com/filename.out`

  Writes specified `filename.out` file on Amazon S3 web storage service to the bucket `bucketname` using the `access_key_id` as the of ID of access key and `secret_access_key` as the personal access key.

Writing to Output Port

- `port:$0.FieldName:discrete`

  If this URL is used, output port of the **Writer** must be connected to another component. Output metadata must contain a `FieldName` of one of the following data types: `string`, `byte` or `cbyte`. Each data record that is received by the **Writer** through the input port is processed according to the input metadata, sent out through the optional output port, and written as the value of the specified field of the metadata of the output edge. Next records are parsed in the same way as described here.

Writing to Console

- `-`

  Writes data to `stdout`.

Using Proxy in Writers

- `http:(direct:)//seznam.cz`

Without proxy.

- `http:(proxy://user:password@212.93.193.82:443)//seznam.cz`

  Proxy setting for http protocol.

- `ftp:(proxy://user:password@proxyserver:1234)//seznam.cz`

  Proxy setting for ftp protocol.

- `ftp:(proxy://proxyserver:443)//server/path/file.dat`

  Proxy setting for ftp protocol.

- `sftp:(proxy://66.11.122.193:443)//user:password@server/path/file.dat`

  Proxy setting for sftp protocol.

Writing to Dictionary

- `dict:keyName:discrete`[1)]

  Writes data to dictionary. Creates `ArrayList<byte[]>`

- `dict:keyName:stream`[2)]

  Writes data to dictionary. Creates `WritableByteChannel`

Legend:

1): The `discrete` processing type uses byte array for storing data.

2): The `stream` processing type uses an output stream that must be created before running a graph (from Java code).

Sandbox Resource as Data Source

A sandbox resource, whether it is a shared, local or partitioned sandbox, is specified in the graph under the fileURL attributes as a so called sandbox URL like this:

```
sandbox://data/path/to/file/file.dat
```

where "data" is code for sandbox and "path/to/file/file.dat" is the path to the resource from the sandbox root. URL is evaluated by CloverETL Server during graph execution and a component (reader or writer) obtains the opened stream from the server. This may be a stream to a local file or to some other remote resource. Thus, a graph does not have to run on the node which has local access to the resource. There may be more sandbox resources used in the graph and each of them may be on a different node. In such cases, CloverETL Server would choose the node with the most local resources to minimalize remote streams.

The sandbox URL has a specific use for parallel data processing. When the sandbox URL with the resource in a *partitioned sandbox* is used, that part of the graph/phase runs in parallel, according to the node allocation specified by the list of partitioned sandbox locations. Thus, each worker has its own local sandbox resource. CloverETL Server evaluates the sandbox URL on each worker and provides an open stream to a local resource to the component.

Viewing Data on Writers

After an output file has been created, you can view its data on **Writers** using the context menu. To do that, right-click the desired component and select **View data** from the context menu.

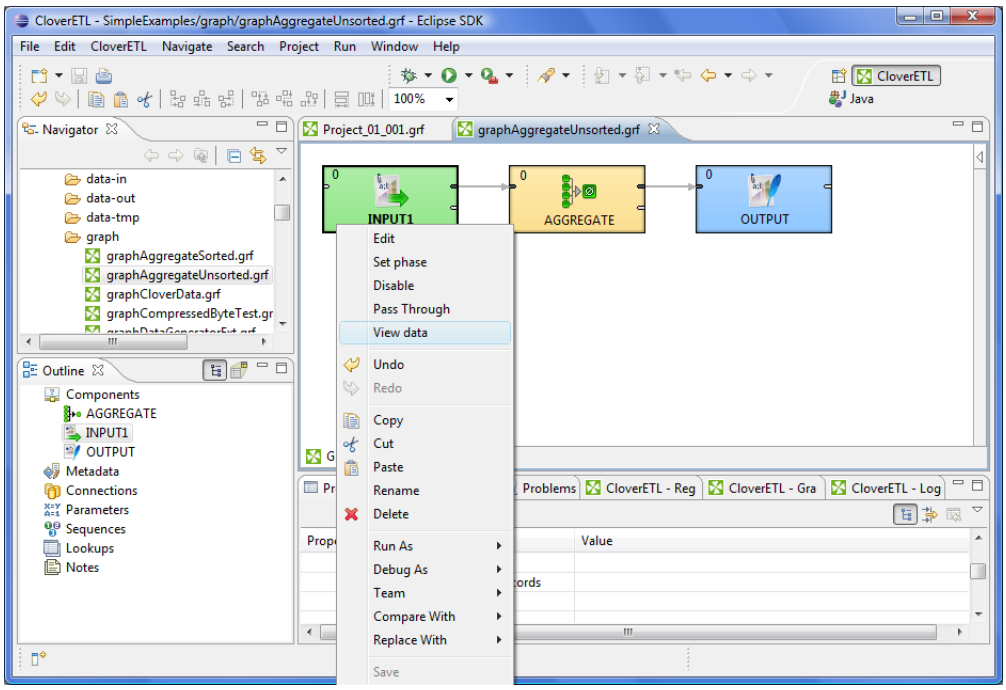


Figure 38.1. Viewing Data on Components

Now you need to choose whether you want to see data as plain text or grid. If you select the **Plain text** option, you can select **Charset**, but you cannot select any filter expression. Windows with results are modal, so that user can view data from components at the same time. To differ between results window title provides info about viewing edge in format GRAPH.name:COMPONENT.name.

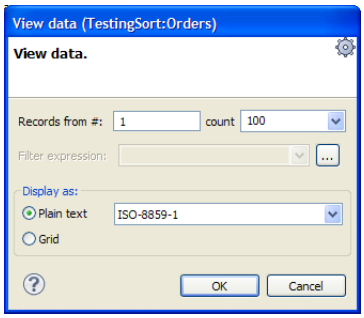


Figure 38.2. Viewing Data as Plain Text

On the other hand, if you select the **Grid** option, you can select **Filter expression**, but no **Charset**.

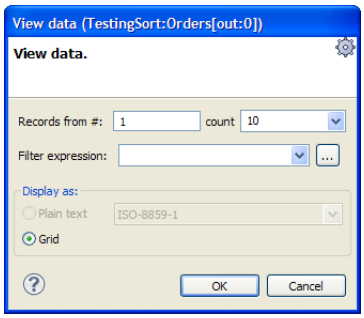


Figure 38.3. Viewing Data as Grid

The result can be as follows in the **Plain text** mode:

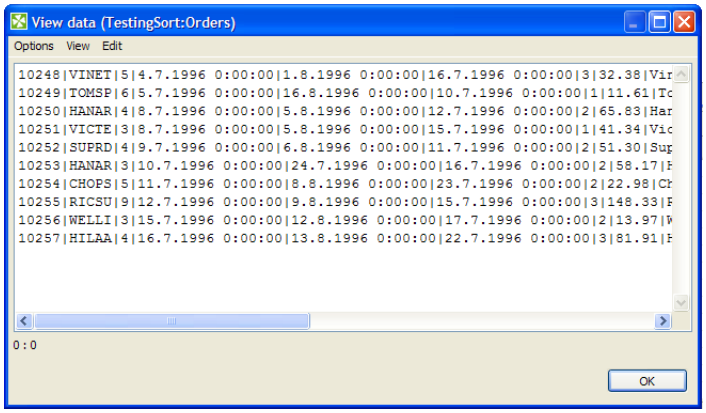
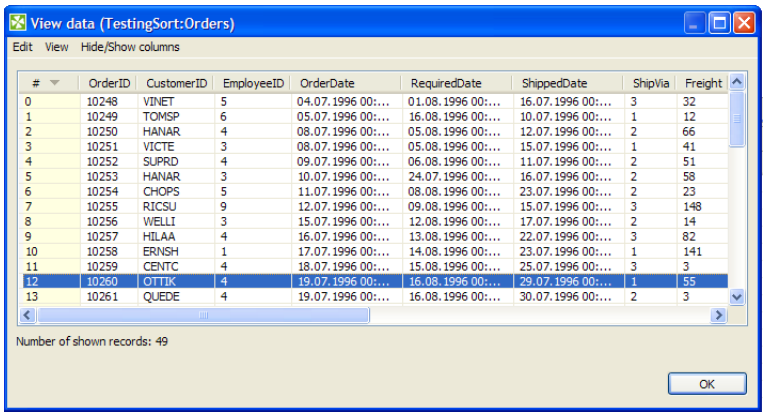


Figure 38.4. Plain Text Data Viewing

Or in the **Grid** mode, it can be like the following:



#	OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight
0	10248	VINET	5	04.07.1996 00:...	01.08.1996 00:...	16.07.1996 00:...	3	32
1	10249	TOMSP	6	05.07.1996 00:...	16.08.1996 00:...	10.07.1996 00:...	1	12
2	10250	HANAR	4	08.07.1996 00:...	05.08.1996 00:...	12.07.1996 00:...	2	66
3	10251	VICTE	3	08.07.1996 00:...	05.08.1996 00:...	15.07.1996 00:...	1	41
4	10252	SUPRD	4	09.07.1996 00:...	06.08.1996 00:...	11.07.1996 00:...	2	51
5	10253	HANAR	3	10.07.1996 00:...	24.07.1996 00:...	16.07.1996 00:...	2	58
6	10254	CHOPS	5	11.07.1996 00:...	08.08.1996 00:...	23.07.1996 00:...	2	23
7	10255	RICSU	9	12.07.1996 00:...	09.08.1996 00:...	15.07.1996 00:...	3	148
8	10256	WELLI	3	15.07.1996 00:...	12.08.1996 00:...	17.07.1996 00:...	2	14
9	10257	HILAA	4	16.07.1996 00:...	13.08.1996 00:...	22.07.1996 00:...	3	82
10	10258	ERNSH	1	17.07.1996 00:...	14.08.1996 00:...	23.07.1996 00:...	1	141
11	10259	CENTC	4	18.07.1996 00:...	15.08.1996 00:...	25.07.1996 00:...	3	3
12	10260	OTTIK	4	19.07.1996 00:...	16.08.1996 00:...	29.07.1996 00:...	1	55
13	10261	QUEDE	4	19.07.1996 00:...	16.08.1996 00:...	30.07.1996 00:...	2	3

Figure 38.5. Grid Data Viewing

The same can be done in some of the **Readers**. See Viewing Data on Writers (p. 266).

Output Port Writing

Some **Writers** allow to write data to the optional output port.

Here we provide the list of **Writers** allowing output port writing:

- **UniversalDataWriter**
- **XLSDataWriter**
- **XMLWriter**
- **StructuredDataWriter**

The attributes for the output port writing in these components may be defined using the URL File Dialog (p. 59).

If you connect the optional output port of any of these **Writers** to an edge, you must also connect the other side of this edge to another component. Metadata of this edge must contain the specified `FieldName` of `string`, `byte` or `cbyte` data type.

Then you must set the **File URL** attribute of such **Writer** to `port:$0.FieldName[:processingType]`.

Here `processingType` is optional and can be set to one of the following: `discrete` or `stream`. If it is not set explicitly, it is `discrete` by default.

When a graph runs, data is read through the input according to the input metadata, processed by the **Writer** according to the specified processing type and sent subsequently to the other component through the optional output port of the **Writer**.

- `discrete`

  Each data record that is received through the input port is processed according to the input metadata, sent out through the optional output port, and written as the value of the specified field of the metadata of the output edge. Next records are parsed in the same way as described here.

- `stream`

  Each data record that is received through the input port is processed in the same way as in case of `discrete` processing type, but another field containing `null` value is added to the end of the output. Such `null` values mean `eof` when multiple files are read again from input port using `stream` processing type. See Input Port Reading (p. 257) for more information about reading with `stream` processing type.

How and Where Data Should Be Written

When you specify some **File URL**, you also need to decide how the following attributes should be set:

- **Append**

  It is very important to decide whether the records should be appended to the existing file (**Append**) or whether the file should be replaced. This attribute is set to `false` by default ("do not append, replace the file").

  This attribute is available in the following **Writers**:

  - **Trash** (the **Debug append** attribute)
  - **UniversalDataWriter**
  - **CloverDataWriter**
  - **XLSDataWriter** (the **Append to the sheet** attribute)
  - **StructuredDataWriter**
  - **XMLWriter**

- **Create directories**

  If you specify some directory in the **File URL** that still does not exist, you must set the **Create directories** attribute to `true`. Such directory will be created. Otherwise, the graph would fail. Remember that the default value of **Create directories** is `false`!

  This attribute is available in the following **Writers**:

  - **Trash**
  - **UniversalDataWriter**
  - **CloverDataWriter**
  - **XLSDataWriter**
  - **StructuredDataWriter**
  - **XMLWriter**

- **Exclude fields**

You can use this attribute to exclude the values of some fields from writing. This attribute should be created using a key wizard and it is used to specify the fields that should not be written to the output. Its form is a sequence of field names separated by semicolon. For example, if you part your output into more files using **Partition key**, you can specify the same fields whose values would not be written to the output files.

- **UniversalDataWriter**
- **XLSDataWriter**

Selecting Output Records

When you set up **Writers**, you may want to limit the records that should be written.

The following **Writers** allow you to limit the number of written records by using the **Number of skipped records** and/or **Max number of records** attributes. What is called **mapping** below, is a subtree which should be mapped from input ports and written to the output file.

- **UniversalDataWriter**
- **CloverDataWriter**
- **XLSDataWriter**
- **StructuredDataWriter**
- **XMLWriter** (the **Number of skipped mappings** and **Max number of mappings** attributes)

Partitioning Output into Different Output Files

Three components allow you to part the incoming data flow and distribute the records among different output files. These components are the following: **UniversalDataWriter**, **XLSDataWriter** and **StructuredDataWriter**.

If you want to part the data flow and write the records into different output files depending on a key value, you must specify the key for such a partition (**Partition key**). It has the form of a sequence of incoming record field names separated by semicolon.

In addition to this, you can also select only some incoming records. This can be done by using a lookup table (**Partition lookup table**). The records whose **Partition key** equals the values of lookup table key are saved to the specified output files, those whose key values do not equal to lookup table key values are either saved to the file specified in the **Partition unassigned file name** attribute or discarded (if no **Partition unassigned file name** is specified).

Remember that if all incoming records are assigned to the values of lookup table, the file for unassigned records will be empty (even if it is defined).

Such lookup table will also serve to group together selected data records into different output files and give them the names. The **Partition output fields** attribute must be specified. It is a sequence of lookup table fields separated by semicolon.

The **File URL** value will only serve as the base name for the output file names. Such base name is concatenated with distinguishing names or numbers. If some partitioning is specified (if **Partition key** is defined), hash signs can be used in **File URL** as placeholder(s) for distinguishing names or numbers. These hash signs must only be used in the file name part of **File URL**.

Important

You need to differentiate between hash sign and dollar sign usage.

- **Hash sign**

  Hash sigh should be used when each of multiple output files only contains records correspoding to the value of specified **Partition key**.

- **Dollar sign**

  Dollar sigh should be used when each of multiple output files contains only a specified number of records based on the **Records per file** attribute.

The hash(es) can be located in any place of this file part of **File URL**, even in its middle. For example: `path/output#.xls` (in case of the output XLS file). If no hash is contained in **File URL**, distinguishing names or numbers will be appended to the end of the file base name.

If **Partition file tag** is set to `Number file tag`, output files are numbered and the count of hashes used in **File URL** means the count of digits for these distinguishing numbers. This is the default value of **Partition file tag**. Thus, ### can go from `000` to `999`.

If **Partition file tag** is set to `Key file tag`, single hash must be used in **File URL**.at most. Distinguishing names are used.

These distinguishing names will be created as follows:

If the **Partition key** attribute (or the **Partition output fields** attribute) is of the following form: `field1;field2;...;fieldN` and the values of these fields are the following: `valueofthefield1`, `valueofthefield2`, ..., `valueofthefieldN`, all the values of the fields are converted to strings and concatenated. The resulting strings will have the following form: `valueofthefield1valueofthefield2...valueofthefieldN`. Such resulting strings are used as distinguishing names and each of them is inserted to the **File URL** into the place marked with hash. Or appended to the end of **File URL** if no hash is used in **File URL**.

For example, if `firstname;lastname` is the **Partition key** (or **Partition output fields**), you can have the output files as follows:

- `path/outjohnsmith.xls`, `path/outmarksmith.xls`, `path/outmichaelgordon.xls`, etc. (if **File URL** is `path/out#.xls` and **Partition file tag** is set to `Key file tag`).
- Or `path/out01.xls`, `path/out02.xls`. etc. (if **File URL** is `path/out##.xls` and **Partition file tag** is set to `Number file tag`).

In **XLSDataWriter** and **UniversalDataWriter**, there is another attribute: **Exclude fields**.

It is a sequence of field names separated by semicolon that should not be written to the output. It can be used when the same fields serve as a part of **Partition key**.

If you are partitioning data using any of these two writers and **Partition file tag** is set to **Key file tag**, values of **Partition key** are written to the names of these files. At the same time, the same values should be written to corresponding output file.

In order to avoid the files whose names would contain the same values as those written in them, you can select the fields that will be excluded when writing to these files. You need to choose the **Exclude fields** attribute.

These fields will only be part of file or sheet names, but will not be written to the contents of these files.

Subsequently, when you will read these files, you will be able to use an autofilling function (`source_name` for **UniversalaDataReader** or **XLSDataReader**, or `sheet_name` for **XLSDataReader**) to get such value from either file name or sheet name (when you have previously set **Sheet name** to `$<field name>`).

In other words, when you have files created using **Partition key** set to `City` and the output files are `London.txt`, `Stockholm.txt`, etc., you can get these values (`London`, `Stockholm`, etc.) from these names. The `City` field values do not need to be contained in these files.

Note

If you want to use the value of a field as the path to an existing file, type the following as the **File URL** attribute in **Writer**:

```
//#
```

This way, if the value of the field used for partitioning is `path/to/my/file/filename.txt`, it will be assigned to the output file as its name. For this reason, the output file will be located in `path/to/my/file` and its name will be `filename.txt`.

Java Interfaces for Writers

- JMSWriter (p. 386) allows optionally a transformation, which can only be written in Java.

  See Java Interfaces for JMSWriter (p. 388) for more information about the interface.

Chapter 39. Common Properties of Transformers

These components have both input and output ports. They can put together more data flows with the same metadata (**Concatenate**, **SimpleGather**, and **Merge**), remove duplicate records (**Dedup**), filter data records (**ExtFilter** and **EmailFilter**), create samples from input records (**DataSampler**), sort data records (**ExtSort**, **FastSort**, and **SortWithinGroups**), multiplicate existing data flow (**SimpleCopy**) split one data flow into more data flows (**Partition** at all, but optionally also **Dedup**, **ExtFilter**, also **Reformat**), intersect two data flows (even with different metadata on inputs) (**DataIntersection**), aggregate data information (**Aggregate**), and perform much more complicated transformations of data flows (**Reformat**, **Denormalizer**, **Pivot**, **Normalizer**, **MetaPivot**, **Rollup**, and **XLSTransformer**).

Metadata can be propagated through some of these transformers, whereas the same is not possible in such components that transform data flows in a more complicated manner. You must have the output metadata defined prior to configuring these components.

Some of these transformers use transformations that have been described above. See Defining Transformations (p. 234) for detailed information about how transformation should be defined.

- Some **Transformers** can have a transformation attribute defined, it may be optional or required. For information about transformation templates for transformations written in CTL see:

  CTL Templates for Transformers (p. 274)

- Some **Transformers** can have a transformation attribute defined, it may be optional or required. For information about transformation interfaces that must be implemented in transformations written in Java see:

  Java Interfaces for Transformers (p. 275)

Here we present an overview of all **Transformers**:

Table 39.1. Transformers Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
SimpleCopy (p. 517)	-	✖	1	1-n	-	-
ExtSort (p. 472)	-	✖	1	1-n	-	-
FastSort (p. 474)	-	✖	1	1-n	-	-
SortWithinGroups (p. 519)	-	✔	1	1-n	-	-
Dedup (p. 454)	-	✔	1	1-2	-	-
ExtFilter (p. 470)	-	✖	1	1-2	-	-
Address Doctor 5 (p. 441)	-	✖	1	1-2	-	-
EmailFilter (p. 465)	-	✖	1	0-2	-	-
Concatenate (p. 448)	✔	✖	1-n	1	-	-
SimpleGather (p. 518)	✔	✖	1-n	1	-	-
Merge (p. 479)	✔	✔	2-n	1	-	-
Partition (p. 491)	-	✖	1	1-n	yes/no[1)]	yes/no[1)]
DataIntersection (p. 449)	✖	✔	2	3	✔	✔
Aggregate (p. 445)	-	✖	1	1	-	-

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Reformat (p. 502)	-	✖	1	1-n	✔	✔
Denormalizer (p. 456)	-	✖	1	1	✔	✔
Pivot (p. 498)	-	✖	1	1	✔	✔
Normalizer (p. 484)	-	✖	1	1	✔	✔
MetaPivot (p. 481)	-	✖	1	1	-	-
Rollup (p. 505)	-	✖	1	1-n	✔	✔
DataSampler (p. 452)	-	✖	1	n	-	-
XSLTransformer (p. 521)	-	✖	1	1	-	-

Legend

1) **Partition** can use either the transformation or two other attributes (**Ranges** or **Partition key**). A transformation must be defined unless one of these is specified.

CTL Templates for Transformers

- Partition (p. 491) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** is defined.

  See Java Interfaces for Partition (and ClusterPartitioner) (p. 497) for more information about the transformation template.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping does not need to be done, records are mapped automatically.

- DataIntersection (p. 449) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for DataIntersection (p. 451) for more information about the transformation template.

- Reformat (p. 502) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Reformat (p. 503) for more information about the transformation template.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

- Denormalizer (p. 456) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Denormalizer (p. 458) for more information about the transformation template.

- Normalizer (p. 484) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Normalizer (p. 485) for more information about the transformation template.

- Rollup (p. 505) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for Rollup (p. 507) for more information about the transformation template.

Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

Java Interfaces for Transformers

- Partition (p. 491) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** is defined.

  See Java Interfaces for Partition (and ClusterPartitioner) (p. 497) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping does not need to be done, records are mapped automatically.

- DataIntersection (p. 449) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for DataIntersection (p. 451) for more information about the interface.

- Reformat (p. 502) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for Reformat (p. 504) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

- Denormalizer (p. 456) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for Denormalizer (p. 464) for more information about the interface.

- Normalizer (p. 484) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for Normalizer (p. 490) for more information about the interface.

- Rollup (p. 505) requires a transformation which can be written in both CTL and Java.

  See Java Interfaces for Rollup (p. 515) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping must be defined for such port.

Chapter 40. Common Properties of Joiners

These components have both input and output ports. They serve to put together the records with potentially different metadata according to the specified key and the specified transformation.

The first input port is called master (driver), the other(s) are called slave(s).

They can join the records incoming through two input ports (**ApproximativeJoin**), or at least two input ports (**ExtHashJoin**, **ExtMergeJoin**, and **RelationalJoin**). The others can also join the records incoming through a single input port with those from lookup table (**LookupJoin**) or database table (**DBJoin**). In them, their slave data records are considered to be incoming through a virtual second input port.

Three of these **Joiners** require that incoming data are sorted: **ApproximativeJoin**, **ExtMergeJoin**, and **RelationalJoin**.

Unlike all of the other **Joiners**, **RelationalJoin** joins data records based on the non-equality conditions. All the others require that key fields on which they are joined have the same values so that these records may be joined.

ApproximativeJoin, **DBJoin**, and **LookupJoin** have optional output ports also for nonmatched master data records. **ApproximativeJoin** has optional output ports for nonmatched both master and slave data records.

Metadata cannot be propagated through these components. You must first select the right metadata or create them by hand according to the desired result. Only then you can define the transformation. For some of the output edges you can also select the metadata on the input, but neither these metadata can be propagated through the component.

These components use a transformations that are described in the section concerning transformers. See Defining Transformations (p. 234) for detailed information about how transformation should be defined. All of the transformations in **Joiners** use common transformation template (CTL Templates for Joiners (p. 278)) and common Java interface (Java Interfaces for Joiners (p. 281)).

Here we present a brief overview of links to these options:

- Join Types (p. 277)
- Slave Duplicates (p. 277)
- CTL Templates for Joiners (p. 278)
- Java Interfaces for Joiners (p. 281)

Here we present an overview of all **Joiners**:

Table 40.1. Joiners Comparison

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
ApproximativeJoin (p. 524)	✖	✔	1	2-4	✔	✔	✔
ExtHashJoin (p. 535)	✖	✖	1-n	1	✖	✖	✔
ExtMergeJoin (p. 541)	✖	✔	1-n	1	✖	✖	✔
LookupJoin (p. 546)	✖	✖	1 (virtual)	1-2	✔	✖	✔
DBJoin (p. 532)	✖	✖	1 (virtual)	1-2	✔	✖	✔

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
RelationalJoin (p. 549)	✖	✔	1	1	✖	✖	✖

Join Types

These components can work under the following three processing modes:

- **Inner Join**

  In this processing mode, only the master records in which the values of **Join key** fields equal to the values of their slave counterparts are processed and sent out through the output port for joined records.

  For **ApproximativeJoin**, the name of this attribute is **Matching key**.

  The unmatched master records can be sent out through the optional output port for master records without a slave (in **ApproximativeJoin**, **LookupJoin** or **DBJoin** only).

  The unmatched slave records can be sent out through the optional output port for slave records without a master (in **ApproximativeJoin** only).

- **Left Outer Join**

  In this processing mode, only the master records in which the values of **Join key** fields do not equal to the values of their slave counterparts are processed and sent out through the output port for joined records.

  For **ApproximativeJoin**, the name of this attribute is **Matching key**.

  The unmatched slave records can be sent out through the optional output port for slave records without a master (in **ApproximativeJoin** only).

- **Full Outer Join**

  In this processing mode, all records, both the masters and the slaves, regardless of whether the values of **Join key** fields are equal to the values of their slave counterparts or not, are processed and sent out through the output port for joined records.

  For **ApproximativeJoin**, the name of this attribute is **Matching key**.

Important

Full outer join mode is not allowed in **LookupJoin** and **DBJoin**.

Note

Remember that **Joiners** parse each pair of records (master and slave) in which the same fields of the **Join key** attribute have `null` values as if these `nulls` were different. Thus, these records do not match one another in such fields and are not joined.

Slave Duplicates

In **Joiners**, sometimes more slave records have the same values of corresponding fields of **Join key** (or **Matching key**, in **ApproximativeJoin**). These slaves are called duplicates. If such duplicate slave records are allowed, all

of them are parsed and joined with the master record if they match any. If the duplicates are not allowed, only one of them or at least some of them is/are parsed (if they match any master record) and the others are discarded.

Different **Joiners** allow to process slave duplicates in a different way. Here we present a brief overview of how these duplicates are parsed and what can be set in these components or other tools:

- **ApproximativeJoin**

  All records with duplicate slaves (the same values of **Matching key**) are always processed.

- **Allow slave duplicates** attribute is included in the following **Joiners** (It can be set to `true` or `false`.):

  - **ExtHashJoin**

    Default is `false`. Only the **first** record is processed, the others are discarded.

  - **ExtMergeJoin**

    Default is `true`. If switched to `false`, only the **last** record is processed, the others are discarded.

  - **RelationalJoin**

    Default is `false`. Only the **first** record is processed, the others are discarded.

- **SQL query** attribute is included in **DBJoin**. **SQL query** allows to specify the exact number of slave duplicates explicitly.

- **LookupJoin** parses slave duplicates according to the setting of used lookup table in the following way:

  - **Simple lookup table** has also the **Allow key duplicate** attribute. Its default value is `true`. If you uncheck the checkbox, only the **last** record is processed, the others are discarded.

  - **DB lookup table** allows to specify the exact number of slave duplicates explicitly.

  - **Range lookup table** does not allow slave duplicates. Only the **first** slave record is used, the others are discarded.

  - **Persistent lookup table** does not allow slave duplicates. Nevertheless, it has the **Replace** attribute. By default, new slave records overwrite the old ones, which are discarded. By default, the **last** slave record remains, the others are discarded. If you uncheck the checkbox, the **first** remains and the others are discarded.

  - **Aspell lookup table** allows that all slave duplicates are used. No limitation of the number of duplicates is possible.

CTL Templates for Joiners

This transformation template is used in every **Joiner** and also in **Reformat** and **DataIntersection**.

Here is an example of how the **Source** tab for defining the transformation in CTL looks.

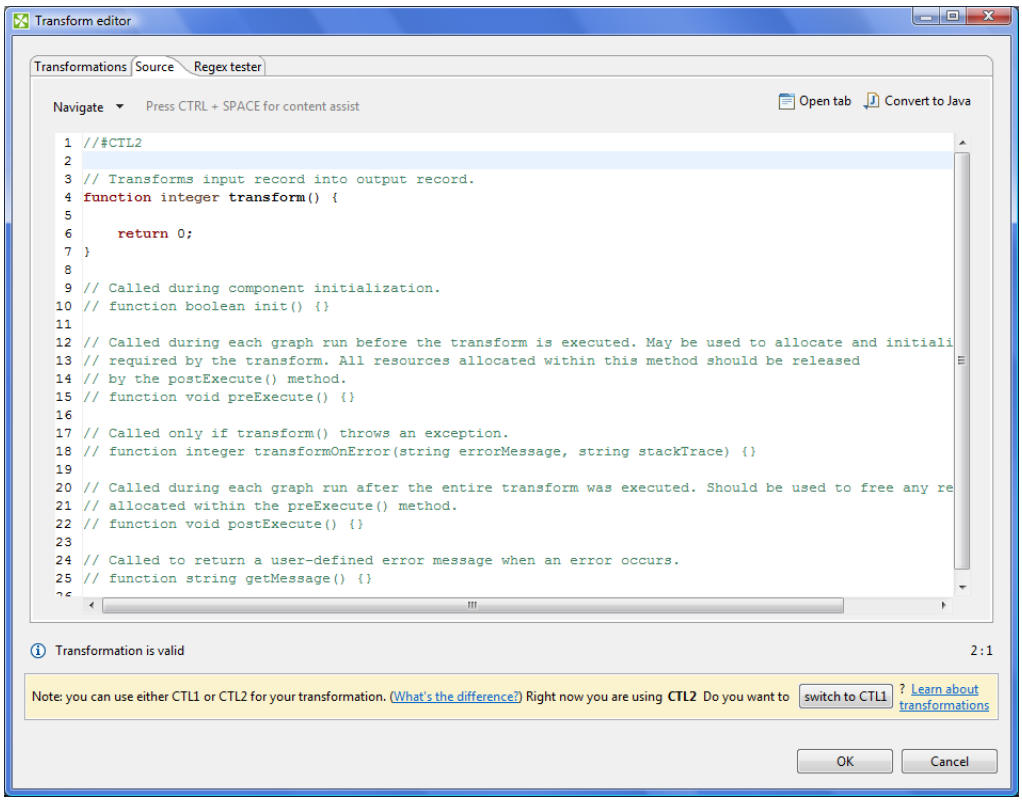


Figure 40.1. Source Tab of the Transform Editor in Joiners

Table 40.2. Functions in Joiners, DataIntersection, and Reformat

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer transform()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly for each set of joined or intersected input records (**Joiners** and **DataIntersection**), and for each input record (**Reformat**).

CTL Template Functions	
Description	Allows you to map input fields to the output fields using a script. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transform() {`
	`  $0.name = $0.name;`
	`  $0.address = $city + $0.street + $0.zip;`
	`  $0.country = toUpper($0.country);`
	`  return ALL;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.name = $0.name;`
	`  $0.address = $city + $0.street + $0.zip;`
	`  $0.country = "country was empty";`
	`  printErr(stackTrace);`
	`  return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user
Invocation	Called in any time specified by user (called only when `transform()` returns value less than or equal to -2).
Returns	`string`

CTL Template Functions	
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Input records or fields and output records or fields**

  Both inputs and outputs are accessible within the `transform()` and `transformOnError()` functions only.

- All of the other CTL template functions allow to access neither inputs nor outputs.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Joiners

This is used in every **Joiner** and also in **Reformat** and **DataIntersection**.

The transformation implements methods of the `RecordTransform` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of the `RecordTransform` interface:

- `boolean init(Properties parameters, DataRecordMetadata[] sourcesMetadata, DataRecordMetadata[] targetMetadata)`

  Initializes reformat class/function. This method is called only once at the beginning of transformation process. Any object allocation/initialization should happen here.

- `int transform(DataRecord[] sources, DataRecord[] target)`

  Performs reformat of source records to target records. This method is called as one step in transforming flow of records. See Return Values of Transformations (p. 238) for detailed information about return values and their meaning.

- `int transformOnError(Exception exception, DataRecord[] sources, DataRecord[] target)`

Performs reformat of source records to target records. This method is called as one step in transforming flow of records. See Return Values of Transformations (p. 238) for detailed information about return values and their meaning. Called only if `transform(DataRecord[], DataRecord[])` throws an exception.

- `void signal(Object signalObject)`

  Method which can be used for signalling into transformation that something outside happened. (For example in aggregation component key changed.)

- `Object getSemiResult()`

  Method which can be used for getting intermediate results out of transformation. May or may not be implemented.

Chapter 41. Common Properties of Cluster Components

These components serve to work on Clusters of **CloverETL Servers**.

Here we present an overview of the two **Cluster components**:

Table 41.1. Cluster Components Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
ClusterGather (p. 554)	✔	✖	1-n (virtual)	1	-	-
ClusterPartitioner (p. 556)	-	✖	1	1-n (virtual)	yes/no[1)]	yes/no[1)]

Legend

1) **ClusterPartitioner** can use either the transformation or two other attributes (**Ranges** or **Partition key**). A transformation must be defined unless one of these is specified.

CTL Templates for Cluster Components

- ClusterPartitioner (p. 556) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** is defined.

  See Java Interfaces for ClusterPartitioner (p. 558) for more information about the transformation template.

  Remember that this component sends each record through the virtual output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping does not need to be done, records are mapped automatically.

  Records are sent through these virtual ports to Cluster nodes.

Java Interfaces for Cluster Components

- ClusterPartitioner (p. 556) requires a transformation (which can be written in both CTL and Java) unless **Partition key** or **Ranges** is defined.

  See Java Interfaces for ClusterPartitioner (p. 558) for more information about the interface.

  Remember that this component sends each record through the connected output port whose number is equal to the value returned by the transformation (Return Values of Transformations (p. 238)). Mapping does not need to be done, records are mapped automatically.

  Records are sent through these virtual ports to Cluster nodes.

Chapter 42. Common Properties of Others

These components serve to fulfil some tasks that has not been mentioned already. We will describe them now. They have no common properties as they are heterogeneous group.

Only JavaExecute (p. 571) requires that a transformation is defined in this component. It must implement the following interface:

Java Interfaces for JavaExecute (p. 572)

Here we will present an overview of all **Others**:

Table 42.1. Others Comparison

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
SystemExecute (p. 583)	-	✖	0-1	0-1	-	✖	✖
JavaExecute (p. 571)	-	-	0	0	-	✔	✖
DBExecute (p. 564)	-	✖	0-1	0-2	-	✖	✖
RunGraph (p. 575)	-	✖	0-1	1-2	-	✖	✖
HTTPConnector (p. 568)	-	✖	0-1	0-1	-	✖	✖
WebServiceClient (p. 586)	-	✖	0-1	0-N	no[2)]	✖	✖
CheckForeignKey (p. 560)	✖	✖	2	1-2	-	✖	✖
SequenceChecker (p. 579)	-	✖	1	1-n	✔	✖	✖
LookupTableReaderWriter (p. 573)	-	✖	0-1	0-n	✔	✖	✖
SpeedLimiter (p. 581)	-	✖	1	1-n	✔	✖	✖

Legend

1) Component sends each data record to all connected output ports.

2) Component sends processed data records to the connected output ports as specified by mapping.

Go now to Chapter 49, Others (p. 559).

Chapter 43. Custom Components

Apart from components provided by **CloverETL** defaultly, you can write your own components. For the step-by-step instructions go to **Creating a Custom Component** document located at our documentation page.

Part VII. Component Reference

Chapter 44. Readers

We assume that you already know what components are. See Chapter 17, Components (p. 84) for an overview.

Only some of the components in a graph are initial nodes. These are called **Readers**.

Readers can read data from input files (both local and remote), receive it from the connected optional input port, or read it from a dictionary. One component only generates data. Since it is also an initial node, we will describe it here.

Components can have different properties. But they also can have some in common. Some properties are common for all of them, others are common for most of the components, or they are common for **Readers** only. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

We can distinguish **Readers** according to what they can read:

- One component only generates data:
  - DataGenerator (p. 298) generates data.

Other **Readers** read data from files.

- Flat files:
  - UniversalDataReader (p. 333) reads data from flat files (delimited or fixed length).
  - ParallelReader (p. 326) reads data from delimited flat files using more threads.
  - ComplexDataReader (p. 291) reads data from really ugly flat files whose structure is heterogeneous or mutually dependent and it uses a neat GUI to achieve that.
  - MultiLevelReader (p. 322) reads data from flat files with a heterogeneous structure.
- Other files:
  - CloverDataReader (p. 289) reads data from files in Clover binary format.
  - XLSDataReader (p. 338) reads data from XLS or XLSX files.
  - DBFDataReader (p. 305) reads data from dBase files.
  - XMLExtract (p. 342) reads data from XML files using SAX technology.
  - XMLXPathReader (p. 354) reads data from XML files using XPath queries.

Other **Readers** unload data from databases.

- Databases:
  - DBInputTable (p. 307) unloads data from database using JDBC driver.
  - QuickBaseRecordWriter (p. 408) reads data from the **QuickBase** online database.
  - QuickBaseImportCSV (p. 406) reads data from the **QuickBase** online database using queries.
  - LotusReader (p. 320) reads data from **Lotus Notes** or **Lotus Domino** database.

Other **Readers** receive JMS messages or read directory structure.

- JMS messages:
  - JMSReader (p. 314) converts JMS messages into data records.
- Directory structure:
  - LDAPReader (p. 317) converts directory structure into data records.
- Email messages:
  - EmailReader (p. 311) Reads email messages.

CloverDataReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

CloverDataReader reads data stored in our internal binary Clover data format files.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
CloverDataReader	clover binary file	0	1-n	yes	no	no	no	no	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

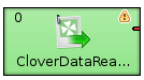
Abstract

CloverDataReader reads data stored in our internal binary Clover data format files. It can also read data from compressed files, console, or dictionary.

Note

Since 2.9 version of **CloverETL** **CloverDataWriter** writes also a header to output files with the version number. For this reason, **CloverDataReader** expects that files in Clover binary format contain such a header with the version number. **CloverDataReader** 2.9 cannot read files written by older versions of **CloverETL** nor these older versions can read data written by **CloverDataWriter** 2.9.

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	yes	For correct data records	Any[1)]
	1-n	no	For correct data records	Output 0

Legend:

1): Metadata can use Autofilling Functions (p. 117).

CloverDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (flat file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Index file URL[1)]		Name of the index file, including path. See Supported File URL Formats for Readers (p. 251). See also Output File Structure (p. 363) for more information about index file names.	
Advanced			
Number of skipped records		Number of records to be skipped. See Selecting Input Records (p. 258).	0-N
Max number of records		Maximum number of records to be read. See Selecting Input Records (p. 258).	0-N
Deprecated			
Start record		Has exclusive meaning: Last record before the first that is already read. Has lower priority than **Number of skipped records**.	0 (default) | 1-n
Final record		Has inclusive meaning: Last record to be read. Has lower priority than **Max number of records**.	all (default) | 1-n

Legend:

1) If this attribute is not specified, all records must be read.

ComplexDataReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the appropriate **Reader** for your purpose, see Readers Comparison (p. 251).

Short Summary

ComplexDataReader reads non-homogeneous data from files.

Component	Data source	Input ports	Output ports	Each to all outputs	Different to different outputs	Transformation	Transf. req.	Java	CTL
ComplexDataReader	flat file	1	1-n	no	yes	yes	yes	yes	yes

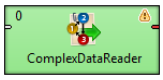
Abstract

ComplexDataReader reads non-homogeneous data from files containing multiple metadata, using the concept of states and transitions and optional lookahead (selector).

The user-defined states and their transitions impose the order of metadata used for parsing the file - presumably following the file's structure.

The component uses the **Data policy** attribute as described in Data Policy (p. 260).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any (Out0)[1)]
	1-N	no	For correct data records	Any (Out1-OutN)

Legend:

1): Metadata on output ports can use Autofilling Functions (p. 117).

ComplexDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	The data source(s) which **ComplexDataReader** should read from. The source can be a flat file, the console, an input port or a dictionary. See Supported File URL Formats for Readers (p. 251).	
Transform		The definition of the state machine that carries out the reading. The settings dialog opens in a separate window that is described in Advanced Description (p. 293).	
Charset		The encoding of records that are read.	ISO-8859-1 (default) | <any encoding>
Data policy		Determines steps that are done when an error occurs. See Data Policy (p. 260) for details. Unlike other Readers, `Controlled` **Data Policy** is not implemented. `Lenient` allows you to skip redundant columns in metadata with a record delimiter (but not incorrect lines).	Strict (default) | Lenient
Trim strings		Specifies whether leading and trailing whitespaces should be removed from strings before inserting them to data fields. See Trimming Data (p. 335).	false (default) | true
Quoted strings		Fields containing a special character (comma, newline, or double quotes) have to be enclosed in quotes. If `true`, these special characters inside the quoted string are not treated as delimiters and the quotes are removed.	false (default) | true
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**.	both (default) | " | '
Advanced			
Skip leading blanks		Specifies whether leading whitespace characters (spaces etc.) will be skipped before inserting input strings to data fields. If you leave it default, the value of **Trim strings** is used. See Trimming Data (p. 335).	false (default) | true
Skip trailing blanks		Specifies whether trailing whitespace characters (spaces etc.) will be skipped before inserting input strings to data fields. If you leave it default, the value of **Trim strings** is used. See Trimming Data (p. 335).	false (default) | true

Attribute	Req	Description	Possible values
Max error count		The maximum number of tolerated error records on the input. The attribute is applicable only if `Controlled` **Data Policy** is being used.	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter character, it will be interpreted as a single delimiter if this attribute is `true`.	false (default) | true
Verbose		By default, not so complex error notification is provided and the performance is fairly high. However, if switched to `true`, more detailed information with lower performance will be provided.	false (default) | true
Selector code	1)	If you decide to use a selector, here you can write its code in Java. A selector is only an optional feature in the transformation. It supports decision-making when you need to look ahead at the data file. See Advanced Description (p. 293).	
Selector URL	1)	The name and path to an external file containing a selector code written in Java. To learn more about the Selector, see Advanced Description (p. 293).	
Selector class	1)	The name of an external class containing the Selector. To learn more about the Selector, see Advanced Description (p. 293).	
Transform URL		The path to an external file which defines state transitions in the state machine.	
Transform class		The name of a Java class that defines state transitions in the state machine.	
Selector properties		Allows you to instantly edit the current Selector in the **State transitions** window.	
State metadata		Allows you to instantly edit the metadata and states assigned to them in the **State transitions** window.	

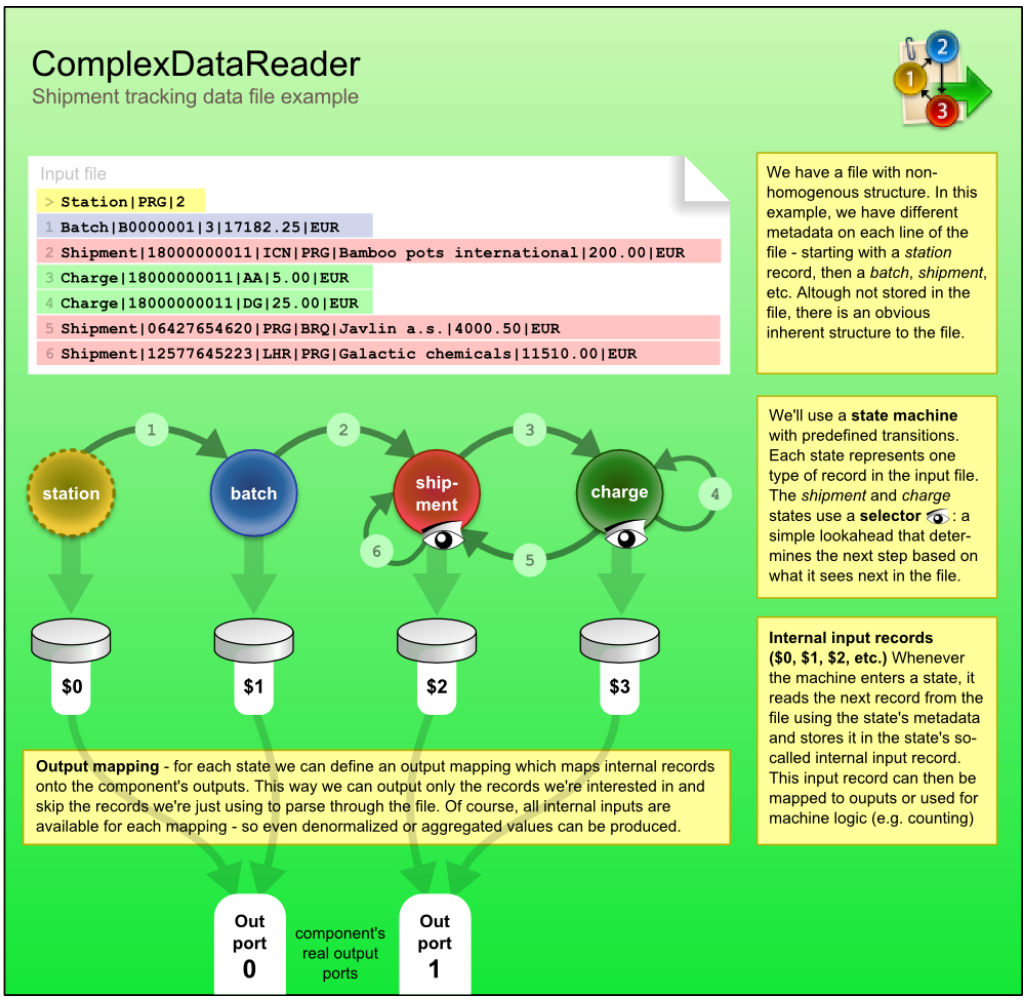
Legend:

1): If you do not define any of these three attributes, the default **Selector class** (`PrefixInputMetadataSelector`) will be used.

Advanced Description

Reading heterogeneous data is generally not an easy task. The data may mix various data formats, delimiters, fields and record types. On top of that, records and their semantics can be dependent on eachother. For example, a record of type `address` can mean a person's address if the preceding record is a `person`, or company's address in the case where our `address` follows a `company`.

MultiLevelReader and **ComplexDataReader** are very similar components in terms of what they can achieve. In **MultiLevelReader** you needed to program the whole logic as a Java transform (in the form of AbstractMultiLevelSelector extension) but, in ComplexDataReader even the trickiest data structures can be configured using the powerful GUI. A new concept of states and transitions has been introduced in **ComplexDataReader**, and the parsing logic is implemented as a simple CTL2 script.



Transitions between states can either be given explicitly - for example, state 3 always follows 2, computed in CTL - for example, by counting the number of entries in a group, or you can "consult" the helping tool to choose the transition. The tool is called **Selector** and it can be understood as a magnifying glass that looks ahead at the upcoming data without actually parsing it.

You can either custom-implement the selector in Java or just use the default one. The default selector uses a table of prefixes and their corresponding transitions. Once it encounters a particular prefix it evaluates all transitions and returns the first matching target state.

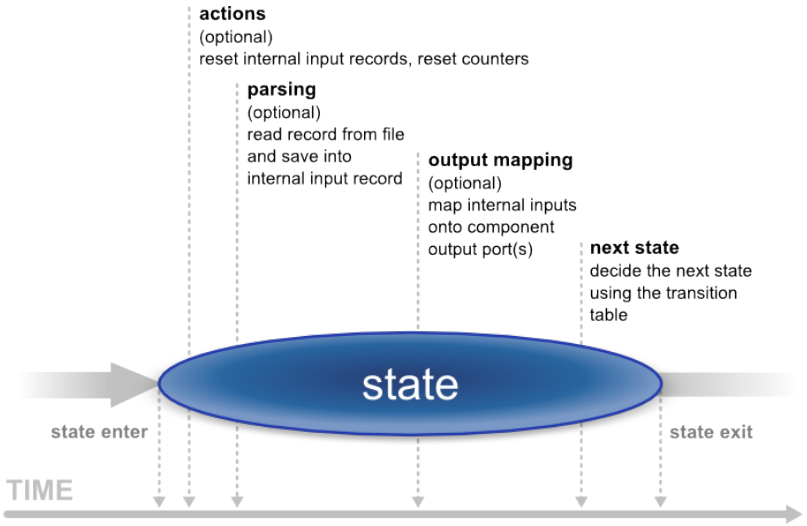
Now let us look a little bit closer on what happens in a state (see picture below). As soon as we enter a state, its **Actions** are performed. Available actions are:

- **Reset counter** - resets a counter which stores how many times the state has been accessed
- **Reset record** - reset the number of records located in internal storages. Thus, it ensures that various data read do not mix with each other.

Next, **Parsing** of the input data is done. This reads a record in from the file and stores it in the state's internal input.

After that comes **Output**, which involves mapping the internal inputs to the component's output ports. This is the only step in which data is sent out of the component.

Finally, there is **Transition** which defines how the machine changes to the next state.



Last but not least, writing the whole reading logics in CTL is possible as well. See CTL in ComplexDataReader (p. 296) for reference.

Video - How to Work with ComplexDataReader

Instead of reading tons of material, why not take a look at a video walkthrough. After watching it, you should have a clear notion of how to use and configure the **ComplexDataReader** component.

See ComplexDataReader example video:

http://www.cloveretl.com/resources/repository/complexdatareader-shipments

Designing State Machine

To start designing the machine, edit the **Transform** attribute. A new window opens offering these tabs: **States**, **Overview**, **Selector**, **Source** and other tabs representing states labeled **$stateNo stateLabel**, e.g. "$0 myFirstState".

On the left hand side of the **States** tab, you can see a pane with all the **Available metadata** your graph works with. In this tab, you design new states by dragging metadata to the right hand side's **States** pane. At the bottom, you can set the **Initial state** (the first state) and the **Final state** (the machine switches to it shortly before terminating its execution or if you call **Flush and finish**). The final state can serve mapping data to the output before the automaton terminates (especially handy for treating the last records of your input). Finally, in the centre there is the **Expression editor** pane, which supports **Ctrl**+**Space** content assist and lets you directly edit the code.

In the **Overview** tab, the machine is graphically visualised. Here you can **Export Image** to an external file or **Cycle View Modes** to see other graphical representations of the same machine. If you click **Undock**, the whole view will open in a separate window that is regularly refreshed.

In state tabs (e.g. "$0 firstState") you define the outputs in the **Output ports** pane. What you see in **Output field** is in fact the (fixed) output metadata. Next, you define **Actions** and work with the **Transition table** at the bottom pane in the tab. Inside the table, there are **Conditions** which are evaluated top-down and **Target states** assigned to them. These are these values for **Target states**:

- **Let selector decide** - the selector determines which state to go to next
- **Flush and finish** - this causes a regular ending of the machine's work
- **Fail** - the machine fails and stops its execution. (e.g it comes across an invalid record)
- A particular state the machine changes to.

The **Selector** tab allows you to implement your own selector or supply it in an external file/Java class.

Finally, the **Source** tab shows the code the machine performs. For more information, see CTL in ComplexDataReader (p. 296)

CTL in ComplexDataReader

The machine can be specified in three ways. First, you can design it as a whole through the GUI. Second, you can create a Java class that describes it. Third, you can write its code in CTL inside the GUI by switching to the **Source** tab in **Transform**, where you can see the source code the machine performs.

Important

Please note you do not have to handle the source code at all. The machine can be configured entirely in the other graphical tabs of this window.

Changes made in **Source** take effect in remaining tabs if you click **Refresh states**. If you want to synchronise the source code with states configuration, click **Refresh source**.

Let us now outline significant elements of the code:

Counters

There are the `counterStateNo` variables which store the number of times a state has been accessed. There is one such variable for each state and their numbering starts with 0. So e.g. counter2 stores how many times state $2 was accessed. The counter can be reset in **Actions**.

Initial State Function

`integer initialState()` - determines which state of the automaton is the first one initiated. If you return `ALL`, it means **Let selector decide**, i.e. it passes the current state to the selector that determines which state will be next (if it cannot do that, the machine fails)

Final State Function

`integer finalState(integer lastState)` - specifies the last state of the automaton. If you return `STOP`, it means the final state is not defined.

Functions In Every State

Each state has two major functions describing it:

- `nextState`
- `nextOutput`

`integer nextState_stateNo()` returns a number saying which state follows the current state (`stateNo`). If you return `ALL`, it means **Let selector decide**. If you return `STOP`, it means **Flush and finish**.

Example 44.1. Example State Function

```
nextState_0() {
 if(counter0 > 5) {
  return 1; // if state 0 has been accessed more than five times since
            // the last counter reset, go to state 1
 }
 return 0;  // otherwise stay in state 0
}
```

`nextOutput_stateNo(integer seq)` - the main output function for a particular state (`stateNo`). It calls the individual `nextOutput_stateNo_seq()` service functions according to the value of `seq`. The `seq` is a

counter which stores how many times the `nextOutput_stateNo` function has been called so far. At last, it calls `nextOutput_stateNo_default(integer seq)` which typically returns `STOP` meaning everything has been sent to the output and the automaton can change to the next state.

`integer nextOutput_stateNo_seq()` - maps data to output ports. In particular, the function can look like e.g. `integer nextOutput_1_0()` meaning it defines mapping for state $1 and `seq` equal to 0 (i.e. this is the first time the function has been called). The function returns a number. The number says which port has been served by this function.

Global Next State Function

`integer nextState(integer state))` - calls individual `nextState()` functions according to the current state

Global Next Output Function

`integer nextOutput(integer state, integer seq)` - calls individual `nextOutput()` functions according to the current state and the value of `seq`.

DataGenerator

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

DataGenerator generates data.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
DataGenerator	generated	0	1-N	no	yes	yes	1)	yes	yes

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

Abstract

DataGenerator generates data according to some pattern instead of reading data from file, database, or any other data source. To generate data, a generate transformation may be defined. It uses a CTL template for **DataGenerator** or implements a `RecordGenerate` interface. Its methods are listed below. Component can send different records to different output ports using Return Values of Transformations (p. 238).

Icon

Ports

Port type	Number	Required	Description	Metadata
Output	0	yes	For generated data records	Any[1)]
	1-N	no	For generated data records	Output 0

Legend:

1): Metadata on all output ports can use Autofilling Functions (p. 117).

DataGenerator Attributes

Attribute	Req	Description	Possible values
Basic			
Generator	1)	Definition of records should be generated written in the graph in CTL or Java.	
Generator URL	1)	Name of external file, including path, containing the definition of the way how records should be generated written in CTL or Java.	
Generator class	1)	Name of external class defining the way how records should be generated.	
Number of records to generate	yes	Number of records to be generated.	
Deprecated			
Record pattern	2)	String consisting of all fields of generated records that are constant. It does not contain values of random or sequence fields. See Record Pattern (p. 300) for more information. User should define random and sequence fields first. See Random Fields (p. 301)and Sequence Fields (p. 300)for more information.	
Random fields	2)	Sequence of individual field ranges separated by semicolon. Individual ranges are defined by their minimum and maximum values. Minimum value is included in the range, maximum value is excluded from the range. Numeric data types represent numbers generated at random that are greater than or equal to the minimum value and less than the maximum value. If they are defined by the same value for both minimum and maximum, these fields will equal to such specified value. Fields of string and byte data type are defined by specifying their minimum and maximum length. See Random Fields (p. 301) for more information. Example of one individual field range: `$salary:=random("0","20000")`.	
Sequence fields	2)	Fields generated by sequence. They are defined as the sequence of individual field mappings (`$field:=IdOfTheSequence`) separated by semicolon. The same sequence ID can be repeated and used for more fields at the same time. See Sequence Fields (p. 300) for more information.	
Random seed	2)	Sets the seed of this random number generator using a single `long` seed. Assures that values of all fields remain stable on each graph run.	0-N

Legend:

1): One of these transformation attributes should be specified instead of the deprecated attributes marked by number 2. However, these new attributes are optional. Any of these transformation attributes must use a CTL template for **DataGenerator** or implement a `RecordGenerate` interface.

See CTL Scripting Specifics (p. 301) or Java Interfaces for DataGenerator (p. 304) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

2): These attributes are deprecated now. Define one of the transformation attributes marked by number 1 instead.

Advanced Description

DataGenerator Deprecated Attributes

If you do not define any of these three attributes, you must instead define the fields which should be generated at random (**Random fields**) and which by sequence (**Sequence fields**) and the others that are constant (**Record pattern**).

- **Record Pattern**

  Record pattern is a string containing all constant fields (all except random and sequential fields) of the generated records in the form of delimited (with delimiters defined in metadata on the output port) or fixed length (with sizes defined in metadata on the output port) record.

- **Sequence Fields**

  Sequence fields can be defined in the dialog that opens after clicking the **Sequence fields** attribute. The **Sequences** dialog looks like this:

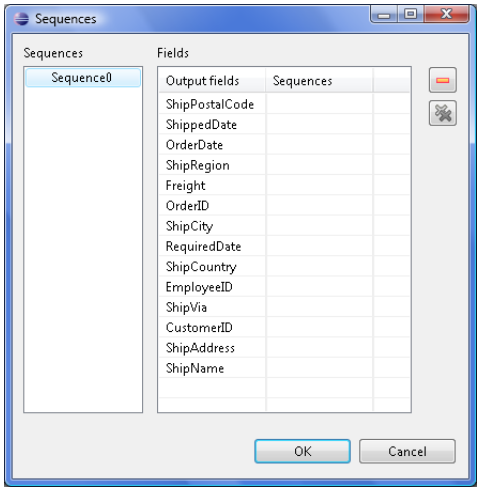


Figure 44.1. Sequences Dialog

This dialog consists of two panes. There are all of the graph sequences on the left and all clover fields (names of the fields in metadata) on the right. Choose the desired sequence on the left and drag and drop it to the right pane to the desired field.

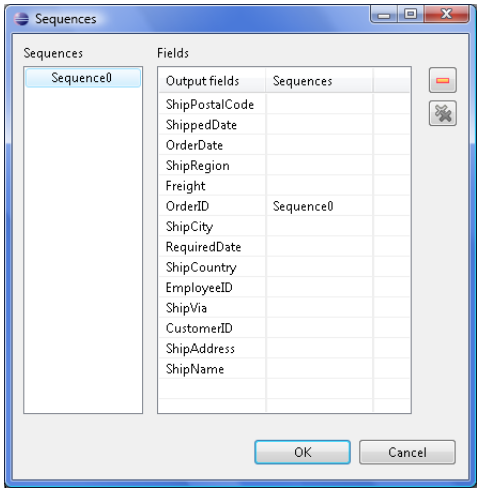


Figure 44.2. A Sequence Assigned

Remember that it is not necessary to assign the same sequence to different clover fields. But, of course, it is possible. It depends only on your decision. This dialog contains two buttons on its right side. For cancelling any selected assigned mapping or all assigned mappings.

- **Random Fields**

  This attribute defines the values of all fields whose values are generated at random. For each of the fields you can define its ranges. (Its minimum and maximum values.) These values are of the corresponding data types according to metadata. You can assign random fields in the **Edit key** dialog that opens after clicking the **Random fields** attribute.

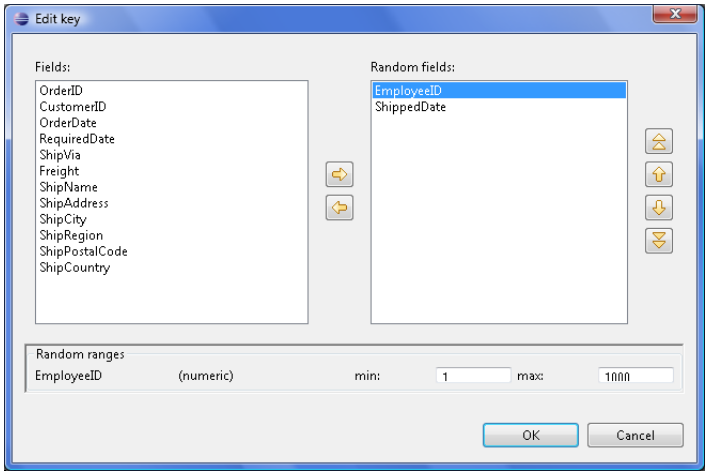


Figure 44.3. Edit Key Dialog

There are the **Fields** pane on the left, the **Random fields** on the right and the **Random ranges** pane at the bottom. In the last pane, you can specify the ranges of the selected random field. There you can type specific values. You can move fields between the **Fields** and **Random fields** panes as was described above - by clicking the **Left arrow** and **Right arrow** buttons.

CTL Scripting Specifics

When you define any of the three new, transformation attributes, you must specify a transformation that assigns values to output fields. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using the easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

DataGenerator uses the following transformation template:

CTL Templates for DataGenerator

This transformation template is used only in **DataGenerator**.

Once you have written your transformation in CTL, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

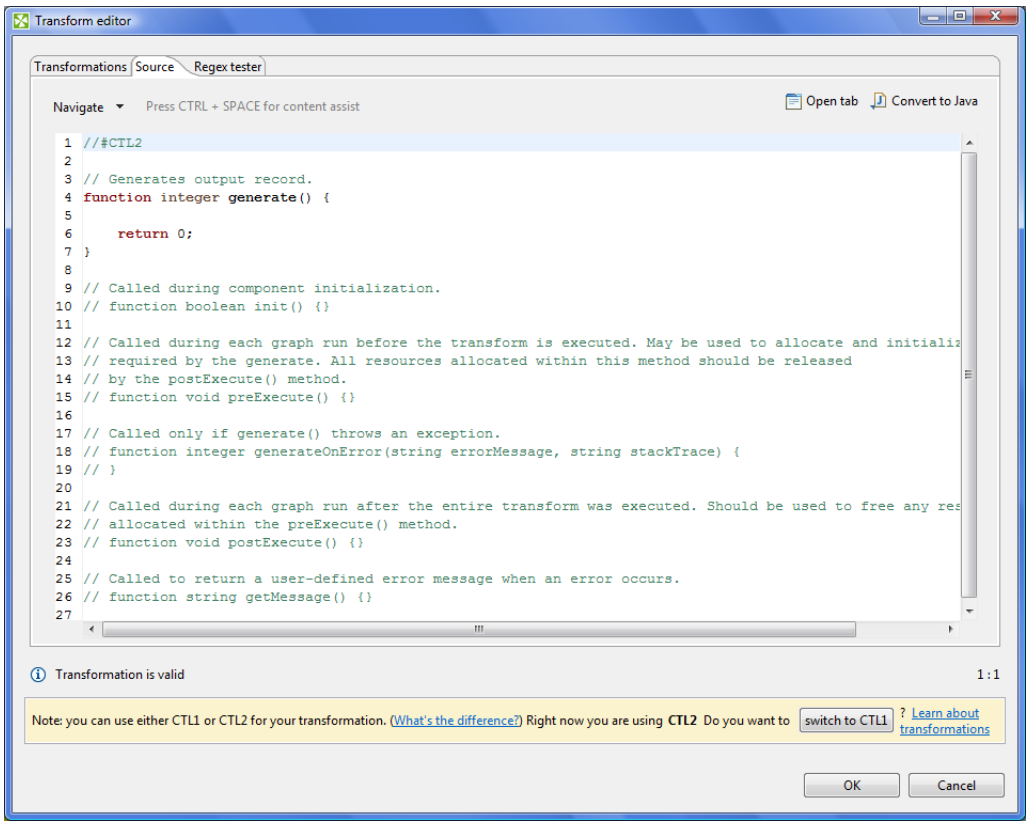


Figure 44.4. Source Tab of the Transform Editor in DataGenerator

Table 44.1. Functions in DataGenerator

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer generate()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly for each output record
Description	Defines the structure and values of all fields of output record. If any part of the `generate()` function for some output record causes fail of the `generate()` function, and if user has defined another function (`generateOnError()`), processing continues in this `generateOnError()` at the place where `generate()` failed. If `generate()` fails and user has not defined any `generateOnError()`, the whole graph will fail. The `generateOnError()` function gets the information gathered by `generate()` that was get from previously successfully processed code. Also error message and stack trace are passed to `generateOnError()`.

CTL Template Functions	

Example	`function integer generate() {`
	`  myTestString = iif(randomBool(),"1","abc");`
	`  $0.name = randomString(3,5) + " " randomString(5,7);`
	`  $0.salary = randomInteger(20000,40000);`
	`  $0.testValue = str2integer(myTestString);`
	`  return ALL;`
	`}`

integer generateOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `generate()` throws an exception.
Description	Defines the structure and values of all fields of output record. If any part of the `generate()` function for some output record causes fail of the `generate()` function, and if user has defined another function (`generateOnError()`), processing continues in this `generateOnError()` at the place where `generate()` failed. If `generate()` fails and user has not defined any `generateOnError()`, the whole graph will fail. The `generateOnError()` function gets the information gathered by `generate()` that was get from previously successfully processed code. Also error message and stack trace are passed to `generateOnError()`.

Example	`function integer generateOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.name = randomString(3,5) + " " randomString(5,7);`
	`  $0.salary = randomInteger(20000,40000);`
	`  $0.stringTestValue = "myTestString is abc";`
	`  return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user (called only when either `generate()` or `generateOnError()` returns value less than or equal to -2).
Invocation	Called in any time specified by user
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the generate. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No

CTL Template Functions	
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Output records or fields**

  Output records or fields are accessible within the `generate()` and `generateOnError()` functions only.

- All of the other CTL template functions do not allow to access outputs.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for DataGenerator

The transformation implements methods of the `RecordGenerate` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of `RecordGenerate` interface:

- `boolean init(Properties parameters, DataRecordMetadata[] targetMetadata)`

  Initializes generate class/function. This method is called only once at the beginning of generate process. Any object allocation/initialization should happen here.

- `int generate(DataRecord[] target)`

  Performs generator of target records. This method is called as one step in generate flow of records.

  ### Note

  This method allows to distribute different records to different connected output ports according to the value returned for them. See Return Values of Transformations (p. 238) for more information about return values and their meaning.

- `int generateOnError(Exception exception, DataRecord[] target)`

  Performs generator of target records. This method is called as one step in generate flow of records. Called only if `generate(DataRecord[])` throws an exception.

- `void signal(Object signalObject)`

  Method which can be used for signaling into generator that something outside happened.

- `Object getSemiResult()`

  Method which can be used for getting intermediate results out of generation. May or may not be implemented.

DBFDataReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

DBFDataReader reads data from fixed-length dbase files.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
DBFDataReader	dBase file	0-1	1-n	yes	no	no	no	no	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

Abstract

DBFDataReader reads data from fixed-length dbase files (local or remote). It can also read data from compressed files, console, input port, or dictionary.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any[1)]
	1-n	no	For correct data records	Output 0

Legend:

1): Metadata on output ports can use Autofilling Functions (p. 117).

DBFDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (dbase file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Charset		Encoding of records that are read.	IBM850 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient
Advanced			
Number of skipped records		Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Max number of records		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Number of skipped records per source		Number of records to be skipped from each source file. See Selecting Input Records (p. 258).	Same as in Metadata (default) | 0-N
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 258).	0-N
Incremental file	1)	Name of the file storing the incremental key, including path. See Incremental Reading (p. 258).	
Incremental key	1)	Variable storing the position of the last read record. See Incremental Reading (p. 258).	

Legend:

1) Either both or neither of these attributes must be specified.

DBInputTable

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

DBInputTable unloads data from database using JDBC driver.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
DBInputTable	database	0	1-n	✔	✖	✖	✖	✖	✖

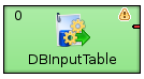
[1] Sending each data record to every connected output port

[2] Sending data records to output ports according to Return Values of Transformations (p. 238)

Abstract

DBInputTable unloads data from a database table using an SQL query or by specifying a database table and defining a mapping of database columns to Clover fields. It can send unloaded records to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	SQL queries	
Output	0	✔	for correct data records	equal metadata [1]
	1-n	✖	for correct data records	

[1] Output metadata can use Autofilling Functions (p. 117)

DBInputTable Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	✔	ID of the database connection to be used to access the database	
Query URL	[1]	Name of external file, including path, defining SQL query.	
SQL query	[1]	SQL query defined in the graph. See SQL Query Editor (p. 309) for detailed information.	
Query source charset		Encoding of external file defining SQL query.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient
Advanced			
Fetch size		Specifies the number of records that should be fetched from the database at once.	20 | 1-N
Incremental file	[2]	Name of the file storing the incremental key, including path. See Incremental Reading (p. 258).	
Incremental key	[2]	Variable storing the position of the last read record. See Incremental Reading (p. 258).	

[1] At least one of these attributes must be specified. If both are defined, only **Query URL** is applied.

[2] Either both or neither of these attributes must be specified.

Advanced Description

Defining Query Attributes

- **Query Statement without Mapping**

  If you do not want to map db fields to Clover fields, in case you have extracted metadata from one db table, you only need to type `select * from table [where dbfieldJ = ? and dbfieldK = somevalue]`.

  See SQL Query Editor (p. 309) for information about how **SQL query** can be defined.

- **Query Statement with Mapping**

  If you want to map database fields to clover fields even for multiple tables, the query will look like this:

```
select                                  $cloverfieldA:=table1.dbfieldP,
$cloverfieldC:=table1.dbfieldS,   ...   ,   $cloverfieldM:=table2.dbfieldU,
$cloverfieldM:=table3.dbfieldV   from   table1,   table2,   table3   [where
table1.dbfieldJ = ? and table2.dbfieldU = somevalue]
```

  See SQL Query Editor (p. 309) for information about how **SQL query** can be defined.

Dollar Sign in DB Table Name

- Remember that if any database table contains a dollar sign in its name, it will be transformed to double dollar signs in the generated query. Thus, each query must contain even numbers of dollar signs in the db table

(consisting of adjacent pairs of dollars). Single dollar signs contained in the name of db table are replaced by double dollar sign in the query in the name of the db table.

Important

Remember also, when connecting to MS SQL Server, it is recommended to use jTDS http://jtds.sourceforge.net driver. It is an open source 100% pure Java JDBC driver for Microsoft SQL Server and Sybase. It is faster than Microsoft's driver.

SQL Query Editor

For defining the **SQL query** attribute, **SQL query editor** can be used.

The editor opens after clicking the **SQL query** attribute row:

On the left side, there is the **Database schema** pane containing information about schemas, tables, columns, and data types of these columns.

Displayed schemas, tables, and columns can be filtered using the values in the **ALL** combo, the **Filter in view** textarea, the **Filter**, and **Reset** buttons, etc.

You can select any columns by expanding schemas, tables and clicking **Ctrl**+**Click** on desired columns.

Adjacent columns can also be selected by clicking **Shift**+**Click** on the first and the list item.

Then you need to click **Generate** after which a query will appear in the **Query** pane.

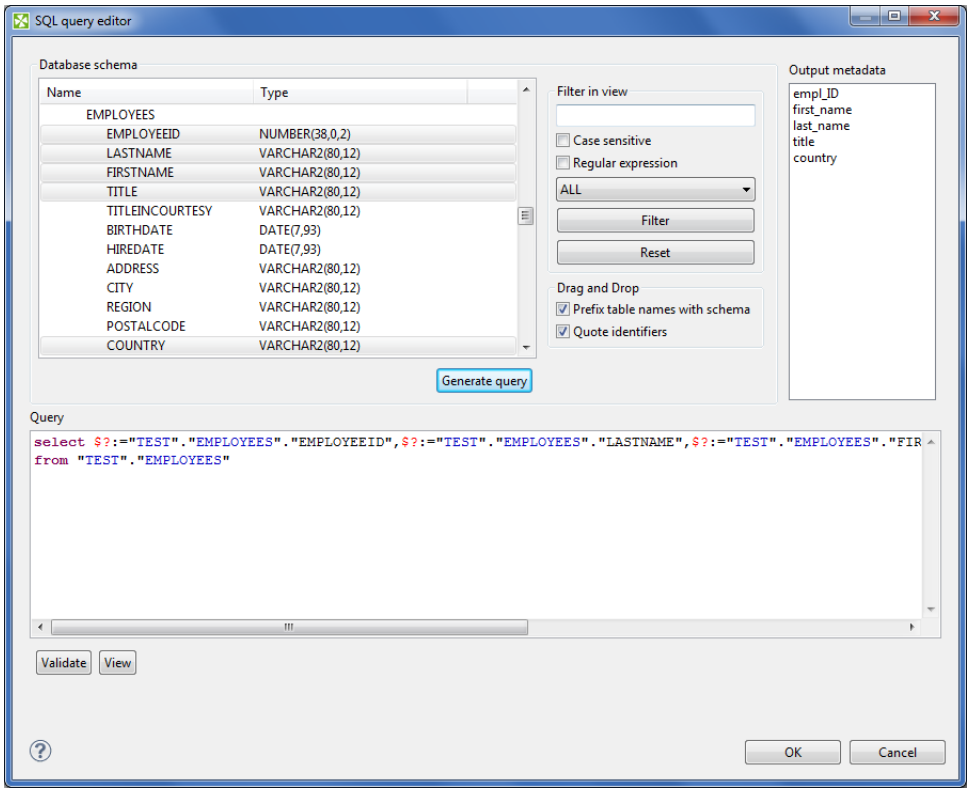


Figure 44.5. Generated Query with Question Marks

The query may contain question marks if any db columns differ from output metadata fields. Output metadata are visible in the **Output metadata** pane on the right side.

Drag and drop the fields from the **Output metadata** pane to the corresponding places in the **Query** pane and then manually remove the "$:=" characters. See following figure:

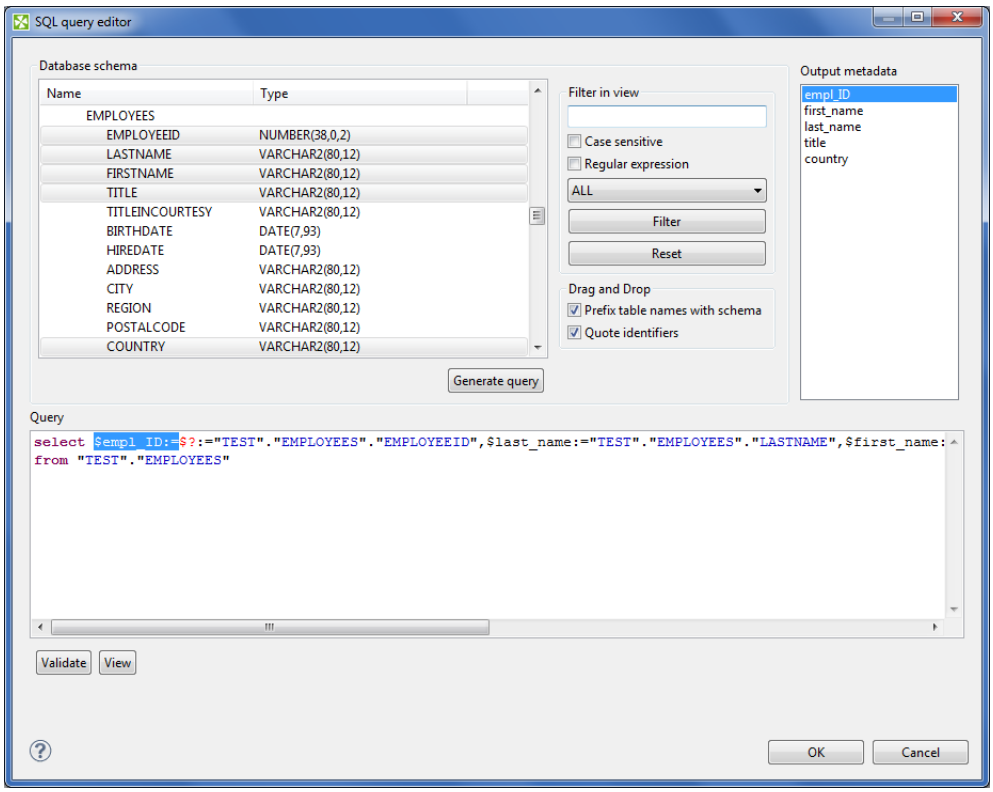


Figure 44.6. Generated Query with Output Fields

You can also type a `where` statement to the query.

Two buttons underneath allow you to validate the query (**Validate**) or view data in the table (**View**).

EmailReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251) .

Short Summary

EmailReader reads a store of email messages, either locally from a delimited flat file, or on an external server

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
EmailReader	✖	✖	1	2	-	-

Abstract

EmailReader is a component that enables the reading of online or local email messages.

This component parses email messages and writes their attributes out to two attached output ports. The first port, the content port, outputs relevant information about the email and body. The second port, the attachment port, writes information relevant to any attachments that the email contains.

The content port will write one record per email message. The attachment port can write multiple records per email message; one record for each attachment it encounters.

Icon

Ports

When looking at ports, it is necessary that use-case scenarios be understood. This component has the ability to read data from a local source, or an external server. The component decides which case to use based on whether there is an edge connected to the single input port.

Case One: If an edge is attached to the input port, the component assumes that it will be reading data locally. In many cases, this edge will come from a UniversalDataReader. In this case, a file can contain multiple email message bodies, separated by a chosen delimeter, and each message will be passed one by one into the emal reader for parsing and processing.

Case Two: If an edge is not connected the the input port, the component assumes that messages will be read from an external server. In this case, the user *must* enter related attributes, such as the server host and protocol parameters, as well as any relevant username and/or password.

Port type	Number	Required	Description	Metadata
Input	0	✖	For inputting email messages from a flat file	String field
Output	0	✖	The content port	Any
	1	✖	The attachment port	Any

EmailReader Attributes

Whether many of the attributes are required or not depends solely on the configuration of the Component. See the ports section. In Case two, where an edge is not connected to the input port, many attributes are required in order to connect to the external server. The user at minimum must choose a protocol and enter a hostname for the server. Usually a username and password will also be required.

Attribute	Req	Description	Possible values
Basic			
Server Type		Protocol utilized to connect to a mail server. Options are POP3 and IMAP. In most cases, IMAP should be selected if possible, as it is an improvement over POP3.	POP3, IMAP
Server Name		The hostname of the server.	E.G. imap.google.com
User Name		Username to connect to the server (if authorization is required)	
Password		Password to connect to server (if authorization is required)	
Fetch Messages		Filters messages based on their status. The option "ALL" will read every message located on the server, regardless of its status. "NEW" fetches only messages that have not been read.	NEW,ALL
Advanced			
Server Port		Specifies the port used to connect to an external server. If left blank, a default port will be used.	Integers
Security		Specifies the security protocol used to connect to the server.	NONE,SSL,STARTTLS, SSL +STARTTLS
Temp File URL		Specifies a directory for temporary storage of any files found inside of attachments. These filenames may be attained from the output "attachment" port's filename attribute. The default directory is the current CloverETL project's temporary directory, denoted ${DATATMP_DIR}	

Attribute	Req	Description	Possible values
POP3 Cache File		Specifies the URL of a file used to keep track of which messages have been read. pop3 servers by default have no way of keeping track of read/unread messages. If one wishes to fetch only unread messages, they must download all of the messages-ids from the server, and then compare them with a list of message-ids that have already been read. Using this method, only the messages that do not appear in this list are actually downloaded, thus saving bandwidth. This file is simply a delimited text file, storing the unique message-ids of messages that have already been read. Even if "ALL" messages is chosen, the user should still provide a cache file, as it will be populated by the messages read. Note: the pop cache file is universal; it can be shared amongst many inboxes, or the user can choose to maintain a separate cache for different mailboxes.	

Tips&Tricks

- Be sure you have dedicated enough memory to your Java Virtual Machine (JVM). Depending on the size of your message attachments (if you choose to read them), you may need to allocate up to 512M to CloverETL so that it may effectively process the data.

Performance Bottlenecks

- *Quantity of messages to process from an external server* **EmailReader** must connect to an external server, therefore one may reach bandwidth limitations. Processing a large amount of messages which contain large attachments may bottleneck the application, waiting for the content to be downloaded. Use the "NEW" option whenever possible, and maintain a POP3 cache if using the POP3 protocol.

JMSReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

JMSReader converts JMS messages into Clover data records.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
JMSReader	jms messages	0	1	yes	no	yes	no	yes	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

Abstract

JMSReader receives JMS messages, converts them into Clover data records and sends these records to the connected output port. Component uses a processor transformation which implements a `JmsMsg2DataRecord` interface or inherits from a `JmsMsg2DataRecordBase` superclass. Methods of `JmsMsg2DataRecord` interface are described below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Output	0	yes	For correct data records	Any[1)]

Legend:

1): Metadata on the output port may contain a field specified in the **Message body field** attribute. Metadata can also use Autofilling Functions (p. 117).

JMSReader Attributes

Attribute	Req	Description	Possible values
Basic			
JMS connection	yes	ID of the JMS connection to be used.	
Processor code	1)	Transformation of JMS messages to records written in the graph in Java.	
Processor URL	1)	Name of external file, including path, containing the transformation of JMS messages to records written in Java.	
Processor class	1)	Name of external class defining the transformation of JMS messages to records. The default processor value is sufficient for most cases. It can process both `javax.jms.TextMessage` and `javax.jms.BytesMessage`.	JmsMsg2DataRecordProperties (default) | other class
JMS message selector		Standard JMX "query" used to filter the JMS messages that should be processed. In effect, it is a string query using message properties and syntax that is a subset of SQL expressions. See http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html for more information.	
Processor source charset		Encoding of external file containing the transformation in Java.	ISO-8859-1 (default) | other encoding
Message charset		Encoding of JMS messages contents. This attribute is also used by the default processor implementation (`JmsMsg2DataRecordProperties`). And it is used for `javax.jms.BytesMessage` only.	ISO-8859-1 (default) | other encoding
Advanced			
Max msg count		Maximum number of messages to be received. 0 means without limitation. See Limit of Run (p. 316) for more information.	0 (default) | 1-N
Timeout		Maximum time to receive messages in milliseconds. 0 means without limitation. See Limit of Run (p. 316) for more information.	0 (default) | 1-N
Message body field		Name of the field to which message body should be written. This attribute is used by the default processor implementation (`JmsMsg2DataRecordProperties`). If no **Message body field** is specified, the field whose name is `bodyField` will be filled with the body of the message. If no field for the body of the message is contained in metadata, the body will not be written to any field.	bodyField (default) | other name

Legend:

1) One of these may be set. Any of these transformation attributes implements a `JmsMsg2DataRecord` interface.

See Java Interfaces for JMSReader (p. 316) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

Limit of Run

It is also important to decide whether you want to limit the number of received messages and/or time of processing. This can be done by using the following setting:

- **Limited Run**

  If you specify the maximum number of messages (**Max msg count**), the timeout (**Timeout**) or both, the processing will be limited by number of messages, or time of processing, or both of these attributes. They need to be set to positive values.

  When the specified number of messages is received, or when the process lasts some defined time, the process stops. Whichever of them will be achieved first, such attribute will be applied.

  #### Note

  Remember that you can also limit the graph run by using the `endOfInput()` method of `JmsMsg2DataReader` interface. It returns a boolean value and can also limit the run of the graph. Whenever it returns `false`, the processing stops.

- **Unlimited Run**

  On the other hand, if you do not specify either of these two attributes, the processing will never stop. Each of them is set to 0 by default. Thus, the processing is limited by neither the number of messages nor the elapsed time. This is the default setting of **JMSReader**.

Java Interfaces for JMSReader

The transformation implements methods of the `JmsMsg2DataRecord` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of `JmsMsg2DataRecord` interface:

- `void init(DataRecordMetadata metadata, Properties props)`

  Initializes the processor.

- `boolean endOfInput()`

  May be used to end processing of input JMS messages when it returns `false`. See Limit of Run (p. 316) for more information.

- `DataRecord extractRecord(Message msg)`

  Transforms JMS message to data record. `null` indicates that the message is not accepted by the processor.

- `String getErrorMsg()`

  Returns error message.

LDAPReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

LDAPReader reads information from an LDAP directory.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
LDAPReader	LDAP directory tree	0	1-n	no	no	no	no	no	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

Abstract

LDAPReader reads information from an LDAP directory and converting it to Clover data records. It provides the logic for extracting the results of a search and converts them into Clover data records. The results of the search must have the same `objectClass`.

Only string and byte Clover data fields are supported. String is compatible with most of ldap usual types, byte is necessary, for example, for `userPassword` ldap type reading.

Icon

Ports

Port type	Number	Required	Description	Metadata
Output	0	yes	For correct data records	Any[1)]
	1-n	no	For correct data records	Output 0

Legend:

1): Metadata on the output must precisely describe the structure of the read object. Metadata can use Autofilling Functions (p. 117).

LDAPReader Attributes

Attribute	Req	Description	Possible values
Basic			
LDAP URL	yes	LDAP URL of the directory.	`ldap://host:port/`
Base DN	yes	Base *Distinguished Name* (the root of your LDAP tree). It is a comma separated list of *attribute=value* pairs reffering to any location with the directory, e.g., if `ou=Humans,dc=example,dc=com` is the root of the subtree to be search, entries representing people from *example.com* domain are to be found.	
Filter	yes	*attribute=value* pairs as a filtering condition for the search. All entries matching the filter will be returned, e.g., `mail=*` returns every entry which has an email address, while `objectclass=*` is the standard method for returning all entries matching a given *base* and scope because all entries have values for *objectclass*.	
Scope		Scope of the search request. By default, only one `object` is searched. If `onelevel`, the level immediately below the distinguished name, if `subtree`, the whole subtree below the distinguished name is searched.	object (default) | onelevel | subtree
User		User DN to be used when connecting to the LDAP directory. Similar to the following: `cn=john.smith,dc=example,dc=com`	
Password		Password to be used when connecting to the LDAP directory.	
Advanced			
Multi-value separator		**LDAPReader** can handle keys with multiple values. These are delimited by this string or character. <none> is special escape value which turns off this functionality, then only the first value is read. This attribute can only be used for string data type. When byte type is used, the first value is the only one that is read.	"|" (default) | other character or string
Alias handling		to control how aliases (leaf entries pointing to another object in the namespace) are dereferenced	always | never | finding (default)| searching
Referral handling		By default, links to other servers are ignored. If `follow`, the referrals are processed.	ignore (default) | follow

Advanced Description

- **Alias Handling**

Searching the entry an alias entry points to is known as *dereferencing* an alias. Setting the **Alias handling** attribute, you can control the extent to which entries are searched:

- `always`: Always dereference aliases.
- `never`: Never dereference aliases.
- `finding`: Dereference aliases in locating the base of the search but not in searching subordinates of the base.
- `searching`: Dereference aliases in searching subordinates of the base but not in locating the base

Tips & Tricks

- *Improving search performance:* If there are no alias entries in the LDAP directory that require dereferencing, choose **Alias handling** `never` option.

LotusReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Chapter 44, Readers (p. 287).

Short Summary

LotusReader reads data from a **Lotus Domino server**. Data is read from **Views**, where each view entry is read as a single data record.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
LotusReader	Lotus Domino	0	1	✖	✖	✖	✖	✖	✖

[1] Sending each data record to every connected output port

[2] Sending data records to output ports according to Return Values of Transformations (p. 238)

Abstract

LotusReader is a component which can read data records from Lotus databases. The reading is done by connection to a database stored on a **Lotus Domino server**.

The data is read from what is in Lotus called a **View**. Views provide tabular structure to the data in Lotus databases. Each row of a view is read by the **LotusReader** component as a single data record.

The user of this component needs to provide the Java library for connecting to Lotus. The libary can be found in the installations of Lotus Notes and Lotus Domino. The **LotusReader** component is not able to communicate with Lotus unless the path to this library is provided or the library is placed on the user's classpath. The path to the library can be specified in the details of Lotus connection (see Chapter 23, Lotus Connections (p. 165)).

Icon

Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for read data records	

LotusReader Attributes

Attribute	Req	Description	Possible values
Basic			
Domino connection	✔	ID of the connection to the Lotus Domino database.	
View	✔	The name of the View in Lotus database from which the data records will be read.	
Advanced			
Multi-value read mode	✖	Reading strategy that will be used for reading Lotus multi-value fields. Either only the first field of the multi-value will be read or all values will be read and then separated by user-specified separator.	Read all values (default) | Read first value only
Multi-value separator	✖	A string that will be used to separate values from multi-value Lotus fields.	";" (default) | "," | ":" | "|" | "\t" | other character or string

MultiLevelReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

MultiLevelReader reads data from flat files with a heterogeneous structure.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
MultiLevelReader	flat file	1	1-n	no	yes	yes	yes	yes	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

Abstract

MultiLevelReader reads information from flat files with a heterogeneous and complicated structure (local or remote which are delimited, fixed-length, or mixed). It can also read data from compressed flat files, console, input port, or dictionary.

Unlike **UniversalDataReader** or the two deprecated readers (**DelimitedDataReader** and **FixLenDataReader**), **MultiLevelReader** can read data from flat files whose structure contains different structures including both delimited and fixed length data records even with different numbers of fields and different data types. It can separate different types of data records and send them through different connected output ports. Input files can also contain non-record data.

Component also uses the **Data policy** option. See Data Policy (p. 260) for more detailed information.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any(Out0)[1)]
	1-N	no	For correct data records	Any(Out1-OutN)[1)]

Legend:

1): Metadata on all output ports can use Autofilling Functions (p. 117).

MultiLevelReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (flat file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Charset		Encoding of records that are read.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Lenient
Selector code	1)	Transformation of rows of input data file to data records written in the graph in Java.	
Selector URL	1)	Name of external file, including path, defining the transformation of rows of input data file to data records written in Java.	
Selector class	1)	Name of external class defining the transformation of rows of input data file to data records.	PrefixMultiLevelSelector (default) | other class
Selector properties		List of the `key=value` expressions separated by semicolon when the whole is surrounded by flower brackets. Each value is the number of the port through which data records should be sent out. Each key is a serie of characters from the beginning of the row contained in the flat file that enable differentiate groups of records.	
Advanced			
Number of skipped records		Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Max number of records		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Number of skipped records per source		Number of records to be skipped from each source file. See Selecting Input Records (p. 258).	Same as in Metadata (default) | 0-N
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 258).	0-N

Legend:

1): If you do not define any of these three attributes, the default **Selector class** (`PrefixMultiLevelSelector`) will be used.

`PrefixMultiLevelSelector` class implements `MultiLevelSelector` interface. The interface methods can be found below.

See Java Interfaces for MultiLevelReader (p. 324) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

Selector Properties

You also need to set some series of parameters that should be used (**Selector properties**). They map individual types of data records to output ports. All of the properties must have the form of a list of the `key=value` expressions separated by semicolon. The whole sequence is in curly brackets. To specify these **Selector properties**, you can use the dialog that opens after clicking the button in this attribute row. By clicking the **Plus** button in this dialog, you can add new key-value pairs. Then you only need to change both the default name and the default value. Each value must be the number of the port through which data records should be sent out. Each key is a series of characters from the beginning of the row contained in the flat file that enable differentiate groups of records.

Java Interfaces for MultiLevelReader

Following are the methods of the `MultiLevelSelector` interface:

- `int choose(CharBuffer data, DataRecord[] lastParsedRecords)`

  A method that peeks into `CharBuffer` and reads characters until it can either determine metadata of the record which it reads, and thus return an index to metadata pool specified in `init()` method, or runs out of data returning `MultiLevelSelector.MORE_DATA`.

- `void finished()`

  Called at the end of selector processing after all input data records were processed.

- `void init(DataRecordMetadata[] metadata, Properties properties)`

  Initializes this selector.

- `int lookAheadCharacters()`

  Returns the number of characters needed to decide (next) record type. Usually it can be any fixed number of characters, but dynamic lookahead size, depending on previous record type, is supported and encouraged whenever possible.

- `int nextRecordOffset()`

  Each call to `choose()` can instrument the parent to skip certain number of characters before attempting to parse a record according to metadata returned in `choose()` method.

- `void postProcess(int metadataIndex, DataRecord[] records)`

  In this method the selector can modify the parsed record before it is sent to corresponding output port.

- `int recoverToNextRecord(CharBuffer data)`

  This method instruments the selector to find the offset of next record which is possibly parseable.

- `void reset()`

  Resets this selector completely. This method is called once, before each run of the graph.

- `void resetRecord()`

  Resets the internal state of the selector (if any). This method is called each time a new choice needs to be made.

ParallelReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

ParallelReader reads data from flat files using multiple threads.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
ParallelReader	flat file	0	1-2	✖	✖	✖	✖	✖	✖

[1] Component sends each data record to all connected output ports.

[2] Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

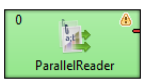
Abstract

ParallelReader reads delimited flat files like CSV, tab delimited, etc., fixed-length, or mixed text files. Reading goes in several parallel threads, which improves the reading speed. Input file is divided into set of chunks and each reading thread parses just records from this part of file. The component can read a single file as well as a collection of files placed on a local disk or remotely. Remote files are accessible via FTP protocol.

According to the component settings and the data structure, either the fast simplistic parser (`SimpleDataParser`) or the robust (`CharByteDataParser`) one is used.

Parsed data records are sent to the first output port. The component has an optional output logging port for getting detailed information about incorrect records. Only if Data Policy (p. 260) is set to `controlled` and a proper **Writer** (**Trash** or **UniversalDataWriter**) is connected to port 1, all incorrect records together with the information about the incorrect value, its location and the error message are sent out through this error port.

Icon



Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for correct data records	any [1]
	1	✘	for incorrect data records	specific structure, see table bellow

[1] Metadata on output port can use Autofilling Functions (p. 117)

Table 44.2. Error Metadata for Parallel Reader

Field Number	Field Content	Data Type	Description
0	record number	integer	position of the erroneous record in the dataset (record numbering starts at 1)
1	field number	integer	position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)
2	raw record	string	erroneous record in raw form (including delimiters)
3	error message	string	error message - detailed information about this error
4	first record offset	long	indicates the initial file offset of the parsing thread

ParallelReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	Attribute specifying what data source(s) will be read. See Supported File URL Formats for Readers (p. 251).	
Charset		Encoding of records that are read in.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient
Trim strings		specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 335) If `true`, the use of the robust parser is forced.	false (default) | true
Quoted strings		Fields that contain a special character (comma, newline, or double quote), must be enclosed in quotes (only single/double quote as a quote character is accepted). If `true`, such special characters inside the quoted string are not treated as delimiters and the quotes are removed.	false (default) | true
Advanced			
Skip leading blanks		specifies whether to skip leading whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 335) If `true`, the use of the robust parser is enforced.	false (default) | true

Attribute	Req	Description	Possible values
Skip trailing blanks		specifies whether to skip trailing whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 335) If `true`, the use of the robust parser is enforced.	false (default) | true
Max error count		maximum number of tolerated error records in input file(s); applicable only if `Controlled` **Data Policy** is set	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter char, it will be interpreted as a single delimiter when setting to `true`.	false (default) | true
Verbose		By default, less comprehensive error notification is provided and the performance is slightly higher. However, if switched to `true`, more detailed information with less performance is provided.	false (default) | true
Level of parallelism		Number of threads used to read input data files. The order of records is not preserved if it is 2 or higher. If the file is too small, this value will be switched to 1 automatically.	2 (default) | 1-n
Distributed file segment reading		In case a graph is running in a **CloverETL Server** environment, component can only process the appropriate part of the file. The whole file is divided into segments by **CloverETL Server** and each cluster worker processes only one proper part of file. By default, this option is turned off.	false (default) | true
Parser		By default, the most appropriate parser is applied. Besides, the parser for processing data may be set explicitly. If an improper one is set, an exception is thrown and the graph fails. See Data Parsers (p. 336)	auto (default) | *<other>*

Advanced Description

- **Quoted strings**

The attribute considerably changes the way your data is parsed. If it is set to `true`, all field delimiters inside quoted strings will be ignored (after the first **Quote character** is actually read). Quote characters will be removed from the field.

Example input:

```
1;"lastname;firstname";gender
```

Output with Quoted strings == true:

{1}, {`lastname;firstname`}, {`gender`}

Output with Quoted strings == false:

{1}, {`"lastname`}, {`firstname";gender`}

QuickBaseRecordReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

QuickBaseRecordReader reads data from **QuickBase** online database.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
QuickBaseRecordReader	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖

[1] Sending each data record to every connected output port

[2] Sending data records to output ports according to Return Values of Transformations (p. 238)

Abstract

QuickBaseRecordReader reads data from the **QuickBase** online database (http://quickbase.intuit.com). Records, the IDs of which are specified in the **Records list** component attribute, are read first. Records with IDs specified in input are read afterward.

The read records are sent through the connected first output port. If the record is erroneous (not present in the database table, e.g.) it can can be sent out through the optional second port if it is connected.

This component wrapps the `API_GetRecordInfo` HTTP interaction (http://www.quickbase.com/api-guide/getrecordinfo.html).

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	for getting application table record IDs to be read	first field: `integer | long`
Output	0	✔	for correct data records	data types and positions of fields must fit the table field types [1]
	1	✖	information about rejected records	Error Metadata for QuickBaseRecordReader (p. 330)[2]

[1] Only `source_row_count` autofilling function returning the record ID can be used.

[2] Error metadata cannot use Autofilling Functions (p. 117).

Table 44.3. Error Metadata for QuickBaseRecordReader

Field number	Field name	Data type	Description
0	<any_name1>	integer | long	ID of the erroneous record
1	<any_name2>	integer | long	error code
2	<any_name3>	string	error message

QuickBaseRecordReader Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see Chapter 22, QuickBase Connections (p. 164)	
Table ID	✔	ID of the table in the QuickBase application data records are to be read from (see the `application_stats` for getting the table ID)	
Records list		List of record IDs (separated by the semicolon) to be read from the specified database table. These records are read first, before the records specified in the input data.	

QuickBaseQueryReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

QuickBaseQueryReader gets records fullfilling given conditions from the **QuickBase** online database table.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
QuickBaseQueryReader	QuickBase	0-1	1-2	✖	✖	✖	✖	✖	✖

[1] Sending each data record to every connected output port

[2] Sending data records to output ports according to Return Values of Transformations (p. 238)

Abstract

QuickBaseQueryReader gets records from the **QuickBase** online database (http://quickbase.intuit.com). You can use the component attributes to define which columns will be returned, how many records will be returned and how they will be sorted, and whether the QuickBase should return structured data. Records that meet the requirements are sent out through the connected output port.

This component wrapps the `API_DoQuery` HTTP interaction (http://www.quickbase.com/api-guide/do_query.html).

Icon

Ports

Port type	Number	Required	Description	Metadata
Output	0	✔	for correct data records	any [1]

[1] Metadata cannot use Autofilling Functions (p. 117)

QuickBaseQueryReader Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see Chapter 22, QuickBase Connections (p. 164)	
Table ID	✔	ID of the table in the QuickBase application data records are to be get from (see the `application_stats` for getting the table ID)	
Query		Determines which records are returned (all, by default) using the form {*<field_id>.<operator>.'<matching_value>'*}	
CList		The *column list* specifies which columns will be included in each returned record and how they are ordered in the returned record aggregate. Use *field_id*s separated by a period.	
SList		The *sort list* determines the order in which the returned records are displayed. Use *field_id* separated by a period.	
Options		Options used for data records that are read. See Options (p. 332) for more information.	

Advanced Description

Options

Options attributes can be as follows:

- `skp-n`

  Specifies `n` records from the beginning that should be skipped.

- `num-n`

  Specifies `n` records that should be read.

- `sortorder-A`

  Specifies the order of sorting as ascending.

- `sortorder-D`

  Specifes the order of sorting as descending.

- `onlynew`

  This parameter cannot be used by anonymous user. The component reads only new records. The results can be different for different users.

UniversalDataReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

UniversalDataReader reads data from flat files.

Component	Data source	Input ports	Output ports	Each to all outputs [1]	Different to different outputs [2]	Transformation	Transf. req.	Java	CTL
UniversalDataReader	flat file	0-1	1-2	✖	✖	✖	✖	✖	✖

[1] Sending each data record to every connected output port

[2] Sending data records to output ports according to Return Values of Transformations (p. 238)

Abstract

UniversalDataReader reads data from flat files such as CSV (comma-separated values) file and delimited, fixed-length, or mixed text files. The component can read a single file as well as a collection of files placed on a local disk or remotely. Remote files are accessible via HTTP, HTTPS, FTP, or SFTP protocols. Using this component, ZIP and TAR archives of flat files can be read. Also reading data from stdin (console), input port, or dictionary is supported.

Parsed data records are sent to the first output port. The component has an optional output logging port for getting detailed information about incorrect records. Only if Data Policy (p. 260) is set to `controlled` and a proper **Writer** (**Trash** or **UniversalDataWriter**) is connected to port 1, all incorrect records together with the information about the incorrect value, its location and the error message are sent out through this error port.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✖	for Input Port Reading (p. 257)	include specific `byte`/ `cbyte`/ `string` field
Output	0	✔	for correct data records	any [1]
	1	✖	for incorrect data records	specific structure, see table below

[1] Metadata on output port 0 can use Autofilling Functions (p. 117)

The optional logging port for incorrect records has to define the following metadata structure - the record contains exactly four fields (named arbitrarily) of given types in the following order:

Table 44.4. Error Metadata for UniversalDataReader

Field number	Field name	Data type	Description
0	recordID	integer	position of the erroneous record in the dataset (record numbering starts at 1)
1	fieldID	integer	position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)
2	data	string | byte | cbyte	erroneous record in raw form (including delimiters)
3	error	string | byte | cbyte	error message - detailed information about this error

UniversalDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	path to data source (flat file, console, input port, dictionary) to be read specified, see Supported File URL Formats for Readers (p. 251).	
Charset		character encoding of input records (character encoding does not apply on byte fields if the record type is `fixed`)	ISO-8859-1 (default) | <other encodings>
Data policy		specifies how to handle misformatted or incorrect data, see Data Policy (p. 260)	strict (default) | controlled | lenient
Trim strings		specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 335) below	default | true | false
Quoted strings		Fields that contain a special character (comma, newline, or double quote), must be enclosed in quotes (only single/double quote as a quote character is accepted). If `true`, such special characters inside the quoted string are not treated as delimiters and the quotes are removed.	false (default) | true
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**.	both (default) | " | '
Advanced			
Skip leading blanks		specifies whether to skip leading whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 335).	default | true | false
Skip trailing blanks		specifies whether to skip trailing whitespace (blanks e.g.) before setting input strings to data fields. If not explicitly set (i.e., having the default value), the value of **Trim strings** attribute is used. See Trimming Data (p. 335).	default | true | false

Attribute	Req	Description	Possible values
Number of skipped records		how many records/rows to be skipped from the source file(s); see Selecting Input Records (p. 258).	0 (default) - N
Max number of records		how many records to be read from the source file(s) in turn; all records are read by default; See Selecting Input Records (p. 258).	1 - N
Number of skipped records per source		how many records/rows to be skipped from each source file. By default, the value of **Skip source rows** record property in output port 0 metadata is used. In case the value in metadata differs from the value of this attribute, the **Number of skipped records per source** value is applied, having a higher priority. See Selecting Input Records (p. 258).	0 (default)- N
Max number of records per source		how many records/rows to be read from each source file; all records from each file are read by default; See Selecting Input Records (p. 258).	1 - N
Max error count		maximum number of tolerated error records in input file(s); applicable only if `Controlled` **Data Policy** is set	0 (default) - N
Treat multiple delimiters as one		If a field is delimited by a multiplied delimiter char, it will be interpreted as a single delimiter when setting to `true`.	false (default) | true
Incremental file	[1]	Name of the file storing the incremental key, including path. See Incremental Reading (p. 258).	
Incremental key	[1]	Variable storing the position of the last read record. See Incremental Reading (p. 258).	
Verbose		By default, less comprehensive error notification is provided and the performance is slightly higher. However, if switched to `true`, more detailed information with less performance is provided.	false (default) | true
Parser		By default, the most appropriate parser is applied. Besides, the parser for processing data may be set explicitly. If an improper one is set, an exception is thrown and the graph fails. See Data Parsers (p. 336)	auto (default) | *<other>*
Deprecated			
Skip first line		By default, the first line is not skipped, if switched to `true` (if it contains a header), the first line is skipped.	false (default) | true

[1] Either both or neither of these attributes must be specified

Advanced Description

- **Trimming Data**

1. Input strings are implicitly (i.e., the **Trim strings** attribute kept at the `default` value) processed before converting to value according to the field data type as follows:
   - Whitespace is removed from both the start and the end in case of `boolean`, `date`, `decimal`, `integer`, `long`, or `number`.
   - Input string is set to a field including leading and trailing whitespace in case of `byte`, `cbyte`, or `string`.
2. If the **Trim strings** attribute is set to `true`, all leading and trailing whitespace characters are removed. A field composed of only whitespaces is transformed to null (zero length string). The `false` value implies preserving all leading and trailing whitespace characters. Remember that input string representing a numerical data type or boolean can not be parsed including whitespace. Thus, use the `false` value carefully.

3. Both the **Skip leading blanks** and **Skip trailing blanks** attributes have higher priority than **Trim strings**. So, the input strings trimming will be determined by the `true` or `false` values of these attributes, regardless the **Trim strings** value.

- **Data Parsers**

1. `org.jetel.data.parser.SimpleDataParser` - is a very simple but fast parser with limited validation, error handling, and functionality. The following attributes are not supported:

   - **Trim strings**
   - **Skip leading blanks**
   - **Skip trailling blanks**
   - **Incremental reading**
   - **Number of skipped records**
   - **Max number of records**
   - **Quoted strings**
   - **Treat multiple delimiters as one**
   - **Skip rows**
   - **Verbose**

   On top of that, you cannot use metadata containing at least one field with one of these attributes:

   - the field is fixed-length
   - the field has no delimiter or, on the other hand, more of them
   - **Shift** is not null (see Details Pane (p. 143))
   - **Autofilling** set to `true`
   - the field is byte-based

2. `org.jetel.data.parser.DataParser` - an all-round parser working with any reader settings
3. `org.jetel.data.parser.CharByteDataParser` - can be used whenever metadata contain byte-based fields mixed with char-based ones. A byte-based field is a field of one of these types: `byte, cbyte` or any other field whose `format` property starts with the "`BINARY:`" prefix. See Binary Formats (p. 109).
4. `org.jetel.data.parser.FixLenByteDataParser` - used for metadata with byte-based fields only. It parses sequences of records consisting of a fixed number of bytes.

Note

Choosing `org.jetel.data.parser.SimpleDataParser` while using **Quoted strings** will cause the **Quoted strings** attribute to be ignored.

Tips & Tricks

- *Handling records with large data fields:* **UniversalDataReader** can process input strings of even hundreds or thousands of characters when you adjust the field and record buffer sizes. Just increase the following properties according to your needs: Record.MAX_RECORD_SIZE for record serialization,

DataParser.FIELD_BUFFER_LENGTH for parsing, and DataFormatter.FIELD_BUFFER_LENGTH for formatting. Finally, don't forget to increase the DEFAULT_INTERNAL_IO_BUFFER_SIZE variable to be at least 2*MAX_RECORD_SIZE. Go to Changing Default CloverETL Settings (p. 76) to get know how to change these property variables.

General examples

- *Processing files with headers:*If the first rows of your input file do not represent real data but field labels instead, set the **Number of skipped records** attribute. If a collection of input files with headers is read, set the **Number of skipped records per source**
- *Handling typist's error when creating the input file manually:*If you wish to ignore accidental errors in delimiters (such as two semicolons instead of a single one as defined in metadata when the input file is typed manually), set the **Treat multiple delimiters as one** attribute to `true`. All redundant delimiter chars will be replaced by the proper one.

XLSDataReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

XLSDataReader reads data from XLS or XLSX files.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
XLSDataReader	XLS(X) file	0-1	1-n	yes	no	no	no	no	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation. See Return Values of Transformations (p. 238) for more information.

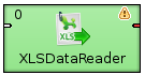
Abstract

XLSDataReader reads data from the specified sheet(s) of XLS or XLSX files (local or remote). It can also read data from compressed files, console, input port, or dictionary.

Note

Remember that **XLSDataReader** stores all data in memory and has high memory requirements.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any[1)]
	1-n	no	For correct data records	Output 0

Legend:

1): Metadata can use Autofilling Functions (p. 117).

XLSDataReader Attributes

Attribute	Req	Description	Possible values
Basic			
Type of parser		Specifies the parser to be used. By default, component guesses according the extension (XLS or XLSX).	Auto (default) | XLS | XLSX
File URL	yes	Attribute specifying what data source(s) will be read (input file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Sheet name	1)	Name of the sheet to be read. Wild cards `?` and `*` can be used in the name.	
Sheet number	1)	Numbers of the sheet to be read. Numbering starts from 0. Sequence of numbers separated by comma and/or got together with a hyphen. Following patterns can be used: `number`, `minNumber-maxNumber`, `*-maxNumber`, `minNumber-*`. Example: `*-5,9-11,17-*`.	
Charset		Encoding of records that are read.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient
Metadata row		Number of the row containing the names of the columns. By default, the header of the sheet is used as metadata row. See Mapping and Metadata (p. 340) for more information.	0 (default) | 1-N
Field mapping		Mapping of XLS fields to Clover fields. Expressed as a sequence of individual mappings for Clover fields separated from each other by semicolon. Each individual mapping looks like this: `$CloverField:=#XLSColumnCode` or `$CloverField:=XLSColumnName`. See Mapping and Metadata (p. 340) for more information.	
Advanced			
Number of skipped records		Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Max number of records		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Number of skipped records per source		Number of records to be skipped from each source file. See Selecting Input Records (p. 258).	Same as in Metadata (default) | 0-N

Attribute	Req	Description	Possible values
Max number of records per source		Maximum number of records to be read from each source file. See Selecting Input Records (p. 258).	0-N
Max error count		Maximum number of allowed errors for the `Controlled` value of **Data Policy** before the graph fails.	0 (default) | 1-N
Incremental file	2)	Name of the file storing the incremental key, including path. See Incremental Reading (p. 258).	
Incremental key	2)	Variable storing the position of the last read record. See Incremental Reading (p. 258).	
Deprecated			
Start row		Has inclusive meaning: First row that is read. Has lower priority than **Number of skipped records**.	0 (default) | 1-n
Final row		Has exclusive meaning: First row that is not already read following the last row that still has been read. Has lower priority than **Max number of records**.	all (default) | 1-n

Legend:

1) One of these attributes must be specified. **Sheet name** has higher priority.

2) Either both or neither of these attributes must be specified.

Advanced Description

Mapping and Metadata

If you want to specify some mapping (**Field mapping**), click the row of this attribute. After that, a button appears there and when you click this button, the following dialog will open:

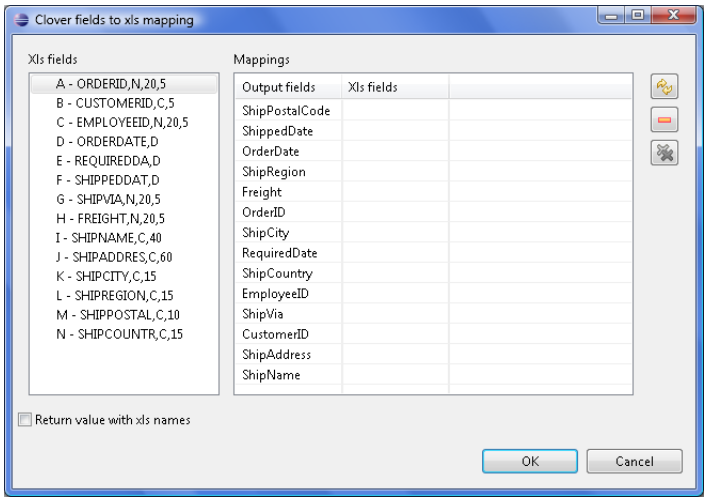


Figure 44.7. XLS Mapping Dialog

This dialog consists of two panes: **XLS fields** on the left and **Mappings** on the right. At the right side of this dialog, there are three buttons: for automatic mapping, canceling one selected mapping and canceling all mappings. You must select an xls field from the left pane, push the left mouse button, drag to the right pane (to the **XLS fields** column) and release the button. This way, the selected xls field has been mapped to one of the output clover fields. Repeat the same with the other xls fields too. (Or you can click the **Auto mapping** button.)

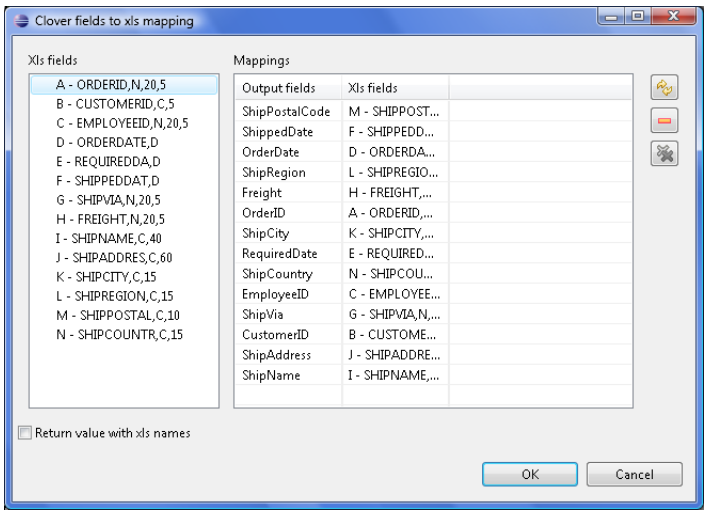


Figure 44.8. XLS Fields Mapped to Clover Fields

Note that xls fields are derived automatically from xls column names when extracting metadata from the XLS file.

When you confirm the mapping by clicking **OK**, the resulting **Field mapping** attribute will look like this (for example): `$OrderDate:=#D;$OrderID:=#A`

On the other hand, if you check the **Return value with xls names** checkbox on the **XLS mapping** dialog, the same mapping will look like this: `$OrderDate:=ORDERDATE,D;$OrderID:=ORDERID,N,20,5`

You can see that the **Field mapping** attribute is a sequence of single mappings separated from semicolon from each other.

Each single mapping consists of an assignment of a clover field name and xls field. The Clover field is on the left side of the assignment and it is preceded by dollar sign, the xls field is on the right side of the assignment and it is either the code of xls column preceded by hash, or the xls field as shown in the **Xls fields** pane.

You must remember that you do not need to read and send out all xls columns, you can even read and only send out some of them.

Example 44.2. Field Mapping in XLSDataReader

- **Mapping with Column Codes**

  ```
  $first_name:=#B;$last_name:=#D;$country:=#E
  ```

- **Mapping with Column Names (XLS Fields)**

  ```
  $first_name:=f_name;$last_name:=l_name;$country:=country
  ```

XMLExtract

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

XMLExtract reads data from XML files.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
XMLExtract	XML file	0-1	1-n	no	yes	no	no	no	no

Legend

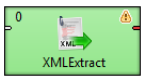
1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation (**DataGenerator** and **MultiLevelReader**). See Return Values of Transformations (p. 238) for more information. **XMLExtract** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Abstract

XMLExtract reads data from XML files using SAX technology. It can also read data from compressed files, console, input port, and dictionary. This component is faster than **XMLXPathReader** which can read XML files too.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any[1)]
	1-n	2)	For correct data records	Any[1)] (each port can have different metadata)

Legend:

1): Metadata on each output port does not need to be the same. Each metadata can use Autofilling Functions (p. 117).

2): Other output ports are required if mapping requires that.

XMLExtract Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (XML file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Use nested nodes		By default, nested elements are also mapped to output ports automatically. If set to `false`, an explicit `<Mapping>` tag must be created for each such nested element.	true (default) | false
Mapping URL	1)	Name of external file, including path, defining mapping of XML structure to output ports. See XMLExtract Mapping Definition (p. 344) for more information.	
Mapping	1)	Mapping of XML structure to output ports. See XMLExtract Mapping Definition (p. 344) for more information.	
XML Schema		URL of the file that should be used for creating the **Mapping** definition. See XMLExtract Mapping Editor and XSD Schema (p. 348) for more information.	
Trim strings		By default, white spaces from the beginning and the end of the elements values are removed. If set to `false`, they are not removed.	true (default) | false
Advanced			
XML features		Sequence of individual expressions of one of the following form: `nameM:=true` or `nameN:=false`, where each `nameM` is an XML feature that should be validated. These expressions are separated from each other by semicolon. See XML Features (p. 260) for more information.	
Number of skipped mappings		Number of mappings to be skipped continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Max number of mappings		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 258).	0-N

Legend:

1) One of these must be specified. If both are specified, **Mapping URL** has higher priority.

Advanced Description

Example 44.3. Mapping in XMLExtract

```
<Mappings>
    <Mapping element="employee" outPort="0" xmlFields="salary" cloverFields="basic_salary">
        <Mapping element="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
        <Mapping element="benefits" outPort="2"
                                    parentKey="empID;jobID" generatedKey="empID;jobID"
                                    sequenceField="seqKey" sequenceId="Sequence0">
            <Mapping element="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
        </Mapping>
        <Mapping element="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
            <Mapping element="customer" outPort="5"
                                    parentKey="projName;projManager;inProjectID;Start"
                                    generatedKey="joinedKey"/>
        </Mapping>
    </Mapping>
</Mappings>
```

XMLExtract Mapping Definition

1. Every **Mapping** definition (both the contents of the file specified in the **Mapping URL** attribute and the **Mapping** attribute) consists of a pair of the start and the end `<Mappings>` tags. Both the start and the end `<Mappings>` tag are empty, without any other attributes.

2. This pair of `<Mappings>` tags surrounds all of the nested `<Mapping>` tags. Each of these `<Mapping>` tags contains some XMLExtract Mapping Tag Attributes (p. 345). See also XMLExtract Mapping Tags (p. 344) for more information.

3. **XMLExtract Mapping Tags**

   - **Empty Mapping Tag (Without a Child)**

     `<Mapping element="[prefix:]nameOfElement"` XMLExtract Mapping Tag Attributes (p. 345) `/>`

     This corresponds to the following node of XML structure:

     ```
     <[prefix:]nameOfElement>ValueOfTheElement</[prefix:]nameOfElement>
     ```

   - **Non-Empty Mapping Tags (Parent with a Child)**

     `<Mapping element="[prefix:]nameOfElement"` XMLExtract Mapping Tag Attributes (p. 345) `>`

     ```
     (nested Mapping elements (only children, parents with one or more
     children, etc.)

     </Mapping>
     ```

     This corresponds to the following XML structure:

     ```
     <[prefix:]nameOfElement elementAttributes>

     (nested elements (only children, parents with one or more children, etc.)

     </[prefix:]nameOfElement>
     ```

4. Nested structure of `<Mapping>` tags copies the nested structure of XML elements in input XML files. See example below.

Example 44.4. From XML Structure to Mapping Structure

- **If XML Structure Looks Like This:**

```
<[prefix:]nameOfElement>
    <[prefix1:]nameOfElement1>ValueOfTheElement11</[prefix1:]nameOfElement1>
    ...
    <[prefixK:]nameOfElementM>ValueOfTheElementKM</[prefixK:]nameOfElementM>
    <[prefixL:]nameOfElementN>
        <[prefixA:]nameOfElementE>ValueOfTheElementAE</[prefixA:]nameOfElementE>
        ...
        <[prefixR:]nameOfElementG>ValueOfTheElementRG</[prefixR:]nameOfElementG>
    </[prefixK:]nameOfElementN>
</[prefix:]nameOfElement>
```

- **Mapping Can Look Like This:**

```
<Mappings>
    <Mapping element="[prefix:]nameOfElement" attributes>
        <Mapping element="[prefix1:]nameOfElement1" attributes11/>
        ...
        <Mapping element="[prefixK:]nameOfElementM" attributesKM/>
        <Mapping element="[prefixL:]nameOfElementN" attributesLN>
            <Mapping element="[prefixA:]nameOfElementE" attributesAE/>
            ...
            <Mapping element="[prefixR:]nameOfElementG" attributesRG/>
        </Mapping>
    </Mapping>
</Mappings>
```

However, **Mapping** does not need to copy all of the XML structure, it can start at the specified level inside the XML file. In addition, if the default setting of the **Use nested nodes** attribute is used (`true`), it also allows mapping of deeper nodes without needing to create separate child `<Mapping>` tags for them).

Important

Remember that mapping of nested nodes is possible only if their names are unique within their parent and confusion is not possible.

5. **XMLExtract Mapping Tag Attributes**

- `element`

  Required

  Each mapping tag must contain one `element` attribute. The value of this element must be a node of the input XML structure, eventually with a prefix (namespace).

  `element="[prefix:]name"`

- `outPort`

  Optional

  Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

  If the `<Mapping>` tag does not contain any `outPort` attribute, it only serves to identify where the deeper XML nodes are located.

  Example: `outPort="2"`

Important

The values from any level can also be sent out using a higher parent `<Mapping>` tag (when default setting of **Use nested nodes** is used and their identification is unique so that confusion is not possible).

- `parentKey`

  The `parentKey` attribute serves to identify the parent for a child.

  Thus, `parentKey` is a sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe.

  These fields are used in metadata on the port specified for such higher level element, they are filled with corresponding values and this attribute (`parentKey`) only says what fields should be copied from parent level to child level as the identification.

  For this reason, the number of these metadata fields and their data types must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

  Example: `parentKey="first_name;last_name"`

  The values of these parent clover fields are copied into clover fields specified in the `generatedKey` attribute.

- `generatedKey`

  The `generatedKey` attribute is filled with values taken from the parent element. It specifies the parent of the child.

  Thus, `generatedKey` is a sequence of metadata fields on the specified child level separated by semicolon, colon, or pipe.

  These metadata fields are used on the port specified for this child element, they are filled with values taken from parent level, in which they are sent to those metadata fields of the `parentKey` attribute specified in this child level. It only says what fields should be copied from parent level to child level as the identification.

  For this reason, the number of these metadata fields and their data types must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

  Example: `generatedKey="f_name;l_name"`

  The values of these clover fields are taken from clover fields specified in the `parentKey` attribute.

- `sequenceField`

  Sometimes a pair of `parentKey` and `generatedKey` does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has mupliple children of the same element name.

  In such a case, these children may be given numbers as the identification.

  By default (if not defined otherwise by a created sequence), children are numbered by integer numbers starting from 1 with step 1.

  This attribute is the name of metadata field of the specified level in which the distinguishing numbers are written.

  It can serve as `parentKey` for the next nested level.

Example: `sequenceField="sequenceKey"`

- `sequenceId`

  Optional

  Sometimes a pair of `parentKey` and `generatedKey` does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has mupliple children of the same element name.

  In such a case, these children may be given numbers as the identification.

  If this sequence is defined, it can be used to give numbers to these child elements even with different starting value and different step. It can also preserve values between subsequent runs of the graph.

  Id of the sequence.

  Example: `sequenceId="Sequence0"`

  ### Important

  Sometimes there may be a parent which has multiple children of the same element name. In such a case, these children cannot be indentified using the parent information copied from `parentKey` to `generatedKey`. Such information is not sufficient. For this reason, a sequence may be defined to give distinguishing numbers to the mutliple child elements.

- `xmlFields`

  If the names of XML nodes or attributes should be changed, it has to be done using a pair of `xmlFields` and `cloverFields` attributes.

  A sequence of element or attribute names on the specified level can be separated by semicolon, colon, or pipe.

  The same number of these names has to be given in the `cloverFields` attribute.

  Do not foget the values have to correspond to the specified data type.

  Example: `xmlFields="salary;spouse"`

  What is more, you can reach further than the current level of XML elements and their attributes. Use the "../" string to reference "the parent of this element". See Source Tab (p. 349) for more information.

  ### Important

  By default, XML names (element names and attribute names) are mapped to metadata fields by their name.

- `cloverFields`

  If the names of XML nodes or attributes should be changed, it must be done using a pair of `xmlFields` and `cloverFields` attributes.

  Sequence of metadata field names on the specified level are separated by a semicolon, colon, or pipe.

  The number of these names must be the same in the `xmlFields` attribute.

  Also the values must correspond to the specified data type.

  Example: `cloverFields="SALARY;SPOUSE"`

Important

By default, XML names (element names and attribute names) are mapped to metadata fields by their name.

- `skipRows`

  Optional

  Number of elements which must be skipped. By default, nothing is skipped.

  Example: `skipRows="5"`

  ### Important

  Remember that also nested (child) elements are skipped when their parent is skipped.

- `numRecords`

  Optional

  Number of elements which should be read. By default, all are read.

  Example: `numRecords="100"`

XMLExtract Mapping Editor and XSD Schema

In addition to writing the mapping code yourself, you can set the **XML Schema** attribute. It is the URL of a file containing an XSD schema that can be used for creating the **Mapping** definition.

When using an XSD, the mapping can be performed visually in the **Mapping** dialog. It consists of two tabs: the **Mapping** tab and the **Source** tab. The **Mapping** attribute can be defined in the **Source** tab, while in the **Mapping** tab you can work with your **XML Schema**.

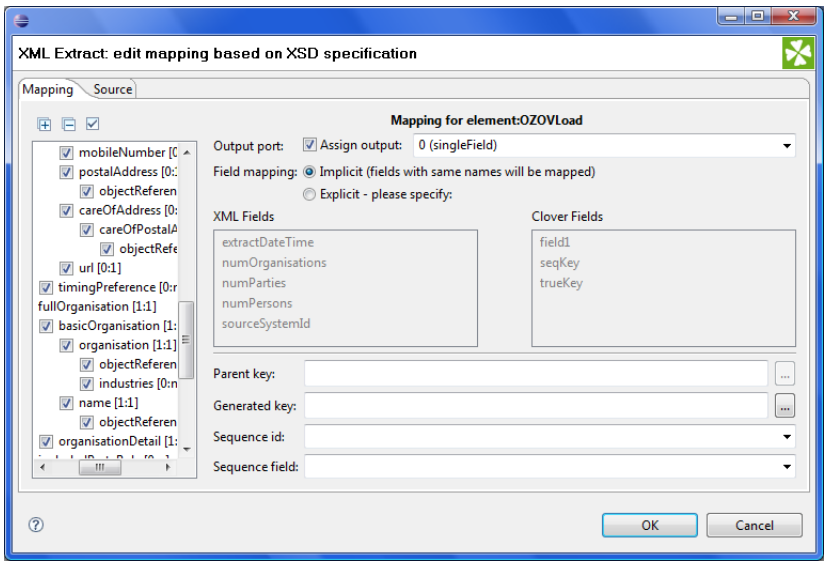


Figure 44.9. The Mapping Dialog for XMLExtract

In the pane on the left hand side of the **Mapping** tab, you can see a tree structure of the file. Every element shows how many elements of the level are contained in the file. In this pane, you need to check the elements that should be mapped to the output ports.

At the top, you specify output port for each selected element by checking the check box. You can then choose from the list ouf output ports labeled `port_number(metadata)`, e.g. "3(customer)".

On the right hand side, you can see both **XML Fields** and **Clover Fields**. You either map them to each other according to their names (implicit mapping) or you map them explicitly. Please note that in XML Fields, not only elements but also their parent elements are visible (as long as parents have some fields) and can be mapped. In the picture below, the "customer" element is selected but we are allowed to leap over its parent element "project" to "employee" whose field "emplID" is actually mapped. Consequently, that enables you to create the whole mapping in a much easier way than if you used the **Parent key** and **Generated key** properties.

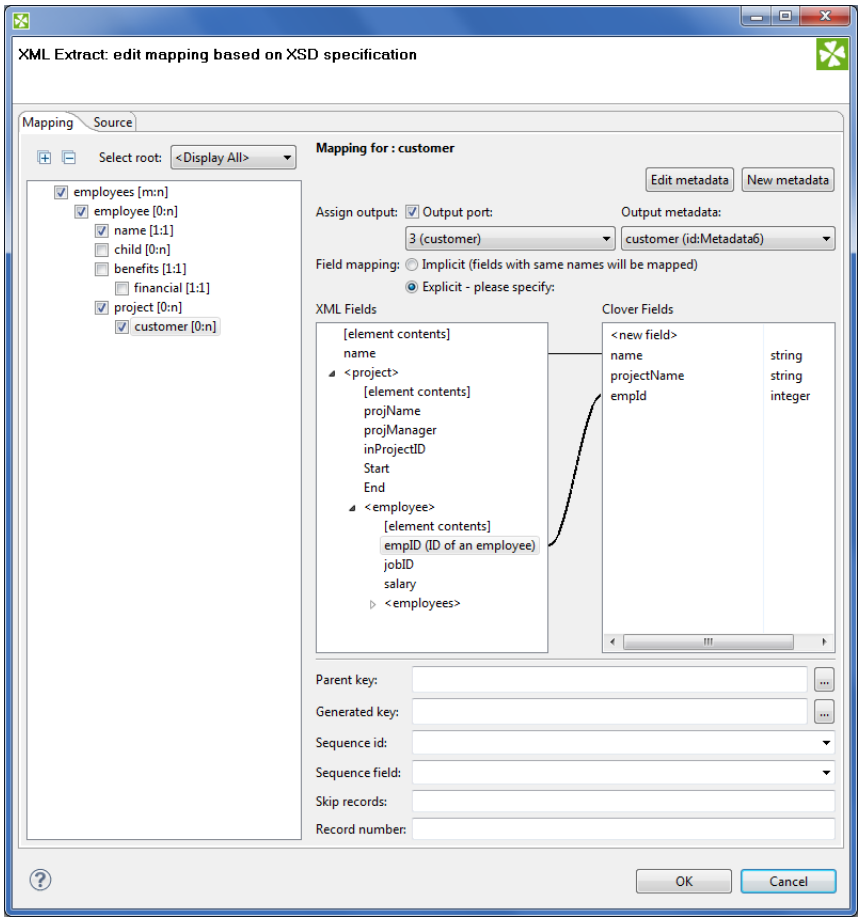


Figure 44.10. Parent Elements Visible in XML Fields

Anyway, in the **Mapping** tab, you can specify all ordinary mapping properties: **Parent key**, **Generated key**, **Sequence id** and/or **Sequence field**.

Source Tab

Once you define all elements, specify output ports and mapping and other properties, you can switch to the **Source** tab. The mapping source code is displayed there. Its structure is the same as described in the preceding sections.

If you want to map an element to XML fields of its parents, use the "../" string (like in the file system) preceding the field name. Every "../" stands for "this element's parent", so "../../" would mean the element's parent's parent and so on. Examine the example below which relates to Figure 44.10, Parent Elements Visible in XML Fields (p. 349). The "../../empID" is a field of "employee" as made available to the currently selected element "customer".

```
<Mapping element="employee">
 <Mapping element="project">
  <Mapping element="customer"
   xmlFields="name;../../empID"
```

```
    cloverFields="name;empId"/>
 </Mapping>
</Mapping>
```

There's one thing that one should keep in mind when referencing parent elements particularly if you relay on the **Use nested nodes** component property to be set to `true`: To reference one parent level using "../" actually means to reference that ancestor element (over more parents) in read XML which is defined in direct parent `<Mapping>` of `<Mapping>` with the "../" parent reference.

An example is always a good thing so here it goes. Lets recall mapping from last example. We will omit one of its `<Mapping>` elements and notice how also the parent field reference had to be changed accordingly.

```
<Mapping element="employee">
 <Mapping element="customer"
  xmlFields="name;../empID"
  cloverFields="name;empId"/>
</Mapping>
```

Usage of Dot In Mapping

Since version 3.1.0 it is possible to map the value of an element using the '.' dot syntax. The dot can be used in the `xmlFields` attribute just the same way as any other XML element/attribute name. In the visual mapping editor, the dot is represented in the XML Fields tree as the element's contents.

The dot represents 'the element itself' (its name). Every other occurence of the element's name in mapping (as text, e.g. "customer") represents the element's subelement or attribute.

The following chunk of code maps the value of element `customer` on metadata field `customerValue` and the value of `customer`'s parent element `project` on metadata field `projectValue`.

```
<Mapping element="project">
 <Mapping element="customer" outPort="0"
  xmlFields=".;../."
  cloverFields="customerValue;projectValue"/>
</Mapping>
```

The element value consists of the text enclosed between the element's start and end tag only if it has no child elements. If the element has child element(s), then the element's value consists of the text between the element's start tag and the start tag of its first child element.

Important

Remember that element values are mapped to Clover fields by their names. Thus, the `<customer>` element mentioned above would be mapped to Clover field named `customer` automatically (implicit mapping).

However, if you want to *rename* the `<customer>` element to a Clover field with another name (explicit mapping), the following construct is necessary:

```
    <Mapping ... xmlFields="customer" cloverFields="newFieldName"   />
```

Moreover, when you have an XML file containg an element and an attribute of the same name:

```
    <customer customer="JohnSmithComp">
    ...
    </customer>
```

you can map both the element and the attribute value to two different fields:

```
<Mapping element="customer" outPort="2"
   xmlFields=".;customer"
   cloverFields="customerElement;customerAttribute"/>
</Mapping>
```

You could even come across a more complicated situation stemming from the example above - the element has an attribute and a subelement all of the same name. The only thing to do is add another mapping at the end of the construct. Notice you can optionally send the subelement to a different output port than its parent. The other option is to leave the mapping blank, but you have to handle the subelement somehow:

```
<Mapping element="customer" outPort="2"
  xmlFields=".;customer"
  cloverFields="customerElement;customerAttribute"/>
  <Mapping element="customer" outPort="4" /> // customer's subelement called 'customer' as well
</Mapping>
```

Remember the explicit mapping (renaming fields) shown in the examples has a higher priority than the implicit mapping.

Templates

Source tab is the only place where templates can be used. Templates are useful when reading a lot of nested elements or recursive data in general.

A template consists of a declaration and a body. The body stretches from the declaration on (up to a potential template reference, see below) and can contain arbitrary mapping. The declaration is an element containing the `templateId` attribute. See example template declaration:

```
<Mapping element="category"  templateId="myTemplate">
 <Mapping element="subCategory"
  xmlFields="name"
  cloverFields="subCategoryName"/>
</Mapping>
```

To use a template, fill in the `templateRef` attribute with an existing `templateId`. Obviously, you have to declare a template first before referencing it. The effect of using a template is that the whole mapping starting with the declaration is copied to the place where the template reference appears. The advantage is obvious: every time you need to change a code that often repeats, you make the change on one place only - in the template. See a basic example of how to reference a template in your mapping:

```
<Mapping templateRef="myTemplate" />
```

Furthermore, a template reference can appear inside a template declaration. The reference should be placed as the last element of the declaration. If you reference the same template that is being declared, you will create a recursive template.

You should always keep in mind how the source XML looks like. Remember that if you have n levels of nested data you should set the `nestedDepth` attribute to n. Look at the example:

```
<Mapping element="myElement"  templateId="nestedTempl">

  <!-- ... some mapping ... -->

 <Mapping templateRef="nestedTempl" nestedDepth="3"/>

</Mapping> <!-- template declaration ends here -->
```

Note

The following chunk of code:

```
 <Mapping templateRef="unnestedTempl" nestedDepth="3" />
```

can be imagined as

```
 <Mapping templateRef="unnestedTempl">
  <Mapping templateRef="unnestedTempl">
   <Mapping templateRef="unnestedTempl">
   </Mapping>
  </Mapping>
 </Mapping>
```

and you can use both ways of nesting references. The latter one with three nested references can produce unexpected results when inside a template declaration, though. As we step deeper and deeper, each `templateRef` copies its template code. BUT when e.g. the 3rd reference is active, it has to copy the code of the two references above it first, then it copies its own code. That way the depth in the tree increases very quickly (exponentially). Luckily, to avoid confusion, you can always wrap the declaration with an element and use nested references outside the declaration. See the example below, where the "wrap" element is effectively used to separate the template from references. In that case, 3 references do refer to 3 levels of nested data.

```
<Mapping element="wrap">
 <Mapping element="realElement" templateId="unnestedTempl"

 <!-- ... some mapping ... -->

 </Mapping>  <!-- template declaration ends here -->
</Mapping> <!-- end of wrap -->

<Mapping templateRef="unnestedTempl">
 <Mapping templateRef="unnestedTempl">
  <Mapping templateRef="unnestedTempl">
  </Mapping>
```

```
  </Mapping>
 </Mapping>
```

In summary, working with `nestedDepth` instead of nested template references always grants transparent results. Its use is recommended.

XMLXPathReader

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 37, Common Properties of Readers (p. 250)

If you want to find the right **Reader** for your purposes, see Readers Comparison (p. 251).

Short Summary

XMLXPathReader reads data from XML files.

Component	Data source	Input ports	Output ports	Each to all outputs[1)]	Different to different outputs[2)]	Transformation	Transf. req.	Java	CTL
XMLXPathReader	XML file	0-1	1-n	no	yes	no	no	no	no

Legend

1) Component sends each data record to all connected output ports.

2) Component sends different data records to different output ports using return values of the transformation (**DataGenerator** and **MultiLevelReader**). See Return Values of Transformations (p. 238) for more information. **XMLExtract** and **XMLXPathReader** send data to ports as defined in their **Mapping** or **Mapping URL** attribute.

Abstract

XMLXPathReader reads data from XML files using DOM technology. It can also read data from compressed files, console, input port, and dictionary. This component is slower than **XMLExtract**, which can read XML files too.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For port reading. See Reading from Input Port (p. 254).	One field (`byte`, `cbyte`, `string`).
Output	0	yes	For correct data records	Any[1)]
	1-n	2)	For correct data records	Any[1)] (each port can have different metadata)

Legend:

1): The metadata on each of the output ports does not need to be the same. Each of these metadata can use Autofilling Functions (p. 117).

2): Other output ports are required if mapping requires that.

XMLXPathReader Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying what data source(s) will be read (XML file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 251).	
Charset		Encoding of records that are read.	ISO-8859-1 (default) | <other encodings>
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient
Mapping URL	1)	Name of external file, including path, defining mapping of XML structure to output ports. See XMLXPathReader Mapping Definition (p. 356) for more information.	
Mapping	1)	Mapping of XML structure to output ports. See XMLXPathReader Mapping Definition (p. 356) for more information.	
Advanced			
XML features		Sequence of individual expressions of one of the following form: `nameM:=true` or `nameN:=false`, where each `nameM` is an XML feature that should be validated. These expressions are separated from each other by semicolon. See XML Features (p. 260) for more information.	
Number of skipped mappings		Number of mappings to be skipped continuously throughout all source files. See Selecting Input Records (p. 258).	0-N
Max number of mappings		Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 258).	0-N

Legend:

1) One of these must be specified. If both are specified, **Mapping URL** has higher priority.

Advanced Description

Example 44.5. Mapping in XMLXPathReader

```
<Context xpath="/employees/employee" outPort="0">
    <Mapping nodeName="salary" cloverField="basic_salary"/>
    <Mapping xpath="name/firstname" cloverField="firstname"/>
    <Mapping xpath="name/surname" cloverField="surname"/>
    <Context xpath="child" outPort="1" parentKey="empID" generatedKey="parentID"/>
    <Context xpath="benefits" outPort="2" parentKey="empID;jobID" generatedKey="empID;jobID"
                              sequenceField="seqKey" sequenceId="Sequence0">
        <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
    </Context>
    <Context xpath="project" outPort="4" parentKey="empID;jobID" generatedKey="empID;jobID">
        <Context xpath="customer" outPort="5" parentKey="projName;projManager;inProjectID;Start"
                                  generatedKey="joinedKey"/>
    </Context>
</Context>
```

Note

Nested structure of `<Context>` tags is similar to the nested structure of XML elements in input XML files.

However, **Mapping** attribute does not need to copy all XML structure, it can start at the specified level inside the whole XML file.

XMLXPathReader Mapping Definition

1. Every **Mapping** definition (both the contents of the file specified in the **Mapping URL** attribute and the **Mapping** attribute) consists of `<Context>` tags which contain also some attributes and allow mapping of element names to Clover fields.

2. Each `<Context>` tag can surround a serie of nested `<Mapping>` tags. These allow to rename XML elements to Clover fields.

3. Each of these `<Context>` and `<Mapping>` tags contains some XMLXPathReader Context Tag Attributes (p. 357) and XMLXPathReader Mapping Tag Attributes (p. 358), respectively.

4. **XMLXPathReader Context Tags and Mapping Tags**

   - **Empty Context Tag (Without a Child)**

     `<Context xpath="xpathexpression"` XMLXPathReader Context Tag Attributes (p. 357) `/>`

   - **Non-Empty Context Tag (Parent with a Child)**

     `<Context xpath="xpathexpression"` XMLXPathReader Context Tag Attributes (p. 357) `>`

     `(nested Context and Mapping elements (only children, parents with one or more children, etc.)`

     `</Context>`

   - **Empty Mapping Tag (Renaming Tag)**

     - `xpath` is used:

       `<Mapping xpath="xpathexpression"` XMLXPathReader Mapping Tag Attributes (p. 358) `/>`

     - `nodeName` is used:

```
<Mapping nodeName="elementname" XMLXPathReader Mapping Tag Attributes (p. 358) />
```

5. **XMLXPathReader Context Tag and Mapping Tag Attributes**

   1) **XMLXPathReader Context Tag Attributes**

   - `xpath`

     Required

     The xpath expression can be any XPath query.

     Example: `xpath="/tagA/.../tagJ"`

   - `outPort`

     Optional

     Number of output port to which data is sent. If not defined, no data from this level of **Mapping** is sent out using such level of **Mapping**.

     Example: `outPort="2"`

   - `parentKey`

     Both `parentKey` and `generatedKey` must be specified.

     Sequence of metadata fields on the next parent level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `generatedKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

     Example: `parentKey="first_name;last_name"`

     Equal values of these attributes assure that such records can be joined in the future.

   - `generatedKey`

     Both `parentKey` and `generatedKey` must be specified.

     Sequence of metadata fields on the specified level separated by semicolon, colon, or pipe. Number and data types of all these fields must be the same in the `parentKey` attribute or all values are concatenated to create a unique string value. In such a case, key has only one field.

     Example: `generatedKey="f_name;l_name"`

     Equal values of these attributes assure that such records can be joined in the future.

   - `sequenceId`

     When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

     Id of the sequence.

     Example: `sequenceId="Sequence0"`

   - `sequenceField`

     When a pair of `parentKey` and `generatedKey` does not insure unique identification of records, a sequence can be defined and used.

A metadata field on the specified level in which the sequence values are written. Can serve as `parentKey` for the next nested level.

Example: `sequenceField="sequenceKey"`

- `namespacePaths`

  Optional

  Default namespaces that should be used for the `xpath` attribute specified in the `<Context>` tag.

  Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.

Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

2) **XMLXPathReader Mapping Tag Attributes**

- `xpath`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag.

  XPath query.

  Example: `xpath="tagA/.../salary"`

- `nodeName`

  Either `xpath` or `nodeName` must be specified in `<Mapping>` tag. Using `nodeName` is faster than using `xpath`.

  XML node that should be mapped to Clover field.

  Example: `nodeName="salary"`

- `cloverField`

  Required

  Clover field to which XML node should be mapped.

  Name of the field in the corresponding level.

  Example: `cloverFields="SALARY"`

- `trim`

  Optional

  Specifies whether leading and trailing white spaces should be removed. By default, it removes both leading and trailing white spaces.

  Example: `trim="false"` (white spaces will not be removed)

- `namespacePaths`.

  Optional

  Default namespaces that should be used for the `xpath` attribute specified in the `<Mapping>` tag.

  Pattern: `namespacePaths='prefix1="URI1";...;prefixN="URIN"'`

  Example: `namespacePaths='n1="http://www.w3.org/TR/html4/";n2="http://ops.com/"'`.

Note

Remember that if the input XML file contains a default namespace, this `namespacePaths` must be specified in the corresponding place of the **Mapping** attribute. In addition, `namespacePaths` is inherited from the `<Context>` element and used by the `<Mapping>` elements.

Chapter 45. Writers

We assume that you already know what components are. See Chapter 17, Components (p. 84) for brief information.

Only some of the components in a graph are terminal nodes. These are called **Writers**.

Writers can write data to output files (both local and remote), send it through the connected optional output port, or write it to dictionary. One component only discards data. Since it is also a terminal node, we describe it here.

Components can have different properties. But they also can have something in common. Some properties are common for all of them, others are common for most of the components, or they are common for **Writers** only. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

We can distinguish **Writers** according to what they can write:

- One component discards data:
  - Trash (p. 414) discards data.

Other **Writers** write data to files.

- Flat files:
  - UniversalDataWriter (p. 416) writes data to flat files (delimited or fixed length).
- Other files:
  - CloverDataWriter (p. 362) writes data to files in Clover binary format.
  - XLSDataWriter (p. 419) writes data to XLS or XLSX files.
  - StructuredDataWriter (p. 410) writes data to files with user-defined structure.
  - XMLWriter (p. 422) creates XML files from input data records.

Other **Writers** loads data into database.

- Databases Writers:
  - DBOutputTable (p. 369) loads data into database using JDBC driver.
  - QuickBaseRecordWriter (p. 408) writes data into the **QuickBase** online database.
  - QuickBaseImportCSV (p. 406) writes data into the **QuickBase** online database.
  - LotusWriter (p. 391) writes data into **Lotus Notes** and **Lotus Domino** databases.
- High-Speed Database Specific Writers (Bulk Loaders):
  - DB2DataWriter (p. 364) loads data into DB2 database using DB2 client.
  - InfobrightDataWriter (p. 381) loads data into Infobright database using Infobright client.
  - InformixDataWriter (p. 383) loads data into Informix database using Informix client.
  - MSSQLDataWriter (p. 393) loads data into MSSQL database using MSSQL client.

- MySQLDataWriter (p. 396) loads data into MYSQL database using MYSQL client.
- OracleDataWriter (p. 399) loads data into Oracle database using Oracle client.
- PostgreSQLDataWriter (p. 403) loads data into PostgreSQL database using PostgreSQL client.

Other **Writers** send e-mails, JMS messages or write directory structure.

- E-mails:
  - EmailSender (p. 377) converts data records into e-mails.
- JMS messages:
  - JMSWriter (p. 386) converts data records into JMS messages.
- Directory structure:
  - LDAPWriter (p. 389) converts data records into a directory structure.

CloverDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

CloverDataWriter writes data to files in our internal binary Clover data format.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
CloverDataWriter	Clover binary file	1	0	no	no	no	no

Abstract

CloverDataWriter writes data to files (local or remote) in our internal binary Clover data format. It can also compress output files, write data to console, or dictionary.

Note

Since 2.9 version of **CloverETL** **CloverDataWriter** writes also a header to output files with the version number. For this reason, **CloverDataReader** expects that files in Clover binary format contain such a header with the version number. **CloverDataReader** 2.9 cannot read files written by older versions of **CloverETL** nor these older versions can read data written by **CloverDataWriter** 2.9.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For received data records	Any

CloverDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying where received data will be written (Clover data file, console, dictionary). See Supported File URL Formats for Writers (p. 263) See also Output File Structure (p. 363) for more information.	
Append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Save metadata		By default, no file with metadata definition is saved. If set to `true`, metadata is saved to metadata file. See Output File Structure (p. 363) for more information.	false (default) | true
Save index		By default, no file with indices of records is saved. If set to `true`, the index of records is saved to an index file. See Output File Structure (p. 363) for more information.	false (default) | true
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Compress level		Sets the compression level. By default, zip compression level is used. Level `0` means archiving without compression.	-1 (default) | 0-9
Number of skipped records		Number of records to be skipped. See Selecting Output Records (p. 270).	0-N
Max number of records		Maximum number of records to be written to the output file. See Selecting Output Records (p. 270).	0-N

Advanced Description

Output File Structure

- **Non-Archived Output File(s)**

  If you do not archive and/or compress the created file(s), the output file(s) will be saved separately with the following name(s): `filename` (for the file with data), `filename.idx` (for the file with index) and `filename.fmt` (for the file with metadata). In all of the created name(s), `filename` contains its extension (if it has any) in all of these three created file(s) names.

- **Archived Output File(s)**

  If the output file is archived and/or compressed (independently on the type of the file), it has the following internal structure: `DATA/filename`, `INDEX/filename.idx` and `META/filename.fmt`. Here, `filename` includes its extension (if it has any) in all of these three names.

  Example 45.1. Internal Structure of Archived Output File(s)

  `DATA/employees.clv`, `INDEX/employees.clv.idx`, `META/employees.clv.fmt`.

DB2DataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

DB2DataWriter loads data into DB2 database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
DB2DataWriter	database	0-1	0-1	no	no	no	no

Abstract

DB2DataWriter loads data into database using DB2 database client. It can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component. If you connect some other component to the optional output port, it can serve to log the information about errors. DB2 database client must be installed and configured on localhost. Server and database must be cataloged as well.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any
Output	0	no	For information about incorrect records	Error Metadata for DB2DataWriter (p. 365)[2)]

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

2): **Error Metadata** cannot use Autofilling Functions (p. 117).

Table 45.1. Error Metadata for DB2DataWriter

Field number	Field name	Data type	Description
0	<any_name1>	integer	Number of incorrect record (records are numbered starting from 1)
1	<any_name2>	integer	number of incorrect field (for delimited records), fields are numbered starting from 1 | offset of incorrect field (for fixed-length records)
2	<any_name3>	string	Error message

DB2DataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File metadata		Metadata of external file. Must only be delimited. Each column except the last one is followed by an identical, one char delimiter. The last delimiter following the last column is `\n`. Delimiter must not be a part of any field value.	
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
User name	yes	Database user.	
Password	yes	Password for database user.	
Load mode		Mode of the action performed when loading data. See Load mode (p. 367) for more information.	insert (default) | replace | restart | terminate
Field mapping	1)	Sequence of individual mappings (`$CloverField:=DBField`) separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 367) for more information.	
Clover fields	1)	Sequence of Clover fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 367) for more information.	
DB fields	1)	Sequence of DB fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 367) for more information.	
Advanced			
Loader input file	2)	Name of input file to be loaded, including path. See Loader input file (p. 368) for more information.	
Parameters		All parameters that can be used as parameters by `load` method. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains the delimiter as its part, such value must be double quoted.	
Rejected records URL (on server)		Name of the file, including path, on DB2 server where rejected records will be saved. Must be located in the directory owned by database user.	

Attribute	Req	Description	Possible values
Batch file URL		URL of the file where the `connect`, `load` and `disconnect` commands for db2 load utility are stored. Normally the batch file is automatically generated, stored in current directory and deleted after the load finishes. If the **Batch file URL** is specified, component tries to use it as is (generates it only if it does not exist or if its length is 0) and does not delete it after the load finishes. (It is reasonable to use this attribute in connection with the **Loader input file** attribute, because batch file contains the name of temporary data file which is generated at random, if not provided explicitly). Path must not contain white spaces.	
DB2 command interpreter		Interpreter that should execute script with DB2 commands (`connect`, `load`, `disconnect`). Its form must be the following: `interpreterName [parameters] ${} [parameters]`. This `${}` expression must be replaced by the name of this script file.	
Use pipe transfer		By default, data from input port is written to a temporary file and then it is read by the component. If set to `true`, on Unix data records received through the input port are sent to pipe instead of a temporary file.	false (default) | true
Column delimiter		The first one char field delimiter from **File metadata** or the metadata on the input edge (if **File metadata** is not specified). Character used as a delimiter for each column in data file. Delimiter must not be contained as a part of a field value. The same delimiter can be set by specifying the value of the `coldel` parameter in the **Parameters** attribute. If **Column delimiter** is set, `coldel` in **Parameters** is ignored.	
Number of skipped records		Number of records to be skipped. By default, no records are skipped. This attribute is applied only if data is received through the input port. Otherwise, it is ignored.	0 (default) | 1-N
Max number of records		Maximum number of records to be loaded into database. The same can be set by specifying the value of the `rowcount` parameter in the **Parameters** attribute. If `rowcount` is set in **Parameters**, the **Max number of records** attribute is ignored.	all (default) | 0-N
Max error count		Maximum number of records after which the load stops. If some number is set explicitely and when it is reached, process can continue in RESTART mode. In REPLACE mode, process continues from the beginning. The same number can be specified with the help of `warningcount` in the **Parameters** attribute. If `warningcount` is specified, **Max error count** is ignored.	all (default) | 0-N
Max warning count		Maximum number of printed error messages and/or warnings.	999 (default) | 0-N

Legend:

1) See Mapping of Clover Fields to DB Fields (p. 367) for more information about their relation.

2) If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 368) for more information.

Advanced Description

Mapping of Clover Fields to DB Fields

- **Field Mapping is Defined**

  If a **Field mapping** is defined, the value of each Clover field specified in this attribute is inserted to such DB field to whose name this Clover field is assigned in the **Field mapping** attribute.

- **Both Clover Fields and DB Fields are Defined**

  If both **Clover fields** and **DB fields** are defined (but **Field mapping** is not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field which lies on the same position in the **DB fields** attribute.

  Number of Clover fields and DB fields in both of these attributes must equal to each other. The number of either part must equal to the number of DB fields that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

  Pattern of **DB fields**:

  ```
  DBFieldA;...;DBFieldM
  ```

- **Only Clover Fields are Defined**

  If only the **Clover fields** attribute is defined (but **Field mapping** and/or **DB fields** are not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field whose position in DB table is equal.

  Number of Clover fields specified in the **Clover fields** attribute must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **Mapping is Performed Automatically**

  If neither **Field mapping**, **Clover fields**, nor **DB fields** are defined, the whole mapping is performed automatically. The value of each Clover field of Metadata is inserted into the same position in DB table.

  Number of all Clover fields must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

Load mode

- `insert`

  Loaded data is added to the database table without deleting or changing existing table content.

- `replace`

  All data existing in the database table is deleted and new loaded data is inserted to the table. Neither the table definition nor the index definition are changed.

- `restart`

Previously interrupted load operation is restarted. The load operation automatically continues from the last consistency point in the load, build, or delete phase.

- `terminate`

  Previously interrupted load operation is terminated and rolled back to the moment when it started even if consistency points had been passed.

Loader input file

Loader input file is the name of input file with data to be loaded, including its path. Normally this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead. Remember that DB2 client must be installed and configured on localhost (see DB2 client setup overview). Server and database must be cataloged as well.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or `named pipe` is used instead of temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, the file must exist, must be specified and must contain data that should be loaded into database. It is not deleted nor overwritten.

DBOutputTable

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

DBOutputTable loads data into database using JDBC driver.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
DBOutputTable	database	1	0-2	✖	✖	✖	✖

Abstract

DBOutputTable loads data into database using JDBC driver. It can also send out rejected records and generate autogenerated columns for some of the available databases.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	Records to be loaded into the database	Any
Output	0	✖	For rejected records	Based on Input 0[1)]
	1	✖	For returned values	Any[2)]

Legend:

1): Metadata on output port 0 may contain any number of fields from input (same names and types) along with up to two additional fields for error information. Input metadata are mapped automatically according to their name(s) and

type(s). The two error fields may have any names and must be set to the following Autofilling Functions (p. 117): `ErrCode` and `ErrText`.

2): Metadata on output port 1 must include at least the fields returned by the `returning` statement specified in the query (for example, `returning $outField1:=$inFieldA,$outField2:=update_count, $outField3:=$inFieldB`). Remember that fields are not mapped by names automatically. A mapping must always be specified in the `returning` statement. Number of returned records is equal to the the number of incoming records.

DBOutputTable Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	✔	ID of the DB connection to be used.	
Query URL	1)	Name of external file, including path, defining SQL query. See Query or DB Table is Defined (p. 371) for more information.	
SQL query	1)	SQL query defined in the graph. See Query or DB Table is Defined (p. 371) for more information. See also SQL Query Editor (p. 374).	
DB table	1)	Name of DB table. See Query or DB Table is Defined (p. 371) for more information.	
Field mapping	2)	Sequence of individual mappings (`$CloverField:=DBField`) separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 372) for more information.	
Clover fields	2)	Sequence of Clover fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 372) for more information.	
DB fields	2)	Sequence of DB fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 372) for more information.	
Query source charset		Encoding of external file defining SQL query.	ISO-8859-1 (default) | <other encodings>
Batch mode		By default, batch mode is not used. If set to `true`, batch mode is turned on. Supported by some databases only. See Batch Mode and Batch Size (p. 373) for more information.	false (default) | true
Advanced			
Batch size		Number of records that can be sent to database in one batch update. See Batch Mode and Batch Size (p. 373) for more information.	25 (default) | 1-N
Commit		Defines after how many records (without an error) commit is performed. If set to `MAX_INT`, commit is never performed by the component, i.e., not untill the connection is closed during graph freeing. This attribute is ignored if **Atomic SQL query** is defined.	100 (default) | 1-MAX_INT
Max error count		Maximum number of allowed records. When this number is exceeded, graph fails. By default, no error is allowed. If set to `-1`, all errors are allowed. See Errors (p. 373) for more information.	0 (default) | 1-N | -1

Attribute	Req	Description	Possible values
Action on error		By default, when the number of errors exceeds **Max error count**, correct records are commited into database. If set to `ROLLBACK`, no commit of the current batch is performed. See Errors (p. 373) for more information.	COMMIT (default) | ROLLBACK
Atomic SQL query		Sets atomicity of executing SQL queries. If set to `true`, all SQL queries for one record are executed as atomic operation, but the value of the **Commit** attribute is ignored and commit is performed after each record. See Atomic SQL Query(p. 374) for more information.	false (default) | true

[1)] One of these attributes must be specified. If more are defined, **Query URL** has the highest priority and **DB table** the lowest one. See Query or DB Table is Defined (p. 371) for more information.

[2)]See Mapping of Clover Fields to DB Fields (p. 372) for more information about their relation.

Advanced Description

Query or DB Table is Defined

- **A Query is Defined (SQL Query or Query URL)**
  - **The Query Contains Clover Fields**

    Clover fields are inserted to the specified positions of DB table.

    This is the most simple and explicit way of defining the mapping of Clover and DB fields. No other attributes can be defined.

    See also SQL Query Editor (p. 374).
  - **The Query Contains Question Marks**

    Question marks serve as placeholders for Clover field values in one of the ways shown below. See Mapping of Clover Fields to DB Fields (p. 372) for more information.

    See also SQL Query Editor (p. 374).

Example 45.2. Examples of Queries

Statement	Form
Derby, Infobright, Informix, MSSQL2008, MSSQL2000-2005, MySQL[1)]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]
DB2, Oracle[2)]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := dbf3[, $out1field3 := dbf7]]
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?) [returning $out1field1 := dbf3[, $out1field3 := dbf7]]

Statement	Form
PostgreSQL, SQLite, Sybase[3)]	
insert (with clover fields)	insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm)
insert (with question marks)	insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)
All databases[4)]	
update	update mytable set dbf1 = $in0field1, ..., dbfn = $in0fieldn [returning $out1field1 := $in0field3[, $out1field2 := update_count][, $out1field3 := $in0field7]]
delete	delete from mytable where dbf1 = $in0field1 and ... and dbfj = ? and dbfn = $in0fieldn

Legend:

1)These databasases generate a virtual field called `auto_generated` and map it to one of the output metadata fields as specified in the `insert` statement.

2)These databases return multiple database fields and map them to the output metadata fields as specified in the `insert` statement.

3)These databases do not return anything in the `insert` statement.

4) In the `update` statement, along with the value of the `update_count` virtual field, any number of input metadata fields may be mapped to output metadata fields in all databases.

Important

Remember that the default (**Generic**) JDBC specific does not support auto-generated keys.

- **A DB Table is Defined**

  The mapping of Clover fields to DB fields is defined as shown below. See Mapping of Clover Fields to DB Fields (p. 372) for more information.

Dollar Sign in DB Table Name

- Remember that if any database table contains a dollar sign in its name, it will be transformed to double dollar signs in the generated query. Thus, each query must contain even number of dollar signs in db table (consisting of adjacent pairs of dollars). Single dollar sign contained in the name of db table is replaced by double dollar sign in the query in the name of db table.

  Table whose name is `my$table$` is converted in the query to `my$$table$$`.

Mapping of Clover Fields to DB Fields

- **Field Mapping is Defined**

  If a **Field mapping** is defined, the value of each Clover field specified in this attribute is inserted to such DB field to whose name this Clover field is assigned in the **Field mapping** attribute.

  Pattern of **Field mapping**:

  ```
  $CloverFieldA:=DBFieldA;...;$CloverFieldM:=DBFieldM
  ```

- **Both Clover Fields and DB Fields are Defined**

  If both **Clover fields** and **DB fields** are defined (but **Field mapping** is not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field which lies on the same position in the **DB fields** attribute.

Number of Clover fields and DB fields in both of these attributes must equal to each other. The number of either part must equal to the number of DB fields that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

Pattern of **Clover fields**:

```
CloverFieldA;...;CloverFieldM
```

Pattern of **DB fields**:

```
DBFieldA;...;DBFieldM
```

- **Only Clover Fields are Defined**

  If only the **Clover fields** attribute is defined (but **Field mapping** and/or **DB fields** are not), the value of each Clover field specified in the **Clover fields** attribute is inserted to such DB field whose position in DB table is equal.

  Number of Clover fields specified in the **Clover fields** attribute must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

  Pattern of **Clover fields**:

  ```
  CloverFieldA;...;CloverFieldM
  ```

- **Mapping is Performed Automatically**

  If neither **Field mapping**, **Clover fields**, nor **DB fields** are defined, the whole mapping is performed automatically. The value of each Clover field of Metadata is inserted into the same position in DB table.

  Number of all Clover fields must equal to the number of DB fields in DB table that are not defined in any other way (by specifying clover fields prefixed by dollar sign, db functions, or constants in the query).

Batch Mode and Batch Size

1. **Batch Mode**

   Batch mode speeds up loading of data into database.

2. **Batch Size**

   Remember that some databases return as rejected more records than would correspond to their real number. These databases return even those records which have been loaded into database successfully and send them out through the output port 0 (if connected).

Errors

1. **Max error count**

   Specifies number of errors that are still allowed, but after which graph execution stops. After that, defined **Action on Error** is performed.

2. **Action on Error**

   ```
   COMMIT
   ```

   By default, when maximum number of errors is exceeded, commit is performed for correct records only in some databases. In others, rollback is performed instead. Then, graph stops.

   ```
   ROLLBACK
   ```

On the other hand, if maximum number of errors is exceeded, rollback is performed in all databases, however, only for the last, non-commited records. Then, graph stops. All that has been comitted, cannot be rolled back anymore.

Atomic SQL Query

- **Atomic SQL query** specifies the way how queries consisting of multiple subqueries conserning a single records will be processed.

  By default, each individual subquery is considered separately and in some of these fails, the previous are commited or rolled back according to database.

  If the **Atomic SQL query** attribute is set to `true`, either all subqueries or none of them are commited or rolled back. This assures that all databases behave all in identical way.

Important

Remember also, when connecting to MS SQL Server, it is convenient to use jTDS http://jtds.sourceforge.net driver. It is an open source 100% pure Java JDBC driver for Microsoft SQL Server and Sybase. It is faster than Microsoft's driver.

SQL Query Editor

For defining the **SQL query** attribute, **SQL query editor** can be used.

The editor opens after clicking the **SQL query** attribute row:

On the left side, there is the **Database schema** pane containing information about schemas, tables, columns, and data types of these columns.

Displayed schemas, tables, and columns can be filtered using the values in the **ALL** combo, the **Filter in view** textarea, the **Filter**, and **Reset** buttons, etc.

You can select any columns by expanding schemas, tables and clicking **Ctrl**+**Click** on desired columns.

Adjacent columns can also be selected by clicking **Shift**+**Click** on the first and the list item.

Select one of the following statements from the combo: **insert**, **update**, **delete**.

Then you need to click **Generate** after which a query will appear in the **Query** pane.

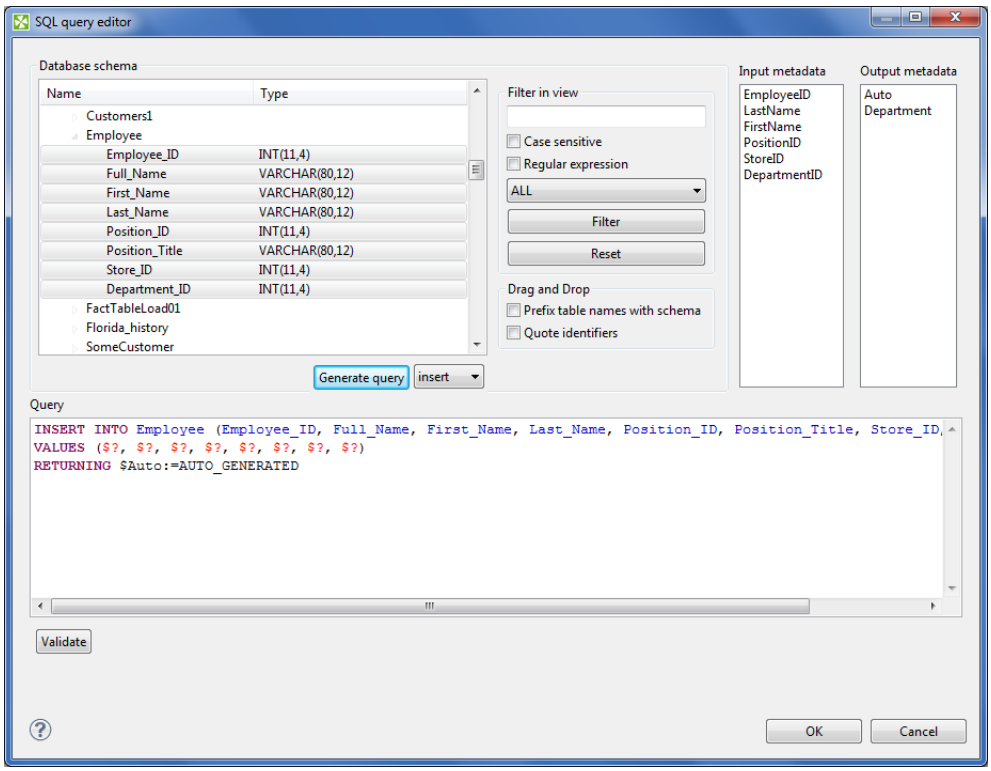


Figure 45.1. Generated Query with Question Marks

The query may contain question marks if any db columns differ from input metadata fields. Input metadata are visible in the **Input metadata** pane on the right side.

Drag and drop the fields from the **Input metadata** pane to the corresponding places in the **Query** pane and manually remove the "$?" characters. See following figure:

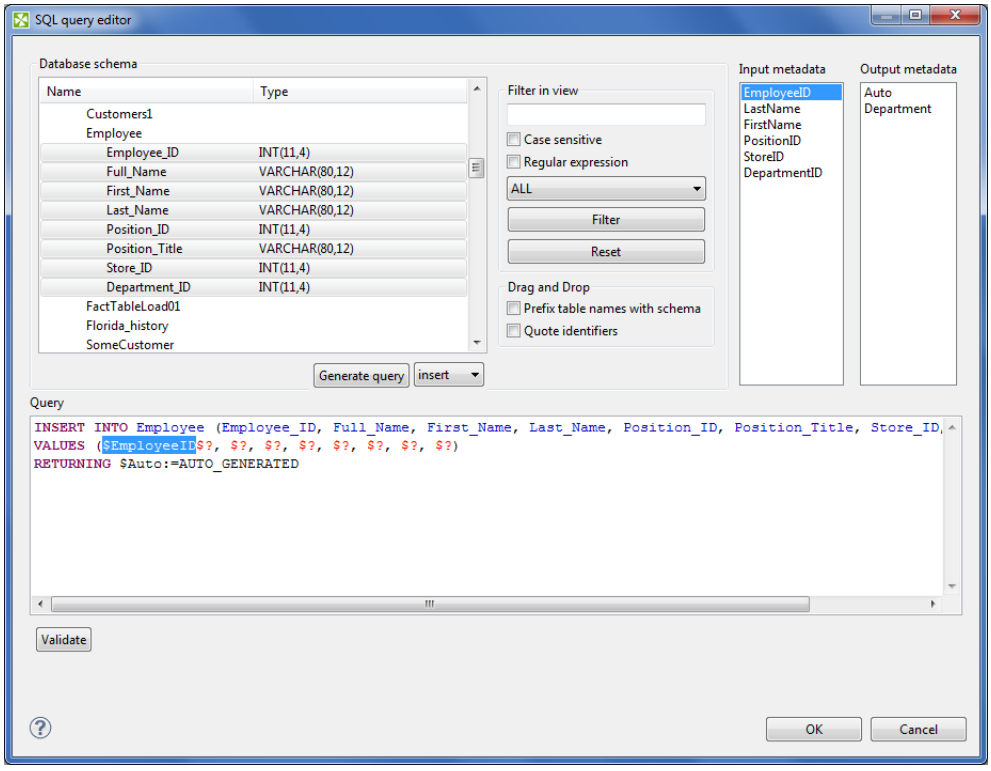


Figure 45.2. Generated Query with Input Fields

If there is an edge connected to the second output port, autogenerated columns and returned fields can be returned.

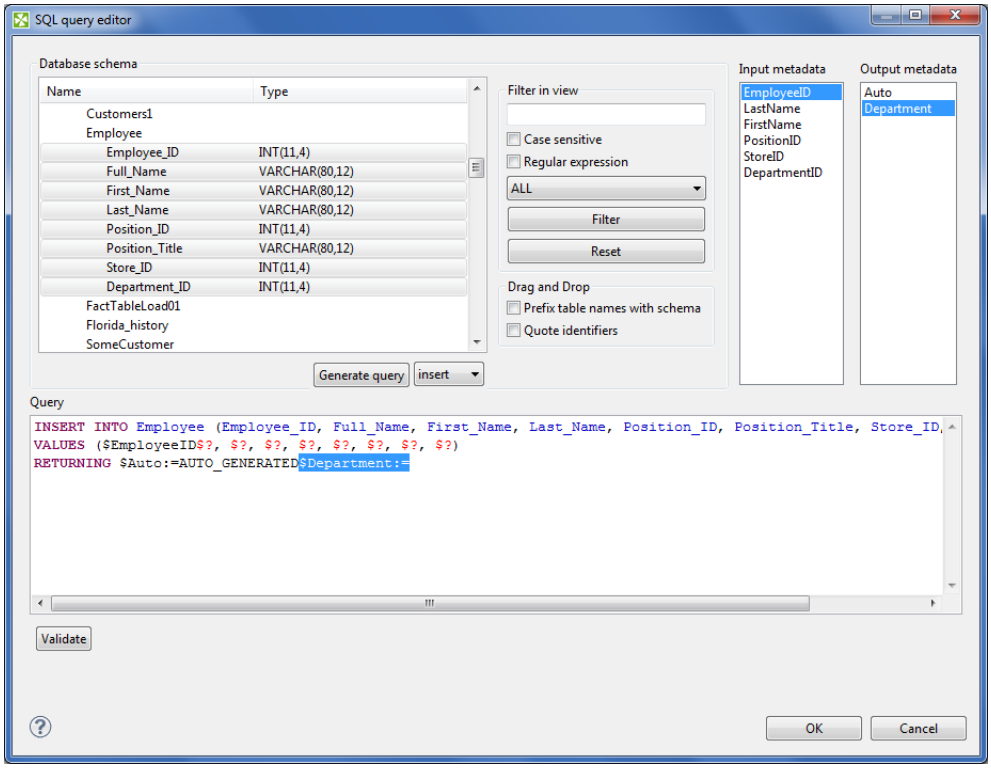


Figure 45.3. Generated Query with Returned Fields

Two buttons allow you to validate the query (**Validate**) or view data in the table (**View**).

EmailSender

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

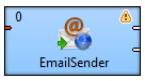
EmailSender sends e-mails.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
EmailSender	flat file	1	0-1	no	no	no	no

Abstract

EmailSender converts data records into e-mails. It can use input data to create the e-mail sender and addressee, e-mail subject, message body, and attachment(s).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For data for e-mails	Any
Output	0	no	For successfully sent e-mails.	Input 0
	1	no	For rejected e-mails.	Input 0 plus field named `errorMessage`[1)]

Legend:

1): If a record is rejected and e-mail is not sent, an error message is created and sent to the `errorMessage` field of metadata on the output 1 (if it contains such a field).

EmailSender Attributes

Attribute	Req	Description	Possible values
Basic			
SMTP server	yes	Name of SMTP server for outgoing e-mails.	
SMTP user		Name of the user for an authenticated SMTP server.	
SMTP password		Password of the user for an authenticated SMTP server.	
Use TLS		By default, TLS is not used. If set to `true`, TLS is turned on.	false (default) | true
Use SSL		By default, SSL is not used. If set to `true`, SSL is turned on.	false (default) | true
Message	yes	Set of properties defining the message headers and body. See E-Mail Message (p. 378) for more information.	
Attachments		Set of properties defining the message attachments. See E-Mail Attachments (p. 379) for more information.	
Advanced			
SMTP port		Number of the port used for connection to SMTP server.	25 (default) | other port
Trust invalid SMTP server certificate		By default, invalid SMTP server certificates are not accepted. If set to `true`, invalid SMTP server certificate (with different name, expired, etc) is accepted.	false (default) | true
Ignore send fail		By default, when an e-mail is not sussessfully sent, graph fails. If set to `true`, graph execution stops even if no mail can be sent successfully.	false (default) | true

Advanced Description

- **E-Mail Message**

  To define the **Message** attribute, you can use the following wizard:

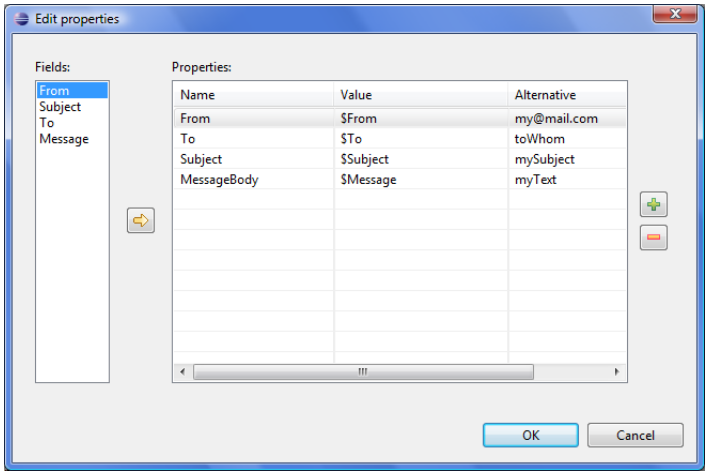


Figure 45.4. EmailSender Message Wizard

In this wizard, you must copy the fields from the **Fields** pane on the left to the **Value** column of the **Properties** pane on the right. Use the **Right arrow** button or drag and drop the selected field to the **Value** column. In

addition, you can also specify alternative values of these attributes (**Alternative** column). In case some field is empty or has null value, such **Alternative** is used instead of the field value.

The resulting value of the **Message** attribute will look like this:

```
From=$From|my@mail.com;Subject=$Subject|mySubject;To=$To|toWhom;MessageBody=$Message|myText
```

- **E-Mail Attachments**

  One of the possible attributes is **Attachments**. It can be specified as a sequence of individual attachments separated by semicolon. Each individual attachment is either file name including its path, or this file name (including path) can also be specified using the value of some input field. Individual attachment can also be specified as a triplet of field name, file name of the attachment and its mime type. These can be specified both explicitly (`[$fieldName, FileName, MimeType]`) or using the field values: `[$fieldNameWithFileContents, $fieldWithFileName, $fieldWithMimeType]`. Each of these three parts of the mentioned triplet can be specified also using a static expression. The attachments must be added to the e-mail using the following **Edit attachments** wizard:

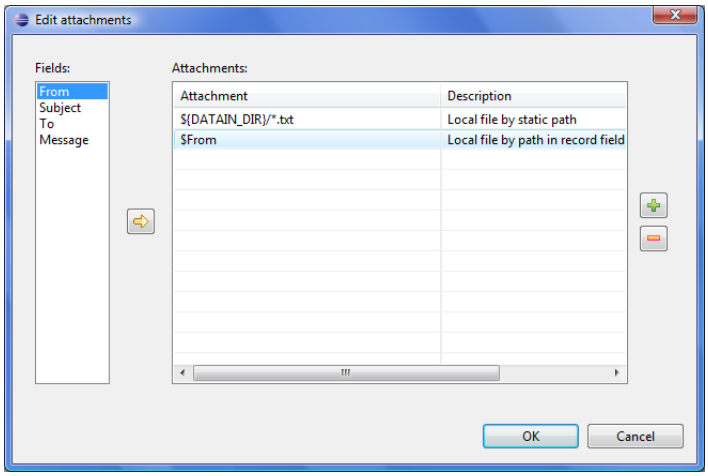


Figure 45.5. Edit Attachments Wizard

You adds the items by clicking the **Plus sign** button, remove by clicking the **Minus sign** button, input fields can be dragged to the **Attachment** column of the **Attachments** pane or the **Arrow** button can be used. If you want to edit any attachment definition, click the corresponding row in the **Attachment** column and the following attribute will open:

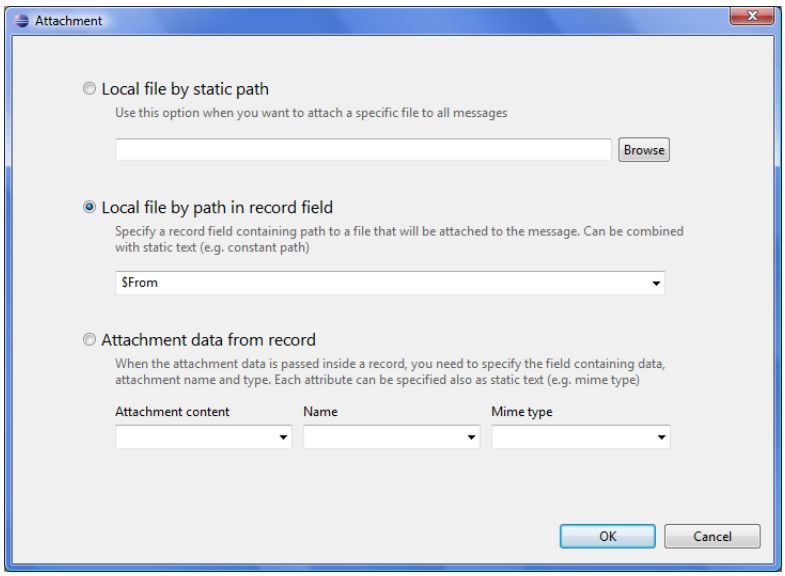


Figure 45.6. Attachment Wizard

In this wizard, you need to locate files, specify them using field names or the mentioned triplet. After clicking **OK**, the attachment is defined.

InfobrightDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

InfobrightDataWriter loads data into Infobright database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
InfobrightDataWriter	database	1	0-1	no	no	no	no

Abstract

InfobrightDataWriter loads data into Infobright database. Only root user can insert data into database with this component. To run this component on Windows, `infobright_jni.dll` must be present in the Java library path. (Can be downloaded at www.infobright.org/downloads/contributions/infobright-core-2_7.zip.)

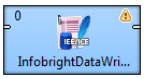
If the hostname is `localhost` or `127.0.0.1`, the load will be done using a local pipe. Otherwise, it will use a remote pipe. The external IP address of the server is not recognized as a local server.

For loading to a remote server you need to start the Infobright remote load agent on the server where Infobright is running. This should be done by executing the `java -jar infobright-core-3.0-remote.jar [-p PortNumber] [-l all | debug | error | info]` command. The output can be redirected to a log file. By default, server is listening at port 5555. The `infobright-core-3.0-remote.jar` is distributed with **CloverETL** or can be downloaded at the Infobright site: www.infobright.org.

By default, `root` is only allowed to connect from `localhost`. You need to add an additional user `root@%` to connect from other hosts. It is recommended to create a different user (not `root`) for loading data. The user requires the `FILE` privilege in order to be able to load data or use the connector:

```
grant FILE on *.* to 'user'@'%';
```

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	Records to be loaded into the database	Any
Output	0	no	For records as they were loaded into database	Corresponding part of metadata on Input 0[1)]

Legend:

1): Only mapped Clover field values can be sent out through the optional output port. Comma must be set as delimiter for each field, `System.getProperty("line.separator")` ("\n" for Unix, "\r\n" for Windows) must be set as record delimiter. Date fields must strictly have the `yyyy-MM-dd` format for dates and the `yyyy-MM-dd HH:mm:ss` format for dates with time.

InfobrightDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	yes	ID of the DB connection object to access the database.	
Database table	yes	Name of DB table where data will be loaded.	
Charset		Charset used for encoding string values to VAR, VARCHAR column types.	ISO-8859-1 (default) | other encoding
Data format		`bh_dataformat` supported by Infobright. Options are: `txt_variable` or `binary` (this is faster, but works with IEE only).	Text (default) | Binary
Advanced			
Clover fields		Sequence of Clover fields separated by semicolon. Only Clover fields listed in this attribute will be loaded into database columns. Position of both Clover field and database column will be the same. Their number should equal to the number of database columns.	
Log file		File for records loaded into database, including path. If this attribute is specified, no data goes to the output port even if it is connected.	
Append data to log file		By default, new record overwrite the older ones. If set to `true`, new records are appended to the older ones.	false (default) | true
Execution timeout		Timeout for load command (in seconds). Has effect only on Windows platform.	15 (default) | 1-N
Check string's and binary's sizes		By default, sizes are not checked before data is passed to database. If set to `true`, sizes are check - should be set to `true` if debugging is supported.	false (default) | true
Remote agent port		Port to be used when connecting to the server.	5555 (default) | otherportnumber

InformixDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

InformixDataWriter loads data into an Informix database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
InformixDataWriter	database	0-1	0-1	no	no	no	no

Abstract

InformixDataWriter loads data into a database using Informix database client (`dbload` utility) or the `load2` free library.

It is very important to have the server with the database on the same computer as both the `dbload` database utility and **CloverETL** and you must be logged in as the root user. The Informix server must be installed and configured on the same machine where **Clover** runs and the user must be logged in as root. The Dbload command line tool must also be available.

InformixDataWriter reads data from the input port or a file. If the input port is not connected to any other component, data must be contained in another file that should be specified in the component.

If you connect a component to the optional output port, rejected records along with information about errors are sent to it.

Another tool is the `load2` free library instead of the `dbload` utility. The `load2` free library can even be used if the server is located on a remote computer.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any
Output	0	no	For information about incorrect records	Input 0 (plus two Error Fields for InformixDataWriter (p. 384)[2)])

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

2): Metadata on the output port 0 contains two additional fields at their end: `number of row`, `error message`.

Table 45.2. Error Fields for InformixDataWriter

Field number	Field name	Data type	Description
LastInputField + 1	<anyname1>	integer	Number of row
LastInputField + 2	<anyname2>	string	Error message

InformixDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to dbload utility	yes	Name of dbload utility, including path. Informix server must be installed and configured on the same machine where Clover runs and the user must be logged in as root. The dbload command line tool must be available.	
Host		Host where database server is located.	
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
Advanced			
Control script		Control script to be used by the dbload utility. If it is not set, the default control script is used instead. Is used only if the **Use load utility** attribute is set to `false`.	
Error log URL		Name of the error log file, including path. If not set, default error log file is used instead.	./error.log
Max error count		Maximum number of allowed records. When this number is exceeded, graph fails.	10 (default) | 0-N
Ignore rows		Number of rows to be skipped. Is used only if the **Use load utility** attribute is set to `true`.	0 (default) | 1-N
Commit interval		Commit interval in number of rows.	100 (default) | 1-N
Column delimiter		One char delimiter used for each column in data. Field values must not include this delimiter as their part. Is used only if the **Use load utility** attribute is set to `false`.	"|" (default) | other character

Attribute	Req	Description	Possible values
Loader input file		Name of input file to be loaded, including path. Normally this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead.	
Use load utility		By default, dbload utility is used to load data to database. If set to `true`, load2 utility is used instead of dbload. The load2 utility must be available.	false (default) | true
User name		Username to be used when connecting to the database. Is used only if the **Use load utility** attribute is set to `true`.	
Password		Password to be used when connecting to the database. Is used only if the **Use load utility** attribute is set to `true`.	
Ignore unique key violation		By default, unique key violation is not ignored. If key values are not unique, graph fails. If set to `true`, unique key violation is ignored. Is used only if the **Use load utility** attribute is set to `true`.	false (default) | true
Use insert cursor		By default, insert cursor is used. Using insert cursor doubles the transfer performance. Is used only if the **Use load utility** attribute is set to `true`. It can be turned off by setting to `false`.	true (default) | false

Advanced Description

Loader input file

Name of input file to be loaded, including path. Normally this file is a temporary storage for data to be passed to dbload utility unless `named pipe` is used instead.

- If it is not set, a loader file is created in Clover or OS temporary directory. The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must exist, must be specified and must contain data that should be loaded into database. It is not deleted or overwritten.

JMSWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

JMSWriter converts Clover data records into JMS messages.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
JMSWriter	jms messages	1	0	yes	no	yes	no

Abstract

JMSWriter receives Clover data records, converts them into JMS messages and sends these messages out. Component uses a processor transformation which implements a `DataRecord2JmsMsg` interface or inherits from a `DataRecord2JmsMsgBase` superclass. Methods of `DataRecord2JmsMsg` interface are described below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For data records	Any[1)]

Legend:

1): Metadata on input port may contain a field specified in the **Message body field** attribute.

JMSWriter Attributes

Attribute	Req	Description	Possible values
Basic			
JMS connection	yes	ID of the JMS connection to be used.	
Processor code		Transformation of records to JMS messages written in the graph in Java.	
Processor URL		Name of external file, including path, containing the transformation of records to JMS messages written in Java.	
Processor class		Name of external class defining the transformation of records to JMS messages. The default processor value (`org.jetel.component.jms.DataRecord2JmsMsgProperties`) is sufficient for most cases. It produces `javax.jms.TextMessage`.	DataRecord2JmsMsgProperties (default) | other class
Processor source charset		Encoding of external file containing the transformation in Java.	ISO-8859-1 (default) | other encoding
Message charset		Encoding of JMS messages contents. This attribute is also used by the default processor implementation (`JmsMsg2DataRecordProperties`). And it is used for `javax.jms.BytesMessage` only.	ISO-8859-1 (default) | other encoding
Advanced			
Message body field		Name of the field of metadata from which the body of the message should be get and sent out. This attribute is used by the default processor implementation (`JmsMsg2DataRecordProperties`). If no **Message body field** is specified, the field whose name is `bodyField` will be used as a resource for the body of the message contents. If no field for the body of the message is contained in metadata (either the specified explicitly or the default one), the processor tries to set a field named *bodyField*, but it's silently ignored if such field doesn't exist in output record metadata. The other fields from metadata serve as resources of message properties with the same names as field names.	bodyField (default) | other name

Legend:

1) One of these may be set. Any of these transformation attributes implements a `DataRecord2JmsMsg` interface.

See Java Interfaces for JMSWriter (p. 388) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Java Interfaces for JMSWriter

The transformation implements methods of the `DataRecord2JmsMsg` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of `DataRecord2JmsMsg` interface:

- `void init(DataRecordMetadata metadata, Session session, Properties props)`

  Initializes the processor.

- `void preExecute(Session session)`

  This is also initialization method, which is invoked before each separate graph run. Contrary the init() procedure here should be allocated only resources for this graph run. All here allocated resources should be released in the `postExecute()` method.

  `session` may be used for creation of JMS messages. Each graph execution has its own session opened. Thus the session set in the `init()` method is usable only during the first execution of graph instance.

- `Message createMsg(DataRecord record)`

  Transforms data record to JMS message. Is called for all data records.

- `Message createLastMsg()`

  This method is called after last record and is supposed to return message terminating JMS output. If it returns null, no terminating message is sent. Since 2.8.

- `String getErrorMsg()`

  Returns error message.

- `Message createLastMsg(DataRecord record)` (deprecated)

  This method is not called explicitly since 2.8. Use `createLastMsg()` instead.

LDAPWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

LDAPWriter writes information to an LDAP directory.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
LDAPWriter	LDAP directory tree	1	0-1	no	no	no	no

Abstract

LDAPWriter writes information to an LDAP directory. It provides the logic to update information on an LDAP directory. An update can be add/delete entries, add/replace/remove attributes. Metadata must match LDAP object attribute name. "DN" metadata attribute is required.

`String`, `byte` and `cbyte` are the only metadata types supported. Most of the LDAP types are compatible with clover string, however, for instance, the `userPassword` LDAP type is necessary to populate from byte data field. LDAP rules are applied : to add an entry, required attributes (even object class) are required in metadata.

Note

LDAP attribute may be multivalued. The default value separator is "|" and is reasonable only for string data fields.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	1	yes	For correct data records	Any[1)]
Output	0-1	no	For rejected records	Input 0

Legend:

1): Metadata on the input must precisely match the LDAP object attribute name. The Distinguished Name metadata attribute is required. As the LDAP attributes are multivalued, their values can be separated by pipe or specified separator. String and byte are the only metadata types supported.

LDAPWriter Attributes

Attribute	Req	Description	Possible values
Basic			
LDAP URL	yes	LDAP URL of the directory. Can be a list of URLs separated by pipe.	pattern: `ldap://host:port/`
Action		Defines the action to be performed with the entry.	replace_attributes (default) | add_entry | remove_entry | remove_attributes
User		User DN to be used when connecting to the LDAP directory. Similar to the following: `cn=john.smith,dc=example,dc=com`.	
Password		Password to be used when connecting to the LDAP directory.	
Advanced			
Multi-value separator		**LDAPWriter** can handle keys with multiple values. These are delimited by this string or character. <none> is special escape value which turns off this functionality, then only first value is written. This attribute can only be used for string data type. When byte type is used, first value is the only one that is written.	"|" (default) | other character or string

LotusWriter

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Chapter 45, Writers (p. 360).

Short Summary

LotusWriter writes data into **Lotus Domino** databases. Data records are stored as Lotus documents in the database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
LotusWriter	Lotus Notes	1	0-1	✖	✖	✖	✖

Abstract

LotusWriter is a component which can write data records to Lotus databases. The writing is be done by connecting to a **Lotus Domino server**.

The data records are written to Lotus database as **Documents**. A document in Lotus is a list of key-value pairs. Every field of written data record will produce one key-value pair, where key will be given by the field name and value by the value of the field.

The user of this component needs to provide a Java library for connecting to Lotus. The libary can be found in the installation of Lotus Notes or Lotus Domino. **LotusWriter** component is not able to communicate with Lotus unless the path to this library is provided or the library is placed on the user's classpath. The path to the library can be specified in the details of Lotus connection (see Chapter 23, Lotus Connections (p. 165)).

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	
Output	0	✖	for invalid data records	

LotusWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Domino connection	✔	ID of the connection to the Lotus Domino database.	
Mode	✖	Write mode. Insert mode always creates new documents in the database. Update mode requires view to be specified. Update operation first finds all documents in the view with same key values as incoming data record. After that either all found documents are updated, or only the first one is updated.	"insert"(default) | "update"
View	✖	The name of the View in Lotus database within which the data records will be updated.	
Advanced			
Mapping	✖	When no mapping is provided, new documents will get the exact same set of fields as retrieved from the input port. With mapping, it is possible to customize which fields will be written to Lotus documents. Fields can be written with different names and order, some can be skipped and some written multiple times with different names. Often it is desirable to work with fields from the **Document** Lotus form. Mapping of input port fields onto Document form fields can be established in this attribute.	docFieldX := inFieldY; ...
Compute with form	✖	When enabled, computation will be launched on the newly created document. The computation is typically defined in a **Document** form. This form is used by the users of Lotus Notes to create new documents. The computation may for example fill-in empty fields with default values or perform data conversions.	true (default) | false
Skip invalid documents	✖	When enabled, documents marked by Lotus as invalid will not be saved into the Lotus database. This setting requires **Compute with form** attribute to be enabled, otherwise validation will not be performed. Validation is performed by the computing Document form action.	true | false (default)
Update mode	✖	Toggles the usage of lazy update mode and behavior when multiple documents are found for update. Lazy update mode only updates the document when values get actually changed - written value (after optional computation) is different from the original value. When multiple documents are found to be updated, either only first one can be updated, or all of them can be updated.	"Lazy, first match"(default) | "Lazy, all matches" | "Eager, first match" | "Eager, all matches"
Multi-value fields	✖	Denotes input fields which should be treated as multi-value fields. Multi-value field will be split into multiple strings by using the separator specified in the **Multi-value separator** attribute. The resulting array of values will then be stored as a multi-value vector into the Lotus database.	semi-colon separated list of input fields
Multi-value separator	✖	A string that will be used to separate values from multi-value Lotus fields.	";" (default) | "," | ":" | "|" | "\t" | other character or string

MSSQLDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

MSSQLDataWriter loads data into MSSQL database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
MSSQLDataWriter	database	0-1	0-1	no	no	no	no

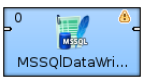
Abstract

MSSQLDataWriter loads data into a database using the MSSQL database client. It reads data through the input port or from a file. If the input port is not connected to any other component, data must be contained in another file that should be specified in the component.

If you connect some other component to the optional output port, it can serve to log the rejected records and information about errors. Metadata on this error port must have the same metadata fields as the input records plus three additional fields at its end: `number of incorrect row` (integer), `number of incorrect column` (integer), `error message` (string).

SQL Server Client Connectivity Components must be installed and configured on the same machine where **CloverETL** runs. The bcp command line tool must be available.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any
Output	0	no	For information about incorrect records	Input 0 (plus three Error Fields for MSSQLDataWriter (p. 394)[2)])

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, the input port must be connected.

2): Metadata on the output port 0 contain three additional fields at their end: `number of incorrect row`, `number of incorrect column`, `error message`.

Table 45.3. Error Fields for MSSQLDataWriter

Field number	Field name	Data type	Description
LastInputField + 1	<anyname1>	integer	Number of incorrect row
LastInputField + 2	<anyname2>	integer	Number of incorrect column
LastInputField + 3	<anyname3>	string	Error message

MSSQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to bcp utility	yes	Name of bcp utility, including path. SQL Server Client Connectivity Components must be installed and configured on the same machine where Clover runs. Bcp command line tool must be available.	
Database	yes	Name of the database where the destination table or view resides.	
Server name		Name of the server to which bcp utility should connect. If bcp utility connects to local named or remote named instance of server, **Server name** should be set to `serverName\instanceName`. If bcp utility connects to local default or remote default instance of server, **Server name** should be set to `serverName`. If it is not set, bcp connects to the local default instance of server on localhost. The same meaning is true for the `serverName` which can be set in **Parameters**. However, if both **Server name** attribute and the `serverName` parameter are set, `serverName` in **Parameters** is ignored.	
Database table	1)	Name of the destination table.	
Database view	1)	Name of the destination view. All columns of the view must refer to the same table.	
Database owner		Owner of table or view. Does not need to be specified if the user performing the operations is the owner. If it is not set and the user is not the owner, SQL Server returns an error message and the process is cancelled.	
User name	yes	Login ID to be used when connecting to the server. The same can be set by specifying the value of the `userName` parameter in the **Parameters** attribute. If set, `userName` in **Parameters** is ignored.	
Password	yes	Password for the login ID to be used when connecting to the server. The same can be set by specifying the value of the `password` parameter in the **Parameters** attribute. If set, `password` in **Parameters** is ignored.	
Advanced			

Attribute	Req	Description	Possible values
Column delimiter		Delimiter used for each column in data. Field values cannot have the delimiter within them. The same can be set by specifying the value of the `fieldTerminator` parameter in the **Parameters** attribute. If set, `fieldTerminator` in **Parameters** is ignored.	\t (default) | any other character
Loader input file		Name of input file to be loaded, including path. See Loader input file (p. 395) for more information.	
Parameters		All parameters that can be used as parameters by the bcp utility. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon. If the value of any parameter contains semicolon as its part, such value must be double quoted. See Parameters (p. 395) for more information.	

Legend:

1) One of these must be specified.

2) If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 395) for more information.

Advanced Description

Loader input file

You can or you must specify another attribute (**Loader input file**), dependent on an edge being connected to the input port. It is the name of input file with data to be loaded, including its path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or `named pipe` is used instead of temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If an input port is not connected, the file must exist, must be specified and must contain data that should be loaded. It is not deleted nor overwritten.

Parameters

You may also want to set some series of parameters that can be used when working with MSSQL (**Parameters**). For example, you can set the number of the port, etc. All of the parameters must have the form of `key=value` or `key` only (if its value is `true`). Individual parameters must be separated from each other by colon, semicolon or pipe. Note that colon, semicolon or pipe can also be a part of some parameter value, but in this case the value must be double quoted.

Among the optional parameters, you can also set `userName`, `password` or `fieldTerminator` for **User name**, **Password** or **Column delimiter** attributes, respectively. If some of the three attributes (**User name**, **Password** and **Column delimiter**) will be set, corresponding parameter value will be ignored.

If you also need to specify the server name, you should do it within parameters. The pattern is as follows: `serverName=[msServerHost]:[msServerPort]`. For example, you can specify both server name and user name in the following way: `serverName=msDbServer:1433|userName=msUser`.

MySQLDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

MySQLDataWriter is a high-speed MySQL table loader. Uses MySQL native client.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
MySQLDataWriter	database	0-1	0-1	no	no	no	no

Abstract

MySQLDataWriter quickly loads data into a database table using native MySQL database client.

It reads data either from the input port or from an input file.

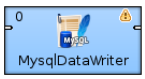
You can attach the **optional output port** and read records which have been reported as rejected.

Reading from input port (input port connected) dumps the data into a temporary file which is then used by `mysql` utility. You can set the temporary file explicitly by setting the **Loader input file** attribute or leave it blank to use default.

Reading from a file (no input connected) uses "Loader input file" attribute as a path to your data file. The attribute is mandatory in this case. The file needs to be in a format recognized by `mysql` utility (see MySQL LOAD DATA).

This component executes MySQL native command-line client (`bin/mysql` or `bin/mysql.exe`). The client must be installed on the same machine as the graph is running on.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any
Output	0	no	For information about incorrect records	Error Metadata for MySQLDataWriter (p. 397)[2)]

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

2): **Error Metadata** cannot use Autofilling Functions (p. 117).

Table 45.4. Error Metadata for MySQLDataWriter

Field number	Field name	Data type	Description
0	<any_name1>	integer	Number of incorrect record (records are numbered starting from 1)
1	<any_name2>	integer	number of incorrect column
2	<any_name3>	string	Error message

MySQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to mysql utility	yes	Name of mysql utility, including path. Must be installed and configured on the same machine where Clover runs. Mysql command line tool must be available.	
Host		Host where database server is located.	localhost (default) | other host
Database	yes	Name of the database into which the records should be loaded.	
Database table	yes	Name of the database table into which the records should be loaded.	
User name	yes	Database user.	
Password	yes	Password for database user.	
Advanced			
Path to control script		Name of command file containing the `LOAD DATA INFILE` statement, including path. See Path to Control Script(p. 398) for more information.	
Lock database table		By default, database is not locked and multiuser access is allowed. If set to `true`, database table is locked to ensure exclusive access and possibly faster loading.	false (default) | true
Ignore rows		Number of rows of data file that should be skipped. By default, no records are skipped. Valid only for input file with data.	0 (default) | 1-N
Column delimiter		Delimiter used for each column in data. Field values must not include this delimiter as their part. By default, tabulator is used.	\t (default) | other character

Attribute	Req	Description	Possible values
Loader input file	1)	Name of input file to be loaded, including path. See Loader input file (p. 398) for more information.	
Parameters		All parameters that can be used as parameters by `load` method. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains the delimiter as its part, such value must be double quoted.	

Legend:

1) If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 398) for more information.

Advanced Description

Path to Control Script

Name of command file containing the `LOAD DATA INFILE` statement (See MySQL LOAD DATA), including path.

- If it is not set, a command file is created in Clover temporary directory and it is deleted after the load finishes.
- If it is set, but the specified command file does not exist, temporary command file is created with the specified name and path and it is not deleted after the load finishes.
- If it is set and the specified command file exists, this file is used instead of command file created by Clover.

Loader input file

Name of input file to be loaded, including path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) or stdin is used instead of temporary file (on Unix). The file is deleted after the load finishes.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must be specified, must exist and must contain data that should be loaded into database. The file is not deleted nor overwritten.

OracleDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

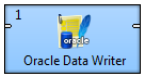
OracleDataWriter loads data into Oracle database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
OracleDataWriter	database	0-1	0-1	no	no	no	no

Abstract

OracleDataWriter loads data into database using Oracle database client. It can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component. If you connect some other component to the optional output port, rejected records are sent to it. Oracle sqlldr database utility must be installed on the computer where **CloverETL** runs.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any
Output	0	no	Rejected records	Input 0

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

OracleDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to sqlldr utility	yes	Name of sqlldr utility, including path.	
TNS name	yes	TNS name identifier.	
User name	yes	Username to be used when connecting to the Oracle database.	
Password	yes	Password to be used when connecting to the Oracle database.	
Oracle table	yes	Name of the database table into which the records should be loaded.	
Advanced			
Control script		Control script for the sqlldr utility. See Control Script (p. 401) for more information.	
Append		Specifies what should be done with database table. See Append Attribute (p. 401) for more information.	append (default) | insert | replace | truncate
Log file name		Name of the file where the process is logged.	${PROJECT}/ loaderinputfile.log
Bad file name		Name of the file where the records causing errors is written.	${PROJECT}/ loaderinputfile.bad
Discard file name		Name of the file where the records not meeting selection criteria is written.	${PROJECT}/ loaderinputfile.dis
DB column names		Names of all columns in the database table.	
Loader input file		Name of input file to be loaded, including path. See Loader Input File (p. 402) for more information.	
Max error count		Maximum number of allowed insert errors. When this number is exceeded, graph fails. If no errors are to be allowed, the attribute should be set to 0. To allow all errors, set this attribute to a very high number.	50 (default) | 0-N
Max discard count		Number of records that can be discarded before the graph stops. If set to 1, even single discarded record stops the graph run.	all (default) | 1-N
Ignore rows		Number of rows of the data file that should be skipped when loading data to database.	0 (default) | 1-N
Commit interval		Conventional path loads only: **Commit interval** specifies the number of rows in the bind array. Direct path loads only: **Commit interval** identifies the number of rows that should be read from the data file before the data is saved. By default, all rows are read and data is all saved at once, at the end of the load.	64 (default for conventional path) | all (default for direct path) | 1-N

Attribute	Req	Description	Possible values
Use file for exchange		By default, on Unix pipe transfer is used. If it is set to `true` and **Loader input file** is not set, temporary file is created and used as data source. By default, on Windows temporary file is created and used as data source. However, since some clients do not need a temporary data file to be created, this attribute can be set to `false` for such clients.	false (default on Unix) | true (default on Windows)
Parameters		All parameters that can be used as parameters by the sqlldr utility. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains semicolon, colon, or pipe as its part, such value must be double quoted.	

Advanced Description

Control Script

Control script for the sqlldr utility.

- If specified, both the **Oracle table** and the **Append** attributes are ignored. Must be specified if input port is not connected. In such a case, **Loader input file** must also be defined.
- If **Control script** is not set, default control script is used.

Example 45.3. Example of a Control script

```
LOAD DATA
INFILE *
INTO TABLE test
append
(
    name TERMINATED BY ';',
    value TERMINATED BY '\n'
)
```

Append Attribute

- **Append (default)**

  Specifies that data is simply appended to a table. Existing free space is not used.

- **Insert**

  Adds new rows to the table/view with the `INSERT` statement. The `INSERT` statement in Oracle is used to add rows to a table, the base table of a view, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or the base table of an object view.

  An `INSERT` statement with a `VALUES` clause adds to the table a single row containing the values specified in the `VALUES` clause.

  An `INSERT` statement with a subquery instead of a `VALUES` clause adds to the table all rows returned by the subquery. Oracle processes the subquery and inserts each returned row into the table. If the subquery selects no rows, Oracle inserts no rows into the table. The subquery can refer to any table, view, or snapshot, including the target table of the `INSERT` statement.

- **Update**

  Changes existing values in a table or in a view's base table.

- **Truncate**

  Removes all rows from a table or cluster and resets the `STORAGE` parameters to the values when the table or cluster was created.

Loader Input File

Name of input file to be loaded, including path.

- If it is not set, a loader file is created in Clover or OS temporary directory (on Windows) (unless **Use file for exchange** is set to `false`) or `named pipe` is used instead of temporary file (in Unix). The created file is deleted after the load finishes.
- If it is set, specified file is created. The created file is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must be specified, must exist and must contain data that should be loaded into database. At the same time, **Control script** must be specified. The file is not deleted nor overwritten.

PostgreSQLDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

PostgreSQLDataWriter loads data into PostgreSQL database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
PostgreSQLDataWriter	database	0-1	0	no	no	no	no

Abstract

PostgreSQLDataWriter loads data into database using PostgreSQL database client. It can read data through the input port or from an input file. If the input port is not connected to any other component, data must be contained in an input file that should be specified in the component. PostgreSQL client utility (`psql`) must be installed and configured on the same machine where **CloverETL** runs.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	Records to be loaded into the database	Any

Legend:

1): If no file containing data for loading (**Loader input file**) is specified, input port must be connected.

PostgreSQLDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Path to psql utility	yes	Name of psql utility, including path. Must be installed and configured on the same machine where **CloverETL** runs. Psql command line tool must be available.	
Host		Host where database server is located.	localhost (default) | other host
Database	yes	Name of the database into which the records should be loaded.	
Database table		Name of the database table into which the records should be loaded.	
User name		PostgreSQL username to be used when connecting to the server.	
Advanced			
Fail on error		By default, graph fails upon each error. If you want to have the standard behavior of PostgreSQL database, you need to switch this attribute to `false`. If set to `false`, graph will run successfully even with some errors as it happens with PostgreSQL database.	true (default) | false
Path to control script		Name of command file containing the `\copy` statement, including path. See Path to Control Script(p. 404) for more information.	
Column delimiter		Delimiter used for each column in data. Field values must not include this delimiter as their part.	tabulator character (default in text mode) | comma (default in CVS mode)
Loader input file		Name of input file to be loaded, including path. See Loader Input File (p. 405) for more information.	
Parameters		All parameters that can be used as parameters by the psql utility or the `\copy` statement. These values are contained in a sequence of pairs of the following form: `key=value`, or `key` only (if the `key` value is the boolean `true`) separated from each other by semicolon, colon, or pipe. If the value of any parameter contains semicolon, colon, or pipe as its part, such value must be double quoted.	

Advanced Description

Path to Control Script

Name of command file containing the `\copy` statement, including path.

- If it is not set, command file is created in Clover temporary directory and it is deleted after the load finishes.
- If it is set, but the specified command file does not exist, temporary file is created with the specified name and path and it is not deleted after the load finishes.

- If it is set and the specified command file exists, this file is used instead of the file created by Clover. The file is not deleted after the load finishes.

Loader Input File

Name of input file to be loaded, including path.

- If input port is connected and this file is not set, no temporary file is created. Data is read from the edge and loaded into database directly.
- If it is set, specified file is created. It is not deleted after data is loaded and it is overwritten on each graph run.
- If input port is not connected, this file must exist and must contain data that should be loaded into database. It is not deleted nor overwritten on another graph run.

QuickBaseImportCSV

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

QuickBaseImportCSV adds and updates **QuickBase** online database table records.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
QuickBaseImportCSV	QuickBase	1	0-2	✖	✖	✖	✖

Abstract

QuickBaseImportCSV receives data records through the input port and writes them into **QuickBase** online database (http://quickbase.intuit.com). Generates record IDs for successfully written records and sends them out through the first optional output port if connected. The first field on this output port must be of string data type. Into this field, generated record IDs will be written. Information about rejected data records can be sent out through the optional second port if connected.

This component wrapps the `API_ImportFromCSV` HTTP interaction (http://www.quickbase.com/api-guide/importfromcsv.html).

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✖	for accepted data records	string field for the table *Record ID#* field values of the imported records
Output	1	✖	for rejected data records	input metadata enriched by up to three Error Fields for QuickBaseImportCSV (p. 407)

Table 45.5. Error Fields for QuickBaseImportCSV

Field number	Field name	Data type	Description
optional[1]	specified in the **Error code output field**	integer | long	error code
optional[1]	specified in the **Error message output field**	string	error message
optional[1]	specified in the **Batch number output field**	integer | long	index (starting from 1) of the failed batch

[1] The error fields must be placed behind the input fields.

QuickBaseImportCSV Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see Chapter 22, QuickBase Connections (p. 164)	
Table ID	✔	ID of the table in the QuickBase application data records are to be written into (see the `application_stats` for getting the table ID)	
Batch size		The maximum number of records in one batch	100 (default) | 1-N
Clist		A period-delimited list of table *field_id*s to which the input data columns map. The order is preserved. Thus, enter a `0` for columns not to be imported. If not specified, the database tries to add unique records. It must be set if editing records. The input data must include a column that contains the record ID for each record that you are updating.	
Error code output field		name of the error metadata field for storing the error code, see Error Fields for QuickBaseImportCSV (p. 407)	
Error message output field		name of the error metadata field for storing the error message, see Error Fields for QuickBaseImportCSV (p. 407)	
Batch number output field		name of the error metadata field for storing the index of the corrupted batch, see Error Fields for QuickBaseImportCSV (p. 407)	

QuickBaseRecordWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

QuickBaseRecordWriter writes data into **QuickBase** online database.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
QuickBaseRecordWriter	QuickBase	1	0-1	✖	✖	✖	✖

Abstract

QuickBaseRecordWriter receives data records through the input port and writes them to a **QuickBase** online database (http://quickbase.intuit.com).

This component wraps the `API_AddRecord` HTTP interaction (http://www.quickbase.com/api-guide/add_record.html).

If the optional output port is connected, rejected records along with the information about the error are sent out through it.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✖	for rejected data records	input metadata enriched by up to two Error Fields for QuickBaseRecordWriter (p. 409)

Table 45.6. Error Fields for QuickBaseRecordWriter

Field number	Field name	Data type	Description
optional[1]	specified in the **error code output field**	integer | long	Error code
optional[1]	specified in the **error message output field**	string	Error message

[1] The error fields must be placed behind the input fields.

QuickBaseRecordWriter Attributes

Attribute	Req	Description	Possible values
Basic			
QuickBase connection	✔	ID of the connection to the QuickBase online database, see Chapter 22, QuickBase Connections (p. 164)	
Table ID	✔	ID of the table in the QuickBase application data records are to be written into (see the `application_stats` for getting the table ID)	
Mapping	✔	List of database table *field_id*s separated by a semicolon the metadata field values are to be written to.	
Error code output field		Name of the field the error code will be stored in, see Error Fields for QuickBaseRecordWriter (p. 409)	
Error message output field		Name of the field the error message will be stored in, see Error Fields for QuickBaseRecordWriter (p. 409)	

StructuredDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

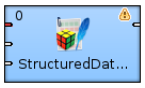
StructuredDataWriter writes data to files with user-defined structure.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
StructuredDataWriter	structured flat file	1-3	0-1	no	no	no	no

Abstract

StructuredDataWriter writes data to files (local or remote, delimited, fixed-length, or mixed) with user-defined structure. It can also compress output files, write data to console, output port, or dictionary.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	Records for body	Any
	1	no	Records for header	Any
	2	no	Records for footer	Any
Output	0	no	For port writing. See Writing to Output Port (p. 265).	One field (`byte`, `cbyte`, `string`).

StructuredDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	Attribute specifying where received data will be written (flat file, console, output port, dictionary). See Supported File URL Formats for Writers (p. 263).	
Charset		Encoding of records written to the output.	ISO-8859-1 (default) | <other encodings>
Append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Body mask		Mask used to write the body of the output file(s). It can be based on the records received through the first input port. See Masks and Output File Structure (p. 412) for more information about definition of **Body mask** and resulting output structure.	Default Body Structure (p. 413) (default) | user-defined
Header mask	1)	Mask used to write the header of the output file(s). It can be based on the records received through the second input port. See Masks and Output File Structure (p. 412) for more information about definition of **Header mask** and resulting output structure.	empty (default) | user-defined
Footer mask	2)	Mask used to write the footer of the output file(s). It can be based on the records received through the third input port. See Masks and Output File Structure (p. 412) for more information about definition of **Footer mask** and resulting output structure.	empty (default) | user-defined
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Records per file		Maximum number of records to be written to one output file.	1-N
Bytes per file		Maximum size of one output file in bytes.	1-N
Number of skipped records		Number of records to be skipped. See Selecting Output Records (p. 270).	0-N
Max number of records		Maximum number of records to be written to all output files. See Selecting Output Records (p. 270).	0-N
Partition key		Key whose values define the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition lookup table	1)	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition file tag		By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 270) for more information.	Number file tag (default) | Key file tag
Partition output fields	1)	Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	

Attribute	Req	Description	Possible values
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 270) for more information.	

Legend:

1) Must be specified if second input port is connected. However, does not need to be based on input data records.

2) Must be specified if third input port is connected. However, does not need to be based on input data records.

Advanced Description

Masks and Output File Structure

- **Output File Structure**

  1. Output file consists of header, body, and footer, in this order.
  2. Each of them is defined by specifying corrresponding mask.
  3. After defining the mask, the mask content is written repeatedly, one mask is written for each incoming record.
  4. However, if the **Records per file** attribute is defined, the output structure is distributed among various output files, but this attribute applies for **Body mask** only. Header and footer are the same for all output files.

- **Defining a Mask**

  Body mask, **Header mask**, and **Footer mask** can be defined in the **Mask** dialog. This dialog opens after clicking corresponding attribute row. In its window, you can see the **Metadata** and **Mask** panes. At the bottom, there is a **Auto XML** button.

  You can define the mask either without field values or with field values.

  Field values are expressed using field names preceded by dollar sign.

  If you click the **Auto XML** button, a simple XML structure appears in the **Mask** pane.

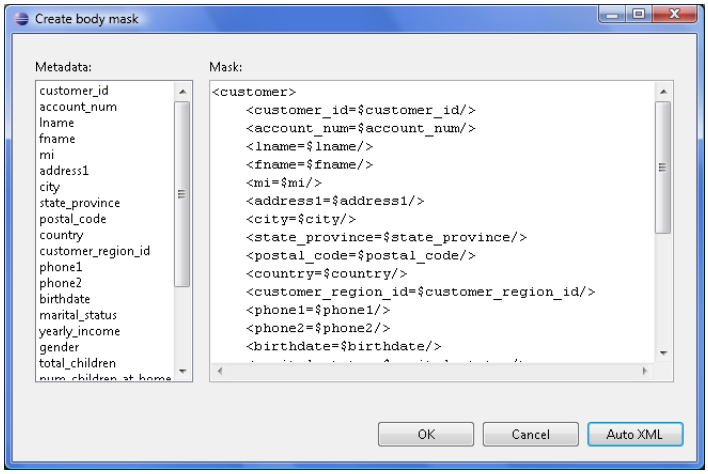


Figure 45.7. Create Mask Dialog

You only need to remove the fields you do not want to save to the output file and you can also rename the suggested left side of the matchings. These have the form of matchings like this: `<sometag=`

`$metadatafield`/>. By default after clicking the **Auto XML** button, you will obtain the XML structure containing expressions like this: `<metadatafield=$metadatafield`/>. Left side of these matchings can be replaced by any other, but the right side must remain the same. You must not change the field names preceded by a dollar sign on the right side of the matchings. They represent the values of fields.

Remember that you do not need to use any XML file as a mask. The mask can be of any other structure.

- **Default Masks**

  1. Default **Header mask** is empty. But it must be defined if second input port is connected.

  2. Default **Footer mask** is empty. But it must be defined if third input port is connected.

  3. Default **Body mask** is empty. However, the resulting default body structure looks like the following:

     ```
     <   recordName   field1name=field1value   field2name=field2value   ...
     fieldNname=fieldNvalue />
     ```

     This structure is written to the output file(s) for all records.

     If **Records per file** is set, only the specified number of records are used for body in each output file at most.

Trash

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

Trash discards data.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
Trash	none	1	0	no	no	no	no

Abstract

Trash discards data. For debugging purpose it can write its data to a file (local or remote), or console. Multiple inputs can be connected for improved graph legibility.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	1-n	yes	For received data records	Any

Trash Attributes

Attribute	Req	Description	Possible values
Basic			
Debug print		By default, all records are discarded. If set to `true`, all records are written to the debug file (if specified), or console. You do not need to switch **Log level** from its default value (`INFO`). This mode is only supported when single input port is connected..	false (default) | true

Attribute	Req	Description	Possible values
Debug file URL		Attribute specifying debug output file. See Supported File URL Formats for Writers (p. 263). If path is not specified, the file is saved to the `${PROJECT}` directory. You do not need to switch **Log level** from its default value (`INFO`).	
Debug append		By default, new records overwrite the older ones. If set to `true`, new records are appended to the older records stored in the output file(s).	false (default) | true
Charset		Encoding of debug output.	ISO-8859-1 (default) | <other encodings>
Advanced			
Print trash ID		By default, trash ID is not written to debug output. If set to `true`, ID of the **Trash** is written to debug file, or console. You do not need to switch **Log level** from its default value (`INFO`).	false (default) | true
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Mode		Trash can run in either Performace or Validate records modes. In Performance mode the raw data is discarded, in Validate records Trash simulates a writer - attempting to deserialize the inputs.	Performance (default) | Validate records

UniversalDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

UniversalDataWriter is a terminative component that writes data to flat files.

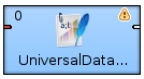
Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
UniversalDataWriter	flat file	1	0-1	✖	✖	✖	✖

Abstract

UniversalDataWriter formats all records from the input port to delimited, fixed-length, or mixed form and writes them to specified flat file(s), such as CSV (comma-separated values) or text file(s). The output data can be stored localy or uploaded via a remote transfer protocol. Also writing ZIP and TAR archives is supported.

The component can write a single file or partitioned collection of files. The type of formatting is specified in metadata for the input port data flow.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for received data records	any
Output	0	✖	for port writing. See Writing to Output Port (p. 265).	include specific `byte`/ `cbyte`/ `string` field

UniversalDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	✔	where the received data to be written (flat file, console, output port, dictionary) specified, see Supported File URL Formats for Writers (p. 263).	
Charset		character encoding of records written to the output	ISO-8859-1 (default) | <other encodings>
Append		If records are printed into an existing non-empty file, they replace the older ones by default (`false`). If set to `true`, new records are appended to the end of the existing output file(s) content.	false (default) | true
Quoted strings		if switched to `true`, all data field values except from `byte` and `cbyte` will be double quoted	false (default) | true
Quote character		Specifies which kind of quotes will be permitted in **Quoted strings**.	both (default) | " | '
Advanced			
Create directories		if set to `true`, non-existing directories in the **File URL** attribute path are created	false (default) | true
Write field names		Field names are not written to the output file(s) defaultly. If set to `true`, names of individual fields are printed to the output as the first row.	false (default) | true
Records per file		Maximum number of records to be written to each output file. If specified, the dollar sign(s) $ (number of digits placeholder) must be part of the file name mask, see Supported File URL Formats for Writers (p. 263)	1 - N
Bytes per file		Maximum size of each output file in bytes. If specified, the dollar sign(s) $ (number of digits placeholder) must be part of the file name mask, see Supported File URL Formats for Writers (p. 263) To avoid splitting a record into two files, max size can be slightly overreached.	1 - N
Number of skipped records		how many records/rows to be skipped before writting the first record to the output file, see Selecting Output Records (p. 270).	0 (default) - N
Max number of records		how many records/rows to be written to all output files, see Selecting Output Records (p. 270).	0-N
Exclude fields		Sequence of field names separated by semicolon that will not be written to the output. Can be used when the same fields serve as a part of **Partition key**.	
Partition key	2)	sequence of field names separated by semicolon defining the records distribution into different output files - records with the same **Partition key** are written to the same output file. According to the selected **Partition file tag** use the proper placeholder ($ or #) in the file name mask, see Partitioning Output into Different Output Files (p. 270)	

Attribute	Req	Description	Possible values
Partition lookup table	[1)]	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition file tag	[2)]	By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 270) for more information.	Number file tag (default) | Key file tag
Partition output fields	[1)]	Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 270) for more information.	

[2)] Either both or neither of these attributes must be specified

[1)] Either both or neither of these attributes must be specified

Tips & Tricks

- *Field size limitation 1:* **UniversalDataWriter** can write fields of a size up to 4kB. To enable bigger fields to be written into a file, increase the DataFormatter.FIELD_BUFFER_LENGTH property, see Changing Default CloverETL Settings (p. 76). Enlarging this buffer does not cause any significant increase of the graph memory consumption.
- *Field size limitation 2:* Another way how to solve the big-fields-to-be-written issue is the utilization of the Normalizer (p. 484) component that can split large fields into several records.

XLSDataWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 263).

Short Summary

XLSDataWriter writes data to XLS or XLSX files.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
XLSDataWriter	XLS(X) file	1	0-1	no	no	no	no

Abstract

XLSDataWriter writes data to XLS or XLSX files (local or remote). It can also compress output files, write data to console, output port, or dictionary.

Note

Remember that **XLSDataWriter** stores all data in memory and has high memory requirements.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For received data records	Any
Output	0	no	For port writing. See Writing to Output Port (p. 265).	One field (`byte`, `cbyte`, `string`).

XLSDataWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Type of parser		Specifies the formatter to be used. By default, component guesses according the extension (XLS or XLSX).	Auto (default) | XLS | XLSX
File URL	yes	Attribute specifying where received data will be written (XLS or XLSX file, console, output port, dictionary). See Supported File URL Formats for Writers (p. 263).	
Sheet name	1)	The name of the sheet to which records are written. If not set, a sheet with default name is created and inserted as the last of all sheets. Can also be a sequence of field names, each of them is prefixed by dollar sign, separated by semicolon, colon, or pipe. Thus, to different values of such sequence, different sheets are created. For example, `$Country;$City`. This way, different countries along with cities create different sheets. Remember that the sheets are created in the order of record values. For this reason, first you should sort records on this field and only then write the records to the output XLS(X) file.	Sheet[0-9]+ (The number in the name is the number of the last sheet with the same name structure: Sheet[0-9]+.)
Sheet number	1)	The number of the sheet to which records are written. If not set, a sheet with default name is created and inserted as the last of all sheets.	0-N
Charset		Encoding of records written to the output.	ISO-8859-1 (default) | <other encodings>
Append to the sheet		By default, new records overwrite the older ones in one sheet. If set to `true`, new records are appended to the older records stored in the sheet(s).	false (default) | true
Metadata row		Number of the row to which the field names should be written. By default, field names are written to the header of the sheet.	0 (default) | 1-N
Advanced			
Create directories		By default, non-existing directories are not created. If set to `true`, they are created.	false (default) | true
Start row		Row of the sheet starting from which the records are written. By default, records are written to the sheet starting from the first row.	1 (default) | 2-N
Start column		Column of the sheet starting from which the records are written. By default, records are written to the sheet starting from the first column.	A (default) | B-*
Records per file		Maximum number of records to be written to one output file.	1-N
Number of skipped records		Number of records to be skipped. See Selecting Output Records (p. 270).	0-N
Max number of records		Maximum number of records to be written to all output files. See Selecting Output Records (p. 270).	0-N
Exclude fields		Sequence of field names separated by semicolon that should not be written to the output. Can be used when the same fields serve as a part of **Partition key** or when the field(s) is(are) selected as **Sheet name** as shown above.	

Attribute	Req	Description	Possible values
Partition key		Key whose values define the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition lookup table	2)	ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition file tag		By default, output files are numbered. If it is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 270) for more information.	Number file tag (default) | Key file tag
Partition output fields	2)	Fields of **Partition lookup table** whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition unassigned file name		Name of the file into which the unassigned records should be written if there are any. If not specified, data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 270) for more information.	

Legend:

1) One of these attributes can be specified. **Sheet name** has higher priority. Before creation of output file, only **Sheet name** can be set. If neither of these is specified, new sheet is created on each graph run.

2) Either both or neither of these attributes must be specified.

Important

Remember that if you want to write data into multiple sheets of the same file, you must write data to each sheet in a separate phase!

XMLWriter

We assume you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 38, Common Properties of Writers (p. 262)

If you want to find the appropriate **Writer** for your purpose, see Writers Comparison (p. 263).

Short Summary

XMLWriter formats records into XML files.

Component	Data output	Input ports	Output ports	Transformation	Transf. required	Java	CTL
XMLWriter	XML file	1-n	0-1	no	no	no	no

Abstract

XMLWriter receives input data records, joins them and formats them into a user-defined XML structure. Even complex mapping is possible and thus the component can create arbitrary nested XML structures.

XMLWriter combines streamed and cached data processing depending on the complexity of the XML structure. This allows to produce XML files of arbitrary size in most cases. However, the output can be partitioned into multiple chunks - i.e. large difficult-to-process XML files can be easily split into multiple smaller chunks.

Standard output options are available: files, compressed files, the console, an output port or a dictionary.

The component needs Eclipse v. 3.6 or higher to run.

Icon

Ports

Port type	Port number	Required	Description	Metadata
Input	0-N	At least one	Input records to be joined and mapped into an XML file	Any (each port can have different metadata)
Output	0	no	For port writing, see Writing to Output Port (p. 265).	One field (`byte`, `cbyte`, `string`).

XMLWriter Attributes

Attribute	Req	Description	Possible values
Basic			
File URL	yes	The target file for the output XML. See Supported File URL Formats for Writers (p. 263).	
Charset		The encoding of an output file generated by XMLWriter.	ISO-8859-1 (default) | <other encodings>
Mapping	1)	Defines how input data is mapped onto an output XML. See Advanced Description (p. 424) for more information.	
Mapping URL	1)	External text file containing the mapping definition. See Creating the Mapping - Mapping Ports and Fields (p. 432) and Creating the Mapping - Source Tab (p. 435) for the mapping file format. Put your mapping to an external file if you want to share a single mapping among multiple graphs.	
XML Schema		The path to an XSD schema. If **XML Schema** is set, the whole mapping can be automatically pre-generated from the schema. To learn how to do it, see Creating the Mapping - Using Existing XSD Schema (p. 435). The schema has to be placed in the `meta` folder.	none (default) | any valid XSD schema
Advanced			
Create directories		If `true`, non existing directories included in the **File URL** path will be automatically created.	false (default) | true
Omit new lines wherever possible		By default, each element is written to a separate line. If set to `true`, new lines are omitted when writing data to the output XML structure. Thus, all XML tags are on one line only.	false (default) | true
Directory for temporary files		Select a path to a directory where temporary files created during the mapping are stored.	(default system temp directory) | any other directory
Cache size		A size of of the database used when caching data from ports to elements (the data is first processed then written). The larger your data is, the larger cache is needed to maintain fast processing.	default: `auto` | e.g. 300MB, 1GB etc.
Sorted input		Tells XMLWriter whether the input data is sorted. Setting the attribute to true declares you want to use the sort order defined in **Sort keys**, see below.	false(default)|true
Sort keys		Tells XMLWriter how the input data is sorted, thus enabling streaming (see Creating the Mapping - Mapping Ports and Fields (p. 432)). The sort order of fields can be given for each port in a separate tab. Working with **Sort keys** has been described in Sort Key (p. 232).	
Records per file		Maximum number of records that are written to a single file. See Partitioning Output into Different Output Files (p. 270)	1-N
Max number of records		Maximum number of records written to all output files. See Selecting Output Records (p. 270).	0-N

Attribute	Req	Description	Possible values
Partition key		A key whose values control the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition lookup table		The ID of a lookup table. The table serves for selecting records which should be written to an output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition file tag		By default, output files are numbered. If this attribute is set to `Key file tag`, output files are named according to the values of **Partition key** or **Partition output fields**. See Partitioning Output into Different Output Files (p. 270) for more information.	Number file tag (default) | Key file tag
Partition output fields		Fields of **Partition lookup table** whose values serve for naming output file(s). See Partitioning Output into Different Output Files (p. 270) for more information.	
Partition unassigned file name		The name of a file that the unassigned records should be written into (if there are any). If it is not given, the data records whose key values are not contained in **Partition lookup table** are discarded. See Partitioning Output into Different Output Files (p. 270) for more information.	

Legend:

1) One of these attributes has to be specified. If both are defined, **Mapping URL** has a higher priority.

Advanced Description

XMLWriter's core part is the mapping editor that lets you visually map input data records onto an XML tree structure (see Figure 45.8, “Mapping Editor” (p. 425)). By dragging the input ports or fields onto XML elements and attributes you map them, effectively populating the XML structure with data.

What is more, the editor gives you direct access to the mapping source where you can virtually edit the output XML file as text. You use special directives to populate the XML with CloverETL data there (see Figure 45.16, Source tab in Mapping editor. (p. 435)).

The XML structure can be provided as an XSD Schema (see the **XML Schema** attribute) or you can define the structure manually from scratch.

You can access the visual mapping editor clicking the "..." button of the **Mapping** attribute.

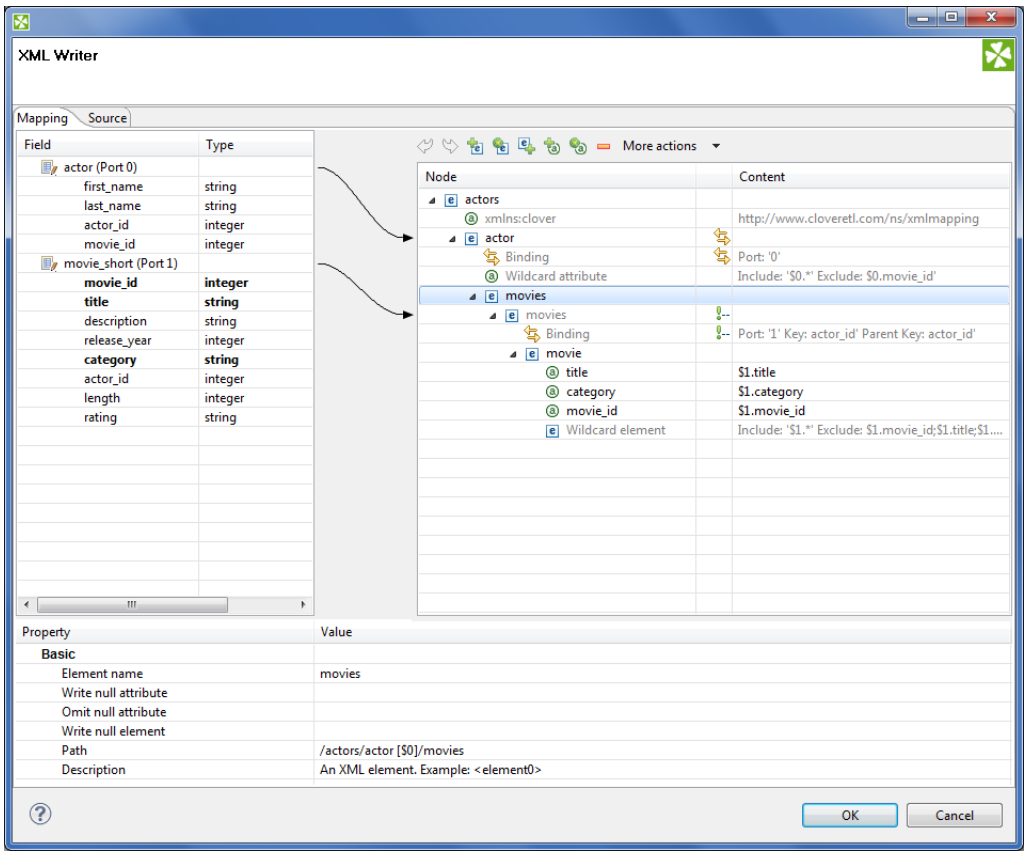


Figure 45.8. Mapping Editor

When inside the editor, notice its two main tabs in the upper left corner of the window:

- **Mapping** - enables you to design the output XML in a visual environment
- **Source** - that is where you can directly edit the XML mapping source code

Changes made in the Mapping tab take immediate effect in the Source tab and vice versa. In other words, both editor tabs allow making equal changes.

When you switch to the **Mapping** tab, you will notice there are three basic parts of the window:

1. Left hand part with **Field** and **Type** columns - represents ports of the input data. Ports are represented by their symbolic names in the **Field** column. Besides the symbolic name, ports are numbered starting from $0 for the first port in the list. Underneath each port, there is a list of all its fields and their data types. Please note neither port names, field names nor their data types can be edited in this section. They all depend merely on the metadata on the XMLWriter's input edge.

2. Right hand part with **Node** and **Content** columns - the place where you define the structure of output elements , attributes , wildcard elements or wildcard attributes and namespaces. In this section, data can be modified either by double-clicking a cell in the **Node** or the **Content** column. The other option is to click a line and observe its **Property** in the bottom part section of the window.

3. Bottom part with **Property** and **Value** columns - for each selected Node, this is where its properties are displayed and modified.

Creating the Mapping - Designing New XML Structure

The mapping editor allows you to start from a completely blank mapping - first designing the output XML structure and then mapping your input data to it. The other option is to use your own XSD schema, see Creating the Mapping - Using Existing XSD Schema (p. 435).

As you enter a blank mapping editor, you can see input ports on the left hand side and a root element on the right hand side. The point of mapping is first to design the output XML structure on the right hand side (data destination). Second, you need to connect port fields on the left hand side (data source) to those pre-prepared XML nodes (see Creating the Mapping - Mapping Ports and Fields (p. 432)).

Let us now look on how to build a tree of nodes the input data will flow to. To add a node, right-click an element, click **Add Child** or **Add Property** and select one of the available options:

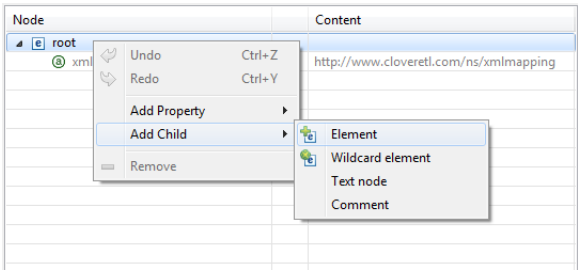


Figure 45.9. Adding Child to Root Element.

Important

For a closer look on adding nodes, manipulating them and using smart drag and drop mouse techniques, see Working with Nodes (p. 431).

Namespace

Adds a **Namespace** as a new `xmlns:prefix` attribute of the selected element. Declaring a Namespace allows you to use your own XML tags. Each Namespace consists of a prefix and an URI. In case of XMLWriter mapping, the root element has to declare the `clover` namespace, whose URI is `http://www.cloveretl.com/ns/xmlmapping`. That grants you access to all special XML mapping tags. If you switch to the **Source** tab, you will easily recognise those tags as they are distinct by starting with `clover:`, e.g. `clover:inport="2"`. Keep in mind that no XML tag beginning with the `clover:` prefix is actually written into the output XML.

Wildcard attribute

Adds a special directive to populate the element with attributes based on **Include** / **Exclude** wildcard patterns instead of mapping these attributes explicitly. This feature is useful when you need to retain metadata independence.

Attribute names are generated from field names of the respective metadata. Syntax: use `$portNumber.field` or `$portName.field` to specify a field, use `*` in the field name for "any string". Use ; to specify multiple patterns.

Example 45.4. Using Expressions in Ports and Fields

`$0.*` - all fields on port 0

`$0.*;$1.*` - all fields on ports 0 and 1 combined

`$0.address*` - all fields beginning with the "address" prefix, e.g. $0.addressState, $0.addressCity, etc.

`$child.*` - all fields on port `child` (the port is denoted by its name instead of an explicit number)

There are two main properties in a Wildcard attribute. At least one of them has to be always set:

- **Include** - defines the inclusion pattern, i.e. which fields should be included in the automatically generated list. That is defined by an expression whose syntax is `$port.field`. A good use of expressions explained above can be made here. **Include** can be left blank provided **Exclude** is set (and vice versa). If **Include** is blank, XMLWriter lets you use all ports that are connected to nodes up above the current element (i.e. all its parents) or to the element itself.

- **Exclude** - lets you specify the fields that you explicitly do not want in the automatically generated list. Expressions can be used here the same way as when working with **Include**.

Example 45.5. Include and Exclude property examples

1. **Include** = `$0.i*`

Exclude = `$0.index`

Include takes all fields from port $0 starting with the 'i' character. Exclude then removes the `index` field of the same port.

2. **Include** = (blank)

Exclude = `$1.*;$0.id`

Include is not given so all ports connected to the node or up above are taken into consideration. Exclude then removes all fields of port $1 and the `id` field of port $0. Condition: ports $0 and $1 are connected to the element or its parents.

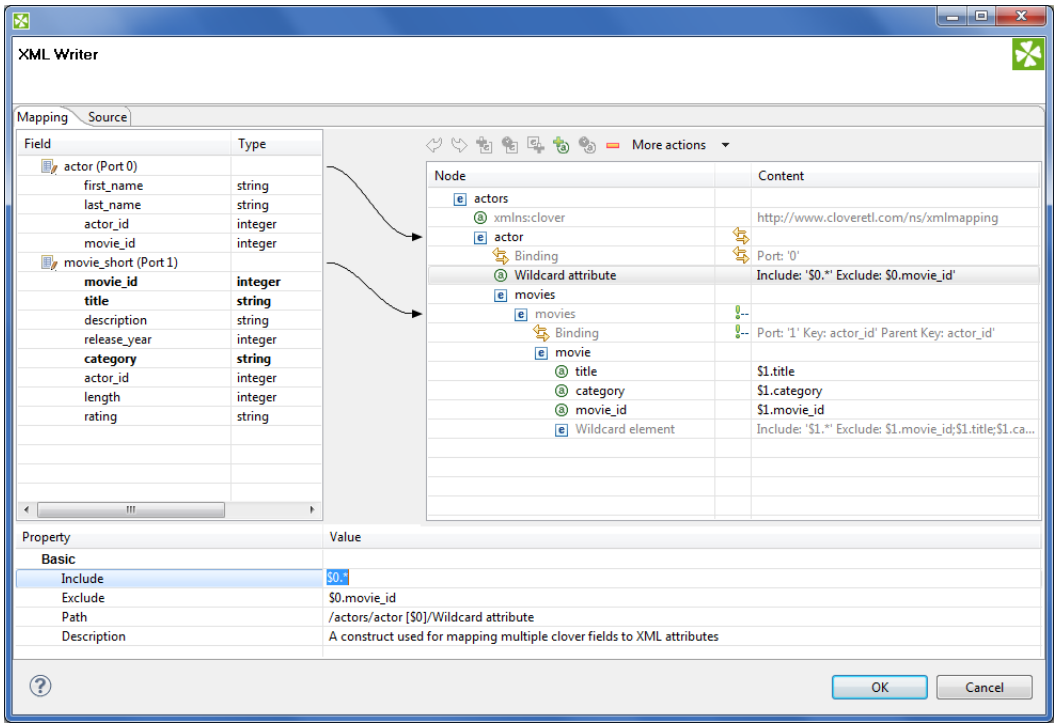


Figure 45.10. Wildcard attribute and its properties.

Attribute

Adds a single attribute to the selected element. Once done, the Attribute name can be changed either by double-clicking it or editing **Attribute name** at the bottom. The attribute **Value** can either be a fixed string or a field value that you map to it. You can even combine static text and multiple field mappings. See example below.

Example 45.6. Attribute value examples

`Film` - the attribute's value is set to the literal string "Film"

`$1.category` - the `category` field of port $1 becomes the attribute value

`ID: '{$1.movie_id}'` - produces "ID: '535'", "ID: '536'" for `movie_id` field values 535 and 536 on port $1. Please note the curly brackets that can optionally delimit the field identifier.

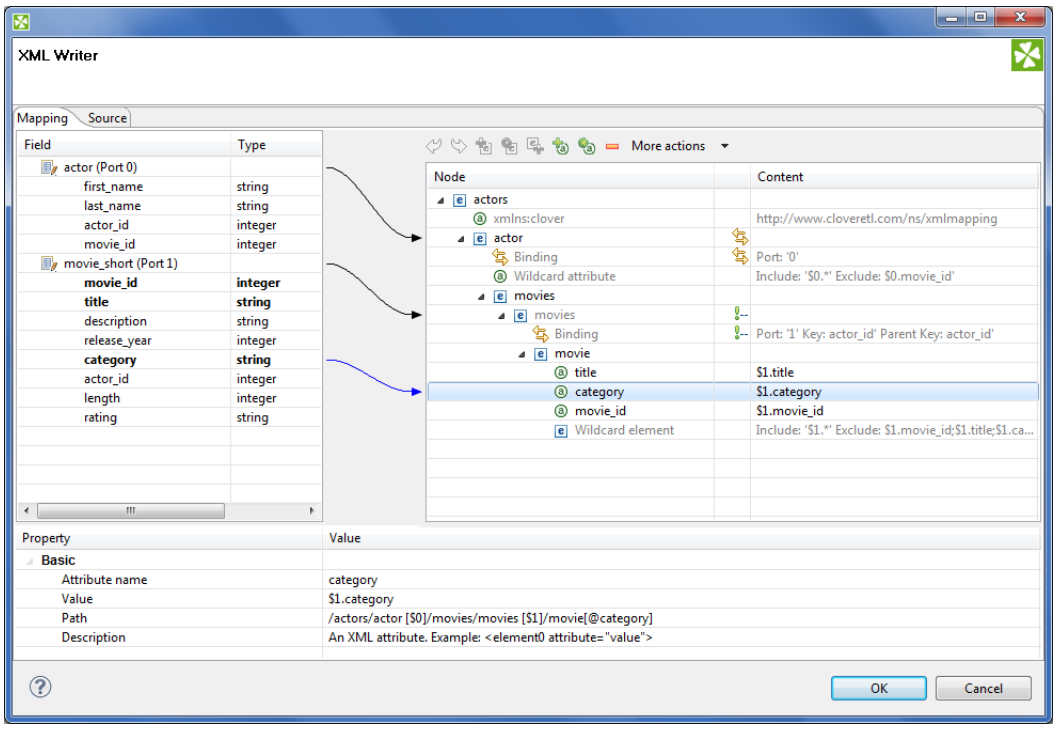


Figure 45.11. Attribute and its properties.

Path and **Description** are common properties for most nodes. They both provide a better overview for the node. In **Path**, you can observe how deep in the XML tree a node is located.

Element

Adds an element as a basic part of the output XML tree.

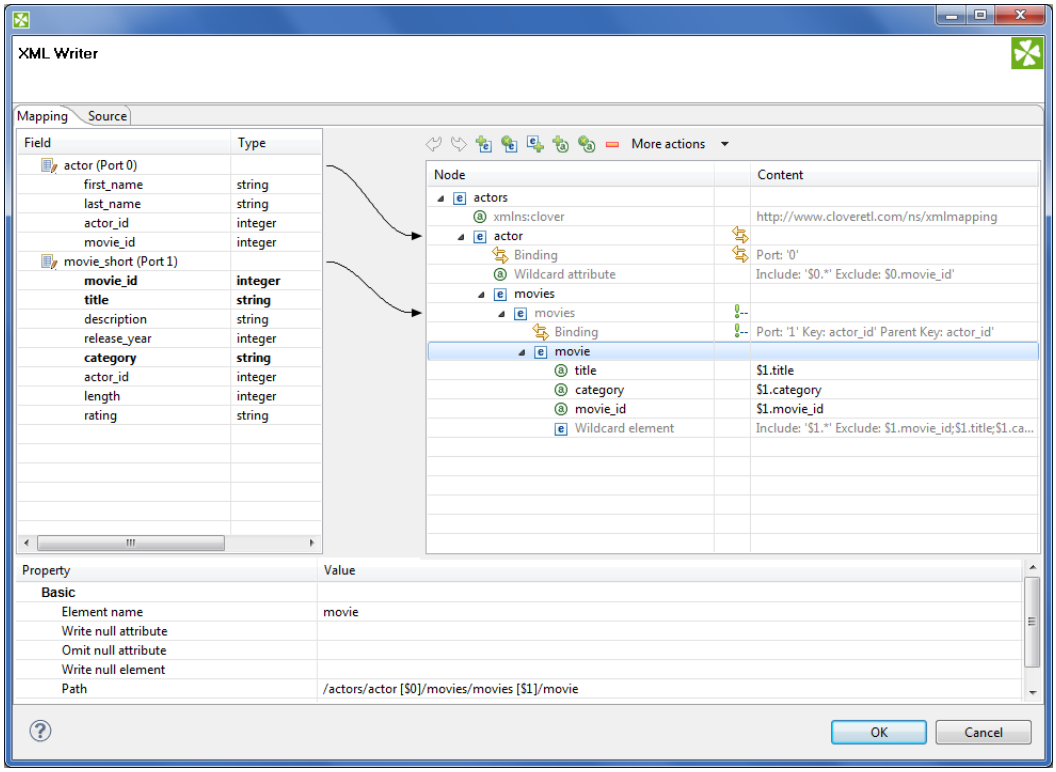


Figure 45.12. Element and its properties.

Depending on an element's location in the tree and ports connected to it, the element can have these properties:

- **Element name** - name of the element as it will appear in the output XML.
- **Value** - element value. You can map a field to an element and it will populate its value. If on the other hand you map a port to an element, you will create a **Binding** (see Creating the Mapping - Mapping Ports and Fields (p. 432)). If **Value** is not present, right-click the element and choose **Add Child - Text node** The element then gets a new field representing its text value. The newly created Text node cannot be left blank.
- **Write null attribute** - by default, attributes with values mapping to NULL will not be put to the output. However, here you can explicitly list names of attributes that will always appear in the output.

  Example 45.7. Writing null attribute

  Let us say you have an element <date> and its attribute "time" that maps to input port 0, field `time` (i.e. <date time="$0.time"/>). For records where the `time` field is empty (null), the default output would be:

```
<date/>
```

  Setting **Write null attribute** to `time` produces:

```
<date time="" />
```

- **Omit null attribute** - in contrast to **Write null attribute**, this one specifies which of the current element's attributes will NOT be written if their values are null. Obviously, such behaviour is default. The true purpose of **Omit null attribute** lies in wildcard expressions in combination with **Write null attribute**.

  Example 45.8. Omitting Null Attribute

  Let us say you have an element with a **Wildcard attribute**. The element is connected to port 2 and its fields are mapped to the wildcard attribute, i.e. **Include**=$2.*. You know that some of the fields contain no data. You would like to write SOME of the empty ones, e.g. `height` and `width`. To achieve that, click the element and set:

  Write null attribute=$2.* - forces writing of all attributes although they are null

  Omit null attribute=$2.height;$2.width - only these attributes will not be written

- **Hide** - in elements having a port connected, set **Hide** to `true` to force the following behaviour: the selected element is not written to the output XML while all its children are. By default, the property is set to `false`. Hidden elements are displayed with a grayish font in the Mapping editor.

Example 45.9. Hide Element

Imagine an example XML:

```
<address>
 <city>Atlanta</city>
 <state>Georgia</state>
</address>
<address>
 <city>Los Angeles</city>
 <state>California</state>
</address>
```

Then hiding the `address` element produces:

```
<city>Atlanta</city>
<state>Georgia</state>
<city>Los Angeles</city>
<state>California</state>
```

- **Partition** - by default, partitioning is done according to the first and topmost element that has a port connected to it. If you have more such elements, set **Partition** to `true` in one of them to distinguish which element governs the partitioning. Please note partitioning can be set only once. That is if you set an element's **Partition** to `true`, you should not set it in either of its subelements (otherwise the graph fails). For a closer look on partitioning, see Partitioning Output into Different Output Files (p. 270).

Example 45.10. Partitioning According to Any Element

In the mapping snippet below, setting **Partition** to `true` on the <invoice> element produces the following behaviour:

<person> will be repeated in every file

<invoice> will be divided (partitioned) into several files

```
<person clover:inPort="0">
 <firstname> </firstname>
 <surname> </surname>
</person>

<invoice clover:inPort="1" clover:partition="true""">
 <customer> </customer>
 <total> </total>
</invoice>
```

Wildcard element

Adds a set of elements. The **Include** and **Exclude** properties influence which elements are added and which not. To learn how to make use of the `$port.field` syntax, please refer to **Wildcard attribute**. Rules and examples described there apply to **Wildcard element** as well. What is more, **Wildcard element** comes with two additional properties, whose meaning is closely related to the one of **Write null attribute** and **Omit null attribute**:

- **Write null element** - use the `$port.field` syntax to determine which elements are written to the output despite their having no content. By default, if an element has no value, it is not written. **Write null element** does not have to be entered on condition that the **Omit null element** is given. Same as in **Include** and **Exclude**, all ports connected to the element or up above are then available. See example below.

- **Omit null element** - use the `$port.field` syntax to skip blank elements. Even though they are not written by default, you might want to use **Omit null element** to skip the blank elements you previously forced to be written in **Write null element**. Alternatively, using **Omit null element** only is also possible. That means you exclude blank elements coming from all ports connected to the element or above.

Example 45.11. Writing and omitting blank elements

Say you aim to create an XML file like this:

```
<person>
 <firstname>William</firstname>
 <middlename>Makepeace</middlename>
 <surname>Thackeray</surname>
</person>
```

but you do not need to write the element representing the middle name for people without it. What you need to do is to create a **Wildcard element**, connect it to a port containing data about people (e.g. port $0 with a `middle` field), enter the **Include** property and finally set:

Write null element = `$0.*`

Omit null element = `$0.middle`

As a result, first names and surnames will always be written (even if blank). Middle name elements will not be written if the `middle` field contains no data.

Text node

Adds content of the element. It is displayed at the very end of an uncollapsed element, i.e. always behind its potential Binding, Wildcard attributes or Attributes. Once again, its value can either be a fixed string, a port's field or their combination.

Comment

Adds a comment. This way you can comment on every node in the XML tree to make your mapping clear and easy-to-read. Every comment you add is displayed in the Mapping editor only. What is more, you can have it written to the output XML file setting the comment's **Write to the output** to true. Examine the **Source** tab to see your comment there, for instance:

```
<!-- clover:write This is my comment in the Source tab. It will be written to the output
XML because I set its 'Write to output' to true. There is no need to worry about the
"clover:write" directive at the beginning as no attribute/element starting with
the "clover" prefix is put to the output.
-->
```

Working with Nodes

Having added the first element, you will notice that every element except for the root provides other options than just **Add Child** (and **Add Property**). Right-click an element to additionally choose from **Add Sibling Before** or **Add Sibling After**. Using these, you can have siblings added either before or after the currently selected element.

Besides the right-click context menu, you can use toolbar icons located above the XML tree view.

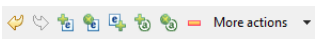


Figure 45.13. Mapping editor toolbar.

The toolbar icons are active depending on the selected node in the tree. Actions you can do comprise:

- **Undo** and **Redo** the last action performed.
- **Add Child Element** under the selected element.
- **Add (child) Wildcard Element** under the selected element.
- **Add Sibling Element After** the selected element.
- **Add Child Attribute** to the selected element
- **Add Wildcard Attribute** to the selected element.
- **Remove** the selected node
- **More actions** - besides other actions described above, you can especially **Add Sibling Before** or **Add Sibling After**

When building the XML tree from scratch (see Creating the Mapping - Designing New XML Structure (p. 425)) why not make use of these tips saving mouse clicks and speeding up your work:

- drag a port and drop it onto an element - you will create a **Binding**, see Creating the Mapping - Mapping Ports and Fields (p. 432)
- drag a field and drop it onto an element - you will add a child element of the same name as the field
- drag an available field (or even more fields) onto an element - you will create a subelement whose name is the field's name. Simultaneously, the element's content is set to `$portNumber.fieldName`.
- drag one or more available ports and drop it onto an element with a **Binding** - you will create a **Wildcard element** whose **Include** will be set to `$portNumber.*`
- combination of the two above - drag a port and a field (even from another port) onto an element with a **Binding** - the port will be turned to **Wildcard element** (**Include**=`$portNumber.*`), while the field becomes a subelement whose content is `$portNumber.fieldName`
- drag an available port/field and drop it onto a Wildcard element/attribute - the port or field will be added to the **Include** directive of the Wildcard element/attribute. If it is a port, it will be added as $0.* (example for port 0). If it is a field, it will be added as $0.priceTotal (example for port 0, field priceTotal).

Every node you add can later be moved in the tree by a simple drag and drop using the left mouse button. That way you can re-arrange your XML tree any way you want. Actions you can do comprise:

- drag an (wildcard) element and drop it on another element - the (wildcard) element becomes a subelement
- drag an (wildcard) attribute and drop it on an element - the element now has the (wildcard) attribute
- drag a text node and drop it on an element - the element's value is now the text node
- drag a namespace and drop it on an element - the element now has the namespace

Removing nodes (such as elements or attributes) in the Mapping editor is also carried out by pressing Delete or right-clicking the node and choosing **Remove**. To select more nodes at once, use **Ctrl**+**click** or **Shift**+**click** .

Any time during your work with the mapping editor, press **Ctrl**+**Z** to Undo the last action performed or **Ctrl**+**Y** to Redo it.

Creating the Mapping - Mapping Ports and Fields

In Creating the Mapping - Designing New XML Structure (p. 425), you have learned how to design the output XML structure your data will flow to. Step two in working with the Mapping editor is connecting the data source to your elements and attributes. The data source is represented by ports and fields on the left hand side of the

Mapping editor window. Remember the **Field** and **Type** columns cannot be modified as they are dependent on the metadata of the XMLWriter's input ports.

To connect a field to an XML node, click a field in the **Field** column, drag it to the right hand part of the window and drop it on an XML node. The result of that action differs according to the node type:

- element - the field will supply data for the element value
- attribute - the field will supply data for the atrribute value
- text node - the field will supply data for the text node
- advanced drag and drop mouse techniques will be discussed below

A newly created connection is displayed as an arrow pointing from a port/field to a node.

To map a port, click a port in the left hand side of the Mapping editor and drag it to the right hand part of the window. Unlike working with fields, a port can only be dropped on an element. Please note that dragging a port on an element DOES NOT map its data but rather instructs the element to repeat itself with each incoming record in that port. As a concequence, a new **Binding** pseudo-element is created, see picture below.

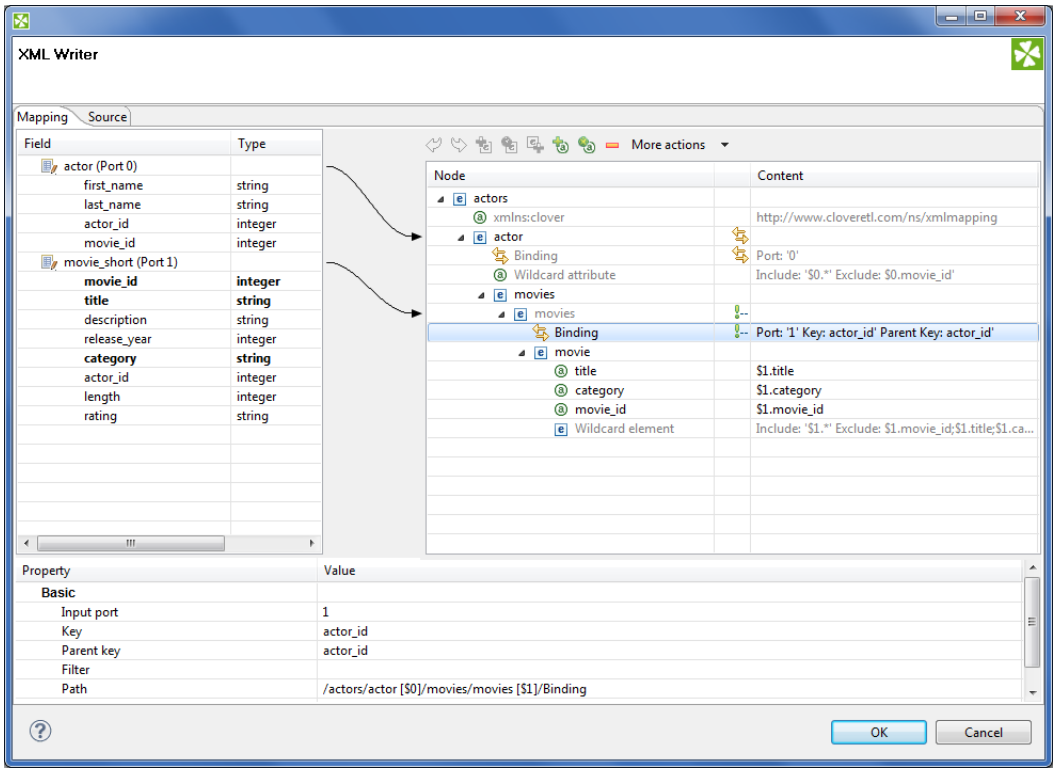


Figure 45.14. Binding of Port and Element.

A **Binding** specifies mapping of an input port to an element. This binding drives the element to repeat itself with every incoming record.

Mouse over a **Binding** to have a tooltip displayed. The tooltip informs you whether the port data is being cached or streamed (affecting overall performance) and which port from. Moreover, in case of caching, you learn how your data would have to be sorted to enable streaming.

Every **Binding** comes with a set of properties:

- **Input port** - the number of the port the data flows flows from. Obviously, you can always easily check which port a node is connected to looking at the arrow next to it.
- **Key** and **Parent key** - the pair of keys determines how the incoming data are joined. In **Key**, enter names of the current element's available fields. In **Parent key**, enter names of fields available to the element's direct parent.

Consequently, the data is joined when the incoming key values equal. Keep in mind that if you specify one of the pair of keys, you have to enter the other one too. To learn which fields are at disposal, click the "..." button located on the right hand side of the key value area. The **Edit key** window will open, enabling you to neatly choose parts of the key by adding them to the **Key parts** list. Naturally, you have to have exactly as many keys as parentKeys, otherwise errors occur.

If fields of **key** and **parentKey** have numerical values, they are compared regardless of their data type. Thus e.g. 1.00 (double) is considered equal to 1 (integer) and these two fields would be joined.

Note

Keys are not mandatory properties. If you do not set them, the element will be repeated for every record incoming from the port it is bound to. Use keys to actually select only some of those records.

- **Filter** - a CTL expression selecting which records are written to the output and which not. See Advanced Description (p. 471) for reference.

To remove a **Binding**, click it and press Delete (alternatively, right-click and select **Remove** or find this option in the toolbar).

Finally, a **Binding** can specify a JOIN between an input port and its parent node in the XML structure (meaning the closest parent node that is bound to an input port). Note that you can join the input with itself, i.e. the element and its parent being driven by the same port. That, however, implies caching and thus slower operation. See the following example:

Example 45.12. Binding that serves as JOIN

Let us have two input ports:

0 - customers (id, name, address)

1 - orders (order_id, customer_id, product, total)

We need some sort of this output:

```
<customer id="1">
  <name>John Smith</name>
  <address>35 Bowens Rd, Edenton, NC (North Carolina)</address>
  <order>
    <product>Towel</product>
    <total>3.99</total>
  </order>
  <order>
    <product>Pillow</product>
    <total>7.99</total>
  </order>
</customer>

<customer id="2">
  <name>Peter Jones</name>
  <address>332 Brixton Rd, Louisville, KY (Kentucky)</address>
  <order>
    <product>Rug</product>
    <total>10.99</total>
  </order>
</customer>
</programlisting>
```

You can see we need to join "orders" with "customer" on (orders.customer_id = customers.id). Port 0 (customers) is bound to the <customer> element, port 1 (orders) is bound to <order> element. Now, this is very easy to setup in the **Binding** pseudoattribute of the nested "order" element. Setting **Key** to "customer_id" and **Parent key** to "id" does exactly the right job.

Creating the Mapping - Using Existing XSD Schema

There is no need to create an XML structure from scratch if you already hold an XSD schema. In that case, you can use the schema to pre-generate the the XML tree. The only thing that may remain is mapping ports to XML nodes, see Creating the Mapping - Mapping Ports and Fields (p. 432).

First of all, start by stating where your schema is. A full path to the XSD has to be set in the **XML Schema** attribute. Second, open the Mapping editor by clicking **Mapping**. Third, when in the editor, choose a root element from the XSD and finally click **Change root element** (see picture below). The XML tree is then automatically generated. Remember you still have to use the `clover` namespace for the process to work properly.

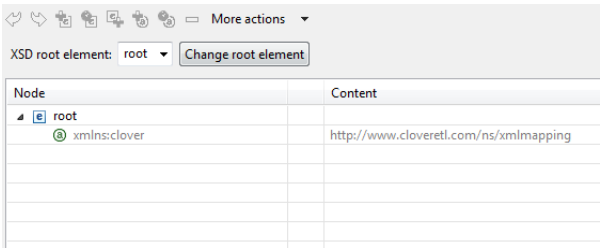


Figure 45.15. Generating XML from XSD root element.

Creating the Mapping - Source Tab

In the **Source** tab of the Mapping editor you can directly edit the XML structure and data mapping. The concept is very simple:

1) write down or paste the desired XML data

2) put data field placeholders (e.g. `$0.field`) into the source wherever you want to populate an element or attribute with input data

3) create port binding and (join) relations - **Input port**, **Key**, **Parent key**

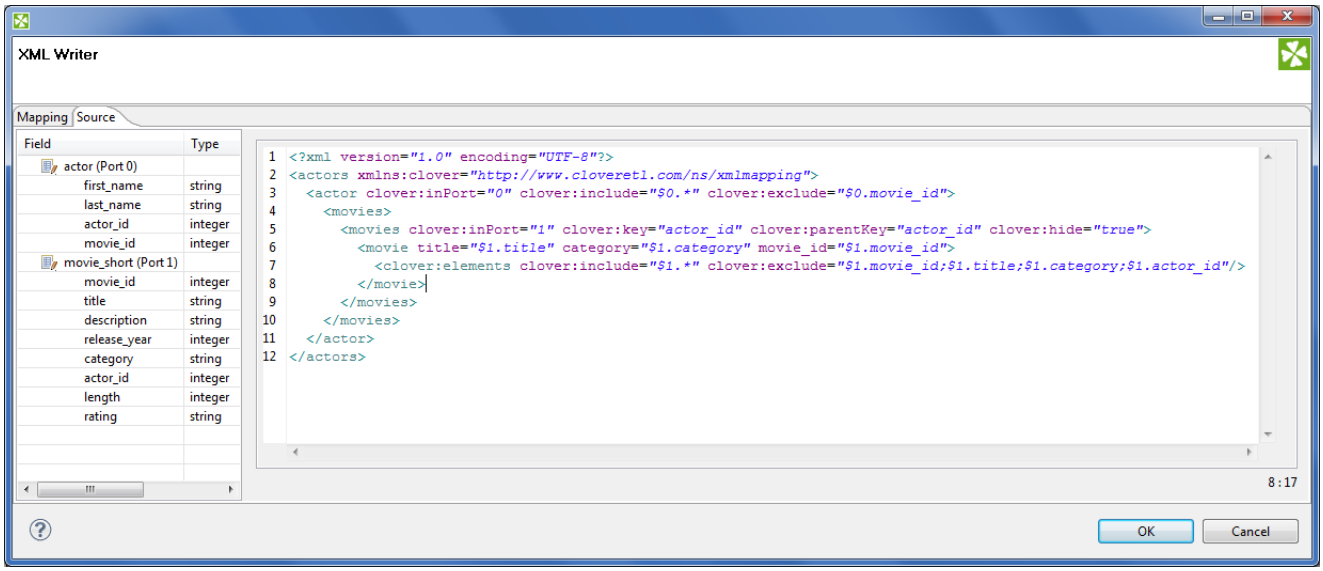


Figure 45.16. Source tab in Mapping editor.

Here you are the same code as in the figure above for your own experiments:

```
<?xml version="1.0" encoding="UTF-8"?>
<actors xmlns:clover="http://www.cloveretl.com/ns/xmlmapping">
  <actor clover:inPort="0" clover:include="$0.*" clover:exclude="$0.movie_id">
    <movies>
      <movies clover:inPort="1" clover:key="actor_id" clover:parentKey="actor_id"
         clover:hide="true">
        <movie title="$1.title" category="$1.category" movie_id="$1.movie_id">
          <clover:elements clover:include="$1.*"
            clover:exclude="$1.movie_id;$1.title;$1.category;$1.actor_id"/>
        </movie>
      </movies>
    </movies>
  </actor>
</actors>
```

Changes made in either of the tabs take immediate effect in the other one. For instance, if you connect port $1 to an element called `invoice` in **Mapping** then switching to **Source**, you will see the element has changed to: `<invoice clover:inPort="1">`.

Source tab supports drag and drop for both ports and fields located on the left hand side of the tab. Dragging a port, e.g. $0 anywhere into the source code inserts the following: `$0.*`, meaning all its fields are used. Dragging a field works the same way, e.g. if you drag field `id` of port $2, you will get this code: `$2.id`.

There are some useful keyboard shortcuts in the **Source** tab. **Ctrl**+**F** brings the **Find/Replace** dialog. **Ctrl**+**L** jumps quickly to a line you type in. Furthermore, a highly interactive **Ctrl**+**Space** Content Assist is available. The range of available options depends on the cursor position in the XML:

I. Inside an element tag - the Content Assist lets you automatically insert the code for **Write attributes when null**, **Omit attributes when null**, **Select input data**, **Exclude attributes**, **Filter input data**, **Hide this element**, **Include attributes**, **Define key**, **Omit when null**, **Define parent key** or **Partition**. On the picture below, please notice you have to insert an extra space after the element name so that the Content Assist could work.

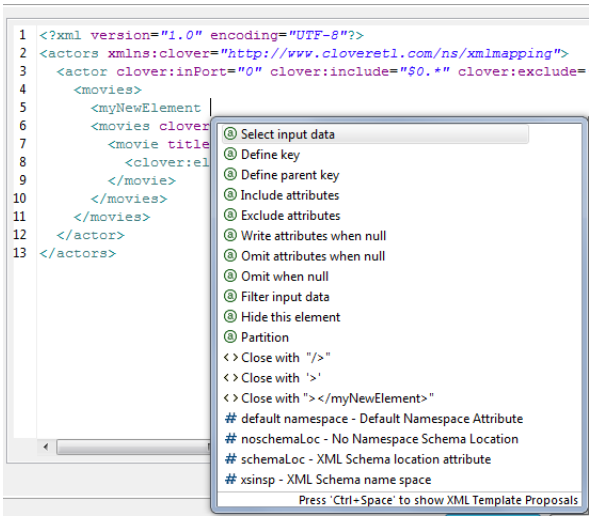


Figure 45.17. Content Assist inside element.

The inserted code corresponds to nodes and their properties as described in Creating the Mapping - Designing New XML Structure (p. 425)

II. Inside the "" quotes - Content Assist lets you smoothly choose values of node properties (e.g. particular ports and fields in **Include** and **Exclude**) and even add Delimiters. Use Delimiters to separate multiple expressions from each other.

III. In a free space in between two elements - apart from inserting a port or field of your choice, you can add **Wildcard element** (as described in Creating the Mapping - Designing New XML Structure (p. 425)), **Insert template** or **Declare template** - see below.

Example 45.13. Insert Wildcard attributes in Source tab

First, create an element. Next, click inside the element tag, press Space, then press **Ctrl**+**Space** choose **Include attributes**. The following code is inserted: `clover:include=""`. Afterwards, you have to determine which port and fields the attributes will be received from (i.e. identical activity to setting the **Include** property in the Mapping tab). Instead of manually typing e.g. `$1.id`, use the Content Assist again. Click inside the "" brackets, press **Ctrl**+**Space** and you will get a list of all available ports. Choose one and press **Ctrl**+**Space** again.

Now that you are done with `include` press Space and then Ctrl+Space again. You will see the Content Assist adapts to what you are doing and where you are. A new option has turned up: **Exclude attributes**. Choose it to insert `clover:exclude=""`. Specifying its value corresponds to entering the **Exclude** property in Mapping.

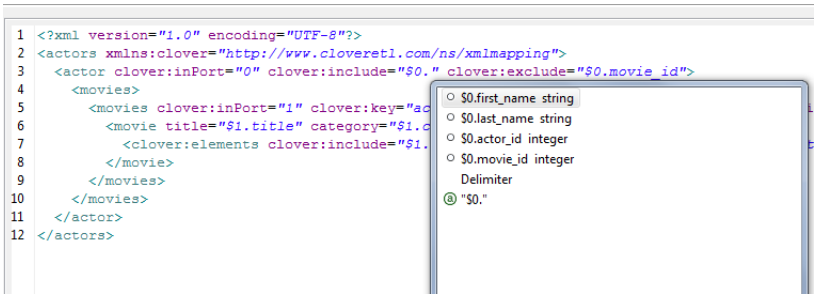


Figure 45.18. Content Assist for ports and fields.

One last thing about the **Source** tab. Sometimes, you might need to work with the `$port.field` syntax a little more. Imagine you have port $0 and its `price` field. Your aim is to send those prices to an element called e.g. `subsidy`. First, you establish a connection between the port and the element. Second, you realize you would like to add the US dollar currency right after the `price` figure. To do so, you just edit the source code like this (same changes can be done in Mapping):

```
<subsidy>$0.price USD</subsidy>
```

However, if you needed to have the "USD" string attached to the price for a reason, use the { } brackets to separate the `$port.field` syntax from additional strings:

```
<subsidy>{$0.price}USD</subsidy>
```

If you ever needed to suppress the dollar placeholder, type it twice. For instance, if you want to print "$0.field" as a string to the output, which would normally map field data coming from port 0, type "$$0.field". That way you will get a sort of this output:

```
<element attribute="$0.field">
```

Templates and Recursion

A template is a piece of code that is used to insert another (larger) block of code. Templates can be inserted into other templates, thus creating recursive templates.

As mentioned above, the **Source** tab's Content Assist allows you to smoothly declare and use your own templates. The option is available when pressing **Ctrl**+**Space** in a free space in between two elements. Afterwards, choose either **Declare template** or **Insert template**.

The **Declare template** inserts the template header. First, you need to enter the template name. Second, you fill it with your own code. Example template could look like this:

```
<clover:template clover:name="templCustomer">
<customer>
 <name>$0.name</name>
 <city>$0.city</city>
 <state>$0.state</state>
</customer>
</clover:template>
```

To insert this template under one of the elements, press **Ctrl**+**Space** and select **Insert template**. Finally, fill in your template name:

```
<clover:insertTemplate clover:name="templCustomer"/>
```

In recursive templates, the `insertTemplate` tag appears inside the template after its potential data. When creating recursive structures, it is crucial to define keys and parent keys. The recursion then continues as long as there are matching `key` and `parentKey` pairs. In other words, the recursion depth is dependent on your input data. Using `filter` can help to get rid of the records you do not need to be written.

Chapter 46. Transformers

We assume that you already know what components are. See Chapter 17, Components (p. 84) for brief information.

Some of the components are intermediate nodes of the graph. These are called **Transformers** or **Joiners**.

For information about **Joiners** see Chapter 47, Joiners (p. 523). Here we will describe **Transformers**.

Transformers receive data through the connected input port(s), process it in the user-specified way and send it out through the connected output port(s).

Components can have different properties. But they also can have some in common. Some properties are common for all of them, others are common for most of the components, or they are common for **Transformers** only. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

We can distinguish **Transformers** according to what they can do.

- One **Transformer** only copies each input data to all connected outputs.
  - SimpleCopy (p. 517) copies each input data record to all connected output ports.
- One **Transformer** passes only some input records to the output.
  - DataSampler (p. 452) passes some input records to the output based on one of the selected filtering strategies.
- One **Transformer** removes duplicate data records.
  - Dedup (p. 454) removes duplicate data. Duplicate data can be sent out through the optional second output port.
- Other components filter data according to the user-defined conditions:
  - ExtFilter (p. 470) compares data with the user-defined condition and sends out records matching this condition. Data records not matching the condition can be sent out through the optional second output port.
  - EmailFilter (p. 465) validates e-mail adresses and sends out the valid ones. Data records with invalid e-mail addresses can be sent out through the optional second output port.
- One Transformer can validate or edit the address format:
  - Address Doctor 5 (p. 441)
- Other **Transformer** sort data each in different way:
  - ExtSort (p. 472) sorts input data.
  - FastSort (p. 474) sorts input data faster than **ExtSort**.
  - SortWithinGroups (p. 519) sorts input data withing groups of sorted data.
- One **Transformer** is able to aggregate information about data:
  - Aggregate (p. 445) aggregates information about input data records.
- One **Transformer** distributes input records among connected output ports:

  - Partition (p. 491) distributes individual input data records among different connected output ports.

- One **Transformer** receives data through two input ports and sends it out through three output ports. Data contained in the first port only, in both ports, or in the second port go to corresponding output port.

  - DataIntersection (p. 449) intersects sorted data from two inputs and sends it out through three connected output ports as defined by the intersection.

- Other **Transformers** can receive data records from multiple input ports and send them all through the unique output port.

  - Concatenate (p. 448) receives data records with the same metadata from one or more input ports, puts them together, and sends them out through the unique output port. Data records from each input port are sent out after all data records from previous input ports.

  - SimpleGather (p. 518) receives data records with the same metadata from one or more input ports, puts them together, and sends them out through the unique output port as fast as possible.

  - Merge (p. 479) receives sorted data records with the same metadata from two or more input ports, sorts them all, and sends them out through the unique output port.

- Other **Transformers** receive data through connected input port, process it in the user-defined way and send it out through the connected output port(s).

  - Denormalizer (p. 456) creates single output data record from a group of input data records.

  - Pivot (p. 498) is a simple form of Denormalizer which creates a pivot table, summarizing input records.

  - Normalizer (p. 484) creates one or more output data record(s) from a single input data record.

  - MetaPivot (p. 481) works similarly to Normalizer, but it always performs the same transformation and the output metadata is fixed to data types.

  - Reformat (p. 502) processes input data in the user-defined way. Can distribute output data records among different or all connected output ports in the user-defined way.

  - Rollup (p. 505) processes input data in the user-defined way. Can create a number of output records from another number of input records. Can distribute output data records among different or all connected output ports in the user-defined way.

  - DataSampler (p. 452) passes only some input records to the output. You can select from one of the available filtering strategies that suits your needs.

- One **Transformer** can transform input data using stylesheets.

  - XSLTransformer (p. 521) transforms input data using stylesheets.

Address Doctor 5

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the appropriate **Transformer** for your purpose, see Transformers Comparison (p. 273).

Short Summary

Address Doctor 5 validates, corrects or completes the address format of your input records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Address Doctor 5	-	no	1	1-2	-	-

Abstract

Address Doctor 5 receives records on its single input port. It then performs a user-defined transformation to the address format (e.g. corrects it or completes its fields). At last it sends the edited address to the first output port. The second output port can optionally be used for records that did not fit the transformation (i.e. a standard error port).

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For transformed data records	Any2
Output	1	no	For records that could not be transformed (error port)	Any2

Address Doctor 5 Attributes

Attribute	Req	Description	Possible values
Basic			
Config file	1)	External file defining the configuration.	
Parameter file	1)	External file defining parameters.	
Configuration	2)	Specifies the address database and its location.	
Parameters	2)	Controls how the transformation is performed.	
Input mapping	yes	Determines what will be processed.	
Output mapping	yes	Controls what will be mapped to the ouptput.	
Element item delimiter		If the whole address is stored on a single line, this attribute specifies which special character separates the address fields.	delimiter is not used (default) | one of these: ; : # | \n \r\n clover_item_delimiter

Legend:

1): If these two are set, you do not have to define **Configuration** and **Parameters**.

2): You either set these two or you define **Config file** and **Parameter file** instead.

Advanced Description

Address Doctor 5 operates by passing the input data and configuration to a third party Address Doctor library. Afterwards, the component maps the outputs from the library back to CloverETL.

Consequently, if you ever get unsure about working with the component, a good place to look for help is the official Address Doctor 5 documentation. It contains the necessary information for a detailed configuration of the **Address Doctor 5** component. CloverETL actually serves as a GUI for setting parameters.

Working with the component can be regarded as fulfilling these tasks:

- telling the graph where Address Doctor libraries (`*.jar` + native libraries) are - they can be passed as a Java Argument or copied to Program Files; please note you have to obtain the libraries yourself - they are not embedded in CloverETL
- obtaining the address database
- setting the component attributes - see Address Doctor 5 Configuration (p. 442)

Address Doctor 5 Configuration

The components is configured in four basic steps setting these attributes:

1. **Configuration** - specifies where the address database is located. Do not forget your database is supplied in one of the modes (e.g. BATCH_INTERACTIVE) and thus you have to set a matching **Type** (applies to **Enrichment** databases set in **Parameters**, too). In order to be able to work with the database, you have to obtain an appropriate **Unlock code** (either universal or specific).

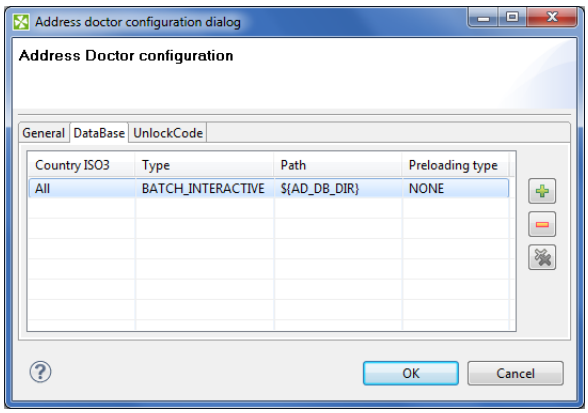


Figure 46.1. DataBase Configuration

2. **Parameters** - controls what transformation will be performed. Particular settings are highly specific and should be consulted with the official Address Doctor 5 documentation.

   For instance in the **Process** tab of the dialogue, you can configure various **Enrichments**. These allow you to add certificates of the address format. The certificates guarantee that a particular address format matches the official format of a national post office. Note that adding Enrichments usually slows the data processing and can optionally require an additional database.

3. **Input mapping** - determines what will be processed. You work with a wizard that lets you do the settings in three basic steps:

   - Select address properties form all Address Doctor internal fields ("metadata") that are permitted on the input. Field names are accompanied by a number in parantheses informing you how many fields can form a property ("output metadata"). For instance "Street name (6)" tells you the street name can be written on up to 6 rows of the input file.

   - Specify the internal mapping of Address Doctor - drag input fields you have chosen in the previous step on the available fields of the **Input mapping**.

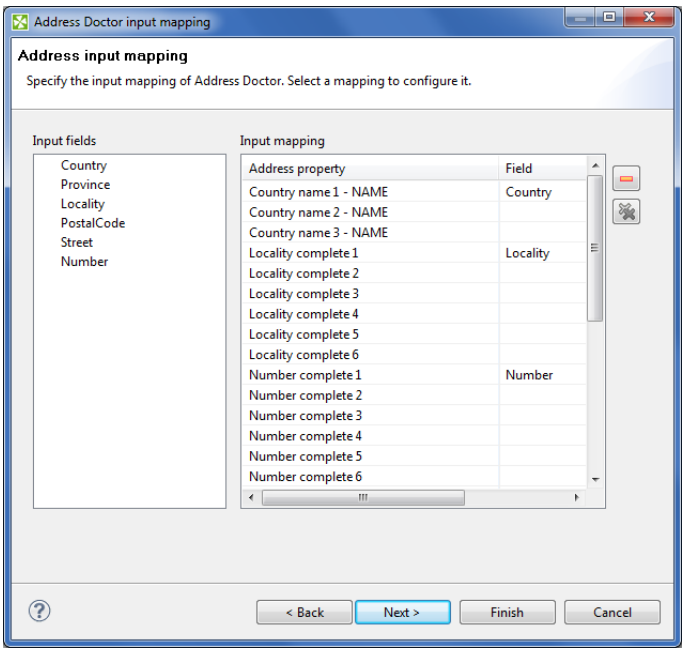


Figure 46.2. Input mapping

- Examine the summary of the input mapping.

4. **Output mapping** - here you decide what will be mapped to the output, i.e. the first output port. Optionally, you can map data to the second "error" port (if no such mapping is done, error codes and error messages are generated).

   Similarly to **Input mapping**, you do the configuration by means of a clear wizard comprising these steps:

   - Select address properties for mapping.
   - Specify particular output mapping. That involves assigning the internal fields you have selected before to output fields. In the **Error port** tab, design a structure of the error output (its fields) that is sent to the second output port if the component cannot perform the address transformation.

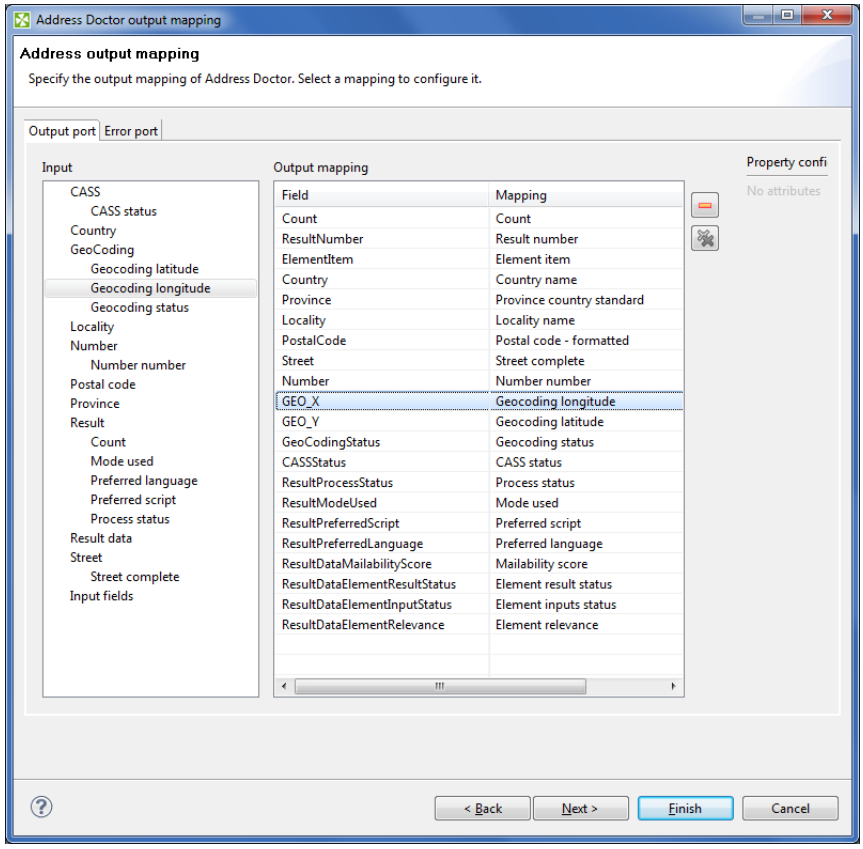


Figure 46.3. Output mapping

- Examine the summary of the output mapping.

A spin-off of working with the component is the so-called *transliteration*. That means you can e.g. input an address in the Cyrillic alphabet and have it converted to the Roman alphabet. No extra database is needed for this task.

Aggregate

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Aggregate computes statistical information about input data records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Aggregate	-	no	1	n	no	no

Abstract

Aggregate receives data records through single input port, computes statistical information about input data records and sends it to single output port.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For statistical information	Any2

Aggregate Attributes

Attribute	Req	Description	Possible values
Basic			
Aggregation mapping		Sequence of individual mappings for output field names separated from each other by semicolon. Each mapping can have the following form: `$outputField:=constant` or `$outputField:=$inputField` (this must be a field name from the **Aggregate key**) or `$outputField:=somefunction($inputField)`.	
Aggregate key		Key according to which the records are grouped. See Group Key (p. 231) for more information.	
Charset		Encoding of incoming data records.	ISO-8859-1 (default) | other encoding
Sorted input		By default, input data records are supposed to be sorted acoording to **Aggregate key**. If they are not sorted as specified, switch this value to `false`.	true (default) | false
Equal NULL		By default, records with null values are considered to be different. If set to `true`, records with null values are considered to be equal.	false (default) | true
Deprecated			
Old aggregation mapping		Mapping that was used in older versions of CloverETL, its use is deprecated now.	

Advanced Description

Aggregate Mapping

When you click the **Aggregation mapping** attribute row, an **Aggregation mapping** wizard opens. In it, you can define both the mapping and the aggregation. The wizard consists of two panes. You can see the **Input field** pane on the left and the **Aggregation mapping** pane on the right.

1. Select each field that should be mapped to output by clicking and drag and drop it to the **Mapping** column in the right pane at the row of the desired output field name. After that, the selected input field appears in the **Mapping** column. This way you can map all the desired input fields to the output fields.

2. In addition to it, for such fields that are not part of **Aggregate key**, you must also define some aggregation function.

   Fields of **Aggregate key** are the only ones that can be mapped to output fields without any function (or with a function).

   Thus, the following mapping can only be done for key fields: `$outField=$keyField`.

   On the other hand, for fields that are not contained in the key, such mapping is not allowed. A function must always be defined for them.

   To define a function for a field (contained in the key or not-contained in it), click the row in the **Function** column and select some function from the combo list. After you select the desired function, click **Enter**.

3. For each output field, a constant may also be assigned to it.

Example 46.1. Aggregation Mapping

```
$Count=count();$AvgWeight:=avg($weight);$OutFieldK:=$KeyFieldM;
$SomeDate:=2008-08-28
```

Here:

1. Among output fields are: `Count`, `AvgWeight`, `OutFieldK`, and `SomeDate`. Output metadata can also have other fields.

2. Among input fields are also: `weight`, and `KeyFieldM`. Input metadata can also have other fields.

3. `KeyFieldM` must be a field from **Aggregate key**. This key may also consist of other fields.

   `weight` is not a field from **Aggregate key**.

   `2008-08-28` is a constant date that is assigned to output date field.

   `count()` and `avg()` are functions that can be applied to inputs. The first does not need any argument, the second need one - which is the value of the `weight` field for each input record.

Concatenate

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Concatenate gathers data records from multiple inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Concatenate	yes	no	1-n	1	-	-

Abstract

Concatenate receives potentially unsorted data records through one or more input ports. (Metadata of all input ports must be the same.) **Concatenate** gathers all the records in the order of input ports and sends them to the single output port. It gathers input records starting with the first input port, continuing with the next one and ending with the last port. Within each input port the records order is preserved.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
	1-n	no	For input data records	Input 0[1)]
Output	0	yes	For gathered data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

DataIntersection

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

DataIntersection intrersects data from two inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
DataIntersection	no	yes	2	3	yes	yes

Abstract

DataIntersection receives sorted data from two inputs, compares the **Join key** values in both of them and processes the records in the following way:

Such input records that are on both input port 0 and input port 1 are processed according to the user-defined transformation and the result is sent to the output port 1. Such input records that are only on input port 0 are sent unchanged to the output port 0. Such input records that are only on input port 1 are sent unchanged to the output port 2.

Records are considered to be on both ports if the values of all **Join key** fields are equal in both of them. Otherwise, they are considered to be records on input 0 or 1 only.

A transformation must be defined. The transformation uses a CTL template for **DataIntersection**, implements a `RecordTransform` interface or inherits from a `DataRecordTransform` superclass. The interface methods are listed below.

Note

Note that this component is similar to **Joiners**: It does not need identical metadata on its inputs and processes records whose **Join key** is equal. Also duplicate records can be sent to transformation or not (**Allow key duplicates**).

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records (data flow A)	Any(In0)[1)]
	1	yes	For input data records (data flow B)	Any(In1)[1)]
Output	0	yes	For not-changed output data records (contained in flow A only)	Input 0[2)]
	1	yes	For changed output data records (contained in both input flows)	Any(Out1)
	2	yes	For not-changed output data records (contained in flow B only)	Input 1[2)]

Legend:

1): Part of them must be equivalent and comparable (**Join key**).

2): Metadata cannot be propagated through this component.

DataIntersection Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key that compares data records from input ports. Only those pairs of records (one from each input) with equal value of this attribute are sent to transformation. See Join key (p. 451) for more information. Records should be sorted in ascending order to get reasonable results.	
Transform	1)	Definition of the way how records should be intersected written in the graph in CTL or Java.	
Transform URL	1)	Name of external file, including path, containing the definition of the way how records should be intersected written in CTL or Java.	
Transform class	1)	Name of external class defining the way how records should be intersected.	
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Equal NULL		By default, records with null values of key fields are considered to be equal. If set to `false`, they are considered to be different from each other.	true (default) | false
Advanced			
Allow key duplicates		By default, all duplicates on inputs are allowed. If switched to `false`, records with duplicate key values are not allowed. If it is `false`, only the **first** record is used for join.	true (default) | false
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Attribute	Req	Description	Possible values
Slave override key		Older form of **Join key**. Contains fields from the second input port only. This attribute is deprecated now and we suggest you use the current form of the **Join key** attribute.	

Legend:

1): One of these must specified. Any of these transformation attributes uses a CTL template for **DataIntersection** or implements a `RecordTransform` interface.

See CTL Scripting Specifics (p. 451) or Java Interfaces for DataIntersection (p. 451) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Join key**

  Expressed as a sequence of individual subexpressions separated from each other by semicolon. Each subexpression is an assignment of a field name from the first input port (prefixed by dollar sign), on the left side, and a field name from the second input port (prefixed by dollar sign), on the right side.

  Example 46.2. Join Key for DataIntersection

  ```
  $first_name=$fname;$last_name=$lname
  ```

  In this **Join key**, `first_name` and `last_name` are fields of metadata on the first input port and `fname` and `lname` are fields of metadata on the second input port.

  Pairs of records containing the same value of this key on both input ports are transformed and sent to the second output port. Records incoming through the first input port for which there is no counterpart on the second input port are sent to the first output port without being changed. Records incoming through the second input port for which there is no counterpart on the first input port are sent to the third output port without being changed.

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify a transformation that assigns a number of output port to each input record.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

CTL Templates for DataIntersection

DataIntersection uses the same transformation teplate as **Reformat** and **Joiners**. See CTL Templates for Joiners (p. 278) for more information.

Java Interfaces for DataIntersection

DataIntersection implements the same interface as **Reformat** and **Joiners**. See Java Interfaces for Joiners (p. 281) for more information.

DataSampler

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the appropriate **Transformer** for your purpose, see Transformers Comparison (p. 273).

Short Summary

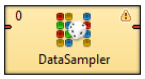
DataSampler passes only some input records to the output. There is a range of filtering strategies you can select from to control the transformation.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
DataSampler	-	no	1	1-N	no	no

Abstract

DataSampler receives data on its single input edge. It then filters input records and passes only some of them to the output. You can control which input records are passed by selecting one of the filtering strategies called **Sampling methods**. The input and output metadata have to match each other.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	Yes	For sampled data records	Input0

DataSampler Attributes

Attribute	Req	Description	Possible values
Basic			
Sampling method	yes	The filtering strategy that determines which records will be passed to the output. Individual strategies you can choose from are described in Advanced Description (p. 453)	Simple| Systematic| Stratified|PPS
Required sample size	yes	The desired size of output data expressed as a fraction of the input. If you want the output to be e.g. 15% (roughly) of the input size, set this attribute to 0.15.	(0; 1)
Sampling key	1)	A field name the **Sampling method** uses to define strata. Field names can be chained in a sequence separated by a colon, semicolon or pipe. Every field can be followed by an order indicator in brackets (**a** for ascending, **d** for descending, **i** for ignore and **r** for automatic estimate).	e.g. Surname(a); FirstName(i); Salary(d)
Advanced			
Random seed		A `long` number that is used in the random generator. It assures that results are random but remain identical on every graph run.	<0; N>

Legend:

1) The attribute is required in all sampling methods except for **Simple**.

Advanced Description

A typical use case for **DataSamper** can be imagined like this. You want to check whether your data transformation works properly. In case you are processing millions of records, it might be useful to get only a few thousands and observe. That is why you will use this component to create a data sample.

DataSampler offers four **Sampling methods** to create a representative sample of the whole data set:

- **Simple** - every record has equal chance of being selected. The filtering is based on a `double` value chosen (approx. uniformly) from the <0.0d; 1.0d) interval. A record is selected if the drawn number is lower than **Required sample size**.
- **Systematic** - has a random start. It then proceeds by selecting every k-th element of the ordered list. The first element and interval derive from **Required sample size**. The method depends on the data set being arranged in a sort order given by **Sampling key** (for the results to be representative). There are also cases you might need to sample an unsorted input. Even though you always have to specify **Sampling key**, remember you can suppress its sort order by setting the order indicator to **i** for "ignore". That ensures the data set's sort order will not be regarded. Example key setting: "InvoiceNumber(i)".
- **Stratified** - if the data set contains a number of distinct categories, the set can be organised by these categories into separate *strata*. Each *stratum* is then sampled as an independent sub-population out of which individual elements are selected on a random basis. At least one record from each stratum is selected.
- **PPS** (Probability Proportional to Size Sampling) - probability for each record is set to proportional to its *stratum* size up to a maximum of 1. Strata are defined by the value of the field you have chosen in **Sampling key**. The method then uses **Systematic** sampling for each group of records.

Comparing the methods, **Simple** random sampling is the simplest and quickest one. It suffices in most cases. **Systematic** sampling with no sorting order is as fast as **Simple** and produces a strongly representative data probe, too. **Stratified** sampling is the trickiest one. It is useful only if the data set can be split into separate groups of reasonable sizes. Otherwise the data probe is much bigger than requested. For a deeper insight into sampling methods in statistics, see Wikipedia.

Dedup

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Dedup removes duplicate records.

Component	Same input metadata	Sorted inputs [1]	Inputs	Outputs	Java	CTL
Dedup	-	✔	1	0-1	-	-

[1] Input records may be sorted only partially, i.e., the records with the same value of the **Dedup key** are grouped together but the groups are not ordered

Abstract

Dedup reads data flow of records grouped by the same values of the **Dedup key**. The key is formed by field name(s) from input records. If no key is specified, the component behaves like the Unix `head` or `tail` command. The groups don't have to be ordered.

The component can select the specified number of the first or the last records from the group or from the whole input. Only those records with no duplicates can be selected too.

The deduplicated records are sent to output port 0. The duplicate records may be sent through output port 1.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✔	for deduplicated data records	equal input metadata [1]
	1	✘	for duplicate data records	

[1] Metadata can be propagated through this component.

Dedup Attributes

Attribute	Req	Description	Possible values
Basic			
Dedup key		Key according to which the records are deduplicated. By default, i.e., if the **Dedup key** is not set, the in **Number of duplicates** attribute specified number of records from the beginning or the end of all input records is preserved while removing the others. If the **Dedup key** is set, only specified number of records with the same values in fields specified as the **Dedup key** is picked up. See Dedup key (p. 455).	
Keep		Defines which records will be preserved. If `First`, those from the beginning. If `Last`, those from the end. Records are selected from a group or the whole input. If `Unique`, only records with no duplicates are selected.	First (default) | Last | Unique
Equal NULL		By default, records with null values of key fields are considered to be equal. If `false`, they are considered to be different.	true (default) | false
Number of duplicates		Maximum number of duplicate records to be selected from each group of adjacent records with equal key value or , if key not set, maximum number of records from the beginning or the end of all records. Ignored if `Unique` option selected.	1 (default) | 1-N

Advanced Description

- **Dedup key**

  The component can process sorted input data as well as partially sorted ones. When setting the fields composing the **Dedup key**, choose the proper **Order** attribute:

  1. *Ascending* - if the groups of input records with the same key field value(s) are sorted in ascending order
  2. *Descending* - if the groups of input records with the same key field value(s) are sorted in descending order
  3. *Auto* - the sorting order of the groups of input records is guessed from the first two records with different value in the key field, i.e., from the first records of the first two groups.
  4. *Ignore* - if the groups of input records with the same key field value(s) are not sorted

Denormalizer

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Denormalizer creates single output record from one or more input records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Denormalizer	-	✖	1	1	✔	✔

Abstract

Denormalizer receives sorted data through single input port, checks **Key** values, creates one output record from one or more adjacent input records with the same **Key** value.

A transformation must be defined. The transformation uses a CTL template for **Denormalizer**, implements a `RecordDenormalize` interface or inherits from a `DataRecordDenormalize` superclass. The interface methods are listed below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	any
Output	0	✔	for denormalized data records	any

Denormalizer Attributes

Attribute	Req	Description	Possible values
Basic			
Key	[1]	Key that creates groups of input data records according to its value. Adjacent input records with the same value of **Key** are considered to be members of one group. One output record is composed from members of such group. See Key (p. 457) for more information.	
Group size	[1]	Group may be defined by exact number of its memebers. E.g. each five records form a single group. The input record count MUST me a multiple of `group size`. This is mutually exclusive with `key` atribute.	a number
Denormalize	[2]	Definition of how to denormalize records, written in the graph in CTL.	
Denormalize URL	[2]	Name of external file, including path, containing the definition of how to denormalize records, written in CTL or Java.	
Denormalize class	[2]	Definition of how to denormalize records, written in the graph in Java.	
Sort order		Order in which groups of input records are expected to be sorted. See Sort order (p. 457)	Auto (default) | Ascending | Descending | Ignore
Equal NULL		By default, records with null values of key fields are considered to be equal. If `false`, they are considered to be different.	true (default) | false
Denormalize source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

[1] Either `key` or `group size` must be specified, `group size` having higher priority

[2] One of them must specified. Any of these transformation attributes uses the CTL template for **Denormalizer** or implements a `RecordDenormalize` interface.

See CTL Scripting Specifics (p. 458) or Java Interfaces for Denormalizer (p. 464) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Key**

  Expressed as a sequence of field names separated from each other by semicolon, colon, or pipe.

  Example 46.3. Key for Denormalizer

  ```
  first_name;last_name
  ```

  In this **Key**, `first_name` and `last_name` are fields of metadata on input port.

- **Sort order**

If the records are denormalized by the **Key**, i.e., not by the **Group size**, the input records must be grouped according to the **Key** field value. Then, depending on the sorting order of the groups, select the proper **Sort order**:

1. *Auto* - the sorting order of the groups of input records is guessed from the first two records with different value in the key field, i.e., from the first records of the first two groups.

2. *Ascending* - if the groups of input records with the same key field value(s) are sorted in ascending order

3. *Descending* - if the groups of input records with the same key field value(s) are sorted in descending order

4. *Ignore* - if the groups of input records with the same key field value(s) are not sorted

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify the way how input should be transformed into output.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

Once you have written your transformation, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

You can open the transformation definition as another tab of the graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

CTL Templates for Denormalizer

Here is an example of how the **Source** tab for defining the transformation looks.

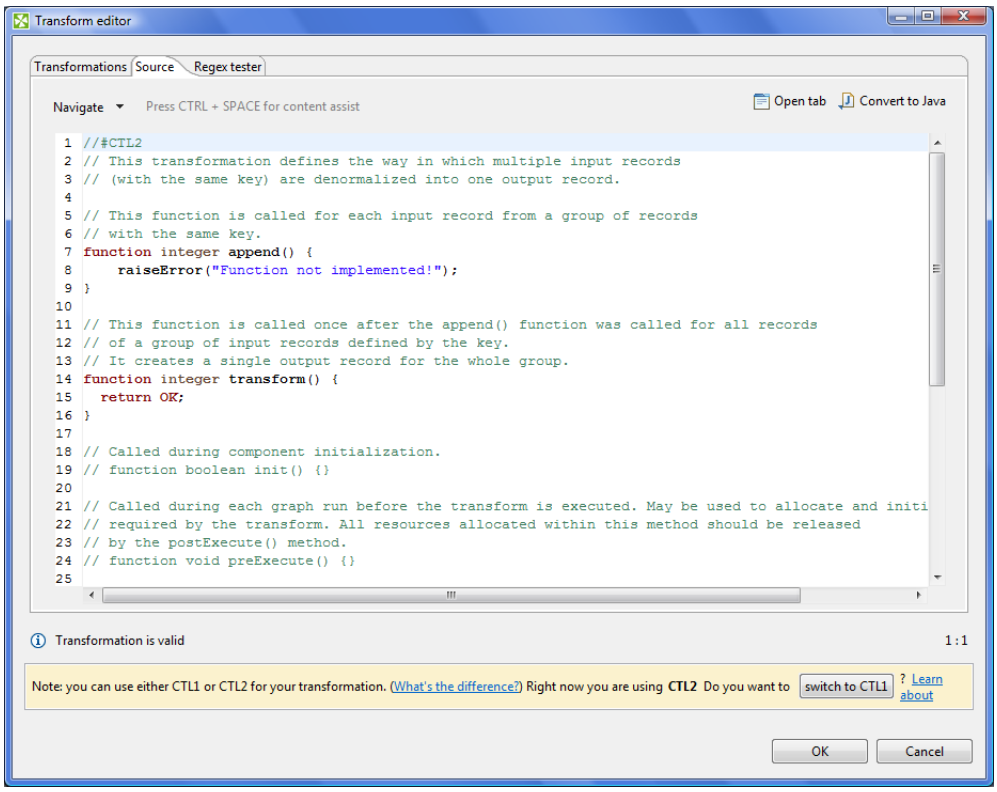


Figure 46.4. Source Tab of the Transform Editor in the Denormalizer Component (I)

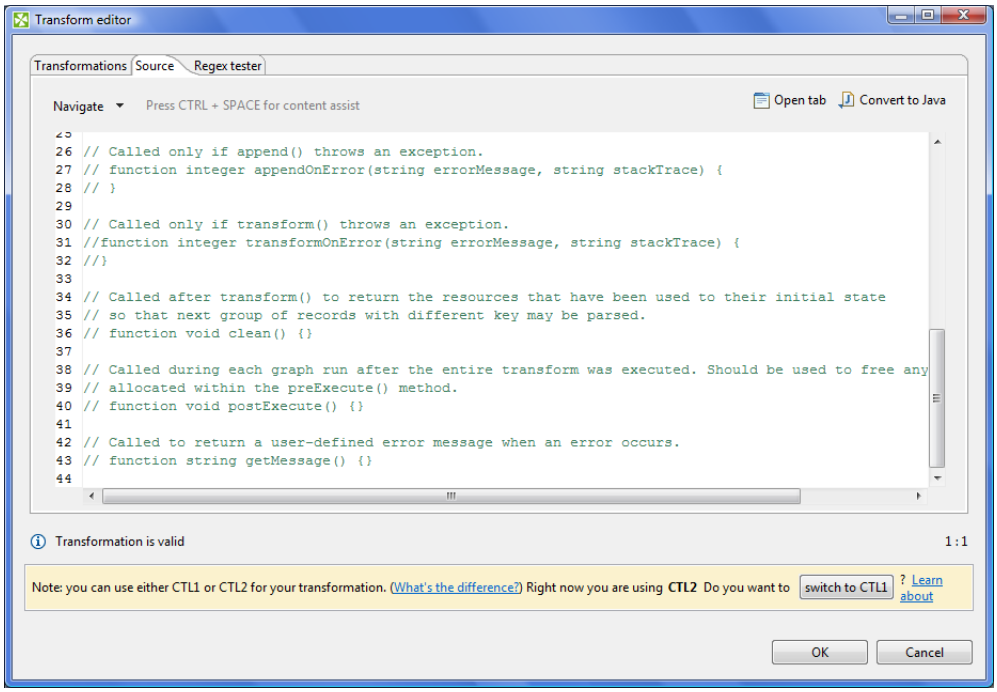


Figure 46.5. Source Tab of the Transform Editor in the Denormalizer Component (II)

Table 46.1. Functions in Denormalizer

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables

CTL Template Functions	
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer append()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly, once for each input record
Description	For the group of adjacent input records with the same **Key** values it appends the information from which composes the resulting output record. If any of the input records causes fail of the `append()` function, and if user has defined another function (`appendOnError()`), processing continues in this `appendOnError()` at the place where `append()` failed. If `append()` fails and user has not defined any `appendOnError()`, the whole graph will fail. The `append()` passes to `appendOnError()` error message and stack trace as arguments.

Example	`function integer append() {`
	`  CustomersInGroup++;`
	`  myLength = length(errorCustomers);`
	`  if(!isInteger($0.OneCustomer)) {`
	`  errorCustomers = errorCustomers`
	`  + iif(myLength > 0 ,"-","")`
	`  + $0.OneCustomer;`
	`  }`
	`  customers = customers`
	`  + iif(length(customers) > 0 ," - ","")`
	`  + $0.OneCustomer;`
	`  groupNo = $GroupNo;`
	`  return CustomersInGroup;`
	`}`

integer transform()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly, once for each output record.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

CTL Template Functions	

Example	`function integer transform() {`
	`  $0.CustomersInGroup = CustomersInGroup;`
	`  $0.CustomersOnError = errorCustomers;`
	`  $0.Customers = customers;`
	`  $0.GroupNo = groupNo;`
	`  return OK;`
	`}`

void clean()	
Required	no
Input Parameters	none
Returns	void
Invocation	Called repeatedly, once for each output record (after this has been created by the **transform()** function).
Description	Returns the component to the initial settings

Example	`function void clean(){`
	`  customers = "";`
	`  errorCustomers = "";`
	`  groupNo = 0;`
	`  CustomersInGroup = 0;`
	`}`

integer appendOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. Positive integer numers are ignored, meaning of 0 and negative values is described in Return Values of Transformations (p. 238)
Invocation	Called if `append()` throws an exception. Called repeatedly for the whole group of records with the same **Key** value.
Description	For the group of adjacent input records with the same **Key** values it appends the information from which it composes the resulting output record. If any of the input records causes fail of the `append()` function, and if user has defined another function (`appendOnError()`), processing continues in this `appendOnError()` at the place where `append()` failed. If `append()` fails and user has not defined any `appendOnError()`, the whole graph will fail. The `appendOnError()` function gets the information gathered by `append()` that was get from previously successfully processed input records. Also error message and stack trace are passed to `appendOnError()`.

Example	`function integer appendOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return CustomersInGroup;`
	`}`

integer transformOnError(Exception exception, stackTrace)	
Required	no

CTL Template Functions	
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  $0.CustomersInGroup = CustomersInGroup;`
	`  $0.ErrorFieldForTransform = errorCustomers;`
	`  $0.CustomersOnError = errorCustomers;`
	`  $0.Customers = customers;`
	`  $0.GroupNo = groupNo;`
	`  return OK;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user
Invocation	Called in any time specified by user (called only when either `append()`, `transform()`, `appendOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.

CTL Template Functions	
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Input records or fields**

  Input records or fields are accessible within the `append()` and `appendOnError()` functions only.

- **Output records or fields**

  Output records or fields are accessible within the `transform()` and `transformOnError()` functions only.

- All of the other CTL template functions allow to access neither inputs nor outputs.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Denormalizer

The transformation implements methods of the `RecordDenormalize` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of the `RecordDenormalize` interface:

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  Initializes denormalize class/function. This method is called only once at the beginning of denormalization process. Any object allocation/initialization should happen here.

- `int append(DataRecord inRecord)`

  Passes one input record to the composing class.

- `int appendOnError(Exception exception, DataRecord inRecord)`

  Passes one input record to the composing class. Called only if `append(DataRecord)` throws an exception.

- `int transform(DataRecord outRecord)`

  Retrieves composed output record. See Return Values of Transformations (p. 238) for detailed information about return values and their meaning. In **Denormalizer**, only `ALL`, `0`, `SKIP`, and **Error codes** have some meaning.

- `int transformOnError(Exception exception, DataRecord outRecord)`

  Retrieves composed output record. Called only if `transform(DataRecord)` throws an exception.

- `void clean()`

  Finalizes current round/clean after current round - called after the transform method was called for the input record.

EmailFilter

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

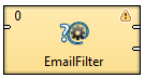
EmailFilter filters input records according to the specified condition.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
EmailFilter	-	no	1	0-2	-	-

Abstract

EmailFilter receives incoming records through its input port and verifies specified fields for valid e-mail addresses. Data records that are accepted as valid are sent out through the optional first output port if connected. Specified fields from the rejected inputs can be sent out through the optional second output port if this is connected to other component. Metadata on the optional second output port may also contain up to two additional fields with information about error.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	no	For valid data records	Input 0[1)]
	1	no	For rejected data records	Any[2)]

Legend:

1): Metadata cannot be propagated through this component.

2): Metadata on the output port 0 contain any of the input data fields plus up to two additional fields. Fields whose names are the same as those in the input metadata are filled in with input values of these fields.

Table 46.2. Error Fields for EmailFilter

Field number	Field name	Data type	Description
FieldA	the **Error field** attribute value	string	Error field
FieldB	the **Status field** attribute value	integer[1)]	Status field

Legend:

1): The following error codes are most common:

- **0** `No error` - e-mail address accepted.
- **1** `Syntax error` - any string that does not conform to e-mail address format specification is rejected with this error code.
- **2** `Domain error` - verification of domain failed for the address. Either the domain does not exist or the DNS system can not determine a mail exchange server.
- **3** `SMTP handshake error` - at `SMTP` level this error code indicates that a mail exchange server for specified domain is either unreachable or the connection failed for other reason (e.g. server to busy, etc.).
- **4** `SMTP verify error` - at `SMTP` level this error code means that server rejected the address as being invalid using the VRFY command. Address is officially invalid.
- **5** `SMTP recipient error` - at `SMTP` level this error code means the server rejected the address for delivery.
- **6** `SMTP mail error` - at `MAIL` level this error indicates that although the server accepted the test message for delivery, an error occurred during send.

EmailFilter Attributes

Attribute	Req	Description	Possible values
Basic			
Field list	yes	List of selected input field names whose values should be verified as valid or non-valid e-mail addresses. Expressed as a sequence of field names separated by colon, semicolon, or pipe.	
Level of inspection		Various methods used for the e-mail address verification can be specified. Each level includes and extends its predecessor(s) on the left. See Level of Inspection (p. 468) for more information.	SYNTAX | DOMAIN (default) | SMTP | MAIL
Accept empty		By default, even empty field is accepted as a valid address. This can be switched off, if it is set to `false`. See Accept Conditions (p. 468) for more information.	true (default) | false
Error field		Name of the output field to which error message can be written (for rejected records only).	
Status field		Name of the output field to which error code can be written (for rejected records only).	
Multi delimiter		Regular expression that serves to split individual field value to multiple e-mail addresses. If empty, each field is treated as a single e-mail address.	[,;] (default) | other

Attribute	Req	Description	Possible values
Accept condition		By default, record is accepted even if at least one field value is verified as valid e-mail address. If set to `STRICT`, record is accepted only if all field values from all fields of the **Field list** are valid. See Accept Conditions (p. 468) for more information.	LENIENT (default) | STRICT
Advanced			
E-mail buffer size		Maximum number of records that are read into memory after which they are bulk processed. See Buffer and Cache Size (p. 468) for more information.	2000 (default) | 1-N
E-mail cache size		Maximum number of cached e-mail address verification results. See Buffer and Cache Size (p. 468) for more information.	2000 (default) | 0 (caching is turned off) | 1-N
Domain cache size		Maximum number of cached DNS query results. Is ignored at `SYNTAX` level.	3000 (default) | 0 (caching is turned off) | 1-N
Domain retry timeout (ms)		Timeout in millisecond for each DNS query attempt. Thus, maximum time in milliseconds spent to resolving equals to **Domain retry timeout** multiplicated by **Domain retry count**.	800 (default) | 1-N
Domain retry count		Number of retries for failed DNS queries.	2 (default) | 1-N
Domain query A records		By default, according to the SMTP standard, if no MX record could be found, A record should be searched. If set to `false`, DNS query is two times faster, however, this SMTP standard is broken..	true (default) | false
SMTP connect attempts (ms,...)		Attempts for connection and HELO. Expressed as a sequence of numbers separated by comma. The numbers are delays between individual attempts to connect.	1000,2000 (default)
SMTP anti-greylisting attempts (s,...)		Anti-greylisting feature. Attempts and delays between individual attempts expressed as a sequence of number separated by comma. If empty, anti-greylisting is turned off. See SMTP Grey-Listing Attempts (p. 469) for more information.	30,120,240 (default)
SMTP retry timeout (s)		TCP timeout in seconds after which a SMTP request fails.	300 (default) | 1-N
SMTP concurrent limit		Maximum number of parallel tasks when anti-greylisting is on.	10 (default) | 1-N
Mail From		The `From` field of a dummy message sent at `MAIL` level.	CloverETL <clover@cloveretl.org> (default) | other
Mail Subject		The `Subject` field of a dummy message sent at `MAIL` level.	Hello, this is a test message (default) | other
Mail Body		The `Body` of a dummy message sent at `MAIL` level.	Hello,\nThis is CloverETL text message.\n\nPlease ignore and don't respond. Thank you, have a nice day! (default) | other

Advanced Description

Buffer and Cache Size

Increasing **E-mail buffer size** avoids unnecessary repeated queries to DNS system and SMTP servers by processing more records in a single query. On the other hand, increasing **E-mail cache size** might produce even better performance since addresses stored in cache can be verified in an instant. However, both parameters require extra memory so set it to the largest values you can afford on your system.

Accept Conditions

By default, even an empty field from input data records specified in the **List of fields** is considered to be a valid e-mail address. The **Accept empty** attribute is set to `true` by default. If you want to be more strict, you can switch this attribute to `false`.

In other words, this means that at least one valid e-mail address is sufficient for considering the record accepted.

On the other hand, in case of **Accept condition** set to `STRICT`, all e-mail addresses in the **List of fields** must be valid (either including or excluding empty values depending on the **Accept empty** attribute).

Thus, be careful when setting these two attributes: **Accept empty** and **Accept condition**. If there is an empty field among fields specified in **List of fields**, and all other non-empty values are verified as invalid addresses, such record gets accepted if both **Accept condition** is set to `LENIENT` and **Accept empty** is set to `true`. However, in reality, such record does not contain any useful and valid e-mail address, it contains only an empty string which assures that such record is accepted.

Level of Inspection

1. **SYNTAX**

   At the first level of validation (`SYNTAX`), the syntax of e-mail expressions is checked and even both non-strict conditions and international characters (except TLD) are allowed.

2. **DOMAIN**

   At the second level of validation (`DOMAIN`) - which is the default one - DNS system is queried for domain validity and mail exchange server information. The following four attributes can be set to optimize the ratio of performance to false-negative responses: **Domain cache size**, **Domain retry timeout**, **Domain retry count**. and **Domain query A records**. The number of queries sent to DNS server is specified by the **Domain retry count** attribute. Its default value is 2. Time interval between individual queries that are sent is defined by **Domain retry timeout** in milliseconds. By default it is 800 milliseconds. Thus, the whole time during which the queries are being resolved is equal to **Domain retry count** x **Domain retry timeout**. The results of queries can be cached. The number of cached results is defined by **Domain cache size**. By default, 3000 results are cached. If you set this attribute to 0, you turn the caching off. You can also decide whether A records should be searched if no MX record is found (**Domain query A records**). By default, it is set to `true`. Thus, A record is searched if MX record is not found. However, you can switch this off by setting the attribute to `false`. This way you can speed the searching two times, although that breaks the SMTP standard.

3. **SMTP**

   At the third level of validation (`SMTP`), attempts are made to connect SMTP server. You need to specify the number of attempts and time intervals between individual attempts. This is defined using the **SMTP connect attempts** attribute. This attribute is a sequence of integer numbers separated by commas. Each number is the time (in seconds) between two attempts to connect the server. Thus, the first number is the interval between the first and the second attempts, the second number is the interval between the second and the third attempts, etc. The default value is three attempts with time intervals between the first and the second attempts equal to 1000 and between the second and the third attempts equal to 2000 milliseconds.

Additionally, the **EmailFilter** component at `SMTP` and `MAIL` levels is capable to increase accuracy and eliminate false-negatives caused by servers incorporating greylisting. Greylisting is one of very common anti-spam techniques based on denial of delivery for unknown hosts. A host becomes known and "greylisted" (i.e. not allowed) when it retries its delivery after specified period of time, usually ranging from 1 to 5 minutes. Most spammers do not retry the delivery after initial failure just for the sake of high performance. **EmailFilter** has an anti-greylisting feature which retries each failed `SMTP/MAIL` test for specified number of times and delays. Only after the last retry fails, the address is considered as invalid.

4. **MAIL**

   At the fourth level (`MAIL`), if all has been successful, you can send a dummy message to the specified e-mail address. The message has the following properties: **Mail From**, **Mail Subject** and **Mail Body**. By default, the message is sent from `CloverETL <clover@cloveretl.org>`, its subject is `Hello, this is a test message`. And its default body is as follows: `Hello,\nThis is CloverETL test message. \n\nPlease ignore and don't respond. Thank you and have a nice day!`

SMTP Grey-Listing Attempts

To turn anti-greylisting feature, you can specify the **SMTP grey-listing attempts** attribute. Its default value is 30,120,240. These numbers means that four attempts can be made with time intervals between them that equal to 30 seconds (between the first and the second), 120 seconds (between the second and the third) and 240 seconds (between the third and the fourth). You can change the default values by any other comma separated sequence of integer numbers. The maximum number of parallel tasks that are performed when anti-greylisting is turned on is specified by the **SMTP concurrent limit** attribute. Its default value is 10.

ExtFilter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

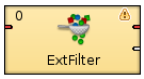
ExtFilter filters input records according to the specified condition.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
ExtFilter	-	no	1	1-2	-	-

Abstract

ExtFilter receives data records through single input port, compares them with the specified filter expression and sends those that are in conformity with this expression to the first output port. Rejected records are sent to the optional second output port.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For allowed data records	Input 0[1)]
	1	no	For rejected data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

ExtFilter Attributes

Attribute	Req	Description	Possible values
Basic			
Filter expression		Expression according to which the records are filtered. Expressed as the sequence of individual expressions for individual input fields separated from each other by semicolon.	

Advanced Description

Filter Expression

When you select this component, you must specify the expression according to which the filtering should be performed (**Filter expression**). The filtering expression consists of some number of subexpressions connected with logical operators (logical `and` and logical `or`) and parentheses for expressing precedence. For these subexpressions there exists a set of functions that can be used and set of comparison operators (greater than, greater than or equal to, less than, less than or equal to, equal to, not equal to). The latter can be selected in the **Filter editor** dialog as the mathematical comparison signs (>, >=, <, <=, ==, !=) or also their textual abbreviations can be used (`.gt.`, `.ge.`, `.lt.`, `.le.`, `.eq.`, `.ne.`). All of the record field values should be expressed by their port numbers preceded by dollar sign, dot and their names. For example, `$0.employeeid`.

Note

You can also use the Partition (p. 491) component as a filter instead of **ExtFilter**. With the Partition (p. 491) component you can define much more sofisticated filter expressions and distribute data records among more output ports.

Or you can use the Reformat (p. 502) component as a filter.

Important

You can use either CTL1, or CTL2 in **Filter Editor**.

The following two options are equivalent:

1. **For CTL1**

```
is_integer($0.field1)
```

2. **For CTL2**

```
//#CTL2
isInteger($0.field1)
```

ExtSort

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

ExtSort sorts input records according to a sort key.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
ExtSort	-	✖	1	1-N	-	-

Abstract

ExtSort changes the order in which records flow through a graph. How to compare two records is specified by a sorting key.

The **Sort key** is defined by one or more input fields and the sorting order (ascending or descending) for each field. The resulting sequence depends also on the key field type: `string` fields are sorted in ASCIIbetical order while the others alphabetically.

The component receives data records through the single input port, sorts them according to specified sort key and copies each of them to all connected output ports.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	the same input and output metadata [1]
Output	0	✔	for sorted data records	
	1-N	✖	for sorted data records	

[1] As all output metadata must be same as the input metadata, they can be propagated through this component.

ExtSort Attributes

Attribute	Req	Description	Possible values
Basic			
Sort key	✔	Key according to which the records are sorted. See Sort Key (p. 232) for more information.	
Advanced			
Buffer capacity		Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.	8000 (default) | 1-N
Number of tapes		Number of temporary files used to perform external sorting. Even number higher than 2.	6 (default) | 2*(1-N)
Temp directories		List of names of temporary directories that are used to create temporary files to perform external sorting separated by semicolon.	java.io.tmpdir system property (default) | other directories
Deprecated			
Sort order		Order of sorting (`Ascending` or `Descending`). Can be denoted by the first letter (`A` or `D`) only. The same for all key fields.	Ascending (default) | Descending
Sorting locale		Locale that should be used for sorting.	none (default) | any locale
Case sensitive		In the default setting of **Case sensitive** (`true`), upper-case and lower-case characters are sorted as distinct characters. Lower-cases precede corresponding upper-cases. If **Case sensitive** is set to `false`, upper-case characters and lower-case characters are sorted as if they were the identical.	true (default) | false
Sorter initial capacity		does the same as **Buffer capacity**	8000 (default) | 1-N

Advanced Description

Sorting Null Values

Remember that **ExtSort** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

FastSort

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

FastSort sorts input records using a sort key. **FastSort** is faster than **ExtSort** but requires more system resources.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
FastSort	-	✖	1	1-N	-	-

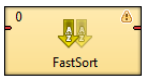
Abstract

FastSort is a high performance sort component reaching the optimal efficiency when enough system resources are available. **FastSort** can be up to 2.5 times faster than **ExtSort** but consumes significantly more memory and temporary disk space.

The component takes input records and sorts them using a sorting key - a single field or a set of fields. You can specify sorting order for each field in the key separately. The sorted output is sent to all connected ports.

Pretty good results can be obtained with the default settings (just the sorting key needs to be specified). However, to achieve the best performance, a number of parameters is available for tweaking.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	the same input and output metadata [1)]
Output	0	✔	for sorted data records	
	1-N	✖	for sorted data records	

[1)] As all output metadata must be same as the input metadata, they can be propagated through this component.

FastSort Attributes

Attribute	Req	Description	Possible values
Basic			
Sort key	✔	List of fields (separated by semicolon) the data records are to be sorted by, including the sorting order for each data field separately, see Sort Key (p. 232)	
Estimated record count	1)	Estimated number of input records to be sorted. A rough guess of the order of magnitude is sufficient, see Estimated Record Count (p. 476).	auto (default) | 1-N
In memory only		If `true`, internal sorting is forced and all attributes except **Sort key** and **Run size** are ignored.	false (default) | true
Temp directories		List of paths to temporary directories for storing sorted runs, separated by semicolons	system TEMP dir (default) | other dir
Advanced			
Run size (records)	1) 2)	Number of records sorted at once in memory. Largely affects speed and memory requirements, see Run Size (p. 476)	auto from (if set) **Estimated record count** | 20,000 default | 1000 - N
Max open files		Limits the number of temp files that can be created during the sorting. Too low number (500 or less) significantly reduces the performance, see Max Open Files (p. 476).	unlimited (default) | 1-N
Concurrency (threads)		Number of worker threads to do the job. The default value ensures the optimal results while overriding the default may even slow the graph run down, see Concurrency (p. 477).	auto (default) | 1-N
Number of read buffers	2)	How many chunks of data will be held in memory at a time, see Number of Read Buffers (p. 477).	auto (default) | 1-N
Average record size (bytes)	2)	Guess on average byte size of records, see Average Record Size (p. 477).	auto (default) | 1-N
Maximum memory (MB, GB)	2)	Rough estimate of maximum memory that can be used, see Maximum Memory (p. 477).	auto (default) | 1-N
Tape buffer (bytes)		Buffer used by a worker for filling the output. Affects the performance slightly, see Tape Buffer (p. 477).	8192 (default) | 1-N
Compress temporary files		If `true`, temporary files are compressed. For more information see Compress Temporary Files (p. 477).	false (default) | true
Deprecated			
Sorting locale		Locale used for correct sorting order	none (default) | any locale

Attribute	Req	Description	Possible values
Case sensitive		By default (**Sorting locale** is `none`), upper-case characters are sorted separately and precede lower-case characters that are sorted separately too. If **Sorting locale** is set, upper- and lower-case characters are sorted together - if **Case sensitive** is `true`, a lower-case precedes corresponding upper-case while `false` preservers the order, data strings appears in the input in.	false (default) | true

[1)]**Estimated record count** is a helper attribute which is used for calculating (rather unnatural) **Run size** automatically as approximately **Estimated record count** to the power 0.66. If **Run size** set explicitly, **Estimated record count** is ignored.
Reasonable **Run size**s vary from 5,000 to 200,000 based on the record size and the total number of records.

[2)]These attributes affect automatic guess of **Run size**. Generally, the following formula must be true:
Number of read buffers * **Average record size** < **Maximum memory**

Advanced Description

Sorting Null Values

Remember that **FastSort** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

FastSort Tweaking

Basically, you do not need to set any of these attributes, however, sometimes you can increase performance by setting them. You may have a limited memory or you need to sort agreat number of records, or these records are too big. In similar cases, you can fit **FastSort** to your needs.

1. **Estimated Record Count**

   Basic attribute which lets **FastSort** know a rough number of records it will have to deal with. The attribute is complementary to **Run size**; you don't need to set it if **Run size** is specified. On the other hand, if you don't want to play with attributes setting much, giving the rough number of records spares memory to be allocated during the graph run. Based on this count, **Maximum memory**, records size, etc., **Run size** is determined.

2. **Run Size**

   The core attribute for **FastSort**; determines how many records form a "run" (i.e., a bunch of sorted records in temp files). The less **Run size**, the more temp files get created, less memory is used and greater speed is achieved. On the other hand, higher values might cause memory issues. There is no rule of thumb as to whether **Run size** should be high or low to get the best performance. Generally, the more records you are about to sort the bigger **Run size** you might want. The rough formula for **Run size** is **Estimated record count**^0.66. Note that memory consumption multiplies with **Number of read buffers** and **Concurrency**. So, higher **Run size**s result in much higher memory footprints.

3. **Max Open Files**

   FastSort uses relatively large numbers of temporary files during its operation. In case you hit quota or OS-specific limits, you can limit the maximum number of files to be created. The following table should give you a better idea:

Dataset size	Number of temp. files	Default Run size	Note
1,000,000	~100	~10,000	
10,000,000	~250	~45,000	
1,000,000,000	20,000 to 2,000	50,000 to 500,000	Depends on available memory

Note that numbers in the table above are not exact and might be different on your system. However, sometimes such large numbers of files might cause problems hitting user quotas or other runtime limitations, see Performance Bottlenecks (p. 477) for a help how to solve such issues.

4. **Concurrency**

   Tells **FastSort** how many runs (chunks) should be sorted at a time in parallel. By default, it is automatically set to 1 or 2 based on the number of CPU cores in your system. Overriding this value makes sense if your system has lots of CPU cores and you think your disk performance can handle working with so many parallel data streams.

5. **Maximum Memory**

   You can set the maximum amount of memory dedicated to a single component. This is a guide for **FastSort** when computing **Run size**, i.e., if **Run size** is set explicitly, this setting is ignored. A unit must be specified, e.g., '200MB', '1gb', etc.

6. **Average Record Size**

   You can set **Average record size** in bytes. If omitted, it will be computed as an average record size from the first 1000 parsed records.

7. **Number of Read Buffers**

   This setting corresponds tightly to the number of threads (**Concurrency**) - must be equal to or greater than **Concurrency**. The more read buffers the less change the workers will block each other. Defaults to **Concurrency** + 2

8. **Compress Temporary Files**

   Along with **Temporary files charset** this option lets you reduce the space required for temporary files. Compression can save a lot of space but affects performance by up to 30% down so be careful with this setting.

9. **Tape Buffer**

   Size (in bytes) of a file output buffer. The default value is 8kB. Decreasing this value might avoid memory exhaustion for large numbers of runs (e.g. when **Run size** is very small compared to the total number of records). However, the impact of this setting is quite small.

Tips & Tricks

- Be sure you have dedicated enough memory to your Java Virtual Machine (JVM). Having plenty of memory available, **FastSort** is capable of doing astonishing job. Remember that the default JVM heap space 64MB can cause **FastSort** to crash. Don't hesitate to increase the memory value up to 2 GB (but still leaving some memory for the operating system). It is well worth it. How to set the JVM is described in Program and VM Arguments (p. 73) section.

Performance Bottlenecks

- *Sorting big records (long string fields, tens or hundreds of fields, etc.):* **FastSort** is greedy for both memory and CPU cores. If the system does not have enough of either, **FastSort** can easily crash with out-of-memory. In this case, use the **ExtSort** component instead.

- *Utilizing more than 2 CPU cores:* Unless you you are able to use really fast disk drives, overriding the default value of **Concurrency** to more than 2 threads does not neccessarily help. It can even slow the process back down a bit as extra memory is loaded for each additional thread.

- *Coping with quotas and other runtime limitations:* In complex graphs with several parallel sorts, even with other graph components also having huge number of open files, `Too many open files` error and graph execution failure may occur. There are two possible solutions to this issue:

1. increase the limit (quota)

   This option is recommended for production systems since there is no speed sacrifice. Typically, setting limit to higher number on Unix systems.

2. force **FastSort** to keep the number of temporary files below some limit

   For regular users on large servers increasing the quota is not an option. Thus, **Max open files** must be set to a reasonable value. **FastSort** then performs intermediate merges of temporary files to keep their number below the limit. However, setting **Max open files** to values, for which such merges are inevitable, often produces significant performance drop. So keep it at the highest possible value. If you are forced to limit **FastSort** to less than a hundred temporary files, even for large datasets, consider using **ExtSort** instead which is designed for performance with limited number of tapes.

Merge

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Merge merges and sorts data records from two or more inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Merge	yes	yes	2-n	1	-	-

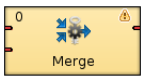
Abstract

Merge receives sorted data records through two or more input ports. (Metadata of all input ports must be the same.) It gathers all input records and sorts them in the same way on the output.

Important

Remember that all key fields must be sorted in ascending order!

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0-1	yes	For input data records	Any
	2-n	no	For input data records	Input 0[1)]
Output	0	yes	For merged data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

Merge Attributes

Attribute	Req	Description	Possible values
Basic			
Merge key	yes	Key according to which the sorted records are merged. (Remember that all key fields must be sorted in ascending order!) See Group Key (p. 231) for more information.[1)]	
Equal NULL		By default, records with null values of key fields are considered to be different. If set to `true`, they are considered to be equal.	false (default) | true

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

MetaPivot

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the appropriate **Transformer** for your purpose, see Transformers Comparison (p. 273).

Short Summary

MetaPivot converts every incoming record into several output records, each one representing a single field from the input.

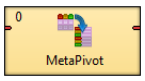
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
MetaPivot	-	no	1	1	no	no

Abstract

On its single input port, **MetaPivot** receives data that do not have to be sorted. Each *field* of the input record is written as a new *line* on the output. The metadata represent data types and are restricted to a fixed format, see Advanced Description (p. 482) All in all, **MetaPivot** can be used to effectively transform your records to a neat data-dependent structure.

Unlike Normalizer (p. 484), which **MetaPivot** is derived from, no transofmation is defined. **MetaPivot** always does the same tranformation: it takes the input records and "rotates them" thus turning input columns to output rows.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For transformed data records	Any2

MetaPivot Attributes

MetaPivot has no component-specific attributes.

Advanced Description

When working with MetaPivot, you have to use a fixed format of the output metadata. The metadata fields represent particular data types. Field names and data types have to be set *exactly as follows* (otherwise unexpected `BadDataFormatException` will occur):

`[recordNo long]` - the serial number of a record (outputs can be later grouped by this)

`[fieldNo integer]` - the current field number: 0...n-1 where n is the number of fields in the input metadata

`[fieldName string]` - name of the field as it appeared on the input

`[fieldType string]` - the field type, e.g. "string", "date", "decimal"

`[valueBoolean boolean]` - the boolean value of the field

`[valueByte byte]` - the byte value of the field

`[valueDate date]` - the date value of the field

`[valueDecimal decimal]` - the decimal value of the field

`[valueInteger integer]` - the integer value of the field

`[valueLong long]` - the long value of the field

`[valueNumber number]` - the number value of the field

`[valueString string]` - the string value of the field

The total number of output records produced by **MetaPivot** equals to (number of input records) * (number of input fields).

You may have noticed some of the fields only make the output look better arranged. That is true - if you needed to omit them for whatever reasons, you can do it. The only three fields that do not have to be included in the output metadata are: `recordNo`, `fieldNo` and `fieldType`.

Example 46.4. Example MetaPivot Transformation

Let us now look at what **MetaPivot** makes with your data. Say you have an input file containg data of various types separated into fields. You have only two records:

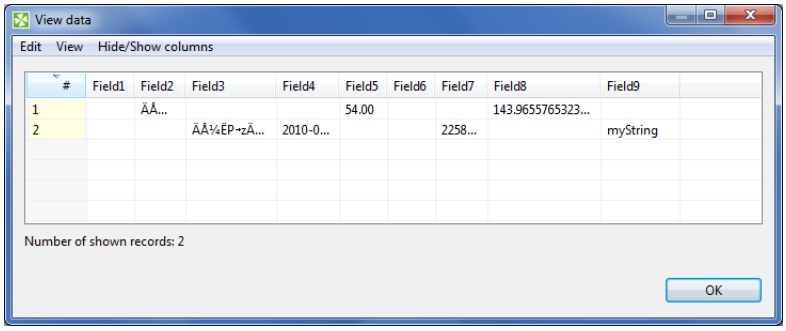


Figure 46.6. Example MetaPivot Input

Sending these data to **MetaPivot** "classifies" the data to output fields corresponding to their data types:

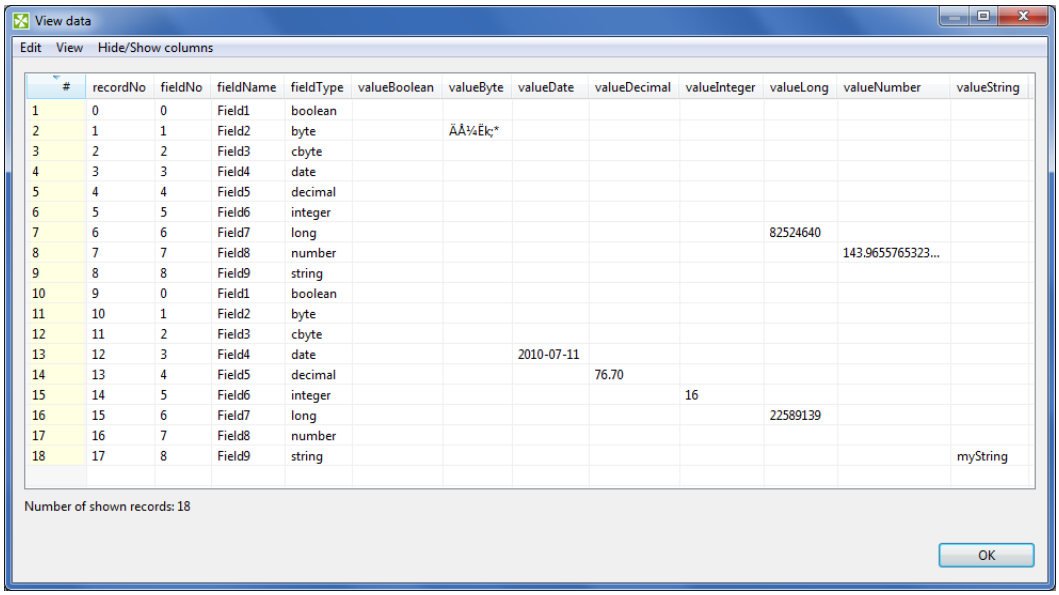


Figure 46.7. Example MetaPivot Output

Thus e.g. "myString" is placed in the **valueString** field or "76.70" in the **valueDecimal**. Since there were 2 records and 9 fields on the input, we have got 18 records on the output.

Normalizer

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Normalizer creates one or more output records from each single input record.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Normalizer	-	no	1	1	yes	yes

Abstract

Normalizer receives potentially unsorted data through single input port, decomposes input data records and composes one or more output records from each input record.

A transformation must be defined. The transformation uses a CTL template for **Normalizer**, implements a `RecordNormalize` interface or inherits from a `DataRecordNormalize` superclass. The interface methods are listed below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For normalized data records	Any2

Normalizer Attributes

Attribute	Req	Description	Possible values
Basic			
Normalize	1)	Definition of the way how records should be normalized written in the graph in CTL or Java.	
Normalize URL	1)	Name of external file, including path, containing the definition of the way how records should be normalized written in CTL or Java.	
Normalize class	1)	Name of external class defining the way how records should be normalized.	
Normalize source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Legend:

1): One of these must specified. Any of these transformation attributes uses a CTL template for **Normalizer** or implements a `RecordNormalize` interface.

See CTL Scripting Specifics (p. 485) or Java Interfaces for Normalizer (p. 490) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify the way how input should be transformed into output.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

Once you have written your transformation, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

CTL Templates for Normalizer

The **Source** tab for defining the transformation looks like this:

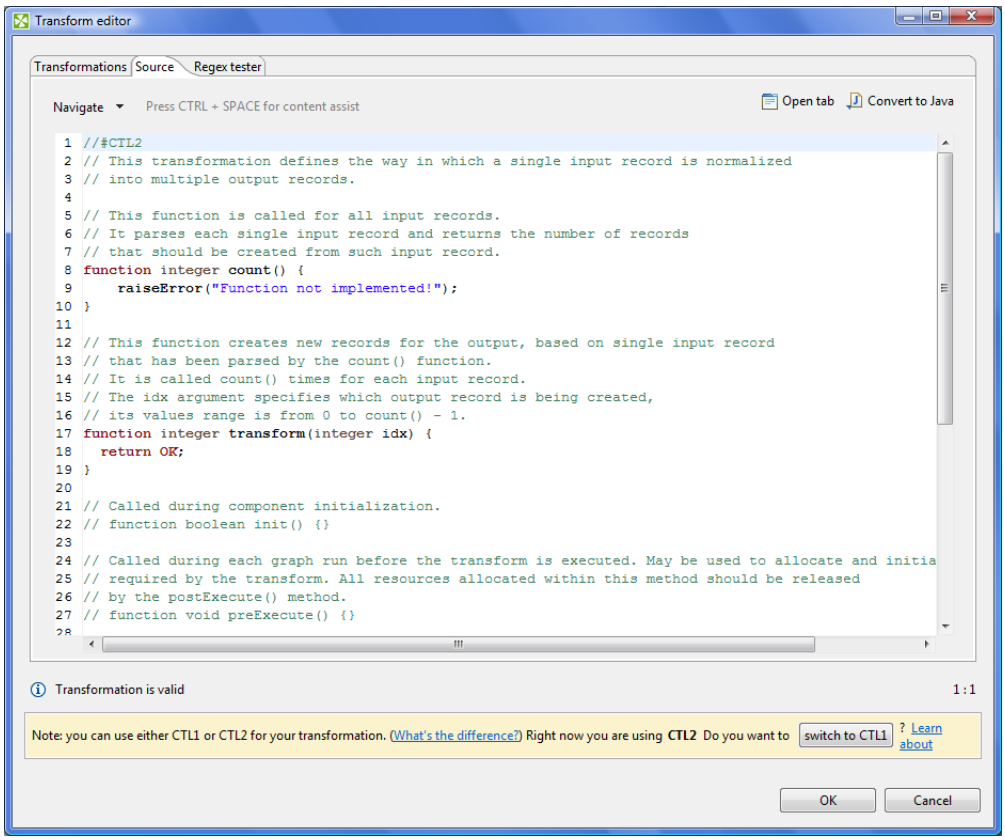


Figure 46.8. Source Tab of the Transform Editor in the Normalizer Component (I)

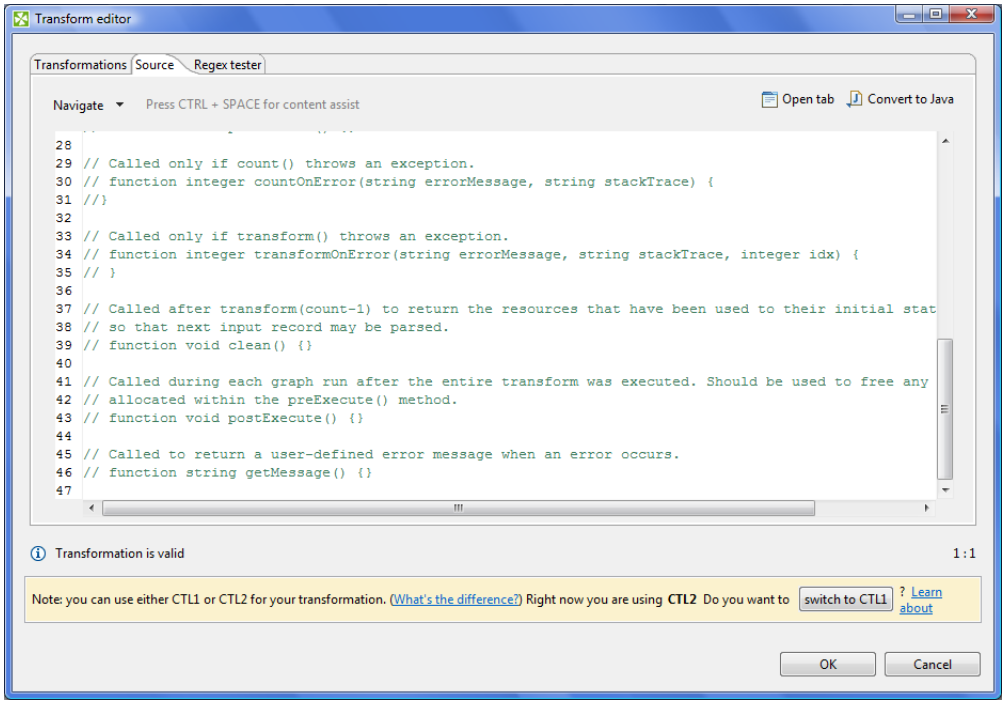


Figure 46.9. Source Tab of the Transform Editor in the Normalizer Component (II)

Table 46.3. Functions in Normalizer

CTL Template Functions	
boolean init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`true` | `false` (in case of `false` graph fails)
integer count()	
Required	yes
Input Parameters	none
Returns	For each input record returns one integer number greater than 0. The returned number is equal to the the amount of new output records that will be created by the **transform()** function.
Invocation	Called repeatedly, once for each input record
Description	For each input record it generates the number of output records that will be created from this input. If any of the input records causes fail of the `count()` function, and if user has defined another function (`countOnError()`), processing continues in this `countOnError()` at the place where `count()` failed. If `count()` fails and user has not defined any `countOnError()`, the whole graph will fail. The `countOnError()` function gets the information gathered by `count()` that was get from previously successfully processed input records. Also error message and stack trace are passed to `countOnError()`.

Example	`function integer count() {`
	`  customers = split($0.customers,"-");`
	`  return length(customers);`
	`}`

integer transform(integer idx)	
Required	yes
Input Parameters	`integer idx` integer numbers from `0` to `count-1` (Here `count` is the number returned by the **transform()** function.)
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly, once for each output record
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

CTL Template Functions	

Example	`function integer transform(integer idx) {`
	`  myString = customers[idx];`
	`  $0.OneCustomer = str2integer(myString);`
	`  $0.RecordNo = $0.recordNo;`
	`  $0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

void clean()	
Required	no
Input Parameters	none
Returns	void
Invocation	Called repeatedly, once for each input record (after the last output record has been created from the input record).
Description	Returns the component to the initial settings

Example	`function void clean() {`
	`  clear(customers);`
	`}`

integer countOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	For each input record returns one integer number greater than 0. The returned number is equal to the the amount of new output records that will be created by the **transform()** function.
Invocation	Called if `count()` throws an exception.
Description	For each input record it generates the number of output records that will be created from this input. If any of the input records causes fail of the `count()` function, and if user has defined another function (`countOnError()`), processing continues in this `countOnError()` at the place where `count()` failed. If `count()` fails and user has not defined any `countOnError()`, the whole graph will fail. The `countOnError()` function gets the information gathered by `count()` that was get from previously successfully processed input records. Also error message and stack trace are passed to `countOnError()`.

Example	`function integer countOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  return 1;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer idx)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`integer idx`

CTL Template Functions	
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer idx) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`  $0.OneCustomerOnError = customers[idx];`
	`  $0.RecordNo = $recordNo;`
	`  $0.OrderWithinRecord = idx;`
	`  return OK;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user
Invocation	Called in any time specified by user (called only when either `count()`, `transform()`, `countOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.

CTL Template Functions	
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Input records or fields**

  Input records or fields are accessible within the `count()` and `countOnError()` functions only.

- **Output records or fields**

  Output records or fields are accessible within the `transform()` and `transformOnError()` functions only.

- All of the other CTL template functions allow to access neither inputs nor outputs.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Normalizer

The transformation implements methods of the `RecordNormalize` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of `RecordNormalize` interface:

- `boolean init(Properties parameters, DataRecordMetadata sourceMetadata, DataRecordMetadata targetMetadata)`

  Initializes normalize class/function. This method is called only once at the beginning of normalization process. Any object allocation/initialization should happen here.

- `int count(DataRecord source)`

  Returns the number of output records which will be created from specified input record.

- `int countOnError(Exception exception, DataRecord source)`

  Called only if `count(DataRecord)` throws an exception.

- `int transform(DataRecord source, DataRecord target, int idx)`

  `idx` is a sequential number of output record (starting from 0). See Return Values of Transformations (p. 238) for detailed information about return values and their meaning. In **Normalizer**, only `ALL`, `0`, `SKIP`, and **Error codes** have some meaning.

- `int transformOnError(Exception exception, DataRecord source, DataRecord target, int idx)`

  Called only if `transform(DataRecord, DataRecord, int)` throws an exception.

- `void clean()`

  Finalizes current round/clean after current round - called after the transform method was called for the input record.

Partition

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Partition distributes individual input data records among different output ports.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Partition	-	no	1	1-n	yes/no[1)]	yes/no[1)]

Legend

1) **Partition** can use either transformation or other two attributes (**Ranges** and/or **Partition key**). A transformation must be defined unless at least one of these is specified.

Abstract

Partition distributes individual input data records among different output ports.

To distribute data records, user-defined transformation, ranges of **Partition key** or RoundRobin algorithm may be used. Ranges of **Partition key** are either those specified in the **Ranges** attribute or calculated hash values. It uses a CTL template for **Partition** or implements a `PartitionFunction` interface. Its methods are listed below. In this component no mapping may be defined since it does not change input data records. It only distributes them unchanged among output ports.

Tip

Note that you can use the **Partition** component as a filter similarly to **ExtFilter**. With the **Partition** component you can define much more sophisticated filter expressions and distribute input data records among more outputs than 2.

Neither **Partition** nor **ExtFilter** allow to modify records.

Important

Partition is high-performance component, thus you cannot modify input and output records - it would result in an error. If you need to do so, consider using **Reformat** instead.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For output data records	Input 0[1)]
	1-N	no	For output data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component.

Partition Attributes

Attribute	Req	Description	Possible values
Basic			
Partition	1)	Definition of the way how records should be distributed among output ports written in the graph in CTL or Java.	
Partition URL	1)	Name of external file, including path, containing the definition of the way how records should be distributed among output ports written in CTL or Java.	
Partition class	1)	Name of external class defining the way how records should be distributed among output ports.	
Ranges	1),2)	Ranges expressed as a sequence of individual ranges separated from each other by semicolon. Each individual range is a sequence of intervals for some set of fields that are adjacent to each other without any delimiter. It is expressed also whether the minimum and maximum margin is included to the interval or not by bracket and parenthesis, respectively. Example of **Ranges**: `<1,9)(,31.12.2008);<1,9)<31.12.2008,);<9,)(,31.12.2008); <9,)<31.12.2008)`.	
Partition key	1),2)	Key according to which input records are distributed among different output ports. Expressed as the sequence of individual input field names separated from each other by semicolon. Example of **Partition key**: `first_name;last_name`.	
Advanced			
Partition source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default) | other encoding
Deprecated			

Attribute	Req	Description	Possible values
Locale		Locale to be used when internationalization is set to `true`. By default, system value is used unless value of **Locale** specified in the `defaultProperties` file is uncommented and set to the desired **Locale**. For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).	system value or specified default value (default) | other locale
Use internationalization		By default, no internationalization is used. If set to `true`, sorting according national properties is performed.	false (default) | true

Legend:

1): If one of these transformation attributes is specified, both **Ranges** and **Partition key** will be ignored since they have less priority. Any of these transformation attributes must use a CTL template for **Partition** or implement a `PartitionFunction` interface.

See CTL Scripting Specifics (p. 493) or Java Interfaces for Partition (and ClusterPartitioner) (p. 497) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

2): If no transformation attribute is defined, **Ranges** and **Partition key** are used in one of the following three ways:

- Both **Ranges** and **Partition key** are set.

  The records in which the values of the fields are inside the margins of specified range will be sent to the same output port. The number of the output port corresponds to the order of the range within all values of the fields.

- **Ranges** are not defined. Only **Partition key** is set.

  Records will be distributed among different output ports as described above. Hash values will be calculated and used as margins as if they were specified in **Ranges**.

- Neither **Ranges** nor **Partition key** are defined.

  RoundRobin algorithm will be used to distribute records among output ports.

CTL Scripting Specifics

When you define any of the three transformation attributes, which is optional, you must specify a transformation that assigns a number of output port to each input record.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

Partition uses the following transformation teplate:

CTL Templates for Partition (or ClusterPartitioner)

This transformation template is used in **Partition**, and **ClusterPartitioner**.

Once you have written your transformation in CTL, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

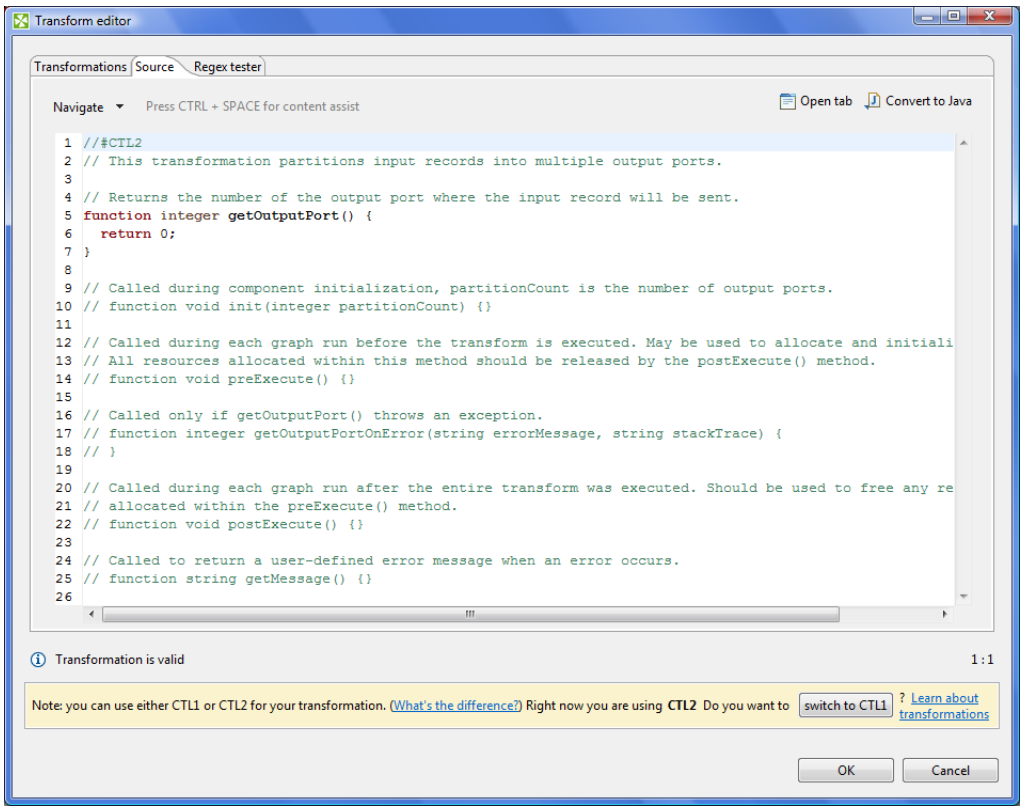


Figure 46.10. Source Tab of the Transform Editor in the Partitioning Component

You can open the transformation definition as another tab of a graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

Table 46.4. Functions in Partition (or ClusterPartitioner)

CTL Template Functions	
void init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`void`
integer getOutputPort()	
Required	yes
Input Parameters	none
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly for each input record

CTL Template Functions	
Description	It does not transform the records, it does not change them nor remove them, it only returns integer numbers. Each of these returned numbers is a number of the output port to which individual record should be sent. In **ClusterPartitioner**, these ports are virtual and mean Cluster nodes. If any part of the `getOutputPort()` function for some output record causes fail of the `getOutputPort()` function, and if user has defined another function (`getOutputPortOnError()`), processing continues in this `getOutputPortOnError()` at the place where `getOutputPort()` failed. If `getOutputPort()` fails and user has not defined any `getOutputPortOnError()`, the whole graph will fail. The `getOutputPortOnError()` function gets the information gathered by `getOutputPort()` that was get from previously successfully processed code. Also error message and stack trace are passed to `getOutputPortOnError()`.

Example	`function integer getOutputPort() {`
	`  switch (expression) {`
	`  case const0 : return 0; break;`
	`  case const1 : return 1; break;`
	`  ...`
	`  case constN : return N; break;`
	`  [default : return N+1;]`
	`  }`
	`}`

integer getOutputPortOnError(string errorMessage, string stackTrace)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `getOutputPort()` throws an exception.
Description	It does not transform the records, it does not change them nor remove them, it only returns integer numbers. Each of these returned numbers is a number of the output port to which individual record should be sent. In **ClusterPartitioner**, these ports are virtual and mean Cluster nodes. If any part of the `getOutputPort()` function for some output record causes fail of the `getOutputPort()` function, and if user has defined another function (`getOutputPortOnError()`), processing continues in this `getOutputPortOnError()` at the place where `getOutputPort()` failed. If `getOutputPort()` fails and user has not defined any `getOutputPortOnError()`, the whole graph will fail. The `getOutputPortOnError()` function gets the information gathered by `getOutputPort()` that was get from previously successfully processed code. Also error message and stack trace are passed to `getOutputPortOnError()`.

CTL Template Functions	

Example	`function integer getOutputPortOnError(`
	`  string errorMessage,`
	`  string stackTrace) {`
	`  printErr(errorMessage);`
	`  printErr(stackTrace);`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user
Invocation	Called in any time specified by user (called only when either `getOutputPort()` or `getOutputPotOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Input records or fields**

  Input records or fields are accessible within the `getOutputPort()` and `getOutputPortOnError()` functions only.

- **Output records or fields**

  Output records or fields are not accessible at all as records are mapped to the output without any modification and mapping.

- All of the other CTL template functions allow to access neither inputs nor outputs.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Partition (and ClusterPartitioner)

The transformation implements methods of the `PartitionFunction` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of `PartitionFunction` interface:

- `void init(int numPartitions,RecordKey partitionKey)`

  Called before `getOutputPort()` is used. The `numPartitions` argument specifies how many partitions should be created. The `RecordKey` argument is the set of fields composing key based on which the partition should be determined.

- `boolean supportsDirectRecord()`

  Indicates whether partition function supports operation on serialized records /aka direct. Returns `true` if `getOutputPort(ByteBuffer)` method can be called.

- `int getOutputPort(DataRecord record)`

  Returns port number which should be used for sending data out. See Return Values of Transformations (p. 238) for more information about return values and their meaning.

- `int getOutputPortOnError(Exception exception, DataRecord record)`

  Returns port number which should be used for sending data out. Called only if `getOutputPort(DataRecord)` throws an exception.

- `int getOutputPort(ByteBuffer directRecord)`

  Returns port number which should be used for sending data out. See Return Values of Transformations (p. 238) for more information about return values and their meaning.

- `int getOutputPortOnError(Exception exception, ByteBuffer directRecord)`

  Returns port number which should be used for sending data out. Called only if `getOutputPort(ByteBuffer)` throws an exception.

Pivot

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the appropriate **Transformer** for your purpose, see Transformers Comparison (p. 273).

Short Summary

Pivot produces a pivot table. The component creates a data summarization record for every group of input records. A group can be identified either by a key or its size.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Pivot	-	no	1	1	yes	yes

Note: When using the key attribute, input records should be sorted, though. See Advanced Description (p. 499).

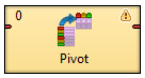
Abstract

The component reads input records and treats them as groups. A group is defined either by a key or a number of records forming the group. **Pivot** then produces a single record from each group. In other words, the component creates a pivot table.

Pivot has two principal attributes which instruct it to treat some input values as output field names and other inputs as output values.

The component is a simple form of Denormalizer (p. 456).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For summarization data records	Any2

Pivot Attributes

Attribute	Req	Description	Possible values
Basic			
Key	1)	The key is a set of fields used to identify groups of input records (more than one field can form a key). A group is formed by a sequence of records with identical key values.	any input field
Group size	1)	The number of input records forming one group. When using Group size, the input data do not have to be sorted. **Pivot** then reads a number of records and transforms them to one group. The number is just the value of Group size.	<1; n>
Field defining output field name	2)	The input field whose value "maps" to a field name on the output.	
Field defining output field value	2)	The input field whose value "maps" to a field value on the output.	
Sort order		Groups of input records are expected to be sorted in the order defined here. The meaning is the same as in Denormalizer, see Sort order (p. 457). Beware that in Pivot, setting this to `Ignore` can produce unexpected results if input is not sorted.	Auto (default) | Ascending | Descending | Ignore
Equal NULL		Determines whether two fields containing null values are considered equal.	true (default) | false
Advanced			
Pivot transformation	3)	Using CTL or Java, you can write your own records transformation here.	
Pivot transformation URL	3)	The path to an external file which defines how to transform records. The transformation can be written in CTL or Java.	
Pivot transformation class	3)	The name of a class that is used for data transformation. It can be written in Java.	
Pivot transformation source charset		The encoding of an external file defining the data transformation.	ISO-8859-1 (default) | any
Deprecated			
Error actions		Defines actions that should be performed when the specified transformation returns an **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file which error messages should be written to. These messages are generated during **Error actions**, see above. If the attribute is not set, messages are written to **Console**.	

Legend:

1): One of the **Key** or **Group size** attributes has to be always set.

2): These two values can either be given as an attribute or in your own transformation.

3): One of these attributes has to be set if you do not control the transformation by means of **Field defining output field name** and **Field defining output field value**.

Advanced Description

You can define the data transformation in two ways:

1) Set the **Key** or **Group size** attributes. See Group Data by Setting Attributes (p. 500).

2) Write the transformation yourself in CTL/Java or provide it in an external file/Java class. See Define Your Own Transformation - Java/CTL (p. 501).

Group Data by Setting Attributes

If you group data using the **Key** attribute your input should be sorted according to **Key** values. To tell the component how your input is sorted, specify **Sort order**. If the **Key** fields appear in the output metadata as well, **Key** values are copied automatically.

While when grouping with the **Group size** attribute, the component ignores the data itself and takes e.g. 3 records (for Group size = 3) and treats them as one group. Naturally, you have to have an adequate number of input records otherwise errors on reading will occur. The number has to be a multiple of **Group size**, e.g. 3, 6, 9 etc. for Group size = 3.

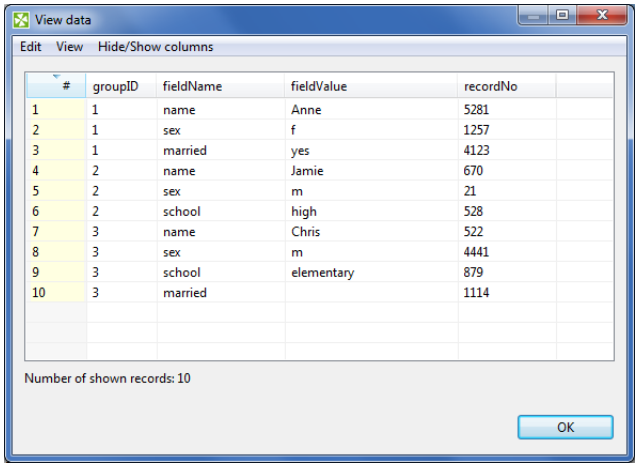
Then there are the two major attributes which describe the "mapping". They say:

- which input field's value will designate the output field - **Field defining output field name**
- which input field's value will be used as a value for that field **Field defining output field value**

As for the output metadata, it is arbitrary but fixed to field names. If your input data has extra fields, they are simply ignored (only fields defined as a value/name matter). Likewise output fields without any corresponding input records will be null.

Example 46.5. Data Transformation with Pivot - Using Key

Let us have the following input txt file with comma-separated values:



#	groupID	fieldName	fieldValue	recordNo
1	1	name	Anne	5281
2	1	sex	f	1257
3	1	married	yes	4123
4	2	name	Jamie	670
5	2	sex	m	21
6	2	school	high	528
7	3	name	Chris	522
8	3	sex	m	4441
9	3	school	elementary	879
10	3	married		1114

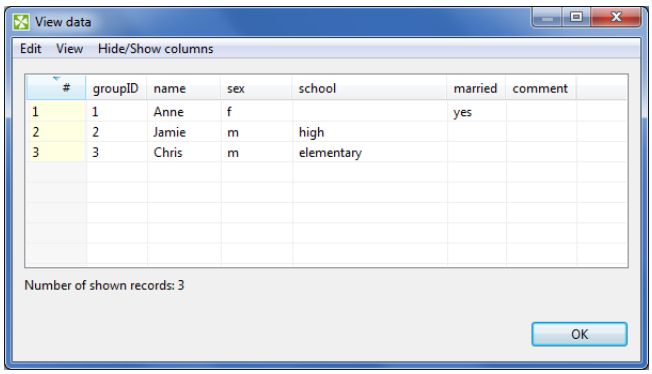
Because we are going to group the data according to the `groupID` field, the input has to be sorted (mind the ascending order of groupIDs). In the **Pivot** component, we will make the following settings:

Key = `groupID` (to group all input records with the same groupID)

Field defining output field name = `fieldName` (to say we want to take output fields' names from this input field)

Field defining output field value = `fieldValue` (to say we want to take output fields' values from this input field)

Processing that data with **Pivot** produces the following output:



Notice the input `recordNo` field has been ignored. Similarly, the output `comment` had no corresponding fields on the input, that is why it remains null. `groupID` makes part in the output metadata and thus was copied automatically.

Note

If the input is not sorted (not like in the example), grouping records according to their count is especially handy. Omit **Key** and set **Group size** instead to read sequences of records that have exactly the number of records you need.

Define Your Own Transformation - Java/CTL

In **Pivot**, you can write the transformation function yourself. That can be done either in CTL or Java, see Advanced attributes in Pivot Attributes (p. 499)

Before writing the transformation, you might want to refer to some of the sections touching the subject:

- Defining Transformations (p. 234)
- writing transformations in **Denormalizer**, the component **Pivot** is derived from: CTL Scripting Specifics (p. 458) and Java Interfaces for Denormalizer (p. 464)

Java

Compared to **Denormalizer**, the **Pivot** component has new significant attributes: `nameField` and `valueField`. These can be defined either as attributes (see above) or by methods. If the transformation is not defined, the component uses `com.opensys.cloveretl.component.pivot.DataRecordPivotTransform` which copies values from `valueField` to `nameField`.

In Java, you can implement your own `PivotTransform` that overrides `DataRecordPivotTransform`. However, you can override only one method, e.g. `getOutputFieldValue`, `getOutputFieldIndex` or others from `PivotTransform` (that extends `RecordDenormalize`).

CTL

In CTL1/2, too, you can set one of the attributes and implement the other one with a method. So you can e.g. set `valueField` and implement `getOutputFieldIndex`. Or you can set `nameField` and implement `getOutputFieldValue`.

In the compiled mode, the `getOutputFieldValue` and `getOutputFieldValueonError` methods cannot be overridden. When the transformation is written in CTL, the default `append` and `transform` methods are always performed before the user defined ones.

For a better understanding, examine the methods' documentation directly in the **Transform editor**.

Reformat

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Reformat manipulates record's structure or content.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Reformat	-	✖	1	1-N	✔	✔

Abstract

Reformat receives potentially unsorted data through single input port, transforms each of them in a user-specified way and sents the resulting record to the port(s) specified by user. Return values of the transformation are numbers of output port(s) to which data record will be sent.

A transformation must be defined. The transformation uses a CTL template for **Reformat**, implements a `RecordTransform` interface or inherits from a `DataRecordTransform` superclass. The interface methods are listed below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	✔	for input data records	Any(In0)
Output	0	✔	for transformed data records	Any(Out0)
	1-n	✖	for transformed data records	Any(OutPortNo)

Reformat Attributes

Attribute	Req	Description	Possible values
Basic			
Transform	1)	Definition of how records should be intersected written in the graph in CTL or Java.	
Transform URL	1)	Name of external file, including path, containing the definition of the way how records should be intersected written in CTL or Java.	
Transform class	1)	Name of external class defining the way how records should be intersected.	
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Legend:

1): One of these must specified. Any of these transformation attributes uses a CTL template for **Reformat** or implements a `RecordTransform` interface.

See CTL Scripting Specifics (p. 503) or Java Interfaces for Reformat (p. 504) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Use Reformat To

- Drop unwanted fields
- Validate fields using functions or regular expressions
- Calculate new or modify existing fields
- Convert data types

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify a transformation that assigns a number of output port to each input record.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

CTL Templates for Reformat

Reformat uses the same transformation teplate as **DataIntersection** and **Joiners**. See CTL Templates for Joiners (p. 278) for more information.

Java Interfaces for Reformat

Reformat implements the same interface as **DataIntersection** and **Joiners**. See Java Interfaces for Joiners (p. 281) for more information.

Rollup

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

Rollup creates one or more output records from one or more input records.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
Rollup	-	no	1	1-N	yes	yes

Abstract

Rollup receives potentially unsorted data through single input port, transforms them and creates one or more output records from one or more input records.

Component can sent different records to different output ports as specified by user.

A transformation must be defined. The transformation uses a CTL template for **Rollup**, implements a `RecordRollup` interface or inherits from a `DataRecordRollup` superclass. The interface methods are listed below.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any(In0)
Output	0	yes	For output data records	Any(Out0)
	1-N	no	For output data records	Any(Out1-N)

Rollup Attributes

Attribute	Req	Description	Possible values
Basic			
Group key		Key according to which the records are considered to be included into one group. Expressed as the sequence of individual input field names separated from each other by semicolon. See Group Key (p. 231) for more information. If not specified, all records are considered to be members of a single group.	
Group accumulator		ID of metadata that serve to create group accumulators. Metadata serve to store values used for transformation of individual groups of data records.	no metadata (default) | any metadata
Transform	1)	Definition of the transformation written in the graph in CTL or Java.	
Transform URL	1)	Name of external file, including path, containing the definition of the transformation written in CTL or Java.	
Transform class	1)	Name of external class defining the transformation.	
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Sorted input		By default, records are considered to be sorted. Either in ascending or descending order. Different fields may even have different sort order. If your records are not sorted, switch this attribute to `false`.	true (default) | false
Equal NULL		By default, records with null values of key fields are considered to be equal. If set to `false`, they are considered to be different from each other.	true (default) | false

Legend:

1): One of these must specified. Any of these transformation attributes uses a CTL template for **Rollup** or implements a `RecordRollup` interface.

See CTL Scripting Specifics (p. 506) or Java Interfaces for Rollup (p. 515) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

CTL Scripting Specifics

When you define any of the three transformation attributes, you must specify the way how input should be transformed into output.

Transformations implement a `RecordRollup` interface or inherit from a `DataRecordRollup` superclass. Below is the list of `RecordRollup` interface methods. See Java Interfaces for Rollup (p. 515) for detailed information this interface.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

Once you have written your transformation, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

You can open the transformation definition as another tab of the graph (in addition to the **Graph** and **Source** tabs of **Graph Editor**) by clicking corresponding button at the upper right corner of the tab.

CTL Templates for Rollup

Here is an example of how the **Source** tab for defining the transformation looks like:

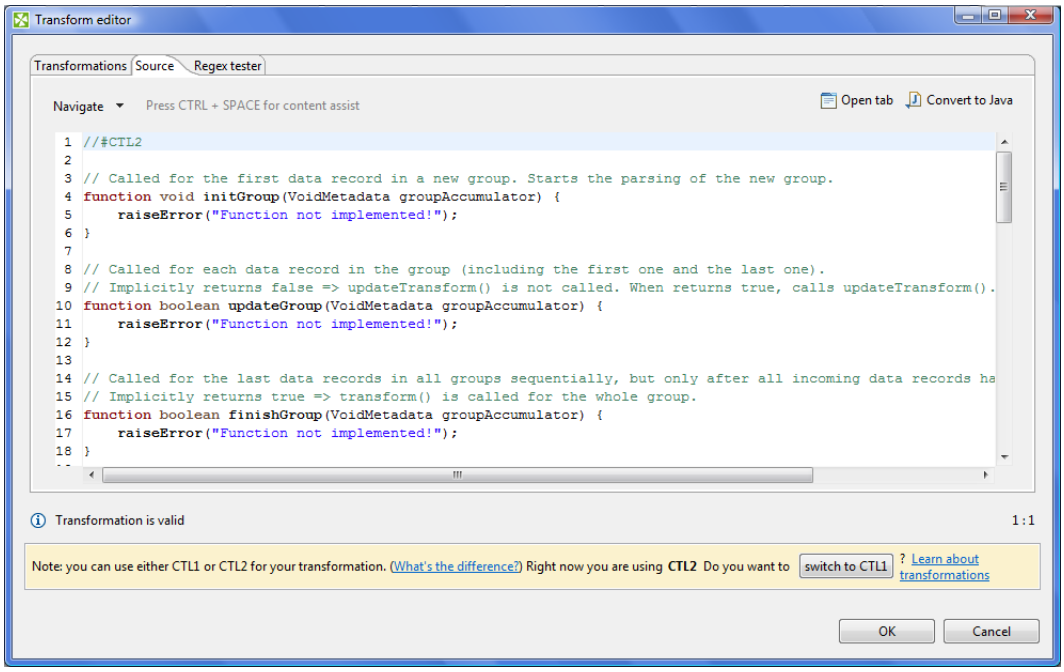


Figure 46.11. Source Tab of the Transform Editor in the Rollup Component (I)

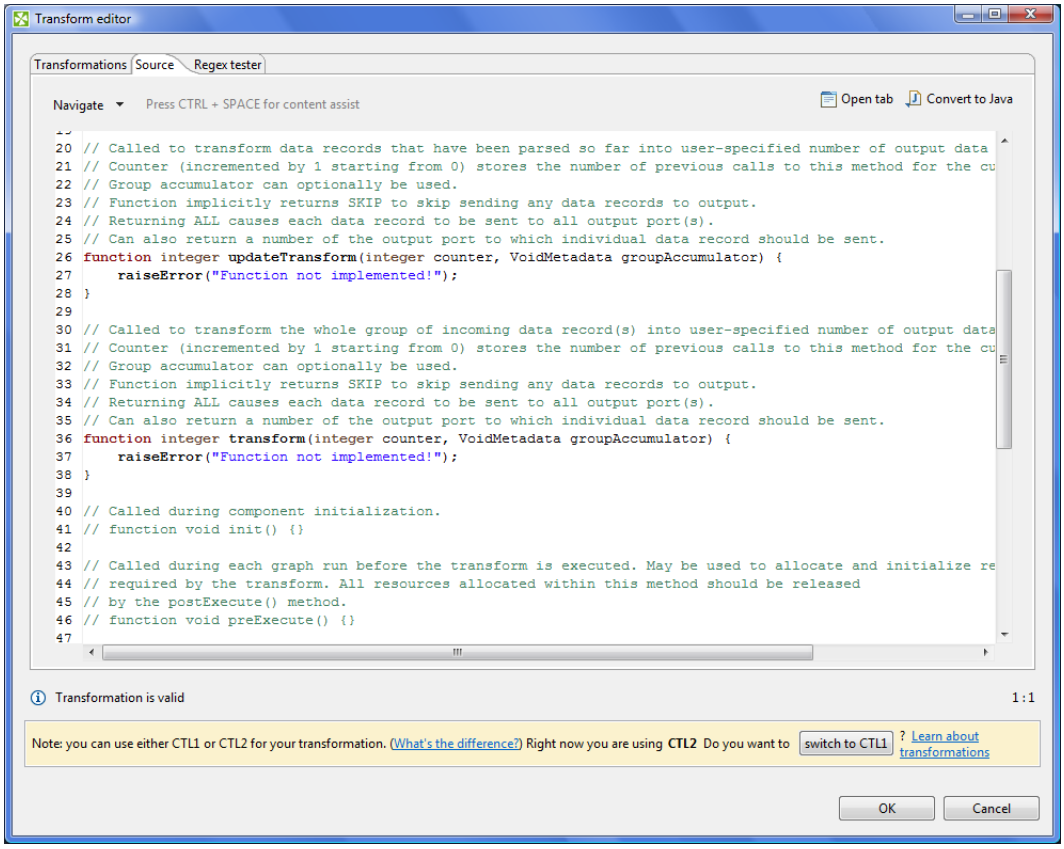


Figure 46.12. Source Tab of the Transform Editor in the Rollup Component (II)

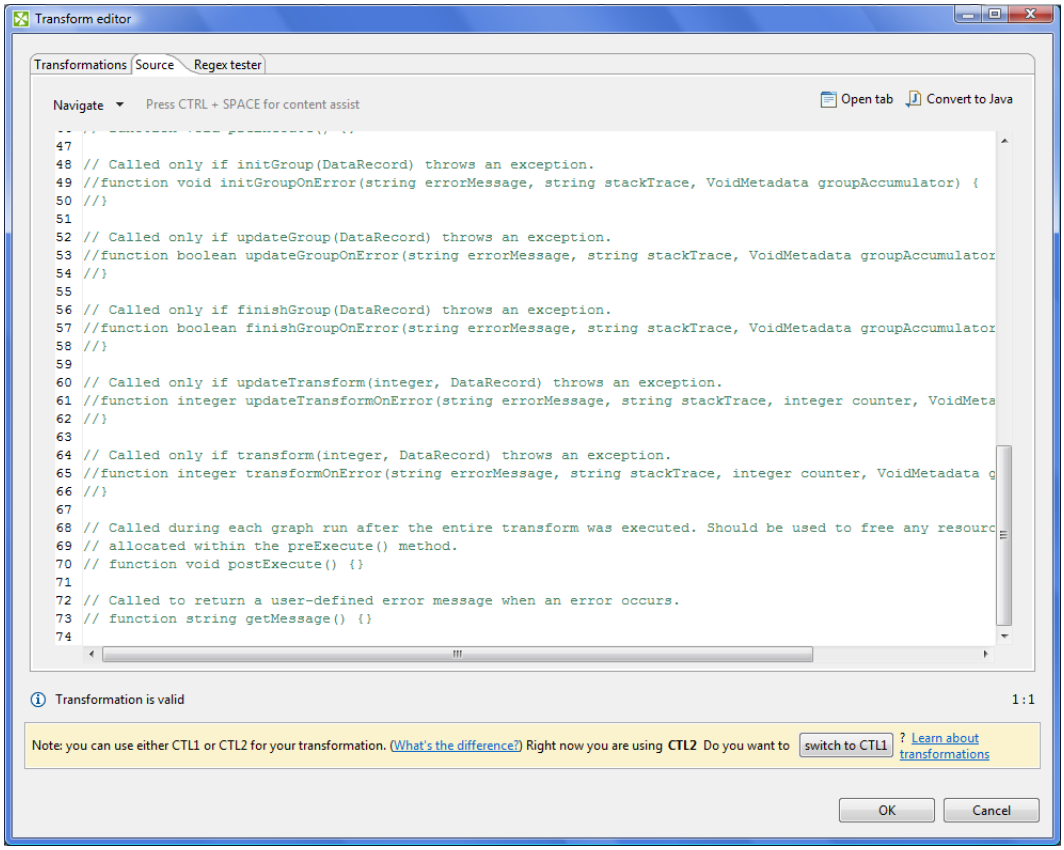


Figure 46.13. Source Tab of the Transform Editor in the Rollup Component (III)

Table 46.5. Functions in Rollup

CTL Template Functions	
void init()	
Required	No
Description	Initialize the component, setup the environment, global variables
Invocation	Called before processing the first record
Returns	`void`
void initGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	void
Invocation	Called repeatedly, once for the first input record of each group. Called before **updateGroup(groupAccumulator)**.
Description	Initializes information for specific group.

Example	`function void initGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count = 0;`
	`  groupAccumulator.totalFreight = 0;`
	`}`

CTL Template Functions	
boolean updateGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	false (**updateTransform(counter,groupAccumulator)** is not called) | true (**updateTransform(counter,groupAccumulator)** is called)
Invocation	Called repeatedly (once for each input record of the group, including the first and the last record) after the **initGroup(groupAccumulator)** function has already been called for the whole group.
Description	Updates information for specific group. If any of the input records causes fail of the `updateGroup()` function, and if user has defined another function (`updateGroupOnError()`), processing continues in this `updateGroupOnError()` at the place where `updateGroup()` failed. If `updateGroup()` fails and user has not defined any `updateGroupOnError()`, the whole graph will fail. The `updateGroup()` passes to `updateGroupOnError()` error message and stack trace as arguments.

Example	`function boolean updateGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.count++;`
	`  groupAccumulator.totalFreight =`
	`  groupAccumulator.totalFreight`
	`  + $0.Freight;`
	`  return true;`
	`}`

boolean finishGroup(<metadata name> groupAccumulator)	
Required	yes
Input Parameters	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	true (**transform(counter,groupAccumulator)** is called) | false (**transform(counter,groupAccumulator)** is not called)
Invocation	Called repeatedly, once for the last input record of each group. Called after **updateGroup(groupAccumulator)** has already been called for all input records of the group.
Description	Finalizes the group information. If any of the input records causes fail of the `finishGroup()` function, and if user has defined another function (`finishGroupOnError()`), processing continues in this `finishGroupOnError()` at the place where `finishGroup()` failed. If `finishGroup()` fails and user has not defined any `finishGroupOnError()`, the whole graph will fail. The `finishGroup()` passes to `finishGroupOnError()` error message and stack trace as arguments.

CTL Template Functions	

Example	`function boolean finishGroup(`
	`  companyCustomers groupAccumulator) {`
	`  groupAccumulator.avgFreight =`
	`  groupAccumulator.totalFreight`
	`  / groupAccumulator.count;`
	`  return true;`
	`}`

integer updateTransform(integer counter, <metadata name> groupAccumulator)	
Required	yes
Input Parameters	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly as specified by user. Called after **updateGroup(groupAccumulator)** returns `true`. The function is called until `SKIP` is returned.
Description	It creates output records based on individual record information. If any part of the `transform()` function for some output record causes fail of the `updateTransform()` function, and if user has defined another function (`updateTransformOnError()`), processing continues in this `updateTransformOnError()` at the place where `updateTransform()` failed. If `updateTransform()` fails and user has not defined any `updateTransformOnError()`, the whole graph will fail. The `updateTransformOnError()` function gets the information gathered by `updateTransform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `updateTransformOnError()`.

Example	`function integer updateTransform(`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= Length) {`
	`  clear(customers);`
	`  return SKIP;`
	`  }`
	`  $0.customers = customers[counter];`
	`  $0.EmployeeID = $0.EmployeeID;`
	`  return ALL;`
	`}`

integer transform(integer counter, <metadata name> groupAccumulator)	
Required	yes

CTL Template Functions	
Input Parameters	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called repeatedly as specified by user. Called after **finishGroup(groupAccumulator)** returns `true`. The function is called until `SKIP` is returned.
Description	It creates output records based on all of the records of the whole group. If any part of the `transform()` function for some output record causes fail of the `transform()` function, and if user has defined another function (`transformOnError()`), processing continues in this `transformOnError()` at the place where `transform()` failed. If `transform()` fails and user has not defined any `transformOnError()`, the whole graph will fail. The `transformOnError()` function gets the information gathered by `transform()` that was get from previously successfully processed code. Also error message and stack trace are passed to `transformOnError()`.

Example	`function integer transform(`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter > 0) return SKIP;`
	`  $0.ShipCountry = $0.ShipCountry;`
	`  $0.Count = groupAccumulator.count;`
	`  $0.AvgFreight =`
	`  groupAccumulator.avgFreight;`
	`  return ALL;`
	`}`

void initGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	void
Invocation	Called if `initGroup()` throws an exception.
Description	Initializes information for specific group.

Example	`function void initGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator)`
	`  printErr(errorMessage);`
	`}`

CTL Template Functions	
boolean updateGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	false (**updateTransform(counter,groupAccumulator)** is not called) | true (**updateTransform(counter,groupAccumulator)** is called)
Invocation	Called if `updateGroup()` throws an exception for a record of the group. Called repeatedly (once for each of the other input records of the group).
Description	Updates information for specific group.

Example	`function boolean updateGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator) {`
	`  printErr(errorMessage);`
	`  return true;`
	`}`

boolean finishGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	true (**transform(counter,groupAccumulator)** is called) | false (**transform(counter,groupAccumulator)** is not called)
Invocation	Called if `finishGroup()` throws an exception.
Description	Finalizes the group information.

Example	`function boolean finishGroupOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  companyCustomers groupAccumulator) {`
	`  printErr(errorMessage);`
	`  return true;`
	`}`

integer updateTransformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
Required	yes

CTL Template Functions	
Input Parameters	`string errorMessage`
	`string stackTrace`
	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `updateTransform()` throws an exception.
Description	It creates output records based on individual record information

Example	`function integer updateTransformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= 0) {`
	`  return SKIP;`
	`  }`
	`  printErr(errorMessage);`
	`  return ALL;`
	`}`

integer transformOnError(string errorMessage, string stackTrace, integer counter, <metadata name> groupAccumulator)	
Required	no
Input Parameters	`string errorMessage`
	`string stackTrace`
	`integer counter` (starts from 0, specifies the number of created records. should be terminated as shown in example below. Function calls end when `SKIP` is returned.)
	`<metadata name> groupAccumulator` (metadata specified by user) If `groupAccumulator` is not defined, `VoidMetadata Accumulator` is used in the function signature.
Returns	Integer numbers. See Return Values of Transformations (p. 238) for detailed information.
Invocation	Called if `transform()` throws an exception.
Description	It creates output records based on all of the records of the whole group.

CTL Template Functions	

Example	`function integer transformOnError(`
	`  string errorMessage,`
	`  string stackTrace,`
	`  integer counter,`
	`  companyCustomers groupAccumulator) {`
	`  if (counter >= 0) {`
	`  return SKIP;`
	`  }`
	`  printErr(errorMessage);`
	`  return ALL;`
	`}`

string getMessage()	
Required	No
Description	Prints error message specified and invocated by user
Invocation	Called in any time specified by user (called only when either `updateTransform()`, `transform()`, `updateTransformOnError()`, or `transformOnError()` returns value less than or equal to -2).
Returns	`string`
void preExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	May be used to allocate and initialize resources required by the transform. All resources allocated within this function should be released by the `postExecute()` function.
Invocation	Called during each graph run before the transform is executed.
void postExecute()	
Required	No
Input parameters	None
Returns	`void`
Description	Should be used to free any resources allocated within the `preExecute()` function.
Invocation	Called during each graph run after the entire transform was executed.

Important

- **Input records or fields**

  Input records or fields are accessible within the `initGroup()`, `updateGroup()`, `finishGroup()`, `initGroupOnError()`, `updateGroupOnError()`, and `finishGroupOnError()` functions.

  They are also accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.

- **Output records or fields**

Output records or fields are accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.

- **Group accumulator**

  Group accumulator is accessible within the `initGroup()`, `updateGroup()`, `finishGroup()`, `initGroupOnError()`, `updateGroupOnError()`, and `finishGroupOnError()` functions.

  It is also accessible within the `updateTransform()`, `transform()`, `updateTansformOnError()`, and `transformOnError()` functions.

- All of the other CTL template functions allow to access neither inputs nor outputs or `groupAccumulator`.

Warning

Remember that if you do not hold these rules, NPE will be thrown!

Java Interfaces for Rollup

The transformation implements methods of the `RecordRollup` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following is the list of the `RecordRollup` interface methods:

- `void init(Properties parameters, DataRecordMetadata inputMetadata, DataRecordMetadata accumulatorMetadata, DataRecordMetadata[] outputMetadata)`

  Initializes the rollup transform. This method is called only once at the beginning of the life-cycle of the rollup transform. Any internal allocation/initialization code should be placed here.

- `void initGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the first data record in a group. Any initialization of the group "accumulator" should be placed here.

- `void initGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the first data record in a group. Any initialization of the group "accumulator" should be placed here. Called only if `initGroup(DataRecord, DataRecord)` throws an exception.

- `boolean updateGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for each data record (including the first one as well as the last one) in a group in order to update the group "accumulator".

- `boolean updateGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for each data record (including the first one as well as the last one) in a group in order to update the group "accumulator". Called only if `updateGroup(DataRecord, DataRecord)` throws an exception.

- `boolean finishGroup(DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the last data record in a group in order to finish the group processing.

- `boolean finishGroupOnError(Exception exception, DataRecord inputRecord, DataRecord groupAccumulator)`

  This method is called for the last data record in a group in order to finish the group processing. Called only if `finishGroup(DataRecord, DataRecord)` throws an exception.

- `int updateTransform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). The output data record will be sent to the output when this method finishes. This method is called whenever the `boolean updateGroup(DataRecord, DataRecord)` method returns `true`. The `counter` argument is the number of previous calls to this method for the current group update. See Return Values of Transformations (p. 238) for detailed information about return values and their meaning.

- `int updateTransformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). Called only if `updateTransform(int, DataRecord, DataRecord)` throws an exception.

- `int transform(int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). The output data record will be sent to the output when this method finishes. This method is called whenever the `boolean finishGroup(DataRecord, DataRecord)` method returns `true`. The `counter` argument is the number of previous calls to this method for the current group. See Return Values of Transformations (p. 238) for detailed information about return values and their meaning.

- `int transformOnError(Exception exception, int counter, DataRecord inputRecord, DataRecord groupAccumulator, DataRecord[] outputRecords)`

  This method is used to generate output data records based on the input data record and the contents of the group "accumulator" (if it was requested). Called only if `transform(int, DataRecord, DataRecord)` throws an exception.

SimpleCopy

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

SimpleCopy copies data to all connected output ports.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
SimpleCopy	-	no	1	1-n	-	-

Abstract

SimpleCopy receives data records through single input port and copies each of them to all connected output ports.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For copied data records	Input 0[1)]
	1-n	no	For copied data records	Output 0[1)]

Legend:

1): Metadata on the output port(s) can be fixed-length or mixed even when those on the input are delimited, and vice versa. Metadata can be propagated through this component. All output metadata must be the same.

SimpleGather

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

SimpleGather gathers data records from multiple inputs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
SimpleGather	yes	no	1-n	1	-	-

Abstract

SimpleGather receives data records through one or more input ports. **SimpleGather** the gathers (demultiplexes) all the records as fast as possible and sends them all to the single output port. Metadata of all input and output ports must be the same.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
	1-n	no	For input data records	Input 0[1)]
Output	0	yes	For gathered data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

SortWithinGroups

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

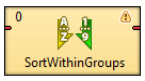
SortWithinGroups sorts input records within groups of records according to a sort key.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
SortWithinGroups	-	yes	1	1-n	-	-

Abstract

SortWithinGroups receives data records (that are grouped according to group key) through single input port, sorts them according to sort key separately within each group of adjacent records and copies each record to all connected output ports.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For sorted data records	Input 0[1)]
	1-n	no	For sorted data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.

SortWithinGroups Attributes

Attribute	Req	Description	Possible values
Basic			
Group key	yes	Key defining groups of records. Non-adjacent records with the same key value are considered to be of different groups and each of these different groups is processed separately and independently on the others. See Group Key (p. 231) for more information.	
Sort key	yes	Key according to which the records are sorted within each group of adjacent records. See Sort Key (p. 232) for more information.	
Advanced			
Buffer capacity		Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.	10485760 (default) | 1-N
Number of tapes		Number of temporary files used to perform external sorting. Even number higher than 2.	8 (default) | 2*(1-N)
Temp directories		List of names of temporary directories that are used to create temporary files to perform external sorting separated by semicolon.	java.io.tmpdir system property (default) | other directories

Advanced Description

Sorting Null Values

Remember that **SortWithinGroups** processes the records in which the same fields of the **Sort key** attribute have `null` values as if these `nulls` were equal.

XSLTransformer

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 39, Common Properties of Transformers (p. 273)

If you want to find the right **Transformer** for your purposes, see Transformers Comparison (p. 273).

Short Summary

XSLTransformer transforms input data records using an XSL transformation.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
XSLTransformer	-	no	1	1	-	-

Abstract

XSLTransformer receives data records through single input port, applies XSL transformation to specified fields and sends the records to single output port.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any1
Output	0	yes	For transformed data records	Any2

XSLTransformer Attributes

Attribute	Req	Description	Possible values
Basic			
XSLT file	1)	External file defining the XSL transformation.	
XSLT	1)	XSL transformation defined in the graph.	

Attribute	Req	Description	Possible values
Mapping	2)	Sequence of individual mappings for output fields separated from each other by semicolon. Each individual mapping has the following form: `$outputField:=transform($inputField)` (if `inputField` should be transformed according to the XSL transformation) or `$outputField:=$inputField` (if `inputField` should not be transformed).	
XML input file or field	2),3)	URL of file, dictionary or field serving as input.	
XML output file or field	2),3)	URL of file, dictionary or field serving as output.	
Advanced			
XSLT file charset		Encoding of external file defining the XSL transformation for all data types except `byte` or `cbyte`. Default encoding is UTF-8.	UTF-8 (default) | other encoding

Legend:

1): One of these attributes must be set. If both are set, **XSLT file** has higher priority.

2): One of these attributes must be set. If more are set, **Mapping** has the highest priority.

3): Either both or neither of them must be set. They are ignored if **Mapping** is defined.

Advanced Description

Mapping

Mapping can be defined using the following wizard.

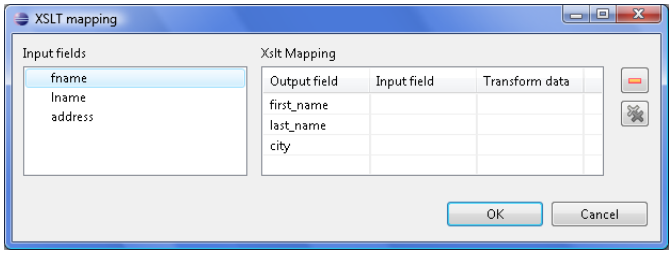


Figure 46.14. XSLT Mapping

Assign the input fields from the **Input fields** pane on the left to the output fields by dragging and dropping them in the **Input field** column of the right pane. Select which of them should be transformed by setting the **Transform data** option to true. By default, fields are not transformed.

The resulting **Mapping** can look like this:

```
$0.first_name:=transform($0.fname);$0.last_name:=$0.lname;$0.city:=$0.address;
```

Figure 46.15. An Example of Mapping

Remember that you must set either the **Mapping** attribute, or a pair of the following two attributes: **XML input file or field** and **XML output file or field**. These define input and output file, dictionary or field. If you set **Mapping**, these two other attributes are ignored even if they are set.

Chapter 47. Joiners

We assume that you already know what components are. See Chapter 17, Components (p. 84) for brief information.

Some components are intermediate nodes of the graph. These are called **Joiners** or **Transformers**.

For information about **Transformers** see Chapter 46, Transformers (p. 439). Here we will describe **Joiners**.

Joiners serve to join data from more data sources according to the key values.

Components can have different properties. But they also can have something in common. Some properties are common for all of them, others are common for most of the components, or they are common for **Joiners** only. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

We can distinguish **Joiners** according to how they process data. All **Joiners** work using key values.

- Some **Joiners** read data from two or more input ports and join them according to the equality of key values.
  - ExtHashJoin (p. 535) joins two or more data inputs according to the equality of key values.
  - ExtMergeJoin (p. 541) joins two or more sorted data inputs according to the equality of key values.
- Other **Joiners** read data from one input port and another data source and join them according to the equality of key values.
  - DBJoin (p. 532) joins one input data source and a database according to the equality of key values.
  - LookupJoin (p. 546) joins one input data source and a lookup table according to the equality of key values.
- One **Joiner** joins data according to the level of conformity of key values.
  - ApproximativeJoin (p. 524) joins two sorted inputs according to the level of conformity of key values.
- One **Joiner** joins data according to the user-defined relation of key values.
  - RelationalJoin (p. 549) joins two or more sorted data inputs according to the user-defined relation of key values (! =, >, >=, <. <=).

ApproximativeJoin

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

Short summary

ApproximativeJoin merges sorted data from two data sources on a common matching key. Afterwards, it distributes records to the output based on a user-specified **Conformity limit**.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
ApproximativeJoin	no	yes	1	2-4	yes	yes	yes

Abstract

ApproximativeJoin is a fuzzy joiner that is usually used in quite special situations. It requires the input be sorted and is very fast as it processes data in the memory. However, it should be avoided in case of large inputs as its memory requirements may be proportional to the size of the input.

The data attached to the first input port is called **master** as in the other Joiners. The second input port is called **slave**.

Unlike other joiners, this component uses two keys for joining. First of all, the records are matched in a standard way using **Matching Key**. Each pair of these matched records is then reviewed again and the conformity (similarity) of these two records is computed using **Join key** and a user-defined algorithm. The conformity level is then compared to **Conformity limit** and each record is sent either to the first (greater conformity) or to the second output port (smaller conformity). The rest of the records is sent to the third and fourth output port.

Icon

Ports

ApproximativeJoin receives data through two input ports, each of which may have a different metadata structure.

The conformity is then computed for matched data records. The records with greater conformity are sent to the first output port. Those with smaller conformity are sent to the second output port. The third output port can optionally be used to capture unmatched master records. The fourth output port can optionally be used to capture unmatched slave records.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1	yes	Slave input port	Any
Output	0	yes	Output port for the joined data with greater conformity	Any, optionally including additional fields: `_total_conformity_` and `_keyName_conformity_`. See Additional fields (p. 530).
	1	yes	Output port for the joined data with smaller conformity	Any, optionally including additional fields: `_total_conformity_` and `_keyName_conformity_`. See Additional fields (p. 530).
	2	no	Optional output port for master data records without slave matches	Input 0
	3	no	Optional output port for slave data records without master matches	Input 1

ApproximativeJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows with the same value of **Matching key** are compared and distributed between the first and the second output port. Depending on the specified **Conformity limit**. See Join key (p. 528).	
Matching key	yes	This key serves to match master and slave records.	
Transform	1)	Transformation in CTL or Java defined in the graph for records with greater conformity.	
Transform URL	1)	External file defining the transformation in CTL or Java for records with greater conformity.	
Transform class	1)	External transformation class for records with greater conformity.	
Transform for suspicious	2)	Transformation in CTL or Java defined in the graph for records with smaller conformity.	
Transform URL for suspicious	2)	External file defining the transformation in CTL or Java for records with smaller conformity.	
Transform class for suspicious	2)	External transformation class for records with smaller conformity.	

Attribute	Req	Description	Possible values
Conformity limit (0,1)		This attribute defines the limit of conformity for pairs of records. To the records with conformity higher than this value the transformation is applied, to those with conformity less than this value, the transformation for suspicious is applied.	0.75 (default) | between 0 and 1
Advanced			
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Deprecated			
Locale		Locale to be used when internationalization is used.	
Case sensitive		If set to `true`, upper and lower cases of characters are considered different. By default, they are processed as if they were equal to each other.	false (default) | true
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Slave override key		In older versions of **CloverETL**, slave part of **Join key**. **Join key** was defined as the sequence of individual expressions consisting of master field names each of them was followed by parentheses containing the 6 parameters mentioned below. These individual expressions were separated by semicolon. The **Slave override key** was a sequence of slave counterparts of the master **Join key** fields. Thus, in the case mentioned above, **Slave override key** would be `fname;lname`, whereas **Join key** would be `first_name(3 0.8 true false false false);last_name(4 0.2 true false false false)`.	
Slave override matching key		In older versions of **CloverETL**, slave part of **Matching key**. **Matching key** was defined as a master field name. Slave override matching key was its slave counterpart. Thus, in the case mentioned above (`$masterField=$slaveField`), **Slave override matching key** would be this `slaveField` only. And **Matching key** would be this `masterField`.	

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

2) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 531) or Java Interfaces (p. 531) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Matching key**

  You can define the **Matching key** using the **Matching key** wizard. You only need to select the desired master (driver) field in the **Master key** pane on the left and drag and drop it to the **Master key** pane on the right in the **Master key** tab. (You can also use the provided buttons.)

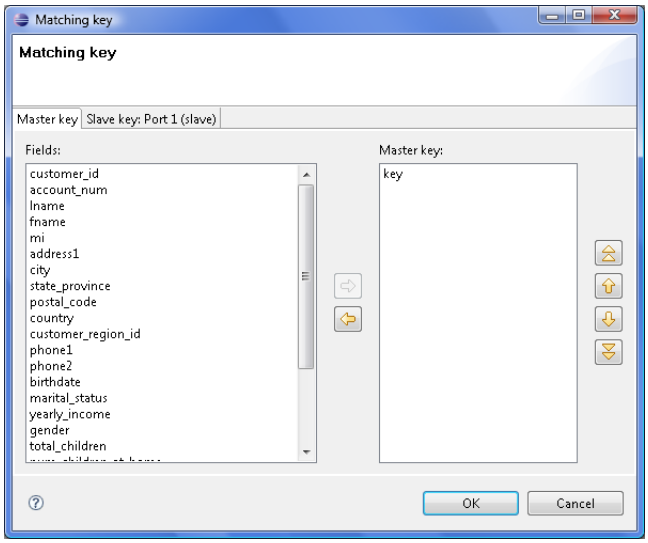


Figure 47.1. Matching Key Wizard (Master Key Tab)

In the **Slave key** tab, you must select one of the slave fields in the **Fields** pane on the left and drag and drop it to the **Slave key field** column at the right from the **Master key field** column (containing the master field the **Master key** tab) in the **Key mapping** pane.

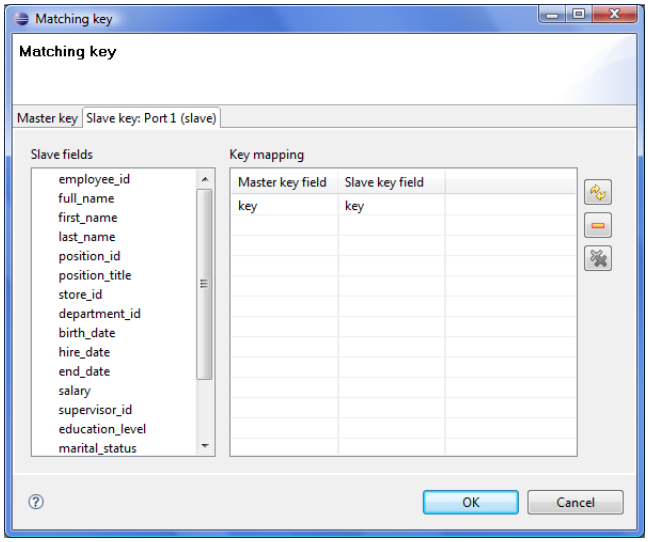


Figure 47.2. Matching Key Wizard (Slave Key Tab)

Example 47.1. Matching Key

Matching key looks like this:

```
$master_field=$slave_field
```

- **Conformity limit**

  You have to define the limit of conformity (**Conformity limit (0,1)**). The defined value distributes incoming records according to their conformity. The conformity can be greater or smaller than the specified limit. You have to define transformations for either group. The records with smaller conformity are marked "suspicious" and sent to port 1, while records with higher conformity go to port 0 ("good match").

  The conformity calculation is a challenge so let us try to explain at least in basic terms. First, groups of records are made based on **Matching key**. Afterwards, all records in a single group are compared to each other according to the **Join Key** specification. The strength of comparison selected in particular **Join key** fields determines what "penalty" characters get (for comparison strength, see Join key (p. 528)):

  - **Identical** - is a character-by-character comparison. The penalty is given for each different character (similar to `String.equals()`).

  - **Tertiary** - ingores differences in lower/upper case (similar to `String.equalsIgnoreCase()`), if it is the only comparison strength activated. If activated together with **Identical**, then a difference in diacritic (e.g. 'c' vs. ' č ') is a full penalty and a difference in case (e.g. 'a' vs. 'A') is half a penalty.

  - **Secondary** - a plain letter and its diacritic derivates for the same language are considered equal. The language used during comparison is taken from the metadata on the field. When no metadata is set on the field, it is treated as `en` and should work identically to **Primary** (i.e. derivatives are treated as equal).

    Example:

    language=`sk`: 'a', 'á', 'ä' are equal because they are all Slovak characters

    language= `sk`: 'a', ' ą ' are different because ' ą ' is a Polish (and not Slovak) character

  - **Primary** - all diacritic-derivates are considered equal regardless of language settings.

    Example:

    language=any: 'a', 'á', 'ä', ' ą ' are equal because they are all derivatives of 'a'

  As a final step, the total conformity is calculated as a weighted average of field conformities.

- **Join key**

  You can define the **Join key** with the help of the **Join key** wizard. When you open the **Join key** wizard, you can see two tabs: **Master key** tab and **Slave key** tab.

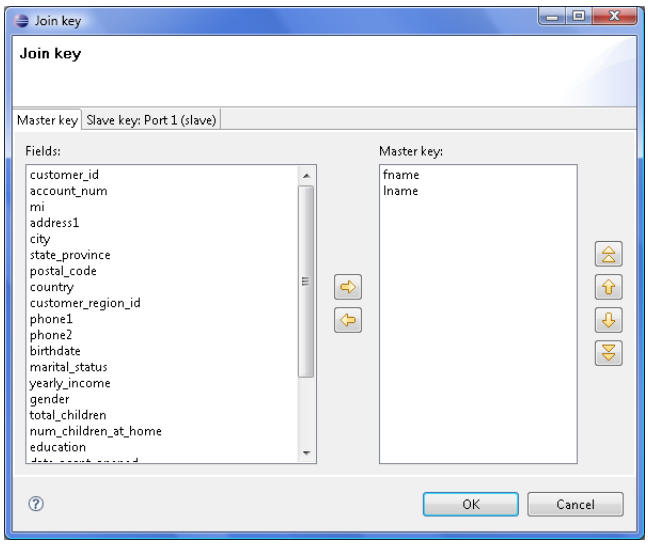


Figure 47.3. Join Key Wizard (Master Key Tab)

In the **Master key** tab, you must select the driver (master) fields in the **Fields** pane on the left and drag and drop them to the **Master key** pane on the right. (You can also use the buttons.)

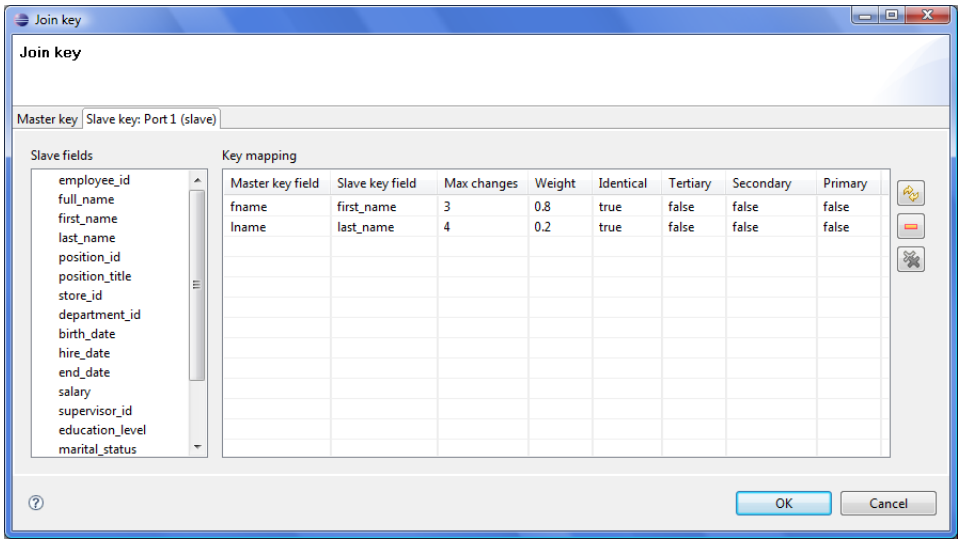


Figure 47.4. Join Key Wizard (Slave Key Tab)

In the **Slave key** tab, you can see the **Fields** pane (containing all slave fields) on the left and the **Key mapping** pane on the right.

You must select some of these slave fields and drag and drop them to the **Slave key field** column at the right from the **Master key field** column (containing the master fields selected in the **Master key** tab in the first step). In addition to these two columns, there are other six columns that should be defined: Maximum changes, Weight and the last four representing strength of comparison.

The maximum changes property contains the integer number that is equal to the the number of letters that should be changed so as to convert one data value to another value. The maximum changes property serves to compute the conformity. The conformity between two strings is 0, if more letters must be changed so as to convert one string to the other.

The weight property defines the weight of the field in computing the similarity. Weight of each field difference is computed as the quotient of the weight defined by user and the sum of the weights defined by user.

The strength of comparison can be identical, tertiary, secondary or primary.

- **identical**

  Only identical letters are considered equal.

- **tertiary**

  Upper and lower case letters are considered equal.

- **secondary**

  Diacritic letters and their Latin equivalents are considered equal.

- **primary**

  Letters with additional features such as a peduncle, pink, ring and their Latin equivalents are considered equal.

In the wizard, you can change any boolean value of these columns by simply clicking. This switches `true` to `false`, and vice versa. You can also change any numeric value by clicking and typing the desired value.

When you click **OK**, you will obtain a sequence of assignments of driver (master) fields and slave fields preceded by dollar sign and separated by semicolon. Each slave field is followed by parentheses containing six mentioned parameters separated by white spaces. The sequence will look like this:

```
$driver_field1=$slave_field1(parameters);...;$driver_fieldN=$slave_fieldN(parameters)
```

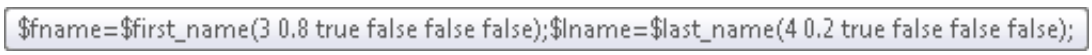


Figure 47.5. An Example of the Join Key Attribute in ApproximativeJoin Component

Example 47.2. Join Key for ApproximativeJoin

`$first_name=$fname(3 0.8 true false false false);$last_name=$lname(4 0.2 true false false false)`. In this **Join key**, `first_name` and `last_name` are fields from the first (master) data flow and `fname` and `lname` are fields from the second (slave) data flow.

- **Additional fields**

  Metadata on the first and second output ports can contain additional fields of numeric data type. Their names must be the following: `"_total_conformity_"` and some number of `"_keyName_conformity_"` fields. In the last field names, you must use the field names of the **Join key** attribute as the `keyName` in these additional field names. To these additional fields the values of computed conformity (total or that for `keyName`) will be written.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

DBJoin

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

Short Summary

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structures.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
DBJoin	no	no	1 (virtual)	1-2	yes	no	yes

Abstract

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structure. It is a general purpose joiner usable in most common situations. It does not require the input to be sorted and is very fast as data is processed in memory.

The data attached to the first input port is called the **master**, the second data source is called **slave**. Its data is considered as if it were incoming through the second (virtual) input port. Each master record is matched to the slave record on one or more fields known as a join key. The output is produced by applying a transformation that maps joined inputs to the output.

Icon

Ports

DBJoin receives data through a single input port and joins it with data from a database table. These two data sources can potentially have different metadata structure.

The joined data is then sent to the first output port. The second output port can optionally be used to capture unmatched master records.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1 (virtual)	yes	Slave input port	Any

Port type	Number	Required	Description	Metadata
Output	0	yes	Output port for the joined data	Any
	1	no	Optional output port for master data records without slave matches. (Only if the **Join type** attribute is set to `Inner join`.) This applies only to **LookupJoin** and **DBJoin**.	Input 0

DBJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 534).	
Left outer join		If set to `true`, also driver records without corresponding slave are parsed. Otherwise, `inner join` is performed.	false (default) | true
DB connection	yes	ID of the DB connection to be used as the resource of slave records.	
DB metadata		ID of DB metadata to be used. If not set, metadata is extracted from database using **SQL query**.	
Query URL	3)	Name of external file, including path, defining SQL query.	
SQL query	3)	SQL query defined in the graph.	
Transform	1), 2)	Transformation in CTL or Java defined in the graph.	
Transform URL	1), 2)	External file defining the transformation in CTL or Java.	
Transform class	1), 2)	External transformation class.	
Cache size		Maximum number of records with different key values that can be stored in memory.	100 (default)
Advanced			
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Legend:

1) One of these transformation attributes should be set. Any of them must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 534) or Java Interfaces (p. 534) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

2) The unique exception is the case when none of these three attributes is specified, but the **SQL query** attribute defines what records will be read from DB table. Values of **Join key** contained in the input records serve to select the records from db table. These are unloaded and sent unchanged to the output port without any transformation.

3) One of these attributes must be specified. If both are defined, **Query URL** has the highest priority.

Advanced Description

- **Join key**

  The **Join key** is a sequence of field names from master data source separated from each other by a semicolon, colon, or pipe. You can define the key in the **Edit key** wizard.

  Order of these field names must correspond to the order of the key fields from database table (and their data types). The slave part of **Join key** must be defined in the **SQL query** attribute.

  One of the query attributes must contain the expression of the following form: `... where field_K=? and field_L=?`.

  Example 47.3. Join Key for DBJoin

```
$first_name;$last_name
```

  This is the master part of fields that should serve to join master records with slave records.

  SQL query must contain the expression that can look like this:

```
... where fname=? and lname=?
```

  Corresponding fields will be compared and matching values will serve to join master and slave records.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

ExtHashJoin

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

Short Summary

General purpose joiner, merges potentionally unsorted data from two or more data sources on a common key.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
ExtHashJoin	no	no	1-n	1	no	no	yes

Abstract

This is a general purpose joiner used in most common situations. It does not require the input be sorted and is very fast as it is processed in memory.

The data attached to the first input port is called the **master** (as usual in other **Joiners**). All remaining connected input ports are called **slaves**. Each master record is matched to all slave records on one or more fields known as the **join key**. The output is produced by applying a transformation that maps joined inputs to the output. For details, see Joining Mechanics (p. 540).

This joiner should be avoided in case of large inputs on the slave port. The reason is slave data is cached in the memory.

Tip

If you have larger data, consider using the **ExtMergeJoin** component. If your data sources are unsorted, use a sorting component first (**ExtSort**, **FastSort**, or **SortWithinGroups**).

Icon

Ports

ExtHashJoin receives data through two or more input ports, each of which may have a different metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1	yes	Slave input port	Any
	2-n	no	Optional slave input ports	Any
Output	0	yes	Output port for the joined data	Any

ExtHashJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 538).	
Join type		Type of the join. See Join Types (p. 277).	Inner (default) | Left outer | Full outer
Transform	1)	Transformation in CTL or Java defined in the graph.	
Transform URL	1)	External file defining the transformation in CTL or Java.	
Transform class	1)	External transformation class.	
Allow slave duplicates		If set to `true`, records with duplicate key values are allowed. If it is `false`, only the **first** record is used for join.	false (default) | true
Advanced			
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Hash table size		Initial size of hash table that should be used when joining data flows. If there are more records that should be joined, hash table can be rehashed, however, it slows down the parsing process. See Hash Tables (p. 540) for more information:	512 (default)
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Attribute	Req	Description	Possible values
Left outer		If set to `true`, `left outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true
Full outer		If set to `true`, `full outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 540) or Java Interfaces (p. 540) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Join key**

  The **Join key** attribute is a sequence of mapping expressions for all slaves separated from each other by hash. Each of these mapping expressions is a sequence of field names from master and slave records (in this order) put together using equal sign and separated from each other by semicolon, colon, or pipe.

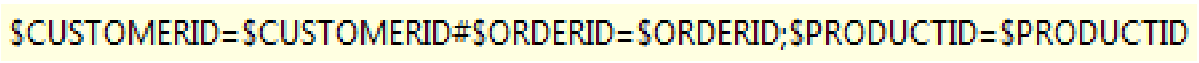


Figure 47.6. An Example of the Join Key Attribute in ExtHashJoin Component

Order of these mappings must correspond to the order of the slave input ports. If some of these mappings is empty or missing for some of the slave input ports, the mapping of the first slave input port is used instead.

Note

Different slaves can be joined with the master using different master fields!

Example 47.4. Slave Part of Join Key for ExtHashJoin

```
$master_field1=$slave_field1;$master_field2=$slave_field2;...;$master_fieldN=$slave_fieldN
```

- If some `$slave_fieldJ` is missing (in other words, if the subexpression looks like this: `$master_fieldJ=`), it is supposed to be the same as the `$master_fieldJ`.
- If some `$master_fieldK` is missing, `$master_fieldK` from the first port is used.

Example 47.5. Join Key for ExtHashJoin

```
$first_name=$fname;$last_name=$lname#=$lname;$salary=;$hire_date=$hdate
```

.

- Following is the part of **Join key** for the first slave data source (input port 1):

  `$first_name=$fname;$last_name=$lname`.

  - Thus, the following two fields from the master data flow are used for join with the first slave data source:

    `$first_name` and `$last_name`.

  - They are joined with the following two fields from this first slave data source:

    `$fname` and `$lname`, respectively.

- Following is the part of **Join key** for the second slave data source (input port 2):

  `=$lname;$salary=;$hire_date=$hdate`.

  - Thus, the following three fields from the master data flow are used for join with the second slave data source:

    `$last_name` (because it is the field which is joined with the `$lname` for the first slave data source), `$salary`, and `$hire_date`.

  - They are joined with the following three fields from this second slave data source:

    `$lname`, `$salary`, and `$hdate`, respectively. (This slave `$salary` field is expressed using the master field of the same name.)

To create the **Join key** attribute, you must use the **Hash Join key** wizard. When you click the **Join key** attribute row, a button appears in this row. By clicking this button you can open the mentioned wizard.

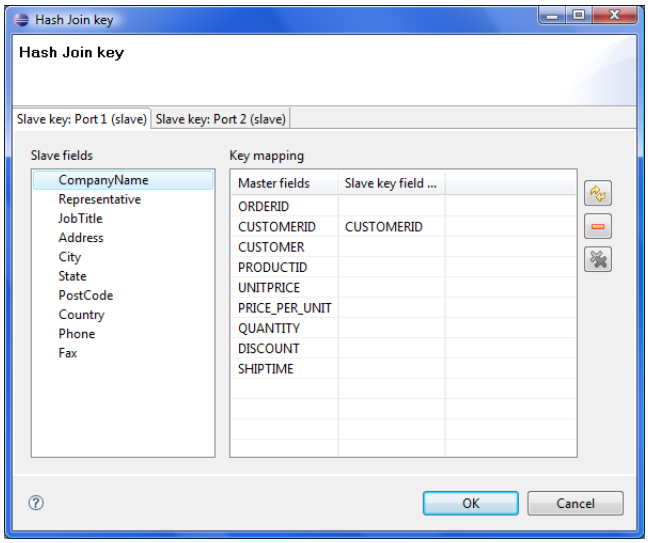


Figure 47.7. Hash Join Key Wizard

In it, you can see the tabs for all of the slave input ports. In each of the slave tab(s) there are two panes. The **Slave fields** pane on the left and the **Key mapping** pane on the right. In the left pane, you can see the list of all

the slave field names. In the right pane, you can see two columns: **Master fields** and **Slave key field mapped**. The left column contains all field names of the driver input port. If you want to map some slave fields to some driver (master) fields, you must select each of the desired slave fields that should be selected in the left pane by clicking its item, and drag and drop it to the **Slave key field mapped** column in the right pane at the row of some driver (master) field to which it should be mapped. It must be done for the selected slave fields. And the same process must be repeated for all slave tabs. Note that you can also use the **Auto mapping** button or other buttons in each tab. Thus, slave fields are mapped to driver (Master) fields according to their names. Note that different slaves can map different number of slave fields to different number of driver (Master) fields.

- **Hash Tables**

  The component first receives the records incoming through the slave input ports, reads them and creates hash tables from these records. These hash tables must be sufficiently small. After that, for each driver record incoming through the driver input port the component looks up the corresponding records in these hash tables. For every slave input port one hash table is created. The records on the input ports do not need to be sorted. If such record(s) are found, the tuple of the driver record and the slave record(s) from the hash tables are sent to transformation class. The transform method is called for each tuple of the master and its corresponding slave records.

  The incoming records do not need to be sorted, but the initialization of the hash tables is time consuming and it may be good to specify how many records can be stored in hash tables. If you decide to specify this attribute, it would be good to set it to the value slightly greater than needed. Nevertheless, for small sets of records it is not necessary to change the default value.

Joining Mechanics

All slave input data is stored in the memory. However, the master data is not. As for memory requirements, you therefore need to consider only the size of your slave data. In consequence, be sure to always set the larger data to the master and smaller inputs as slaves. **ExtHashJoin** uses in-memory hash tables for storing slave records.

Important

Remember each slave port can be joined with the master using different numbers of various master fields.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

ExtMergeJoin

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

Short Summary

General purpose joiner, merges sorted data from two or more data sources on a common key.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
ExtMergeJoin	no	yes	1-n	1	no	no	yes

Abstract

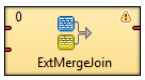
This is a general purpose joiner used in most common situations. It requires the input be sorted and is very fast as there is no caching (unlike **ExtHashJoin**).

The data attached to the first input port is called the **master** (as usual in other **Joiners**). All remaining connected input ports are called **slaves**. Each master record is matched to all slave records on one or more fields known as the **join key**. For a closer look on how data is merged, see Data Merging (p. 544).

Tip

If you want to join different slaves with the master on a key with various key fields, use **ExtHashJoin** instead. But remember slave data sources have to be sufficiently small.

Icon



Ports

ExtMergeJoin receives data through two or more input ports, each of which may have a distinct metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1	yes	Slave input port	Any
	2-n	no	Optional slave input ports	Any
Output	0	yes	Output port for the joined data	Any

ExtMergeJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 543).	
Join type		Type of the join. See Join Types (p. 277).	Inner (default) | Left outer | Full outer
Transform	1)	Transformation in CTL or Java defined in the graph.	
Transform URL	1)	External file defining the transformation in CTL or Java.	
Transform class	1)	External transformation class.	
Allow slave duplicates		If set to `true`, records with duplicate key values are allowed. If it is `false`, only the **first** record is used for join.	false (default) | true
Advanced			
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Ascending ordering of inputs		If set to `true`, incoming records are supposed to be sorted in ascending order. If it is set to `false`, they are descending.	true (default) | false
Deprecated			
Locale		Locale to be used when internationalization is used.	
Case sensitive		If set to `true`, upper and lower cases of characters are considered different. By default, they are processed as if they were equal to each other.	false (default) | true
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	
Left outer		If set to `true`, `left outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

Attribute	Req	Description	Possible values
Full outer		If set to `true`, `full outer` join is performed. By default it is `false`. However, this attribute has lower priority than **Join type**. If you set both, only **Join type** will be applied.	false (default) | true

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 545) or Java Interfaces (p. 545) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Join key**

  You must define the key that should be used to join the records (**Join key**). The records on the input ports must be sorted according to the corresponding parts of the **Join key** attribute. You can define the **Join key** in the **Join key** wizard.

  The **Join key** attribute is a sequence of individual key expressions for the master and all of the slaves separated from each other by hash. Order of these expressions must correspond to the order of the input ports starting with master and continuing with slaves. Driver (master) key is a sequence of driver (master) field names (each of them should be preceded by dollar sign) separated by colon, semicolon or pipe. Each slave key is a sequence of slave field names (each of them should be preceded by dollar sign) separated by colon, semicolon or pipe.

```
$EmployeeID;$CustomerID#$EmployeeID;$CustomerID#$ReportsTo;$CustomerID
```

Figure 47.8. An Example of the Join Key Attribute in ExtMergeJoin Component

  You can use this **Join key** wizard. When you click the **Join key** attribute row, a button appears there. By clicking this button you can open the mentioned wizard.

  In it, you can see the tab for the driver (**Master key** tab) and the tabs for all of the slave input ports (**Slave key** tabs).

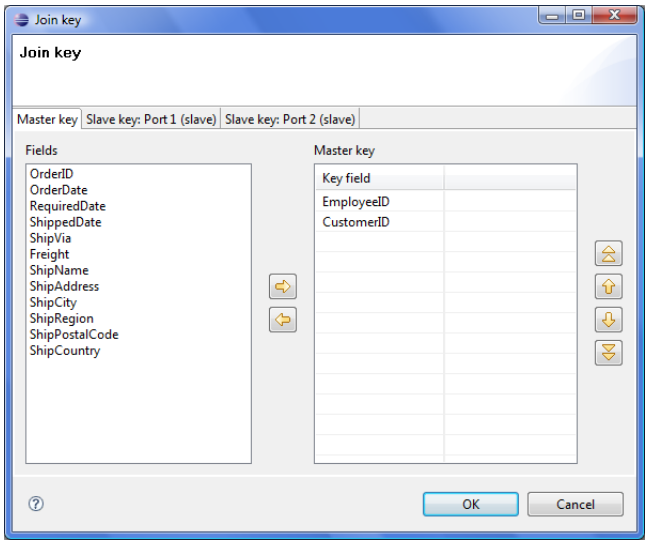


Figure 47.9. Join Key Wizard (Master Key Tab)

In the driver tab there are two panes. The **Fields** pane on the left and the **Master key** pane on the right. You need to select the driver expression by selecting the fields in the **Fields** pane on the left and moving them to the **Master key** pane on the right with the help of the **Right arrow** button. To the selected **Master key** fields, the same number of fields should be mapped within each slave. Thus, the number of key fields is the same for all input ports (both the master and each slave). In addition to it, driver (Master) key must be common for all slaves.

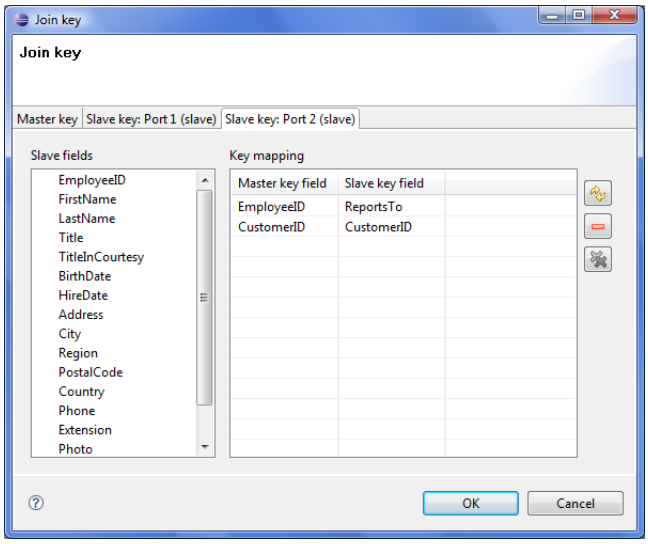


Figure 47.10. Join Key Wizard (Slave Key Tab)

In each of the slave tab(s) there are two panes. The **Fields** pane on the left and the **Key mapping** pane on the right. In the left pane you can see the list of the slave field names. In the right pane you can see two columns: **Master key field** and **Slave key field**. The left column contains the selected field names of the driver input port. If you want to map some driver field to some slave field, you must select the slave field in the left pane by clicking its item, and by pushing the left mouse button, dragging to the **Slave key field** column in the right pane and releasing the button you can transfer the slave field to this column. The same must be done for each slave. Note that you can also use the **Auto mapping** button or other buttons in each tab.

Example 47.6. Join Key for ExtMergeJoin

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

Following is the part of **Join key** for the master data source (input port 0):

`$first_name;$last_name`

- Thus, these fields are joined with the two fields from the first slave data source (input port 1):

  `$fname` and `$lname`, respectively.

- And, these fields are also joined with the two fields from the second slave data source (input port 2):

  `$f_name` and `$l_name`, respectively.

Data Merging

Joining data in **ExtMergeJoin** works the following way. First of all, let us stress again that data on both the master and the slave have to be sorted.

The component takes the first record from the master and compares it to the first one from the slave (with respect to **Join key**). There are three possible comparison results:

- master equals slave - records are joined
- "slave.key < master.key" - the component looks onto the next slave record, i.e. a one-step shift is performed trying to get a matching slave to the current master
- "slave.key > master.key" - the component looks onto the next master record, i.e. a regular one-step shift is performed on the master

Some input data contain sequences of same values. Then they are treated as one unit on the slave (a slave record knows the value of the following record), This happens only if **Allow slave duplicates** has been set to `true`. Moreover, the same-values unit gets stored in the memory. On the master, merging goes all the same by comparing one master record after another to the slave.

Note

In case there are is a large number of duplicate values on the slave, they are stored on your disk.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

LookupJoin

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

For information about lookup tables see Chapter 24, Lookup Tables (p. 166).

Short Summary

General purpose joiner, merges potentionally unsorted data from one data source incoming through the single input port with another data source from lookup table based on a common key.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
LookupJoin	no	no	1 (virtual)	1-2	yes	no	yes

Abstract

This is a general purpose joiner usable in most common situations. It does not require that the input to be sorted and is very fast as it is processed in memory.

The data attached to the first input port is called the **master**, the second data source is called **slave**. Its data is considered as if it were incoming through the second (virtual) input port. Each master record is matched to the slave record on one or more fields known as a join key. The output is produced by applying a transformation that maps joined inputs to the output.

All slave input data is stored in memory, however, the master data is not. Therefore you only need to consider the size of your slave data for memory requirements.

Icon

Ports

LookupJoin receives data through a single input port and joins it with data from lookup table. Either data source may potentially have different metadata structure.

The joined data is then sent to the first output port. The second output port can optionally be used to capture unmatched master records.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1 (virtual)	yes	Slave input port	Any
Output	0	yes	Output port for the joined data	Any
	1	no	Optional output port for master data records without slave matches. (Only if the **Join type** attribute is set to `Inner join`.) This applies only to **LookupJoin** and **DBJoin**.	Input 0

LookupJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 547).	
Left outer join		If set to `true`, also driver records without corresponding slave are parsed. Otherwise, `inner join` is performed.	false (default) | true
Lookup table	yes	ID of the lookup table to be used as the resource of slave records. Number of lookup key fields and their data types must be the same as those of **Join key**. These fields values are compared and matched records are joined.	
Transform	1)	Transformation in CTL or Java defined in the graph.	
Transform URL	1)	External file defining the transformation in CTL or Java.	
Transform class	1)	External transformation class.	
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Advanced			
Free lookup table after finishing		If set to `true`, lookup table is emptied after the parsing finishes.	false (default) | true
Deprecated			
Error actions		Definition of the action that should be performed when the specified transformation returns some **Error code**. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 548) or Java Interfaces (p. 548) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Join key**

You must define the key that should be used to join the records (**Join key**). It is a sequence of field names from the input metadata separated by semicolon, colon or pipe. You can define the key in the **Edit key** wizard.

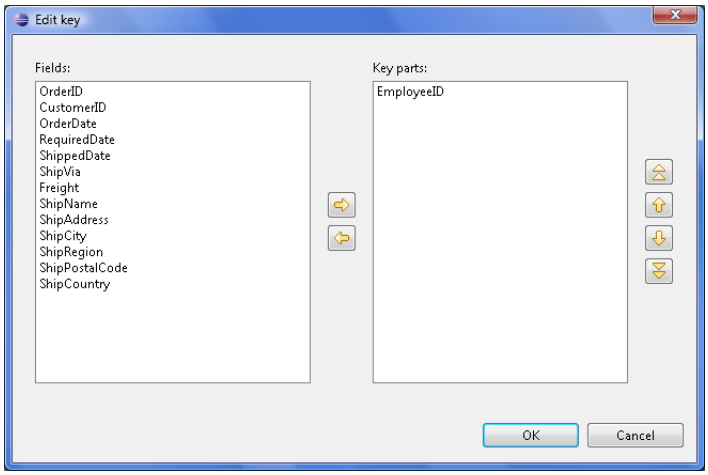


Figure 47.11. Edit Key Wizard

A counterpart of this **Join key** of the input metadata is the **key of lookup table** in lookup tables. It is specified in the lookup table itself.

Example 47.7. Join Key for LookupJoin

```
$first_name;$last_name
```

This is the master part of fields that should serve to join master records with slave records.

Lookup key should look like this:

```
$fname;$lname
```

Corresponding fields will be compared and matching values will serve to join master and slave records.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

RelationalJoin

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 40, Common Properties of Joiners (p. 276)

If you want to find the right **Joiner** for your purposes, see Joiners Comparison (p. 276).

Short Summary

Joiner that merges sorted data from two or more data sources on a common key whose values must differ in these data sources.

Component	Same input metadata	Sorted inputs	Slave inputs	Outputs	Output for drivers without slave	Output for slaves without driver	Joining based on equality
RelationalJoin	no	yes	1	1	no	no	no

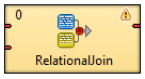
Abstract

This is a joiner usable in situation when data records with different field values should be joined. It requires the input to be sorted and is very fast as it is processed in memory.

The data attached to the first input port is called the **master** as it is also in the other **Joiners**. The other connected input port is called **slave**. Each master record is matched to all slave records on one or more fields known as a join key. The slave records whose values of this join key do not equal to their slave counterparts are joined together with such slaves. The output is produced by applying a transformation that maps joined inputs to the output.

All slave input data is stored in memory, however, the master data is not. Therefore you only need to consider the size of your slave data for memory requirements.

Icon



Ports

RelationalJoin receives data through two input ports, each of which may have a distinct metadata structure.

The joined data is then sent to the single output port.

Port type	Number	Required	Description	Metadata
Input	0	yes	Master input port	Any
	1	yes	Slave input port	Any

Port type	Number	Required	Description	Metadata
Output	0	yes	Output port for the joined data	Any

RelationalJoin Attributes

Attribute	Req	Description	Possible values
Basic			
Join key	yes	Key according to which the incoming data flows are joined. See Join key (p. 550).	
Join relation	yes	Defines the way of joining driver (master) and slave records. See Join relation (p. 552).	`master != slave` | `master(D) < slave(D)` | `master(D) <= slave(D)` | `master(A) > slave(A)` | `master(A) >= slave(A)`
Join type		Type of the join. See Join Types (p. 277).	Inner (default) | Left outer | Full outer
Transform	1)	Transformation in CTL or Java defined in the graph.	
Transform URL	1)	External file defining the transformation in CTL or Java.	
Transform class	1)	External transformation class.	
Transform source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes must use a common CTL template for **Joiners** or implement a `RecordTransform` interface.

See CTL Scripting Specifics (p. 552) or Java Interfaces (p. 552) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Advanced Description

- **Join key**

  You must define the key that should be used to join the records (**Join key**). The records on the input ports must be sorted according to the corresponding parts of the **Join key** attribute. You can define the **Join key** in the **Join key** wizard.

  The **Join key** attribute is a sequence of individual key expressions for the master and all of the slaves separated from each other by hash. Order of these expressions must correspond to the order of the input ports starting with master and continuing with slaves. Driver (master) key is a sequence of driver (master) field names (each of them should be preceded by dollar sign) seperated by a colon, semicolon or pipe. Each slave key is a sequence of slave field names (each of them should be preceded by dollar sign) separated by a colon, semicolon, or pipe.

  ```
  $EmployeeID;$CustomerID#$EmployeeID;$CustomerID#$ReportsTo;$CustomerID
  ```

  Figure 47.12. An Example of the Join Key Attribute in the RelationalJoin Component

  You can use this **Join key** wizard. When you click the **Join key** attribute row, a button appears there. By clicking this button you can open the mentioned wizard.

In it, you can see the tab for the driver (**Master key** tab) and the tabs for all of the slave input ports (**Slave key** tabs).

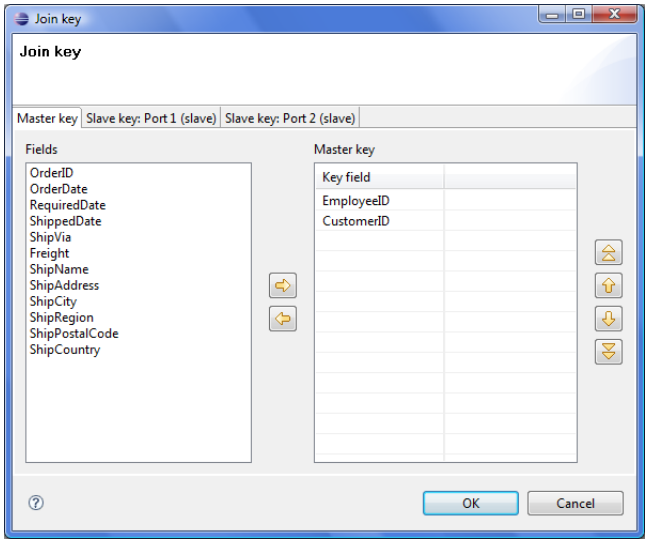


Figure 47.13. Join Key Wizard (Master Key Tab)

In the driver tab there are two panes. The **Fields** pane on the left and the **Master key** pane on the right. You need to select the driver expression by selecting the fields in the **Fields** pane on the left and moving them to the **Master key** pane on the right with the help of the **Right arrow** button. To the selected **Master key** fields, the same number of fields should be mapped within each slave. Thus, the number of key fields is the same for all input ports (both the master and each slave). In addition to it, driver (Master) key must be common for all slaves.

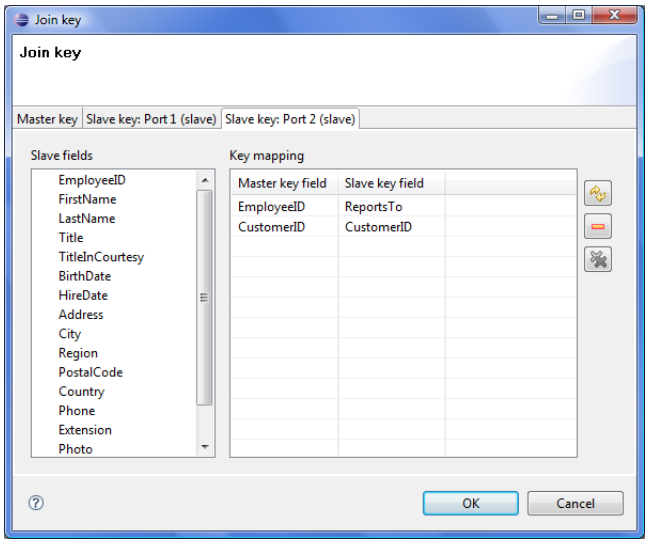


Figure 47.14. Join Key Wizard (Slave Key Tab)

In each of the slave tab(s) there are two panes. The **Fields** pane on the left and the **Key mapping** pane on the right. In the left pane you can see the list of the slave field names. In the right pane you can see two columns: **Master key field** and **Slave key field**. The left column contains the selected field names of the driver input port. If you want to map some driver field to some slave field, you must select the slave field in the left pane by clicking its item, and by pushing the left mouse button, dragging to the **Slave key field** column in the right pane and releasing the button you can transfer the slave field to this column. The same must be done for each slave. Note that you can also use the **Auto mapping** button or other buttons in each tab.

Example 47.8. Join Key for RelationalJoin

```
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

Following is the part of **Join key** for the master data source (input port 0):

`$first_name=$fname;$last_name=$lname`.

- Thus, these fields are joined with the two fields from the first slave data source (input port 1):

  `$fname` and `$lname`, respectively.

- And, these fields are also joined with the two fields from the second slave data source (input port 2):

  `$f_name` and `$l_name`, respectively.

- **Join relation**

  - If both input ports receive data records that are sorted in descending order, slave data records that are greater than or equal to the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is one of the following two: `master(D) < slave (D)` (slaves are greater than master) or `master(D) <= slave(D)` (slaves are greater than or equal to master).

  - If both input ports receive data records that are sorted in ascending order, slave data records that are less than or equal to the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is one of the following two: `master(A) > slave (A)` (slaves are less than driver) or `master(A) >= slave(A)` (slaves are less than or equal to driver).

  - If both input ports receive data records that are unsorted, slave data records that differ from the driver (master) data records are the only ones that are joined with driver data records and sent out through the output port. Corresponding **Join relation** is the following: `master != slave` (slaves are different from driver).

  - Any other combination of sorted order and **Join relation** causes the graph fail.

CTL Scripting Specifics

When you define your join attributes you must specify a transformation that maps fields from input data sources to the output. This can be done using the **Transformations** tab of the **Transform Editor**. However, you may find that you are unable to specify more advanced transformations using this easist approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom field mapping using the simple CTL scripting language.

All **Joiners** share the same transformation template which can be found in CTL Templates for Joiners (p. 278).

Java Interfaces

If you define your transformation in Java, it must implement the following interface that is common for all **Joiners**:

Java Interfaces for Joiners (p. 281)

Chapter 48. Cluster Components

We assume that you already know what components are. See Chapter 17, Components (p. 84) for brief information.

Two of the components serve for graphs that run on Cluster of **CloverETL Servers**. They distribute data records among different nodes or gathers them together. We call these two components **Cluster components**.

Different components can have different properties. But they also can have something in common. Some properties are common for all components, while others are common for most of the components, and also others are common only for some of them. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 41, Common Properties of Cluster Components (p. 283)

We can distinguish either **Cluster** component by what it does:

- ClusterPartitioner (p. 556) distributes data records among various **CloverETL Server** nodes of a single Cluster.
- ClusterGather (p. 554) gathers data records from various **CloverETL Server** nodes of a single Cluster.

ClusterGather

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 41, Common Properties of Cluster Components (p. 283)

Short Summary

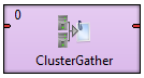
ClusterGather gathers data records from multiple cluster nodes.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
ClusterGather	yes	no	1-n (virtual)	1	-	-

Abstract

ClusterGather receives potentially unsorted data records through single input port. **ClusterGather** gathers records as specified in the **Merge type** attribute.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For gathered data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component.

ClusterGather Attributes

Attribute	Req	Description	Possible values
Basic			
Merging type		The way how records should be gathered. By round robin, sorted by key or gathered as sonn as possible.	Simple gather (default) | Round robin | Merge by key
Merge key	yes	Key according to which the sorted records are merged if **Merge by key** is selected. (Remember that all key fields must be sorted in ascending order!) See Group Key (p. 231) for more information.[1)]	
Equal NULL		By default, records with null values of key fields are considered to be different. If set to `true`, they are considered to be equal.	false (default) | true
Data policy		Determines what should be done when an error occurs. See Data Policy (p. 260) for more information.	Strict (default) | Controlled | Lenient

ClusterPartitioner

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)
- Chapter 41, Common Properties of Cluster Components (p. 283)

Short Summary

ClusterPartitioner distributes individual input data records among different Cluster nodes.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Java	CTL
ClusterPartitioner	-	no	1	1-n (virtual)	yes/no[1)]	yes/no[1)]

Legend

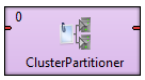
1) **ClusterPartitioner** can use either transformation or other two attributes (**Ranges** and/or **Partition key**). A transformation must be defined unless at least one of these is specified.

Abstract

ClusterPartitioner distributes individual input data records among different Cluster nodes.

To distribute data records, user-defined transformation, ranges of **Partition key** or RoundRobin algorithm may be used. Ranges of **Partition key** are either those specified in the **Ranges** attribute or calculated hash values. It uses a CTL template for **Partition** or implements a `PartitionFunction` interface. Its methods are listed below. In this component no mapping is defined since it does not change input data records. It only distributes them.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For output data records	Input 0[1)]

Legend:

1): Metadata can be propagated through this component.

Partition Attributes

Attribute	Req	Description	Possible values
Basic			
Partition	1)	Definition of the way how records should be distributed among Cluster nodes written in the graph in CTL or Java.	
Partition URL	1)	Name of external file, including path, containing the definition of the way how records should be distributed among Cluster nodes written in CTL or Java.	
Partition class	1)	Name of external class defining the way how records should be distributed among Cluster nodes.	
Ranges	1),2)	Ranges expressed as a sequence of individual ranges separated from each other by semicolon. Each individual range is a sequence of intervals for some set of fields that are adjacent to each other without any delimiter. It is expressed also whether the minimum and maximum margin is included to the interval or not by bracket and parenthesis, respectively. Example of **Ranges**: `<1,9)(,31.12.2008);<1,9)<31.12.2008,);<9,)(,31.12.2008); <9,)<31.12.2008)`.	
Partition key	1),2)	Key according to which input records are distributed among different Cluster nodes. Expressed as the sequence of individual input field names separated from each other by semicolon. Example of **Partition key**: `first_name;last_name`.	
Load balancer queue size		Size of the queue for load balancer.	
Advanced			
Partition source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default) | other encoding

Legend:

1): If one of these transformation attributes is specified, both **Ranges** and **Partition key** will be ignored since they have less priority. Any of these transformation attributes must use a CTL template for **Partition** or implement a `PartitionFunction` interface.

See CTL Scripting Specifics (p. 558) or Java Interfaces for ClusterPartitioner (p. 558) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

2): If no transformation attribute is defined, **Ranges** and **Partition key** are used in one of the following three ways:

- Both **Ranges** and **Partition key** are set.

  The records in which the values of the fields are inside the margins of specified range will be sent to the same Cluster node. The number of the node corresponds to the order of the range within all values of the fields.

- **Ranges** are not defined. Only **Partition key** is set.

  Records will be distributed among different Cluster nodes as described above. Hash values will be calculated and used as margins as if they were specified in **Ranges**.

- Neither **Ranges** nor **Partition key** are defined.

RoundRobin algorithm will be used to distribute records among Cluster nodes.

CTL Scripting Specifics

When you define any of the three transformation attributes, which is optional, you must specify a transformation that assigns a number of Cluster node to each input record.

For detailed information about Clover Transformation Language see Part VIII, CTL - CloverETL Transformation Language (p. 588). (CTL is a full-fledged, yet simple language that allows you to perform almost any imaginable transformation.)

CTL scripting allows you to specify custom transformation using the simple CTL scripting language.

CTL Templates for ClusterPartitioner

ClusterPartitioner uses the same transformation teplate as **Partition**. See CTL Templates for Partition (or ClusterPartitioner) (p. 493).

Once you have written your transformation in CTL, you can also convert it to Java language code by clicking corresponding button at the upper right corner of the tab.

Java Interfaces for ClusterPartitioner

ClusterPartitioner uses the same Java interface as **Partition**. See Java Interfaces for Partition (and ClusterPartitioner) (p. 497).

Chapter 49. Others

We assume that you already know what components are. See Chapter 17, Components (p. 84) for brief information.

Some of the components are slightly different from all those described above. We call this group of components: **Others**.

Others serve to perform multiple and heterogeneous tasks.

Components can have different properties. But they also can have something in common. Some properties are common for all of them, while others are common for most of the components. You should learn:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

As **Others** are heterogeneous group of components, they have no common properties.

We can distinguish each component of the **Others** group according to the task it performs.

- SystemExecute (p. 583) executes system commands.
- JavaExecute (p. 571) executes Java commands.
- DBExecute (p. 564) executes SQL/DML/DDL statements against database.
- RunGraph (p. 575) runs specified **CloverETL** graph(s).
- HTTPConnector (p. 568) sends HTTP requests and receives responses from web server.
- WebServiceClient (p. 586) calls a web-service and maps response to output ports.
- CheckForeignKey (p. 560) checks foreign key values and replaces those invalid by default values.
- SequenceChecker (p. 579) checks whether input data records are sorted.
- LookupTableReaderWriter (p. 573) reads data from a lookup table and/or write data to a lookup table.
- SpeedLimiter (p. 581) slows down data flowing throughout the component.

CheckForeignKey

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

CheckForeignKey checks the validity of foreign key values and replaces invalid values by valid ones.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
CheckForeignKey	-	no	2	1-2	-	no	no

1) Component sends each data record to all connected output ports.

Abstract

CheckForeignKey receives data records through two input ports. The data records on the first input port are compared with those one the second input port. If some value of the specified foreign key (input port 0) is not found within the values of the primary key (input port1), default value is given to the foreign key instead of its invalid value. Then all of the foreign records are sent to the first output port with the new (corrected) foreign key values and the original foreign records with invalid foreign key values can be sent to the optional second output port if it is connected.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For data with foreign key	Any1
	1	yes	For data with primary key	Any2
Output	0	yes	For data with corrected key	Input 0[1)]
	1	no	For data with invalid key	Input 0[1)]

Legend:

1): Metadata cannot be propagated through this component.

CheckForeignKey Attributes

Attribute	Req	Description	Possible values
Basic			
Foreign key	yes	Key that is compared according to which both incoming data flows are compared and data records are distributed among different output ports. See Foreign Key (p. 561) for more information.	
Default foreign key	yes	Sequence of values corresponding to the **Foreign key** data types separated from each other by semicolon. Serves to replace invalid foreign key values. See Foreign Key (p. 561) for more information.	
Equal NULL		By default, records with null values of fields are considered to be different. If set to `true`, nulls are considered to be equal.	false (default) | true
Advanced			
Hash table size		Table for storing key values. Should be higher than the number of records with unique key values.	512 (default) | properties
Deprecated			
Primary key		Sequence of field names from the second input port separated from each other by semicolon. See Deprecated: Primary Key (p. 563) for more information.	

Advanced Description

- **Foreign Key**

  The **Foreign key** is a sequence of individual assignments separated from each other by semicolon. Each of these individual assignments looks like this: `$foreignField=$primaryKey`.

  To define **Foreign key**, you must select the desired fields in the **Foreign key** tab of the **Foreign key definition** wizard. Select the fields from the **Fields** pane on the left and move them to the **Foreign key** pane on the right.

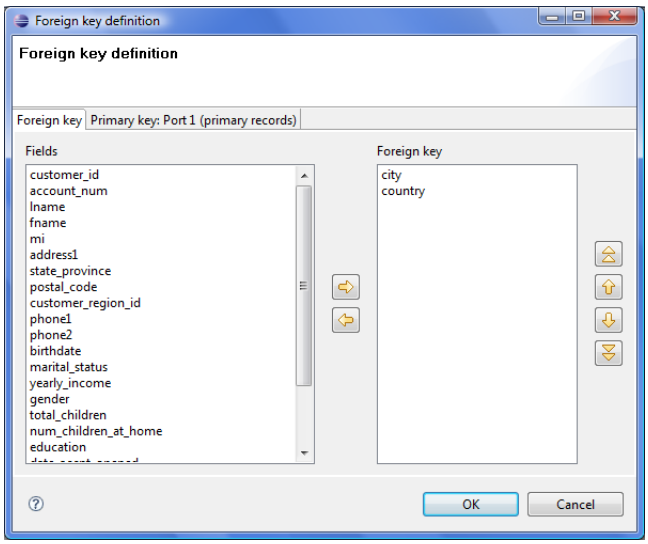


Figure 49.1. Foreign Key Definition Wizard (Foreign Key Tab)

When you switch to the **Primary key** tab, you will see that the selected foreign fields appeared in the **Foreign key** column of the **Foreign key definition** pane.

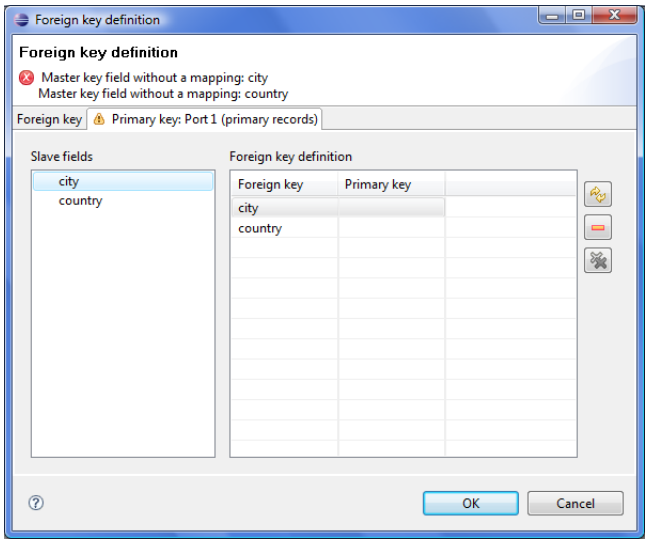


Figure 49.2. Foreign Key Definition Wizard (Primary Key Tab)

You only need to select some primary fields from the left pane and move them to the **Primary key** column of the **Foreign key definition** pane on the right.

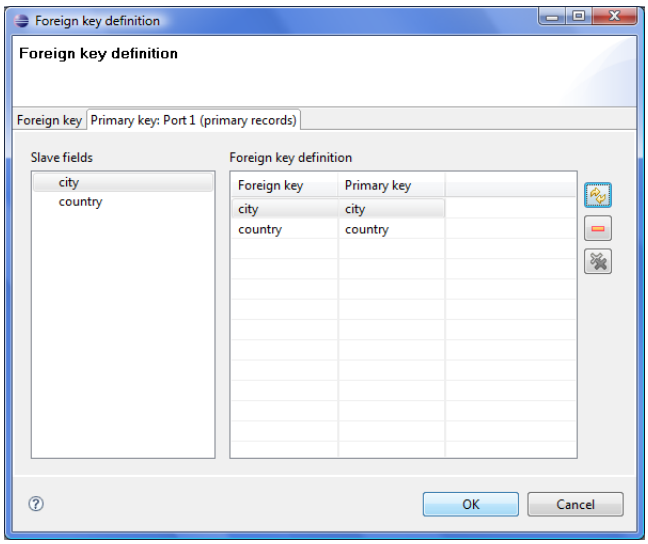


Figure 49.3. Foreign Key Definition Wizard (Foreign and Primary Keys Assigned)

You must also define the default foreign key values (**Default foreign key**). This key is also a sequence of values of corresponding data types separated from each other by semicolon. The number and data types must correspond to metadata of the foreign key.

If you want to define the default foreign key values, you need to click the **Default foreign key** attribute row and type the default values for all fields.

- **Deprecated: Primary Key**

  In older versions of Clover you had to specify both the primary and the foreign keys using the **Primary key** and the **Foreign key** attributes, respectively. They had the form of a sequence of field names separated from each other by semicolon. However, the use of **Primary key** is deprecated now.

DBExecute

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

DBExecute executes SQL/DML/DDL statements against a database.

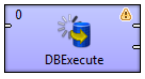
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs [1]	Java	CTL
DBExecute	-	✖	0-1	0-2	-	✖	✖

[1] Component sends each data record to all connected output ports.

Abstract

DBExecute executes specified SQL/DML/DDL statements against a database connected using the JDBC driver. It can execute queries, transactions, call stored procedures, or functions. Input parameters can be received through the single input port and output parameters or result set are sent to the first output port. Error information can be sent to the second output port.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	[1]	Input records for stored procedure or the whole SQL commands	any
Output	0	[2]	Output parameters of stored procedure or result set of the query	any
	1	✖	for error information	based on input metadata [3]

[1] Input port must be connected if **Query input parameters** attribute is specified or if the whole SQL query is received through input port.

[2] Output port must be connected if the **Query output parameters** or **Return set output fields** attribute is required.

[3] Metadata on output port 1 may contain any number of fields from input (same names and types) along with up to two additional fields for error information. Input metadata are mapped automatically according to their name(s) and type(s). The two error fields may have any names and must be set to the following Autofilling Functions (p. 117): `ErrCode` and `ErrText`

DBExecute Attributes

Attribute	Req	Description	Possible values
Basic			
DB connection	yes	ID of the DB connection to be used.	
Query URL	1)	One of these two options: Either the name of external file, including path, defining SQL query with the same characteristics as described in the **SQL query** attribute, or the **File URL** attribute string that is used for port reading. See SQL Query Received from Input Port (p. 566) for details.	
SQL query	1)	SQL query defined in the graph. Contains SQL/DML/DDL statement(s) that should be executed against database. If stored procedure or function with parameters should be called or if output data set should be produced, the form of the statement must be the folowing: *{[? =]call procedureName([?[,?,[...]])}*. (Do not forget enclose the statement in curly brackets!) At the same time, if the input and/or the output parameters are required, corresponding attributes are to be defined for them (**Query input parameters**, **Query output parameters** and/or **Result set output fields**, respectively). In addition, if the query consists of multiple statements, they must be separated from each other by specified **SQL statement delimiter**. Statements will be executed one by one.	
SQL statement delimiter		Delimiter between individual SQL statements in the **SQL query** or **Query URL** attribute. Default delimiter is semicolon.	";" (default) | other character
Print statements		By default, SQL commands are not printed. If set to `true`, they are sent to stdout.	false (default) | true
Transaction set		Specifies whether the statements should be executed in transaction. See Transaction Set (p. 566) for more information. Is applied only if database supports transactions.	SET (default) | ONE | ALL | NEVER_COMMIT
Advanced			
Query source charset		Encoding of external file specified in the **Query URL** attribute.	ISO-8859-1 (default) | <other encodings>
Call as stored procedure		By default, SQL commands are not executed as stored procedure calls unless this attribute is switched to `true`. If they are called as stored procedures, `JDBC CallableStatement` is used.	false (default) | true
Query input parameters		Used when stored procedure/function with input parameters is called. It is a sequence of the following type: `1:=$inputField1;...;n:=$inputFieldN`. Value of each specified input field is mapped to coresponding parameter (whose position in **SQL query** equals to the specified number). This attribute cannot be specified if SQL commands should be received through input port.	

Attribute	Req	Description	Possible values
Query output parameters		Used when stored procedure/function with output parameters or return value is called. It is a equence of the following type: `1:=$outputField1;...;n:=$outputFieldN`. Value of each output parameter (whose position in **SQL query** equals to the specified number) is mapped to the specified field. If the function returns a value, this value is represented by the first parameter.	
Result set output fields		If stored procedure or function returns a set of data, its output will be mapped to given output fields. Attribute is expressed as a sequence of output field names separated from each other by semicolon.	
Error actions		Definition of the action that should be performed when the specified query throws an SQL Exception. See Return Values of Transformations (p. 238).	
Error log		URL of the file to which error messages for specified **Error actions** should be written. If not set, they are written to **Console**.	

Legend:

1): One of these must be set. If both are specified, **Query URL** has higher priority.

Advanced Description

SQL Query Received from Input Port

SQL query can also be received from input port.

In this case, two values of the **Query URL** attribute are allowed:

- SQL command is sent through the input edge.

  The attribute value is: `port:$0.fieldName:discrete`.

  Metadata of this edge has neither default delimiter, nor record delimiter, but **EOF as delimiter** must be set to `true`.

- Name of the file containing the SQL command, including path, is sent through the input edge.

  The attribute value is: `port:$0.fieldName:source`.

For more details about reading data from input port see Input Port Reading (p. 257).

Transaction Set

Options are the following:

- **One statement**

  Commit is performed after each query execution.

- **One set of statements**

  All statements are executed for each input record. Commit is performed after a set of statements.

  For this reason, if an error occurs during the execution of any statement for any of the records, all statements are rolled back for such a record.

- **All statements**

  Commit is performed after all statements only.

  For this reason, if an error occurs, all operations are rolled back.

- **Never commit**

  Commit is not called at all.

  May be called from other component in different phase.

Tips & Tricks

- In general, you shouldn't use the **DBExecute** component for INSERT and SELECT statements. For uploading data to a database, please use the **DBOutputTable** component. And similarly for downloading use the **DBInputTable** component.

Specific Cases

- *Transfering data within a database:* The best practice to load data from one table to another in the same database is to do it inside the database. You can use the **DBExecute** component with a query like this

  ```
  insert into my_table select * from another_table
  ```

  because pulling data out from the database and putting them in is slower as the data has to be parsed during the reading and formatted when writing.

HTTPConnector

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

HTTPConnector sends HTTP requests to a web server and receives responses

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
HTTPConnector	-	no	0-1	0-1	-	no	no

1) Component sends each data record to all connected output ports.

Abstract

HTTPConnector sends HTTP requests to a web server and receives responses. Request is written in a file or in the graph itself or it is received through a single input port, . If request is defined in a file or in the graph, response is written to a response file (single HTTP interaction). If request is received through a port, response is also sent out through a single output port (mutli HTTP interaction) or it can also be written to temporary files and information about these files is sent to the specified output field.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	For URL, parameters for Query, or Request body	Any1[2)]
Output	0	1)	For response or for data records with URL of files containing response	Any2[3)]

Legend:

1): Either both or neither of them must be connected.

2): If connected, **Input field** need not be specified only if the first field (with **URL from field**) is of `string` data type.

3): If connected, **Output field** need not be specified only if the first field is of `string` data type.

HTTPConnector Attributes

Attribute	Req	Description	Possible values
Basic			
URL from input field	[1)]	Name of a `string` field specifying the target URL you wish to retrieve. Field value may contain placeholders in the form `*{<field name>}`. See the section called "Reading of Remote Files" (p. 253) for the URL format. The `HTTP`, `HTTPS`, `FTP` and `SFTP` protocols are supported.	
URL	[1)]	URL of the HTTP server the component connects to. May contain one or more placeholders in the following form: `*{<field name>}`. See the section called "Reading of Remote Files" (p. 253) for the URL format. The `HTTP`, `HTTPS`, `FTP` and `SFTP` protocols are supported.	
Request method		Method of request.	GET (default) | POST
Add input fields as parameters		Specifies whether aditional parameters should be added to the URL.	false (default) | true
Add input fields as parameters to		Specifies whether input fields should be added to the query string or method body. Parameters can only be added to the method body in case that **Request method** is set to POST.	QUERY (default) | BODY
Ignore fields		Specifies which input fields are not added as parameters. List of input fields separated by semicolon is expected.	
Additional HTTP header properties		Additional properties of the request that will be sent to the server. A dialog is used to create it, the final form is a sequence of `key=value` pairs separated by comma and the whole sequence is surrounded by curly braces.	
Charset		Character encoding of the input/output files	ISO-8859-1 (default) | other encoding
Input field	[2)]	Name of the field of the input metadata from which the request content is received. Must be of string data type. May be used for multi HTTP requests.	
Output field	[2)]	Name of the field of the output metadata to which the response response is sent. Must be of string data type. May be used for multi HTTP responses.	
Request content	[3)]	Request content defined directly in the graph. If defined, **Output file URL** must be specified.	
Input file URL	[3)]	URL of the file from which single HTTP request is read. See URL File Dialog (p. 59).	
Output file URL	[3)]	URL of the file to which single HTTP response is written. It is used if either **Input file URL** or **Request content** is defined. See URL File Dialog (p. 59). The output files are not deleted automatically and must be removed by hand or as a part of transformation.	
Append output	false	By default, any new response overwrites the older one. If you switch this attribute to `true`, new response is appended to the olders. Is applied to output files only.	
Advanced			

Attribute	Req	Description	Possible values
Authentication method			HTTP BASIC (default) | HTTP DIGEST | ANY
Username		Username required to connect to the server	
Password		Password required to connect to the server	
Store response file URL to output field		By default, when output port is connected, response is sent to it directly. If this attribute is switched to `true`, response is written to temporary files as specified in the **Directory for response files** and the **Prefix for response names** attributes. Only the information about these temporary files is sent to the specified **Output field** (URLs of the files is sent to the field). This is necessary in case the response body is too large to be stored in a single string data field. Developer is responsible to delete all these temp files.	false (default) | true
Directory for response files		Name of the directory to which all output files with HTTP responses will be written. Output files are not deleted automatically and need to be deleted by hand.	java.io.tmpdir system property (default) | other directory
Prefix for response names		Prefix that will be used in the name of each output file with HTTP response. To this prefix, distinguishing numbers are appended. Output files are not deleted automatically and need to be deleted by hand.	"http-response-" (default) | other prefix

[1)] One of these must be specified. In case that both are specified, **URL** is ignored.

[2)] If both ports are connected, the component makes *Multi HTTP requests*. One input record (request) corresponds to one output record (response). If multi HTTP request is not made, single HTTP request is made instead. If the first input field or the first output field are of `string` data type, corresponding attribute (**Input field** or **Output field**, resp.) does not need to be specified. Data is read and/or written to such fields automatically then.

[3)] Component makes *single HTTP request*. Only one of the **Request content** and **Input file URL** attributes may be specified at once. If single HTTP request is not made, multi HTTP request is made instead.

JavaExecute

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

JavaExecute executes Java commands.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
JavaExecute	-	-	0	0	-	yes	no

1) Component sends each data record to all connected output ports.

Abstract

JavaExecute executes Java commands. Runnable transformation, which is required in the component, implements a `JavaRunnable` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Below is the list of `JavaRunnable` interface methods. See Java Interfaces for JavaExecute (p. 572) for more information.

Icon

Ports

JavaExecute has neither input nor output port.

JavaExecute Attributes

Attribute	Req	Description	Possible values
Basic			
Runnable	1)	Runnable transformation in Java defined in the graph.	
Runnable URL	1)	External file defining the runnable transformation in Java.	
Runnable class	1)	External runnable transformation class.	

Attribute	Req	Description	Possible values
Runnable source charset		Encoding of external file defining the transformation.	ISO-8859-1 (default)
Advanced			
Properties		Properties to be used when executing Java command.	

Legend:

1) One of these must be set. These transformation attributes must be specified. Any of these transformation attributes implements a `JavaRunnable` interface.

See Java Interfaces for JavaExecute (p. 572) for more information.

See also Defining Transformations (p. 234) for detailed information about transformations.

Java Interfaces for JavaExecute

Runnable transformation, which is required in the component, implements a `JavaRunnable` interface and inherits other common methods from the `Transform` interface. See Common Java Interfaces (p. 249).

Following are the methods of the `JavaRunnable` interface:

- `boolean init(Properties parameters)`

  Initializes java class/function. This method is called only once at the beginning of transformation process. Any object allocation/initialization should happen here.

- `void free()`

  This is de-initialization method for this graph element. All resources allocated in the `init()` method should be released here. This method is invoked only once at the end of element existence.

- `void run()`

  The core method, which holds implementation of the Java code to be run by the **JavaExecute** component.

- `void setGraph(TransformationGraph graph)`

  Method which passes into transformation graph instance within which the transformation will be executed. Since `TransformationGraph` singleton pattern was removed, it is NO longer POSSIBLE to access graph's parameters and other elements (e.g. metadata definitions) through `TransformationGraph.getInstance()`.

LookupTableReaderWriter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

LookupTableReaderWriter reads data from lookup table and/or writes it to lookup table.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
LookupTableReaderWriter	-	no	0-1	0-n	yes	no	no

1) Component sends each data record to all connected output ports.

Abstract

LookupTableReaderWriter works in one of the three following ways:

- Receives data through connected single input port and writes it to the specified lookup table.
- Reads data from the specified lookup table and sends it out through all connected output ports.
- Receives data through connected single input port, updates the specified lookup table, reads updated lookup table and sends data out through all connected output ports.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	1)	For data records to be written to lookup table	Any
Output	0-n	1)	For data records to be read from lookup table	Input 0[1)]

Legend:

1): At least one of them must be connected. If input port is connected, component receives data through it and writes it to lookup table. If output port is connected, component reads data from lookup table and sends it out through this port.

LookupTableReaderWriter Attributes

Attribute	Req	Description	Possible values
Basic			
Lookup table	yes	ID of the lookup table to be used as a source of records when component is used as a reader and as a deposit when component is used as a writer or both when it is used as both a reader and a writer.	
Advanced			
Free lookup table after finishing		By default, contents of lookup table are not deleted after graph finishes. If set to `true`, lookup table is emptied after the processing ends.	false (default) | true

RunGraph

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

RunGraph runs **CloverETL** graphs.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
RunGraph	-	no	0-1	1-2	-	no	no

1) Component sends each data record to all connected output ports.

Abstract

RunGraph executes **CloverETL** graphs whose names can be specified in the component attribute or received through the input port.

Icon

Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For graph names and Clover command line arguments	Input Metadata for RunGraph (In-Out Mode) (p. 576)
Output	0	yes	For information about graph execution[1)]	Output Metadata for RunGraph (p. 576)
	1	2)	For information about unsuccessful graph execution	Output Metadata for RunGraph (p. 576)

Legend:

1): Information about successful execution of the specified graph is sent to the first output port if graph is specified in the component itself, or information about both success and fail is sent to it if the graph(s) is(are) specified on the input port.

2): If the name of a single graph that should be executed is specified in the component itself, the second output port must be connected. Data record is sent to it only if the specified graph fails. If the name(s) of one or more graphs that should be executed are received through input port, second output port does not need to be connected. Information about both success or fail is sent to the first output port only.

Table 49.1. Input Metadata for RunGraph (In-Out Mode)

Field number	Field name	Data type	Description
0	<anyname1>	string	Name of the graph to be executed, including path
1	<anyname2>	string	Clover command line argument

Table 49.2. Output Metadata for RunGraph

Field number	Field name	Data type	Description
0	graph	string	Name of the graph to be executed, including path
1	result	string	Result of graph execution (`Finished OK`, `Aborted`, or `Error`)
2	description	string	Detailed description of graph fail
3	message	string	Value of `org.jetel.graph.Result`
4	duration	integer, long, or decimal	Duration of graph execution in milliseconds
5	runID	decimal	Identification of the execution of the graph which runs on **CloverETL Server**.

RunGraph Attributes

Attribute	Req	Description	Possible values
Basic			
Graph URL	1)	Name of one graph, including path, that should be executed by the component. In this case, both output ports must be connected and information about success or fail is sent to the first or the second output port, respectively. (Pipeline mode)	
The same JVM		By default, the same JVM instance is used to run the specified graphs. If switched to `false`, graph(s) run as external processes. When working in the server environment, this attribute always has to be `true`.	true (default) | false
Graph parameters to pass		Parameters that are used by executed graphs. List a sequence separated by semicolon. If **The same JVM** attribute is switched to `false`, this attribute is ignored. See Advanced Description (p. 577)for more information.	
Alternative JVM command	2)	Command line to execute external process. If you want to give more memory to individual graphs that should be run by this **RunGraph** component, type here `java -Xmx1g -cp` or equivalent according to the maximum memory needed by any of the specified graphs.	java -cp (default) | other java command
Advanced			

Attribute	Req	Description	Possible values
Log file URL		Name of the file, including path, containing the log of external processes. The logging will be performed to the specified file independently on the value of **The same JVM** attribute. If **The same JVM** is set `true` (the default setting), logging will also be performed to console. If it is switched to `false`, logging to console will not be performed and logging information will be written to the specified file. See URL File Dialog (p. 59).	
Append to log file	2)	By default, data in the specified log file is overwritten on each graph run.	false (default) | true
Graph execution class	2)	Full class name to execute graph(s).	org.jetel.main.runGraph (default) | other execution class
Command line arguments	2)	Arguments of Java command to be executed when running graph(s).	
Ignore graph fail		By default, if the execution of any of the specified graphs fails (their names are received by **RunGraph** through the input port), the graph with **RunGraph** (that executes them) fails too. If this attribute is set to `true`, fail of each executed graph is ignored. It is also ignored if the graph with **RunGraph** (that executes one other graph) is specified in the component itself as the success information is sent to the first output port and the fail information is sent to the second output port.	false (default) | true

Legend:

1): Must be specified if input port is not connected.

2): These attributes are applied only if **The same JVM** attribute is set to `false`.

Advanced Description

- **Pipeline mode**

  If the component works in pipeline mode (without input edge, with the **Graph URL** attribute specified), the **Command line arguments** attribute must at least specify **CloverETL** plugins in the following way: `-plugins <plugins of CloverETL>`

- **In-out mode**

  If the component works in in-out mode (with input port connected, with empty **Graph URL** attribute) plugins do not need to be specified in the **Command line arguments** attribute.

- **Processing of command line arguments**

  All command line arguments passed to the **RunGraph** component (either as the second field of an input record or as the `cloverCmdLineArgs` component property) are regarded as a space delimited list of arguments which can be quoted. Moreover, the quote character itself can be escaped by backslash.

Example 49.1. Working with Quoted Command Line Arguments

Let us have the the following list of arguments:

```
firstArgument "second argument with spaces" "third argument with spaces and \" a quote"
```

The resulting command line arguments which will be passed to the child JVM are:

1) `firstArgument`

2) `second argument with spaces`

3) `third argument with spaces and " a quote`

Notice in 2) the argument is actually unquoted. That grants an OS-independent approach and smooth run on all platforms.

SequenceChecker

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

SequenceChecker checks the sort order of input data records.

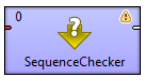
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
SequenceChecker	-	no	1	1-n	yes	no	no

1) Component sends each data record to all connected output ports.

Abstract

SequenceChecker receives data records through single input port, checks their sort order. If this does not correspond to the specified **Sort key**, graph fails. If the sort order corresponds to the specified, data records can optionally be sent to all connected output port(s).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0-n	no	For checked and copied data records	Input 0[1)]

Legend:

1): If data records are sorted properly, they can be sent to the connected output port(s). All metadata must be the same. Metadata can be propagated through this component.

SequenceChecker Attributes

Attribute	Req	Description	Possible values
Basic			
Sort key	yes	Key according to which the records should be sorted. If they are sorted in any other way, graph fails. See Sort Key (p. 232) for more information.	
Unique keys		By default, values of **Sort key** should be unique. If set to `false`, values of **Sort key** can be duplicated.	true (default) | false
Equal NULL		By default, records with null values of fields are considered to be equal. If set to `false`, nulls are considered to be different.	true (default) | false
Deprecated			
Sort order		Order of sorting (`Ascending` or `Descending`). Can be denoted by the first letter (`A` or `D`) only. The same for all key fields. Default sort order is ascending. If records are not sorted this way, graph fails.	Ascending (default) | Descending
Locale		Locale to be used when internationalization is set to `true`. By default, system value is used unless value of **Locale** specified in the `defaultProperties` file is uncommented and set to the desired **Locale**. For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).	system value or specified default value (default) | other locale
Use internationalization		By default, no internationalization is used. If set to `true`, sorting according national properties is performed.	false (default) | true

SpeedLimiter

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

SpeedLimiter slows down data records going through it.

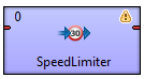
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
SpeedLimiter	-	no	1	1-n	yes	no	no

1) Component sends each data record to all connected output ports.

Abstract

SpeedLimiter receives data records through single input port, delays each input record by specified number of milliseconds and copies each input record of to all connected output ports. Total delay does not depend on the number of output ports, it only depends on the number of input records.

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	yes	For input data records	Any
Output	0	yes	For copied data records	Input 0[1)]
	1-n	no	For copied data records	Input 0[1)]

Legend:

1): Unlike in SimpleCopy (p. 517), metadata on the output must be the same as those on the input. All metadata must be the same. Metadata can be propagated through this component.

SpeedLimiter Attributes

Attribute	Req	Description	Possible values
Basic			
Delay	yes	Delay of processing each input record in milliseconds. Total delay of parsing is equal to the this value multiplicated by the number of input records.	0-N

SystemExecute

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

SystemExecute executes system commands.

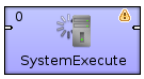
Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
SystemExecute	-	no	0-1	0-1	-	no	no

1) Component sends each data record to all connected output ports.

Abstract

SystemExecute executes commands and arguments specified in the component itself as a separate process. The commands receive standard input through the input port and send standard output to the output port (if the command creates any output).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For standard input of the specified system command (input of the process)	Any1
Output	0	1)	For standard output of the specified system command (output of the process)	Any2

Legend:

1): Standard output must be written to output port or output file. If both output port is connected and output file is specified, output is sent to output port only.

SystemExecute Attributes

Attribute	Req	Description	Possible values
Basic			
System command	yes	Command to be executed by the system. The command is always saved to a tmp file as a script. In case an interpreter is specified, it executes that script. If the command requires an input, it has to be sent to the command through the optional input port. See How it works (p. 585) for details.	
Process input/output charset		Encoding used for formating/parsing data for input/from output of system process.	ISO-8859-1 | <any other>
Output file URL	1)	Name of the file, including path, to which output of the process (together with errors) should be written if the output edge is not connected and if **System command** creates the standard output. See URL File Dialog (p. 59) for more information.	
Append		By default, the contents of output file is always deleted and overwritten by new data. If set to `true`, new output data is appended to the output file without deleting the file contents.	false (default) | true
Command interpreter		Interpreter that should execute the command. If specified, **System command** is saved as a script to a temporary file and executed by this interpreter. Its form must be the following: `<interpreter name> [parameters] ${} [parameters]`. If an interpreter is defined, **System command** is saved to a temporary file and executed as a script. In such a case, the component replaces this `${}` expression by the name of this temporary script file.	
Working directory		Working directory of the component.	current directory (default) | other directory
Advanced			
Number of error lines		Number of lines that are printed if command finishes with errors.	2 (default) | 1-N
Delete tmp batch file		By default, created temporary batch file is deleted after command execution. If set to `false`, it is not deleted.	true (default) | false
Environment		System-dependent mapping from variables to values. Mappings are separated by a colon, semicolon, or pipe. By default, the new value is appended to the environment of the current process. Both `PATH=/home/user/mydir` and `PATH=/home/user/mydir!true` means that `/home/user/mydir` will be appended to the existing `PATH`. Whereas, `PATH=/home/user/mydir!false` replaces the old `PATH` value by the new one (`/home/user/mydir`).	For example: PATH=/home/user/mydir[!true] (default) | PATH=/home/user/mydir!false
Timeout for producer/consumer workers (ms)		Timeout in milliseconds. See Timeout (p. 585) for details.	0 (without limitation) | 1-n

Legend:

1): If the output port is not connected, standard output can only be written to the specified output file. If the output port is connected, output file will not be created and standard output will be sent to the output port.

Advanced Description

How it works

SystemExecute runs the command specified in the **System command** and creates two threads.

- The first thread (producer) reads records from the input edge, serializes them and sends them to `stdin` of the command.
- The second thread (consumer) reads `stdout` of the command, parses it and sends it to the output edge.

Timeout

- When the command ends, component still waits until both the producer and the consumer also finish their work. The time is defined in the **Timeout** attribute.
- By default, timeout is unlimited now. In case of an unexpected deadlock, you can set the timeout to any number of milliseconds.

WebServiceClient

Commercial Component

We assume that you have already learned what is described in:

- Chapter 35, Common Properties of All Components (p. 223)
- Chapter 36, Common Properties of Most Components (p. 231)

If you want to find the right **Other** component for your purposes, see Others Comparison (p. 284).

Short Summary

WebServiceClient calls a web-service.

Component	Same input metadata	Sorted inputs	Inputs	Outputs	Each to all outputs[1)]	Java	CTL
WebServiceClient	-	no	0-1	0-n	no	no	no

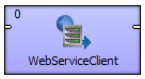
1) Component sends processed data records to the connected output ports as defined by mapping.

Abstract

WebServiceClient sends incoming data record to a web-service and passes the response to the output ports if they are connected. **WebServiceClient** supports `document/literal` styles only.

WebServiceClient supports only SOAP (version 1.1 and 1.2) messaging protocol with document style binding and literal use (document/literal binding).

Icon



Ports

Port type	Number	Required	Description	Metadata
Input	0	no	For request	Any1(In0)
Output	0-N	no[1)]	For response mapped to these ports	Any2(Out#)

Legend:

1): Response does not need to be sent to output if output ports are not connected.

WebServiceClient Attributes

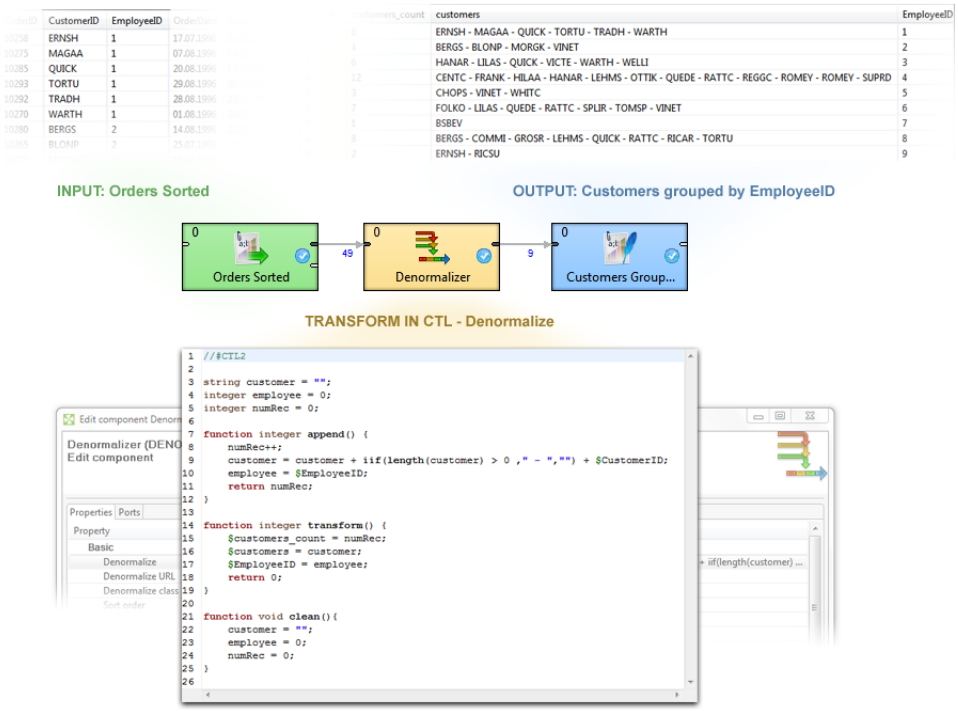
Attribute	Req	Description	Possible values
Basic			
WSD URL	yes	URL of the WSD server to which component will connect.	
Operation name	yes	Name of the operation to be performed.	
Request structure	yes	Structure of the request that is received from input port or written directly to the graph.	
Response mapping		Mapping of successful response to output ports. The same mapping as in **XMLExtract**. See XMLExtract Mapping Definition (p. 344) for more information.	
Fault mapping		Mapping of fault response to output ports. The same mapping as in **XMLExtract**. See XMLExtract Mapping Definition (p. 344) for more information.	
Advanced			
Username		Username to be used when connecting to the server.	
Password		Password to be used when connecting to the server.	
Auth Domain		Authentication domain.	
Auth Realm		Authentication name/realm	

Part VIII. CTL - CloverETL Transformation Language

Chapter 50. Overview

CTL is a proprietary scripting language oriented on data processing in transformation components of **CloverETL**.

It is designed to allow simple and clear notation of how data is processed and yet provide suficient means for its manipulation.



Language syntax resembles Java with some constructs common in scripting languages. Although scripting language itself, CTL code organization into function resembles structure of Java classes with clearly defined methods designating code entry points.

CTL is a high level language in both abstraction of processed data as well as its execution environment. The language shields programmer from the complexity of overall data transformation, while refocusing him to develop a single transformation step as a set of operations applicable onto all processed data records.

Closely integrated with **CloverETL** environment the language also benefits the programmer with uniform access to elements of data transformation located outside the executing component, operations with values of types permissible for record fields and a rich set of validation and manipulation functions.

During transformation execution each component running CTL code uses separate interpreter instance thus preventing possible collisions in heavily parallel multi-threaded execution environment of **CloverETL**.

Basic Features of CTL:

1. **Easy scripting language**

   Clover transformation language (CTL) is very simple scripting language that can serve for writing transformations in great number of **CloverETL** components.

   Although Java can be used in all of these components, working with CTL is much easier.

2. **Used in many CloverETL components**

   CTL can be used in all of the components whose overview is provided in Transformations Overview (p. 237) , except in **JMSReader**, **JMSWriter**, **JavaExecute**, and **MultiLevelReader**.

3. **Used even without knowledge of Java**

   Even without any knowledge of Java user can write the code in CTL.

4. **Almost as fast as Java**

   Transformations written in CTL are almost as fast as those written in Java.

   Source code of CTL2 can even be compiled into Java class.

Two Versions of CTL

Since version 3.0 of **CloverETL**, user can write transformation codes in either of the two CTL versions.

In the following chapters and sections we provide a thorough description of both versions of CTL and the list of their built-in functions.

CTL1 reference and built-in functions:

- Language Reference (p. 606)
- Functions Reference (p. 636)

CTL2 reference and built-in functions:

- Language Reference (p. 667)
- Functions Reference (p. 695)

Note

CTL2 version of Clover transformation language is recommended.

Chapter 51. CTL1 vs. CTL2 Comparison

CTL2 is a new version of **CloverETL** transformation language. It adds many improvements to the CTL concept.

Table 51.1. CTL Version Comparison

Feature	CTL1	CTL2
Strongly typed	✖	✔
Interpreted mode	✔	✔
Compiled mode	✖	✔
Speed	slower	faster

Typed Language

CTL has been redesigned to be strongly typed in CTL2. With variable declarations already containing type information the introduction of type checking mostly affects container types and functions. In CTL2, container types (lists and maps) must declare the element type and user-defined functions must declare return type as well as types of their arguments.

Naturally, strict typing for functions requires introduction of `void` type for functions not returning any value. typing also introduces function overloading in local code as well as in the built-in library.

Arbitrary Order of Code Parts

CTL2 allows to declare variables and functions in any place of the code. Only one condition must be fulfilled - each variable and function must be declared before it is used.

CTL2 also allows to define mapping in any place of the transformation and be followed by other code.

Parts of CTL2 code may be interspersed almost arbitrarily.

Compiled Mode

CTL2 code can be transformed into pure Java which greatly increases speed of the transformation. This is called "compiled mode" and **CloverETL** can do it for you transparently each time you run a graph with CTL2 code in it. The transformation into compiled form is done internally so you do not need to be Java programmer to use it.

Each time you use a component with CTL2 transform and explicitly set it to work in compiled mode, **CloverETL** produces an in-memory Java code from your CTL and runs the Java natively - giving you a great speed increase.

Access to Graph Elements (Lookups, Sequences, ...)

A strict type checking is further extended to validation of lookup tables and sequences access. For lookup tables the actual arguments of lookup operation are validated against lookup table keys, while using the record returned by table in further type checked.

Sequences support three possible return types explicitly set by user: `integer`, `long`, and `string`. In CTL1 records, lookup tables, and sequences were identified by their IDs - in CTL2 they are defined by names. For this reason, names of these graph elements must always be unique in a graph.

Metadata

In CTL2, any metadata is considered to be a data type. This changes the way records are declared in CTL transformation code, as sou can use your metadata names directly in your code to declare a variable:

```
Employee tmpEmployee;
recordName1 myRecord;
```

The following table presents an overview of differences between both versions of CTL.

Table 51.2. CTL Version Differences

CTL1	CTL2
Header (interpreted mode)	
`//#TL`	`//#CTL2`
`//#CTL1`	
Header (compiled mode)	
`unavailable`	`//#CTL2:COMPILED`
Declaration of primitive variables	
`int`	`integer`
`bytearray`	`byte`
Declaration of container variables	
`list myList;`	`<element type>[] myList;` Example: `integer[] myList;`
`map myMap;`	`map[<type of key>, <type of value>] myMap;` Example: `map[string,boolean] myMap;`
Declaration of records	
`record (<metadataID>) myRecord;`	`<metadataName> myRecord;`
Declaration of functions	
`function fName(arg1,arg2) {` `  <functionBody> }`	`function <data type> fName(<type1> arg1,<type2> arg2) {` `  <functionBody> }`
Mapping operator	
`$0.field1 := $0.field1;` (please note `':='` vs `'='`)	`$0.field1 = $0.field1;` (please note `':='` vs `'='`)
Accessing input records	
`@<port No>`	unavailable, may be replaced with: `$<port No>.*`
`@<metadata name>`	unavailable, may be replaced with: `$<metadata name>.*`
Accessing field values	
`@<port No>[<field No>]`	unavailable, may be replaced with: `$<port No>.<field name>`
`@<metadata name>[<field No>]`	unavailable, may be replaced with: `$<metadata name>.<field name>`
`<record variable name>["<field name>"]`	`<record variable name>.<field name>`
Conditional fail expression (interpreted mode only)	
`$0.field1 := expr1 : expr2 : ... : exprN;`	`$0.field1 = expr1 : expr2 : ... : exprN;`
unavailable	`myVar = expr1 : expr2 : ... : exprN;`
unavailable	`myFunction(expr1 : expr2 : ... : exprN)`
Dictionary declaration	

CTL1	CTL2
need not be defined	must always be defined
Dictionary entry types	
string, readable.channel, writable.channel	boolean, byte, date, decimal, integer, long, number, string, readable.channel, writable.channel, object
Writing to dictionary	
signature: `void write_dict(string name, string value)`	syntax: `dictionary.<entry name> = value;`
example 1: `write_dict("customer", "John Smith");`	example: `dictionary.customer = "John Smith";`

example 2:
`string customer;`
`write_dict(customer, "John Smith");`	

signature: `boolean dict_put_str(string name, string value);`	
example 3: `dict_put_str("customer", "John Smith");`	

example 4:
`string customer;`
`dict_put_str(customer, "John Smith");`	

Reading from dictionary	
signature: `string read_dict(string name)`	syntax: `value = dictionary.<entry name>;`

example 1:
`string myString;`
`myString = read_dict("customer");`
	example:
	`string myString;`
	`myString = dictionary.customer;`

example 2:
`string myString;`
`string customer;`
`myString = read_dict(customer);`	

signature: `string dict_get_str(string name)`	

example 3:
`string myString;`
`myString = dict_get_str("customer");`	

example 4:
`string myString;`
`string customer;`
`dict_get_str(customer);`	

Lookup table functions	
`lookup_admin(<lookup ID>,init)`[1)]	unavailable
`lookup(<lookup ID>,keyValue)`	`lookup(<lookup name>).get(keyValue)`
`lookup_next(<lookup ID>)`	`lookup(<lookup name>).next()`
`lookup_found(<lookup ID>)`	`lookup(<lookup name>).count(keyValue)`
`lookup_admin(<lookup ID>,free)`[1)]	unavailable
Sequence functions	
`sequence(<sequence ID>).current`	`sequence(<sequence name>).current()`

CTL1	CTL2
`sequence(<sequence ID>).next`	`sequence(<sequence name>).next()`
`sequence(<sequence ID>).reset`	`sequence(<sequence name>).reset()`
Switch statement	

`switch (Expr) {`
`  case (Expr1) : { StatementA StatementB }`
`  case (Expr2) : { StatementC StatementD }`
`  [default : { StatementE StatementF }]`
`}`
	`switch (Expr) {`
	`  case Const1 : StatementA StatementB break;`
	`  case Const2 : StatementC StatementD break;`
	`  [default : StatementE StatementF]`
	`}`

For loop	

`int myInt;`
`for(Initialization;Condition,Iteration)`
 	
(Initialization, Condition and Iteration are required)
	
	`for(integer myInt;Condition;Iteration)`
	
	(Initialization, Condition and Iteration are optional)

Foreach loop	

`int myInt;`
`list myList;`
`foreach(myInt : myList) Statement`
	
	`integer[] myList;`
	`foreach(integer myInt : myList) Statement`

Error handling	

`string MyException;`
`try Statement1 catch(MyException)`
`[Statement2]`
	unavailable

following set of optional functions can be used in both CTL1 and CTL2:	
`<required template function>OnError()` (e.g. `transformOnError()`, etc.)	
Jump statements	
`break`	`break;`
`continue`	`continue;`
`return Expression`	`return Expression;`
Contained-in operator	

`myVar .in. myContainer`	`in(myVar,myContainer)`
	or
	`myVar.in(myContainer)`

Eval functions	
`eval()`	unavailable
`eval_exp()`	unavailable
Ternary operator	

unavailable
 	
but
 	
`iif(Condition,ExprIfTrue,ExprIfFalse)`
 	
can be used instead
	`Condition ? ExprIfTrue : ExprIfFalse`
	
	but
	
	`iif(Condition,ExprIfTrue,ExprIfFalse)`
	
	also exists

Legend:

1) These functions do nothing since version 3.0 of **CloverETL** and can be removed from the code.

Chapter 52. Migrating CTL1 to CTL2

When you want to migrate any transformation code written in CTL1 to CTL2, you need to make the following steps:

Step 1: Replace the header

Replace the header which is at the beginning of your transformation code.

CTL1 used either `//#TL`, or `//#CTL1` whereas CTL2 uses `//#CTL2` for interpeted mode.

Remember that you can also choose compiled mode which is not available in CTL1. In such a case, header in CTL2 would be: `//#CTL2:COMPILED`

CTL1	CTL2
Interpreted mode	
`//#TL`	`//#CTL2`
`//#CTL1`	
Compiled mode	
unavailable	`//#CTL2:COMPILED`

Step 2: Change declarations of primitive variables (integer, byte, decimal data types)

Both versions of CTL have the same variables, however, key words differ for some of them.

- The `integer` data type is declared using the `int` word in CTL1, whereas it is declared as `integer` in CT2.
- The `byte` data type is declared using the `bytearray` word in CTL1, whereas it is declared as `byte` in CT2.
- The `decimal` data type may contain `Length` and `Scale` in its declaration in CTL1.

  For example, `decimal(15,6) myDecimal;` is valid declaration of a decimal in CTL1 with `Length` equal to 15 and `Scale` equal to 6.

  In CTL2, neither `Length` nor `Scale` may be used in a declaration of `decimal`.

  By default any decimal variable may use up to 32 digits plus decimal dot in its value.

  Only when such decimal is sent to an edge, in which `Length` and `Scale` are defined in metadata (by default they are 8 and 2), precission or length may change.

  Thus, equivalent declaration (in CTL2) would look like this:

  ```
  decimal myDecimal;
  ```

  Decimal field defines these **Length** and **Scale** in metadata. Or uses the default 8 and 2, respectively.

CTL1	CTL2
Integer data type	
`int myInt;`	`integer myInt;`
Byte data type	
`bytearray myByte;`	`byte myByte;`

CTL1	CTL2
Decimal data type	
`decimal(15,6) myDecimal;`	`decimal myDecimal;`
	If such variable should be assigned to a `decimal` field, the field should have defined **Length** and **Scale** to 15 and 6, respectively.

Step 3: Change declarations of structured variables: (list, map, record data types)

- Each `list` is a uniform structure consisting of elements of the same data type.

  The `list` is declared as folows in CTL1 (for example):

  ```
  list myListOfStrings;
  ```

  Equivalent `list` declaration would look like the following in CTL2:

  ```
  string[] myListOfStrings;
  ```

  Declaration of any `list` in CTL2 uses the following syntax:

  ```
  <data type of element>[] identifier
  ```

  Thus, replace the declaration of a `list` of CTL1 with another, valid in CTL2.

- Each `map` is a uniform structure consisting of key-value pairs. Key is always of `string` data type, whereas `value` is of any primitive data type (in CTL1).

  Map declaration may look like this:

  ```
  map myMap;
  ```

  Unlike in CTL1, in addition to `string`, key may be of any other primitive data type in CTL2.

  Thus, in CTL2 you need to specify both key type and value type like this:

  ```
  map[<key type>, <value type>] myMap;
  ```

  In order to rewrite your map declarations fom CTL1 syntax to that of CTL2, replace the older declaation of CTL1:

  ```
  map myMap;
  ```

  with the new of CTL2:

  ```
  map[string, <value type>] myMap;
  ```

  For example, `map[string, integer] myMap;`

- Each `record` is a heterogeneous structure consisting of specified number of fields. Each field can be of different primitive data type. Each field has its name and its number.

  In CTL1, each record may be declared in three different ways:

  Two of them use metadata ID, the third uses port number.

  Unlike in CTL1, where metadata are idetified with their ID, in CTL2 metadata are identified by their *unique* name.

See the table below on how records may be declared in CTL1 and in CTL2.

CTL1	CTL2
Declaration of a list	
`list myList;`	`<element type>[] myList;`
	e.g.: `string[] myList;`
Declaration of a map	
`map myMap;`	`map[<type of key>,<type of value>] myMap;`
	e.g.: `map[string,integer] myMap;`
Declaration of a record	
`record (<metadata ID>) myRecord;`	`<metadata name> myRecord;`
`record (@<port number>) myRecord;`	
`record (@<metadata name>) myRecord;`	

Step 4: Change declarations of functions

1. Add return data types to declarations of functions. (Remember that there is also `void` return type in CTL2.)
2. Add data types of their arguments.
3. Each function that returns any data type other than `void` must end with a `return` statement. Add corresponding `return` statement when necessary.

See following table:

CTL1	CTL2

`function transform(idx) {`
`  <other function body>`
`  $0.customer := $0.customer;`
`}`
	`function integer transform(integer idx) {`
	`  <other function body>`
	`  $0.customer = $0.customer;`
	`  return 0;`
	`}`

Step 5: Change record syntax

In CTL1, @ sign was used in assignments of input records and fields.

In CTL2, other syntax should be used. See the following table:

CTL1	CTL2
Whole record	
`<record variable name> = @<port number>;`	`<record variable name>.* = $<port number>.*;`
`<record variable name> = @<metadata name>`	`<record variable name>.* = $<metadata name>.*;`
Individual field	
`@<port number>[<field number>]`	`$<port number>.<corresponding field name>`
`@<metadata name>[<field number>]`	`$<metadata name>.<corresponding field name>`
`<record variable name>["<field name>"]`	`<record variable name>.<field name>`

Note

Note that you should also change the syntax of `groupAccumulator` usage in **Rollup**.

CTL1	CTL2
`groupAccumulator["<field name>"]`	`groupAccumulator.<field name>`

Step 6: Replace dictionary functions with dictionary syntax

In CTL1, a set of dictionary functions may be used.

In CTL2, dictionary syntax is defined and should be used.

CTL1	CTL2
Writing to dictionary	
`write_dict(string <entry name>, string <entry valu`	`dictionary.<entry name> = <entry value>;`
`dict_put_str(string <entry name>, string <entry value>);`	
Reading from dictionary	
`string myVariable;` `myVariable = read_dict(<entry name>);`	`string myVariable;` `myVariable = dictionary.<entry name>;`
`string myVariable;` `myVariable = dict_get_str(<entry name>);`	

Example 52.1. Example of dictionary usage

CTL1	CTL2
Writing to dictionary	
`write_dict("Mount_Everest", "highest");`	`dictionary.Mount_Everest = "highest";`
`dict_put_str("Mount_Everest", "highest");`	
Reading from dictionary	
`string myVariable;` `myVariable = read_dict("Mount_Everest");`	`string myVariable;` `myVariable = dictionary.Mount_Everest;`
`string myVariable;` `myVariable = dict_get_str("Mount_Everest");`	

Step 7: Add semicolons where necessary

In CT1, `jump`, `continue`, `break`, or `return` statement sometime do not allow terminal semicolon.

In CTL2, it is even required.

Thus, add semicolons to the end of any `jump`, `continue`, `break`, or `return` statement when necessary.

Step 8: Check, replace, or rename some built-in CTL functions

Some CTL1 functions are not available in CTL2. Please check Functions Reference (p. 695) for list of CTL2 functions.

Example:

CTL1 function	CTL2 function
`uppercase()`	`upperCase()`
`bit_invert()`	`bitNegate()`

Step 9: Change switch statements

Replace expressions in the `case` parts of `switch` statement with constants.

Note that CTL1 allows usage of *expressions* in the `case` parts of the `switch` statements, it requires curle braces after each `case` part. Values of one or more expression may even equal to each other, in such a case, all statements are executed.

CTL2 requires usage of *constants* in the `case` parts of the `switch` statements, it does not allow curle braces after each `case` part, and requires a `break` statement at the end of each `case` part. Without such `break` statement, all statements below would be executed. The constant specified in different `case` parts must be different.

CTL version	Switch statement syntax

CTL1	`//#CTL1`
	
	`int myInt;`
	`int myCase;`
	
	`myCase = 1;`
	
	`// Transforms input record into output record.`
	`function transform() {`
	
	`   myInt = random_int(0,1);`
	
	`   switch(myInt) {`
	`      case (myCase-1) : { print_err("two"); print_err("first case1"); }`
	`      case (myCase) : { print_err("three"); print_err("second case1"); }`
	`   }`
	
	`   $0.field1 := $0.field1;`
	
	`   return 0`
	`}`

CTL2	`//#CTL2`
	
	`integer myInt;`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	
	`   myInt = randomInteger(0,1);`
	
	`   switch(myInt) {`
	`      case 0 : printErr("zero"); printErr("first case"); break;`
	`      case 1 : printErr("one"); printErr("second case");`
	`   }`
	
	`   $0.field1 = $0.field1;`
	
	`   return 0;`
	`}`

Step 10: Change sequence and lookup table syntax

In CTL1, metadata, lookup tables, and sequences were identified with their IDs.

In CTL2 they are identified with their names.

Thus, make sure that all metadata, lookup tables, and sequences have unique names. Otherwise, rename them.

The two tables below show how you shout change the code containing lookup table or sequence syntax. Note that these are identified with either IDs (in CTL1) or with their names.

CTL version	Sequence syntax

CTL1	`//#CTL1`
	
	`// Transforms input record into output record.`
	`function transform() {`
	`  $0.field1 := $0.field1;`
	`  $0.field2 := $0.field2;`
	`  $0.field3 := sequence(Sequence0).current;`
	`  $0.field4 := sequence(Sequence0).next;`
	`  $0.field5 := sequence(Sequence0, string).current;`
	`  $0.field6 := sequence(Sequence0, string).next;`
	`  $0.field7 := sequence(Sequence0, long).current;`
	`  $0.field8 := sequence(Sequence0, long).next;`
	
	`  return 0`
	`}`

CTL2	`//#CTL2`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	`  $0.field1 = $0.field1;`
	`  $0.field2 = $0.field2;`
	`  $0.field3 = sequence(seqCTL2).current();`
	`  $0.field4 = sequence(seqCTL2).next();`
	`  $0.field5 = sequence(seqCTL2, string).current();`
	`  $0.field6 = sequence(seqCTL2, string).next();`
	`  $0.field7 = sequence(seqCTL2, long).current();`
	`  $0.field8 = sequence(seqCTL2, long).next();`
	
	`  return 0;`
	`}`

CTL version	Lookup table usage

CTL1	//#CTL1
	
	// variable for storing number of duplicates
	int count;
	int startCount;
	
	// values of fields of the first records
	string Field1FirstRecord;
	string Field2FirstRecord;
	
	// values of fields of next records
	string Field1NextRecord;
	string Field2NextRecord;
	
	// values of fields of the last records
	string Field1Value;
	string Field2Value;
	
	// record with the same metadata as those of lookup table
	record (Metadata0) myRecord;
	
	// Transforms input record into output record.
	function transform() {
	
	  // getting the first record whose key value equals to $0.Field2
	  // must be specified the value of both Field1 and Field2
	  Field1FirstRecord = lookup(LookupTable0,$0.Field2).Field1;
	  Field2FirstRecord = lookup(LookupTable0,$0.Field2).Field2;
	
	  // if lookup table contains duplicate records with the value specified above
	  // their number is returned by the following expression
	  // and assigned to the count variable
	  count = lookup_found(LookupTable0);
	
	  // it is copied to another variable
	  startCount = count;
	
	  // loop for searching the last record in lookup table
	  while ((count - 1) > 0) {
	
	  // searching the next record with the key specified above
	  Field1NextRecord = lookup_next(LookupTable0).Field1;
	  Field2NextRecord = lookup_next(LookupTable0).Field2;
	
	  // decrementing counter
	  count--;
	  }
	
	  // if record had duplicates, otherwise the first record
	  Field1Value = nvl(Field1NextRecord,Field1FirstRecord);
	  Field2Value = nvl(Field2NextRecord,Field2FirstRecord);
	
	  // mapping to the output
	  // last record from lookup table
	  $0.Field1 := Field1Value;
	  $0.Field2 := Field2Value;
	
	  // corresponding record from the edge
	  $0.Field3 := $0.Field1;
	  $0.Field4 := $0.Field2;
	
	  // count of duplicate records
	  $0.Field5 := startCount;
	
	  return 0
	}

CTL version	Lookup table usage

CTL2	`//#CTL2`
	
	`// record with the same metadata as those of lookup table`
	`recordName1 myRecord;`
	
	`// variable for storing number of duplicates`
	`integer count;`
	`integer startCount;`
	
	`// Transforms input record into output record.`
	`function integer transform() {`
	
	`   // if lookup table contains duplicate records,`
	`   // their number is returned by the following expression`
	`   // and assigned to the count variable`
	`   count = lookup(simpleLookup0).count($0.Field2);`
	
	`   // This is copied to startCount`
	`   startCount = count;`
	
	`   // getting the first record whose key value equals to $0.Field2`
	`   myRecord = lookup(simpleLookup0).get($0.Field2);`
	
	`   // loop for searching the last record in lookup table`
	`   while ((count-1) > 0) {`
	
	`      // searching the next record with the key specified above`
	`      myRecord = lookup(simpleLookup0).next();`
	
	`      // decrementing counter`
	`      count--;`
	`   }`
	
	`   // mapping to the output`
	`   // last record from lookup table`
	`   $0.Field1 = myRecord.Field1;`
	`   $0.Field2 = myRecord.Field2;`
	
	`   // corresponding record from the edge`
	`   $0.Field3 = $0.Field1;`
	`   $0.Field4 = $0.Field2;`
	
	`   // count of duplicate records`
	`   $0.Field5 = startCount;`
	
	`   return 0;`
	`}`

Warning

We suggest you better use other syntax for lookup tables.

The reason is that the following expression of CTL2:

```
lookup(Lookup0).count($0.Field2);
```

searches the records through the whole lookup table which may contain a great number of records.

The syntax shown above may be replaced with the following loop:

```
myRecord = lookup(<name of lookup table>).get(<key value>);
while(myRecord != null) {
  process(myRecord);
  myRecord = lookup(<name of lookup table>).next();
}
```

Especially DB lookup tables can return -1 instead of real count of records with specified key value (if you do not set **Max cached size** to a non-zero value).

The `lookup_found(<lookup table ID>)` function for CTL1 is not too recommended either.

Step 11: Change mappings in functions

Rewrite the mappings according to CTL2 syntax. Change mapping operators and remove expressions that use @ as shown above.

CTL version	Transformation with mapping

CTL1	`//#TL`
	
	`int retInt;`
	`function transform() {`
	
	`  if ($0.field3 < 5) retInt = 0; else retInt = 1;`
	
	`  // the following part is the mapping:`
	`  $0.* := $0.*;`
	`  $0.field1 := uppercase($0.field1);`
	`  $1.* := $0.*;`
	`  $1.field1 := uppercase($0.field1);`
	
	`  return retInt`
	`}`

CTL2	`//#CTL2`
	
	`integer retInt;`
	`function integer transform() {`
	
	`  // the following part is the mapping:`
	`  $0.* = $0.*;`
	`  $0.field1 = upperCase($0.field1);`
	`  $1.* = $0.*;`
	`  $1.field1 = upperCase($0.field1);`
	
	`  if ($0.field3 < 5) return = 0; else return 1;`
	`}`

Chapter 53. CTL1

This chapter describes the syntax and the use of CTL1. For detailed information on language reference or built-in functions see:

- Language Reference (p. 606)
- Functions Reference (p. 636)

Example 53.1. Example of CTL1 syntax (Rollup)

```
//#TL
list customers;
int Length;

function initGroup(groupAccumulator) {
}

function updateGroup(groupAccumulator) {
     customers = split($0.customers," - ");
     Length = length(customers);

     return true
}

function finishGroup(groupAccumulator) {
}

function updateTransform(counter, groupAccumulator) {
     if (counter >= Length) {
          remove_all(customers);

          return SKIP;
     }

     $0.customers := customers[counter];
     $0.EmployeeID := $0.EmployeeID;

     return ALL
}

function transform(counter, groupAccumulator) {
}
```

Language Reference

Clover transformation language (CTL) is used to define transformations in many transformation components. (in all **Joiners**, **DataGenerator**, **Partition**, **DataIntersection**, **Reformat**, **Denormalizer**, **Normalizer**, and **Rollup**)

Note

Since the version 2.8.0 of **CloverETL**, you can also use CTL expressions in parameters. Such CTL expressions can use any possibilities of CTL language. However, these CTL expressions must be surrounded by back quotes.

For example, if you define a parameter `TODAY="`today()`"` and use it in your CTL codes, such `${TODAY}` expression will be resolved to the date of this day.

If you want to display a back quote as is, you must use this back quote preceded by back slash as follows: `\``.

Important

CTL1 version is used in such expressions.

This section explains the following areas:

- Program Structure (p. 607)
- Comments (p. 607)
- Import (p. 607)
- Data Types in CTL (p. 608)
- Literals (p. 610)
- Variables (p. 612)
- Operators (p. 613)
- Simple Statement and Block of Statements (p. 618)
- Control Statements (p. 618)
- Functions (p. 623)
- Conditional Fail Expression (p. 625)
- Accessing Data Records and Fields (p. 626)
- Mapping (p. 629)
- Parameters (p. 635)

Program Structure

Each program written in CTL must have the following structure:

```
ImportStatements
VariableDeclarations
FunctionDeclarations
Statements
Mappings
```

Remember that the `ImportStatements` must be at the beginning of the program and the `Mappings` must be at its end. Both `ImportStatements` and `Mappings` may consist of more individual statements or mappings and each of them must be terminated by semicolon. The middle part of the program can be interspersed. Individual declaration of variables and functions and individual statements does not need to be in this order. But they always must use only declared variables and functions! Thus, first you need to declare variable and/or function before you can use it in some statement or another declaration of variable and function.

Comments

Throughout the program you can use comments. These comments are not processed, they only serve to describe what happens within the program.

The comments are of two types. They can be one-line comments or multiline comments. See the following two options:

```
// This is an one-line comment.
```

```
/* This is a multiline comment. */
```

Import

First of all, at the beginning of the program in CTL, you can import some of the existing programs in CTL. The way how you must do it is as follows:

```
import 'fileURL';
```

```
import "fileURL";
```

You must decide whether you want to use single or double quotes. Single quotes do not escape so called escape sequences. For more details see Literals (p. 610) below. For these `fileURL`, you must type the URL of some existing source code file.

But remember that you must import such files at the beginning before any other declaration(s) and/or statement(s).

Data Types in CTL

For basic information about data types used in metadata see Data Types and Record Types (p. 97)

In any program, you can use some variables. Data types in CTL can be the following:

boolean

Its declaration look like this: `boolean` *`identifier`*`;`

bytearray

This data type is an array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

Its declaration looks like this: `bytearray` *`[(size)] identifier`*`;`

date

Its declaration look like this: `date` *`identifier`*`;`

decimal

Its declaration looks like this: `decimal`*`[(length,scale)] identifier`*`;`

The default *`length`* and *`scale`* are `8` and `2`, respectively.

The default values of DECIMAL_LENGTH and DECIMAL_SCALE are contained in the `org.jetel.data.defaultProperties` file and can be changed to other values.

You can cast any float number to the decimal data type by apending the `d` letter to its end.

int

Its declaration looks like this: `int` *`identifier`*`;`

If you apend an `l` letter to the end of any integer number, you can cast it to the long data type

long

Its declaration looks like this: `long` *`identifier`*`;`

Any integer number can be cast to this data type by apending an `l` letter to its end.

number (double)

Its declaration looks like this: `number` *`identifier`*`;`

string

The declaration looks like this: `string` *`identifier`*`;`

list

Each `list` is a container of one the following primitive data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`.

The list data type is indexed by integers starting from 0.

Its declaration looks like this: `list identifier;`

The default list is an empty list.

Examples:

```
list list2; examplelist2[5]=123;
```

Assignments:

- `list1=list2;`

  It means that both lists reference the same elements.

- `list1[ ]=list2;`

  It adds all elements of `list2` to the end of `list1`.

- `list1[ ]="abc";`

  It adds the `"abc"` string to the `list1` as its new last element.

- `list1[ ]=NULL;`

  It removes the last element of the `list1`.

map

This data type is a container of any data type.

The map is indexed by strings.

Its declaration looks like this: `map identifier;`

The default map is an empty map.

Example: `map map1; map1["abc"]=true;`

The assignments are similar to those valid for a list.

record

This data type is a set of fields of data.

The structure of record is based on metadata.

Its declaration can look like one of these options:

1. `record (<metadata ID>) identifier;`
2. `record (@<port number>) identifier;`
3. `record (@<metadata name>) identifier;`

For more detailed information about possible expressions and records usage see Accessing Data Records and Fields (p. 626).

The variable does not have a default value.

It can be indexed by both integer numbers and strings. If indexed by numbers, fields are indexed starting from 0.

Literals

Literals serve to write values of any data type.

Table 53.1. Literals

Literal	Description	Declaration syntax	Example
integer	digits representing integer number	[0-9]+	95623
long integer	digits representing integer numbers with absolute value even greater than 2^{31}, but less than 2^{63}	[0-9]+L?	257L, or 9562307813123123
hexadecimal integer	digits and letters representing integer number in hexadecimal form	0x[0-9A-F]+	0xA7B0
octal integer	digits representing integer number in octal form	0[0-7]*	0644
number (double)	floating point number represented by 64bits in double precision format	[0-9]+.[0-9]+	456.123
decimal	digits representing a decimal number	[0-9]+.[0-9]+D	123.456D
double quoted string	string value/literal enclosed in double quotes; escaped characters [\n,\r,\t, \\, \", \b] get translated into corresponding control chars	"...anything except ["]..."	"hello\tworld\n\r"
single quoted string	string value/literal enclosed in single quotes; only one escaped character [\'] gets translated into corresponding char [']	'...anything except [']...'	'hello\tworld\n\r'
list of literals	list of literals where individual literals can also be other lists/maps/records	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
date	date value	this mask is expected: yyyy-MM-dd	2008-01-01
datetime	datetime value	this mask is expected: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00

Important

You cannot use any literal for `bytearray` data type. If you want to write a `bytearray` value, you must use any of the conversion functions that return `bytearray` and aply it on an argument value.

For information on these conversion functions see Conversion Functions (p. 637)

Important

Remember that if you need to assign decimal value to a decimal field, you should use decimal literal. Otherwise, such number would not be decimal, it would be a double number!

For example:

1. **Decimal value to a decimal field (correct and accurate)**

```
// correct - assign decimal value to decimal field
myRecord.decimalField = 123.56d;
```

2. **Double value to a decimal field (possibly inaccurate)**

```
// possibly inaccurate - assign double value to decimal field
myRecord.decimalField = 123.56;
```

The latter might produce inaccurate results!

Variables

If you define some variable, you must do it by typing data type of the variable, white space, the name of the variable and semicolon.

Such variable can be initialized later, but it can also be initialized in the declaration itself. Of course, the value of the expression must be of the same data type as the variable.

Both cases of variable declaration and initialization are shown below:

```
dataType variable;

...

variable=expression;

dataType variable=expression;
```

Operators

The operators serve to create more complicated expressions within the program. They can be arithmetic, relational and logical. The relational and logical operators serve to create expressions with resulting boolean value. The arithmetic operators can be used in all expressions, not only the logical ones.

All operators can be grouped into three categories:

- Arithmetic Operators (p. 613)
- Relational Operators (p. 615)
- Logical Operators (p. 617)

Arithmetic Operators

The following operators serve to put together values of different expressions (except those of boolean values). These signs can be used more times in one expression. In such a case, you can express priority of operations by parentheses. The result depends on the order of the expressions.

- Addition

+

The operator above serves to sum the values of two expressions.

But the addition of two boolean values or two date data types is not possible. To create a new value from two boolean values, you must use logical operators instead.

Nevertheless, if you want to add any data type to a string, the second data type is converted to a string automatically and it is concatenated with the first (string) summand. But remember that the string must be on the first place! Naturally, two strings can be summed in the same way. Note also that the `concat()` function is faster and you should use this function instead of adding any summand to a string.

You can also add any numeric data type to a date. The result is a date in which the number of days is increased by the whole part of the number. Again, here is also necessary to have the date on the first place.

The sum of two numeric data types depends on the order of the data types. The resulting data type is the same as that of the first summand. The second summand is converted to the first data type automatically.

- Subtraction and Unitary minus

-

The operator serves to subtract one numeric data type from another. Again the resulting data type is the same as that of the minuend. The subtrahend is converted to the minuend data type automatically.

But it can also serve to subtract numeric data type from a date data type. The result is a date in which the number of days is reduced by the whole part of the subtrahend.

- Multiplication

*

The operator serves only to multiplicate two numeric data types.

Remember that during multiplication the first multiplicand determines the resulting data type of the operation. If the first multiplicand is an integer number and the second is a decimal, the result will be an integer number. On the other hand, if the first multiplicand is a decimal and the second is an integer number, the result will be of decimal data type. In other words, order of multiplicands is of importance.

- Division

`/`

The operator serves only to divide two numeric data types. Remember that you must not divide by zero. Dividing by zero throws `TransformLangExecutorRuntimeException` or gives `Infinity` (in case of a number data type)

Remember that during division the numerator determines the resulting data type of the operation. If the nominator is an integer number and the denominator is a decimal, the result will be an integer number. On the other hand, if the nominator is a decimal and the denominator is an integer number, the result will be of decimal data type. In other words, data types of nominator and denominator are of importance.

- Modulus

`%`

The operator can be used for both floating-point data types and integer data types. It returns the remainder of division.

- Incrementing

`++`

The operator serves to increment numeric data type by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is incremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is incremented.

- Decrementing

`--`

The operator serves to decrement numeric data type by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is decremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is decremented.

Relational Operators

The following operators serve to compare some subexpressions when you want to obtain a boolean value result. Each of the mentioned signs can be used. If you choose the `.operator.` signs, they must be surrounded by white spaces. These signs can be used more times in one expression. In such a case you can express priority of comparisons by parentheses.

- Greater than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`>`

`.gt.`

- Greater than or equal to

  Each of the three signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`>=`

`=>`

`.ge.`

- Less than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`<`

`.lt.`

- Less than or equal to

  Each of the three signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`<=`

`=<`

`.le.`

- Equal to

  Each of the two signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`==`

`.eq.`

- Not equal to

  Each of the three signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

```
!=
```

```
<>
```

```
.ne.
```

- Matches regular expression

  The operator serves to compare string and some regular expression. The regular expression can look like this (for example): `"[^a-d].*"` It means that any character (it is expressed by the dot) except `a`, `b`, `c`, `d` (exception is expressed by the `^` sign) (`a-d` means - characters from `a` to `d`) can be contained zero or more times (expressed by `*`). Or, `'[p-s]{5}'` means that `p`, `r`, `s` must be contained exactly five times in the string. For more detailed explanation about how to use regular expressions see `java.util.regex.Pattern`.

```
~=
```

```
.regex.
```

- Contained in

  This operator serves to specify whether some value is contained in the list or in the map of other values.

```
.in.
```

Logical Operators

If the expression whose value must be of boolean data type is complicated, it can consist of some subexpressions (see above) that are put together by logical conjunctions (AND, OR, NOT, .EQUAL TO, NOT EQUAL TO). If you want to express priority in such an expression, you can use parentheses. From the conjunctions mentioned below you can choose either form (for example, `&&` or `and`, etc.).

Every sign of the form `.operator.` must be surrounded by white space.

- Logical AND

`&&`

`and`

- Logical OR

`||`

`or`

- Logical NOT

`!`

`not`

- Logical EQUAL TO

`==`

`.eq.`

- Logical NOT EQUAL TO

`!=`

`<>`

`.ne.`

Simple Statement and Block of Statements

All statements can be divided into two groups:

- **Simple statement** is an expression terminated by semicolon.

  For example:

```
int MyVariable;
```

- **Block of statements** is a series of simple statements (each of them is terminated by semicolon). The statements in a block can follow each other in one line or they can be written in more lines. They are surrounded by curled braces. No semicolon is used after the closing curled brace.

  For example:

```
while (MyInteger<100) {
    Sum = Sum + MyInteger;
    MyInteger++;
}
```

Control Statements

Some statements serve to control the process of the program.

All control statements can be grouped into the following categories:

- Conditional Statements (p. 618)
- Iteration Statements (p. 619)
- Jump Statements (p. 620)

Conditional Statements

These statements serve to branch out the process of the program.

If Statement

On the basis of the `Condition` value this statement decides whether the `Statement` should be executed. If the `Condition` is true, `Statement` is executed. If it is false, the `Statement` is ignored and process continues next after the `if` statement. `Statement` is either simple statement or a block of statements

```
if (Condition) Statement
```

Unlike the previous version of the `if` statement (in which the `Statement`is executed only if the `Condition` is true), other `Statements` that should be executed even if the `Condition` value is false can be added to the `if` statement. Thus, if the `Condition` is true, `Statement1` is executed, if it is false, `Statement2` is executed. See below:

```
if (Condition) Statement1 else Statement2
```

The `Statement2` can even be another `if` statement and also with `else` branch:

```
if (Condition1) Statement1
    else if (Condition2) Statement3
         else Statement4
```

Switch Statement

Sometimes you would have very complicated statement if you created the statement of more branched out `if` statement. In such a case, much more better is to use the `switch` statement.

Now, the `Condition` is evaluated and according to the value of the `Expression` you can branch out the process. If the value of `Expression` is equal to the the value of the `Expression1`, the `Statement1` are executed. The same is valid for the other `Expression : Statement` pairs. But, if the value of `Expression` does not equal to none of the `Expression1,...,ExpressionN`, nothing is done and the process jumps over the `switch` statement. And, if the value of `Expression` is equal to the the values of more `ExpressionK`, more `StatementK` (for different `K`) are executed.

```
switch (Expression) {
    case Expression1 : Statement1
    case Expression2 : Statement2
    ...
    case ExpressionN : StatementN
}
```

In the following case, even if the value of `Expression` does not equal to the values of the `Expression1,...,ExpressionN`, `StatementN+1` is executed.

```
switch (Expression) {
    case Expression1 : Statement1
    case Expression2 : Statement2
    ...
    case ExpressionN : StatementN
    default:StatementN+1
}
```

If the value of the `Expression` in the header of the `switch` function is equal to the the values of more `Expressions#` in its body, each `Expression#` value will be compared to the `Expression` value continuously and corresponding `Statements` for which the values of `Expression` and `Expression#` equal to each other will be executed one after another. However, if you want that only one `Statement#` should be executed for some `Expression#` value, you should put a `break` statement at the end of the block of statements that follow all or at least some of the following expressions: `case Expression#`

The result could look like this:

```
switch (Expression) {
    case Expression1 : {Statement1; break;}
    case Expression2 : {Statement2; break;}
    ...
    case ExpressionN : {StatementN; break;}
    default:StatementN+1
}
```

Iteration Statements

These iteration statements repeat some processes during which some inner `Statements` are executed cyclically until the `Condition` that limits the execution cycle becomes false or they are executed for all values of the same data type.

For Loop

First, the Initialization is set up, after that, the `Condition` is evaluated and if its value is true, the `Statement` is executed and finally the `Iteration` is made.

During the next cycle of the loop, the `Condition` is evaluated again and if it is true, `Statement` is executed and `Iteration` is made. This way the process repeats until the `Condition` becomes false. Then the loop is terminated and the process continues with the other part of the program.

If the `Condition` is false at the beginning, the process jumps over the `Statement` out of the loop.

```
for (Initialization;Condition;Iteration) {
    Statement
}
```

Do-While Loop

First, the `Statement` is executed, then the process depends on the value of `Condition`. If its value is true, the `Statement` is executed again and then the `Condition` is evaluated again and the subprocess either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the end of the loop, even if it is false at the beginning of the subprocess, the `Statement` is executed at least once.

```
do {
    Statement
} while (Condition)
```

While Loop

This process depends on the value of `Condition`. If its value is true, the `Statements` is executed and then the `Condition` is evaluated again and the subprocess either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the start of the loop, if it is false at the beginning of the subprocess, the `Statements` is not executed at all and the loop is jumped over.

```
while (Condition) {
    Statement
}
```

For-Each Loop

The `foreach` statement is executed on all fields of the same data type within a container. Its syntax is as follows:

```
foreach (variable : iterableVariable) {
    Statement
}
```

All elements of the same data type (data type is declared for the `variable` at the beginning of the transformation code) are searched in the `iterableVariable` container. The `iterableVariable` can be a list, a map, or a record. For each variable of the same data type, specified `Statement` is executed. It can be either a simple statement or a block of statements.

Thus, for example, the same `Statement` can be executed for all `string` fields of a record, etc.

Jump Statements

Sometimes you need to control the process in a different way than by decision based on the `Condition` value. To do that, you have the following options:

Break Statement

If you want to stop some subprocess, you can use the following word in the program:

```
break
```

The subprocess breaks and the process jumps to the higher level or to the next `Statements`.

Continue Statement

If you want to stop some iteration subprocess, you can use the following word in the program:

```
continue
```

The subprocess breaks and the process jumps to the next iteration step.

Return Statement

In the functions you can use the `return` word either alone or along with an `expression`. (See the following two options below.) The return statement must be at the end of the function. If it were not at the end, all of the `variableDeclarations`, `Statements` and `Mappings` located after it would be ignored and skipped. The whole function both without the `return` word and with the `return` word alone returns null, whereas the function with the `return expression` returns the value of the `expression`.

```
return
```

```
return expression
```

Error Handling

Clover Transformation Language also provides a simple mechanism for catching and handling possible errors.

As the first step, a string variable must be declared:

```
string MyException;
```

After that, in the code of your transformation, you can use the following `try-catch` statement:

```
try Statement1 catch(MyException) [Statement2]
```

If `Statement1` (a simple statement or a block of statements) is executed successfully, the graph does not fail and the processing continues.

On the other hand, if `Statement1` fails, an exception is thrown and assigned to the `MyException` variable.

The `MyException` variable can be printed or managed other way.

Once the exception is thrown, graph fails unless another `Statement2` (which fixes the failed `Statement1`) is executed.

In addition to it, since version 3.0 of **CloverETL**, CTL1 uses a set of optional `OnError()` functions that exist to each required transformation function.

For example, for required functions (e.g., `append()`, `transform()`, etc.), there exist following optional functions:

`appendOnError()`, `transformOnError()`, etc.

Each of these required functions may have its (optional) couterpart whose name differs from the original (required) by adding the `OnError` suffix.

Moreover, every `<required ctl template function>OnError()` function returns the same values as the original required function.

This way, any exception that is thrown by the original required function causes call of its `<required ctl template function>OnError()` counterpart (e.g., `transform()` fail may call `transformOnError()`, etc.).

In this `transformOnError()`, any incorrect code can be fixed, error message can be printed to Console, etc.

Important

Remember that these `OnError()` functions are not called when the original required functions return **Error codes** (values less then -1)!

If you want that some `OnError()` function is called, you need to use a `raiseError(string arg)` function. Or (as has already been said) also any exception thrown by original required function calls its `OnError()` counterpart.

Functions

You can define your own functions in the following way:

```
function functionName (arg1,arg2,...) {
    variableDeclarations
    Statements
    Mappings
    [return [expression]]
}
```

You must put the return statement at the end. For more information about the return statement see Return Statement (p. 621). Right before it there can be some `Mappings`, the `variableDeclarations` and `Statements` must be at the beginning, the `variableDeclarations` and `Statements` can even be interspersed, but you must remember that undeclared and uninitialized variables cannot be used. So we suggest that first you declare variables and only then specify the `Statements`.

Message Function

Since **CloverETL** version 2.8.0, you can also define a function for your own error messages.

```
function getMessage() {
         return message;
}
```

This `message` variable should be declared as a global string variable and defined anywhere in the code so as to be used in the place where the `getMessage()` function is located. The `message` will be written to console.

Eval

CTL1 offers two functions that enable inline evaluation of a string as if it were CTL code. This way you can interpret text that is e.g stored somewhere in a database as code.

- `eval(string)` - executes the `string` as CTL code
- `eval_exp(string)` - evaluates the `string` as a CTL expression (same as in `eval(string)`) and then returns the value of the result. The value can be saved to a variable.

When using the functions, keep in mind only valid CTL code should be passed to them. So you have to use proper identificators and even terminate expressions with a semicolon. The evaluated expression has a limited scope and cannot see variables declared outside of it.

Example 53.2. Eval() Function Examples

Example 1:

```
int value = eval_exp("2+5"); // the result of the expression is stored to 'value'
```

Example 2:

```
int out; // the variable has to be declared as global

function transform() {

   eval("out = 3 + 5;");
   print_err(out);
   $0.Date := $0.DATE;
   return ALL
}
```

Conditional Fail Expression

You can use a conditional fail expression in CTL1.

However, it can only be used for defining a mapping to an output field.

Important

Remember that (in interpreted mode of CTL2) this expression can be used in multiple ways: for assigning the value to a variable, mapping a value to an output field, or as an argument of a function.

A conditional fail expressions looks like this (for one output field):

```
expression1 : expression2 : expression3 : ... : expressionN;
```

The expressions are evaluated one by one, starting from the first expression and going from left to right.

1. As soon as one of these expressions may successfully be mapped to the output field, it is used and the other expressions are not evaluated.

2. If none of these expressions may be mapped to the output field, graph fails.

Accessing Data Records and Fields

This section describes the way how the record fields should be worked with. As you know, each component may have ports. Both input and output ports are numbered starting from 0.

Metadata of connected edges may be identified by names or IDs.

Metadata consist of fields.

Working with Records and Variables

Now we suppose that ID of metadata of an edge connected to the first port (port 0) - independently of whether it is input or output - is `Customers`, their name is `customers`, and their third field (field 2) is `firstname`.

Following expressions represent the value of the third field (field 2) of the specified metadata:

- `$<port number>.<field number>`

  Example: `$0.2`

  `$0.*` means all fields on the first port (port 0).

- `$<port number>.<field name>`

  Example: `$0.firstname`

- `$<metadata name>.<field number>`

  Example: `$customers.2`

  `$customers.*` means all fields on the first port (port 0).

- `$<metadata name>.<field name>`

  Example: `$customers.firstname`

For input data following syntax can also be used:

- `@<port number>[<field number>]`

  Example: `@0[2]`

  `@0` means the whole input record incoming through the first port (port 0).

- `@<metadata name>[<field number>]`

  Example: `@customers[2]`

  `@customers` means the whole input record whose metadata name is `customers`.

Integer variables can also be used for identifying field numbers of input records. When an integer variable is declared (`int index;`), following is possible:

- `@<port number>[index]`

  Example: `@0[index]`

- `@<metadata name>[index]`

  Example: `@customers[index]`

Remember that metadata name may be the same for multiple edges.

This way you can also create loops. For example, to print out field values on the first input port you can type:

```
int i;
for (i=0; i<length(@0); i++){
     print_err(@0[i]);
}
```

You can also define records in CTL code. Such defitions can look like these:

- `record (metadataID) MyCTLRecord;`

  Example: `record (Customers) MyCustomers;`

- `record (@<port number>) MyCTLRecord;`

  Example: `record (@0) MyCustomers;`

  This is possible for input ports only.

- `record (@<metadata name>) MyCTLRecord;`

  Example: `record (@customers) MyCustomers;`

Records from an input edge can be assigned to records declared in CTL in the following way:

- `MyCTLRecord = @<port number>;`

  Example: `MyCTLRecord = @0;`

Mapping of records to variables looks like this:

- `myVariable = $<port number>.<field number>;`

  Example: `FirstName = $0.2;`

- `myVariable = $<port number>.<field name>;`

  Example: `FirstName = $0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  Example: `FirstName = $customers.2;`

  Remember that metadata names should be unique. Otherwise, use port number instead.

- `myVariable = $<metadata name>.<field name>;`

  Example: `FirstName = $customers.firstname;`

  Remember that metadata names should be unique. Otherwise, use port number instead.

- `myVariable = @<port number>[<field number>];`

  Example: `FirstName = @0[2];`

- `myVariable = @<metadata name>[<field number>];`

  Example: `FirstName = @customers[2];`

  Remember that metadata names should be unique. Otherwise, use port number instead.

Mapping of variables to records can look like this:

- `$<port number>.<field number> := myVariable;`

  Example: `$0.2 := FirstName;`

- `$<port number>.<field name> := myVariable;`

  Example: `$0.firstname := FirstName;`

- `$<metadata name>.<field number> := myVariable;`

  Example: `$customers.2 := FirstName;`

- `$<metadata name>.<field name> := myVariable;`

  Example: `$customers.firstname := FirstName;`

Mapping

Mapping is a part of each transformation defined in some of the **CloverETL** components.

Calculated or generated values or values of input fields are assigned (mapped) to output fields.

1. Mapping assigns a value to an output field.
2. Mapping operator is the following:

   ```
   :=
   ```

3. Mapping must always be defined inside a function.
4. Mapping must be defined at the end of the function and may only be followed by one return statement.
5. Remember that you can also wrap a mapping in a user-defined function which would be subsequently used in *any* place of another function.
6. You can also map different input metadata to different output metadata by field names.

Mapping of Different Metadata (by Name)

When you map input to output like this:

```
$0.* := $0.*;
```

input metadata may even differ from those on the output.

In the expression above, fields of the input are mapped to the fields of the output that have the same name and type as those of the input. The order in which they are contained in respective metadata and the number of all fields in either metadata is of no importance.

Example 53.3. Mapping of Metadata by Name

When you have input metadata in which the first two fields are `firstname` and `lastname`, each of these two fields is mapped to its counterpart on the output. Such output `firstname` field may even be the fifth and `lastname` field be the third, but those two fields of the input will be mapped to these two output fields .

Even if input metadata had more fields and output metadata had more fields, such fields would not be mapped to each other if there did not exist a field with the same name as one of the input fields (independently on the mutual position of the fields in corresponding metadata).

Important

Metadata fields are mapped from input to output by name and data type independently on their order and on the number of all fields!

Use Case 1 - One String Field to Upper Case

To show how mapping works, we provide here a few examples of mappings.

We have a graph with a **Reformat** component. Metadata on its input and output are identical. First two fields (`field1` and `field2`) are of string data type, the third (`field3`) is of integer data type.

1. We want to change the letters of `field1` values to upper case while passing the other two fields unchanged to the output.

2. We also want to distribute records according to the value of `field3`. Those records in which the value of `field3` is less than 5 should be sent to the output port 0, the others to the output port 1.

Examples of Mapping

As the first possibility, we have the mapping for both ports and all fields defined inside the `transform()` function of CTL template.

Example 53.4. Example of Mapping with Individual Fields

The mapping must be defined at the end of a function (the `transform()` function, in this case) and it may only be followed by one return statement.

Since we need that the `return` statement return the number of output port for each record, we must assign it the corresponding value *before* the mapping is defined.

Note that the mappings will be performed for all records. In other words, even when the record will go to the output port 1, also the mapping for output port 0 will be performed, and vice versa.

Moreover, mapping consists of individual fields, which may be complicated in case there are many fields in a record. In the next examples, we will see how this can be solved in a better way.

```
//#TL

// declare variable for returned value (number of output port)
int retInt;

function transform() {
     // create returned value whose meaning is the number of output port.
     if ($0.field3 < 5) retInt = 0; else retInt = 1;

     // define mapping for all ports and for each field
     // (each field is mapped separately)
     $0.field1 := uppercase($0.field1);
     $0.field2 := $0.field2;
     $0.field3 := $0.field3;
     $1.field1 := uppercase($0.field1);
     $1.field2 := $0.field2;
     $1.field3 := $0.field3;

     // return the number of output port
     return retInt
}
```

As the second possibility, we also have the mapping for both ports and all fields defined inside the `transform()` function of CTL template. But now there are wild cards used in the mapping. These passes the records unchanged to the outputs and after this wildcard mapping the fields that should be changed are specified.

Example 53.5. Example of Mapping with Wild Cards

The mapping must also be defined at the end of a function (the `transform()` function, in this case) and it may only be followed by one return statement.

Since we need that return statement returns the number of output port for each record, we must assign it the corresponding value *before* the mapping is defined.

Note that mappings will be performed for all records. In other words, even when the record will go to the output port 1, also the mapping for output port 0 will be performed, and vice versa.

However, now the mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed *below* the mapping with wild cards.

This is useful when there are many unchanged fields and a few that will be changed.

```
//#TL

// declare variable for returned value (number of output port)
int retInt;

function transform() {
     // create returned value whose meaning is the number of output port.
     if ($0.field3 < 5) retInt = 0; else retInt = 1;

     // define mapping for all ports and for each field
     // (using wild cards and overwriting one selected field)
     $0.* := $0.*;
     $0.field1 := uppercase($0.field1);
     $1.* := $0.*;
     $1.field1 := uppercase($0.field1);

     // return the number of output port
     return retInt
}
```

As the third possibility, we have the mapping for both ports and all fields defined outside the `transform()` function of CTL template. Each output port has its own mapping.

Also here, wild cards are used.

The mapping that is defined in separate function for each output port allows the following improvements:

1. Mappings may now be used inside the code in the `transform()` function! Not only at its end.
2. Mapping is performed only for respective output port! In other words, now there is no need to map record to the port 1 when it will go to the port 0, and vice versa.
3. And, there is no need of a variable for the number of output port. Number of output port is defined by constants immediately after corresponding mapping function.

Example 53.6. Example of Mapping with Wild Cards in Separate User-Defined Functions

The mappings must be defined at the end of a function (two separate functions, by one for each output port).

Moreover, mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed below the mapping with wild card. This is of use when there are many unchanged fields and a few that will be changed.

```
//#TL

// define mapping for each port and for each field
// (using wild cards and overwriting one selected field)
// inside separate functions
function mapToPort0 () {
     $0.* := $0.*;
     $0.field1 := uppercase($0.field1);
}

function mapToPort1 () {
     $1.* := $0.*;
     $1.field1 := uppercase($0.field1);
}

// use mapping functions for all ports in the if statement
function transform() {
     if ($0.field3 < 5) {
          mapToPort0();
          return 0
     }
     else {
          mapToPort1();
          return 1
     }
}
```

Use Case 2 - Two String Fields to Upper Case

We have a graph with a **Reformat** component. Metadata on its input and output are identical. First two fields (`field1` and `field2`) are of string data type, the third (`field3`) is of integer data type.

1. We want to change the letters of both the `field1` and the `field2` values to upper case while passing the last field (`field3`) unchanged to the output.

2. We also want to distribute records according to the value of `field3`. Those records in which the value of `field3` is less than 5 should be sent to the output port 0, the others to the output port 1.

Example 53.7. Example of Successive Mapping in Separate User-Defined Functions

Mapping is defined in two separate user-defined functions. The first of them maps the first input field to both output ports. The second maps the other fields to both output fields.

Note that these functions now accept one input parameter of string data type - `valueString`.

In the transformation, a CTL record varibale is declared. Input record is mapped to it and the record is used in the `foreach` loop.

Remember that the number of output port is defined in the `if` that follows the code with the mapping functions.

```
//#TL

string myString;
string newString;
string valueString;

// declare the count variable for counting string fields
int count;

// declare CTL record for the foreach loop
record (@0) CTLRecord;

// declare mapping for field 1
function mappingOfField1 (valueString) {
     $0.field1 := valueString;
     $1.field1 := valueString;
}

// declare mapping for field 2 and 3
function mappingOfField2and3 (valueString) {
     $0.field2 := valueString;
     $1.field2 := valueString;
     $0.field3 := $0.field3;
     $1.field3 := $0.field3;
}

function transform() {

     // count is initialized for each record
     count = 0;

     // input record is assigned to CTL record
     CTLRecord = @0;

     // value of each string field is changed to upper case letters
     foreach (myString : CTLRecord) {
          newString = uppercase(myString);
          // count variable counts the string fields in the record
          count++;

          // mapping is used for fields 1 and the other fields - 2 and 3
          switch (count) {
               case 1 : mappingOfField1(newString);
               case 2 : mappingOfField2and3(newString);
          }
     }

     // output port is selected based on the return value
     if($0.field3 < 5) return 0 else return 1
}
```

Parameters

The parameters can be used in Clover transformation language in the following way: `${nameOfTheParameter}`. If you want such a parameter is considered a string data type, you must surround it by single or double quotes like this: `'${nameOfTheParameter}'` or `"${nameOfTheParameter}"`.

Important

1. Remember that escape sequences are always resolved as soon as they are assigned to parameters. For this reason, if you want that they are not resolved, type double backslashes in these strings instead of single ones.

2. Remember also that you can get the values of environment variables using parameters. To learn how to do it, see Environment Variables (p. 193).

Functions Reference

Clover transformation language has at its disposal a set of functions you can use. We describe them here.

All functions can be grouped into following categories:

- Conversion Functions (p. 637)
- Date Functions (p. 642)
- Mathematical Functions (p. 645)
- String Functions (p. 649)
- Miscellaneous Functions (p. 659)
- Dictionary Functions (p. 661)
- Lookup Table Functions (p. 662)
- Sequence Functions (p. 664)
- Custom CTL Functions (p. 665)

Important

Remember that if you set the **Null value** property in metadata for any `string` data field to any non-empty string, any function that accept `string` data field as an argument and throws NPE when applied on `null` (e.g., `length()`), it will throw NPE when applied on such specific string.

For example, if `field1` has **Null value** property set to `"<null>"`, `length($0.field1)` will fail on the records in which the value of `field1` is `"<null>"` and it will be 0 for empty field.

See Null value (p. 145) for detailed information.

Conversion Functions

Sometimes you need to convert values from one data type to another.

In the functions that convert one data type to another, sometimes a format pattern of a date or any number must be defined. Also locale can have an influence to their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 106).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of these functions:

- `bytearray` **`base64byte`**`(string` *`arg`*`);`

  The `base64byte(string)` function takes one string argument in `base64` representation and converts it to an array of bytes. Its counterpart is the `byte2base64(bytearray)` function.

- `string` **`bits2str`**`(bytearray` *`arg`*`);`

  The `bits2str(bytearray)` function takes an array of bytes and converts it to a string consisting of two characters: `"0"` or `"1"`. Each byte is represented by eight characters ("0" or "1"). For each byte, the lowest bit is at the beginning of these eight characters. The counterpart is the `str2bits(string)` function.

- `int` **`bool2num`**`(boolean` *`arg`*`);`

  The `bool2num(boolean)` function takes one boolean argument and converts it to either integer `1` (if the argument is true) or integer `0` (if the argument is false). Its counterpart is the `num2bool(<numeric type>)` function.

- `<numeric type>` **`bool2num`**`(boolean` *`arg`*`, typename <`*`numeric type`*`>);`

  The `bool2num(boolean, typename)` function accepts two arguments: the first is boolean and the other is the name of any numeric data type. It takes them and converts the first argument to the corresponding `1` or `0` in the `numeric` representation specified by the second argument. The return type of the function is the same as the second argument. Its counterpart is the `num2bool(<numeric type>)` function.

- `string` **`byte2base64`**`(bytearray` *`arg`*`);`

  The `byte2base64(bytearray)` function takes an array of bytes and converts it to a string in `base64` representation. Its counterpart is the `base64byte(string)` function.

- `string` **`byte2hex`**`(bytearray` *`arg`*`);`

  The `byte2hex(bytearray)` function takes an array of bytes and converts it to a string in `hexadecimal` representation. Its counterpart is the `hex2byte(string)` function.

- `long` **`date2long`**`(date` *`arg`*`);`

  The `date2long(date)` function takes one date argument and converts it to a long type. Its value is equal to the the number of milliseconds elapsed from `January 1, 1970, 00:00:00 GMT` to the date specified as the argument. Its counterpart is the `long2date(long)` function.

- `int` **`date2num`**`(date` *`arg`*`, unit` *`timeunit`*`);`

  The `date2num(date, unit)` function accepts two arguments: the first is date and the other is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes these two arguments and converts them to an integer. If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`. Remember that months are numbered starting from `0`. Thus, `date2num(2008-06-12, month)` returns `5`. And `date2num(2008-06-12, hour)` returns `0`.

- `string` **`date2str`**`(date` *`arg`*`, string` *`pattern`*`);`

  The `date2str(date, string)` function accepts two arguments: date and string. The function takes them and converts the date according to the `pattern` specified as the second argument. Thus, `date2str(2008-06-12, "dd.MM.yyyy")` returns the following string: `"12.6.2008"`. Its counterpart is the `str2date(string, string)` function.

- `string` **`get_field_name`**`(record` *`argRecord`*`, integer` *`index`*`);`

  The `get_field_name(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the name of the field with the specified index. Fields are numbered starting from 0.

- `string` **`get_field_type`**`(record` *`argRecord`*`, integer` *`index`*`);`

  The `get_field_type(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the type of the field with the specified index. Fields are numbered starting from 0.

- `bytearray` **`hex2byte`**`(string` *`arg`*`);`

  The `hex2byte(string)` function takes one string argument in `hexadecimal` representation and converts it to an array of bytes. Its counterpart is the `byte2hex(bytearray)` function.

- `date` **`long2date`**`(long` *`arg`*`);`

  The `long2date(long)` function takes one long argument and converts it to a date. It adds the argument number of milliseconds to `January 1, 1970, 00:00:00 GMT` and returns the result as a date. Its counterpart is the `date2long(date)` function.

- `bytearray` **`long2pacdecimal`**`(long` *`arg`*`);`

  The `long2pacdecimal(long)` function takes one argument of long data type and returns its value in the representation of packed decimal number. It is the counterpart of the `pacdecimal2long(bytearray)` function.

- `bytearray` **`md5`**`(bytearray` *`arg`*`);`

  The `md5(bytearray)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its `MD5` hash value.

- `bytearray` **`md5`**`(string` *`arg`*`);`

  The `md5(string)` function accepts one argument of string data type. It takes this argument and calculates its `MD5` hash value.

- `boolean` **`num2bool`**`(<numeric type>` *`arg`*`);`

  The `num2bool(<numeric type>)` function takes one argument of any numeric data type representing 1 or 0 and returns boolean `true` or `false`, respectively.

- `<numeric type>` **`num2num`**`(<numeric type>` *`arg`*`, typename <`*`numeric type`*`>);`

  The `num2num(<numeric type>, typename)` function accepts two arguments: the first is of any numeric data type and the second is the name of any numeric data type. It takes them and converts the first argument value to that of the `numeric` type specified as the second argument. The return type of the function is the same as the second argument. The conversion is successful only if it is possible without any loss of information, otherwise the function throws exception. Thus, `num2num(25.4, int)` throws exception, whereas `num2num(25.0, int)` returns `25`.

- `string` **`num2str`**`(<numeric type>` *`arg`*`);`

  The `num2str(<numeric type>)` function takes one argument of any numeric data type and converts it to its string representation. Thus, `num2str(20.52)` returns `"20.52"`.

- `string` **`num2str`**`(<numeric type>` *`arg`*`, int` *`radix`*`);`

  The `num2str(<numeric type>, int)` function accepts two arguments: the first is of any numeric data type and the second is integer. It takes these two arguments and converts the first to its string representation in the `radix` based numeric system. Thus, `num2str(31, 16)` returns `"1F"`.

- `string` **`num2str`**`(<numeric type>` *`arg`*`, string` *`format`*`);`

  The `num2str(<numeric type>, string)` function accepts two arguments: the first is of any numeric data type and the second is string. It takes these two arguments and converts the first to its string representation using the format specified as the second argument.

- `long` **`pacdecimal2long`**`(bytearray` *`arg`*`);`

  The `pacdecimal2long(bytearray)` function takes one argument of an array of bytes whose meaning is the packed decimal representation of a long number. It returns its value as long data type. It is the counterpart of the `long2pacdecimal(long)` function.

- `bytearray` **`sha`**`(bytearray` *`arg`*`);`

  The `sha(bytearray)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its SHA hash value.

- `bytearray` **`sha`**`(string` *`arg`*`);`

  The `sha(string)` function accepts one argument of string data type. It takes this argument and calculates its SHA hash value.

- `bytearray` **`str2bits`**`(string` *`arg`*`);`

  The `str2bits(string)` function takes one string argument and converts it to an array of bytes. Its counterpart is the `bits2str(bytearray)` function. The string consists of the following characters: Each of them can be either `"1"` or it can be any other character. In the string, each character `"1"` is converted to the bit `1`, all other characters (not only `"0"`, but also `"a"`, `"z"`, `"/"`, etc.) are converted to the bit `0`. If the number of characters in the string is not an integral multiple of eight, the string is completed by "0" characters from the right. Then, the string is converted to an array of bytes as if the number of its characters were integral multiple of eight.

- `boolean` **`str2bool`**`(string` *`arg`*`);`

  The `str2bool(string)` function takes one string argument and converts it to the corresponding boolean value. The string can be one of the following: `"TRUE"`, `"true"`, `"T"`, `"t"`, `"YES"`, `"yes"`, `"Y"`, `"y"`, `"1"`,

"FALSE", "false", "F", "f", "NO", "no", "N", "n", "0". The strings are converted to boolean true or boolean false.

- date **str2date**(string *arg*, string *pattern*);

  The str2date(string, string) function accepts two string arguments. It takes them and converts the first string to the date according to the pattern specified as the second argument. The pattern must correspond to the structure of the first argument. Thus, str2date("12.6.2008", "dd.MM.yyyy") returns the following date: 2008-06-12.

- date **str2date**(string *arg*, string *pattern*, string *locale*, boolean *lenient*);

  The str2date(string, string, string, boolean) function accepts three string arguments and one boolean. It takes the arguments and converts the first string to the date according to the pattern specified as the second argument. The pattern must correspond to the structure of the first argument. Thus, str2date("12.6.2008", "dd.MM.yyyy") returns the following date: 2008-06-12 . The third argument defines the locale for the date. The fourth argument specify whether date interpretation should be lenient (true) or not (false). If it is true, the function tries to make interpretation of the date even if it does not match locale and/or pattern. If this function has three arguments only, the third one is interpreted as locale (if it is string) or lenient (if it is boolean).

- <numeric type> **str2num**(string *arg*);

  The str2num(string) function takes one string argument and converts it to the corresponding numeric value. Thus, str2num("0.25") returns 0.25 if the function is declared with double return type, but the same throws exception if it is declared with integer return type. The return type of the function can be any numeric type.

- <numeric type> **str2num**(string *arg*, typename <*numeric type*>);

  The str2num(string, typename) function accepts two arguments: the first is string and the second is the name of any numeric data type. It takes the first argument and returns its corresponding value in the numeric data type specified by the second argument. The return type of the function is the same as the second argument.

- <numeric type> **str2num**(string *arg*, typename <*numeric type*>, int *radix*);

  The str2num(string, typename, int) function accepts three arguments: string, the name of any numeric data type and integer. It takes the first argument as if it were expressed in the radix based numeric system representation and returns its corresponding value in the numeric data type specified as the second argument. The return type is the same as the second argument. The third argument can be 10 or 16 for number data type as the second argument (however, radix does not need to be specified as the form of the string alone determines whether the string is decimal or hexadecimal string representation of a number), 10 for decimal type as the second argument and any integer number between Character.MIN_RADIX and Character.MAX_RADIX for int and long types as the second argument.

- <numeric type> **str2num**(string *arg*, typename <*numeric type*>, string *format*);

  The str2num(string, typename, string) function accepts three arguments. The first is a string that should be converted to the number, the second is the name of the return numeric data type and the third is the format of the string representation of a number used in the first argument. The type name specified as the second argument can neither be received through the edge nor be defined as variable. It must be specified directly in the function. The function takes the first argument, compares it with the format using system value locale and returns the numeric value of data type specified as the second argument.

- <numeric type> **str2num**(string *arg*, typename <*numeric type*>, string *format*, string *locale*);

  The str2num(string, typename, string, string) function accepts four arguments. The first is a string that should be converted to the number, the second is the name of the return numeric data type, the third is the format of the string representation of a number used in the first argument and the fourth is the locale

that should be used when applying the format. The type name specified as the second argument can neither be received through the edge nor be defined as variable. It must be specified directly in the function. The function takes the first argument, compares it with the format using the locale at the same time and returns the numeric value of data type specified as the second argument.

- `string` **`to_string`**`(<any type>` *`arg`*`);`

  The **`to_string`**`(<any type>)` function takes one argument of any data type and converts it to its string representation.

- `returndatatype` **`try_convert`**`(<any type>` *`from`*`, typename` *`returndatatype`*`);`

  The `try_convert(<any type>, typename)` function accepts two arguments: the first is of any data type and the second is the name of any other data type. The name of the second argument can neither be received through the edge nor be defined as variable. It must be specified directly in the function. The function takes these arguments and tries to convert the first argument to specified data type. If the conversion is possible, the function converts the first argument to data type specified as the second argument. If the conversion is not possible, the function returns null.

- `date` **`try_convert`**`(string` *`from`*`, datetypename` *`date`*`, string` *`format`*`);`

  The `try_convert(string, nameofdatedatatype, string)` function accepts three arguments: the first is of string data type, the second is the name of date data type and the third is a format of the first argument. The `date` word specified as the second argument can neither be received through the edge nor be defined as variable. It must be specified directly in the function. The function takes these arguments and tries to convert the first argument to a date. If the string specified as the first argument corresponds to the form of the third argument, conversion is possible and a date is returned. If the conversion is not possible, the function returns null.

- `string` **`try_convert`**`(date` *`from`*`, stringtypename` *`string`*`, string` *`format`*`);`

  The `try_convert(date, nameofstringdatatype, string)` function accepts three arguments: the first is of date data type, the second is the name of string data type and the third is a format of a string representation of a date. The string word specified as the second argument can neither be received through the edge nor be defined as variable. It must be specified directly in the function. The function takes these arguments and converts the first argument to a string in the form specified by the third argument.

- `boolean` **`try_convert`**`(<any type>` *`from`*`, <any type>` *`to`*`, string` *`pattern`*`);`

  The `try_convert(<any type>, <any type>, string)` function accepts three arguments: two are of any data type, the third is string. The function takes these arguments, tries convert the first argument to the second. If the conversion is successful, the second argument receives the value from the first argument. And the function returns boolean true. If the conversion is not successful, the function returns boolean false and the first and second arguments retain their original values. The third argument is optional and it is used only if any of the first two arguments is string. For example, `try_convert("27.5.1942", dateA, "dd.MM.yyyy")` returns `true` and `dateA` gets the value of the 27 May 1942.

Date Functions

When you work with date, you may use the functions that process dates.

In these functions, sometimes a format pattern of a date or any number must be defined. Also locale can have an influence to their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of the functions:

- `date` **`dateadd`**`(date` *`arg`*`, <numeric type>` *`amount`*`, unit` *`timeunit`*`);`

  The `dateadd(date, <numeric type>, unit)` function accepts three arguments: the first is date, the second is of any numeric data type and the last is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes the first argument, adds the amount of time units to it and returns the result as a date. The `amount` and time `unit` are specified as the second and third arguments, respectively.

- `int` **`datediff`**`(date` *`later`*`, date` *`earlier`*`, unit` *`timeunit`*`);`

  The `datediff(date, date, unit)` function accepts three arguments: two dates and one time unit. It takes these arguments and subtracts the second argument from the first argument. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can be neither received through an edge nor set as variable. The function returns the resulting time difference expressed in time units specified as the third argument. Thus, the difference of two dates is expressed in defined time units. The result is expressed as an integer number. Thus, `date(2008-06-18, 2001-02-03, year)` returns `7`. But, `date(2001-02-03, 2008-06-18, year)` returns `-7`!

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`);`

  The `random_date(date, date)` function accepts two date arguments and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The default `format` is specified in the `defaultProperties` file.

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, long` *`randomSeed`*`);`

  The `random_date(date, date, long)` function accepts two date arguments and one long argument and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value

is used. The default `format` is specified in the `defaultProperties` file. The third argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, string` *`format`*`);`

  The `random_date(date, date, string)` function accepts two date arguments and one string argument and returns a random date between `startDate` and `endDate` corresponding to the `format` specified by the third argument. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used.

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, string` *`format`*`, long` *`randomSeed`*`);`

  The `random_date(date, date, string, long)` function accepts two date arguments, one string and one long arguments and returns a random date between `startDate` and `endDate` corresponding to the `format` specified by the third argument. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The fourth argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, string` *`format`*`, string` *`locale`*`);`

  The `random_date(date, date, string, string)` function accepts two date arguments and two string arguments and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate` corresponding to the `format` and the `locale` specified by the third and the fourth argument, respectively. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`.

- `date` **`random_date`**`(date` *`startDate`*`, date` *`endDate`*`, string` *`format`*`, string` *`locale`*`, long` *`randomSeed`*`);`

  The `random_date(date, date, string, string, long)` function accepts two date arguments, two strings and one long argument returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate` corresponding to the `format` and the `locale` specified by the third and the fourth argument, respectively. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. The fifth argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `date` **`today`**`();`

  The `today()` function accepts no argument and returns current date and time.

- `date` **`trunc`**`(date` *`arg`*`);`

  The `trunc(date)` function takes one date argument and returns the date with the same year, month and day, but hour, minute, second and millisecond are set to 0.

- `long` **`trunc`**`(<numeric type>` *`arg`*`);`

  The `trunc(<numeric type>)` function takes one argument of any numeric data type and returns its truncated long value.

- `null` **`trunc`**`(list` *`arg`*`);`

  The `trunc(list)` function takes one list argument, empties its values and returns null.

- `null` **`trunc`**`(map` *`arg`*`);`

  The `trunc(map)` function takes one map argument, empties its values and returns null.

- `date` **`trunc_date`**`(date` *`arg`*`);`

  The `trunc_date(date)` function takes one date argument and returns the date with the same hour, minute, second and millisecond, but year, month and day are set to 0. The 0 date is 1970-01-01.

Mathematical Functions

You may also want to use some mathematical functions:

- `<numeric type> abs(<numeric type> arg);`

  The `abs(<numeric type>)` function takes one argument of any numeric data type and returns its absolute value.

- `long bit_and(<numeric type> arg1, <numeric type> arg2);`

  The `bit_and(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments and returns the number corresponding to the bitwise `and`. (For example, `bit_and(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` what corresponds to decimal `3`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric representation.

- `long bit_invert(<numeric type> arg);`

  The `bit_invert(<numeric type>)` function accepts one argument of any numeric data type. It takes its integer part and returns the number corresponding to its bitwise `inverted number`. (For example, `bit_invert(11)` returns `-12`.) The function inverts all bits in an argument. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric representation.

- `boolean bit_is_set(<numeric type> arg, <numeric type> Index);`

  The `bit_is_set(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments, determines the value of the bit of the first argument located on the `Index` and returns `true` or `false`, if the bit is `1` or `0`, respectively. (For example, `bit_is_set(11,3)` returns `true`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bit_is_set(11,2)` would return `false`.

- `long bit_lshift(<numeric type> arg, <numeric type> Shift);`

  The `bit_lshift(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments and returns the number corresponding to the original number with some bits added (`Shift` number of bits on the left side are added and set to `0`.) (For example, `bit_lshift(11,2)` returns `44`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`10`) are added and the result is `101100` which corresponds to decimal `44`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- `long bit_or(<numeric type> arg1, <numeric type> arg2);`

  The `bit_or(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments and returns the number corresponding to the bitwise `or`. (For example, `bit_or(11,7)` returns `15`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` what corresponds to decimal `15`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- `long bit_rshift(<numeric type> arg, <numeric type> Shift);`

  The `bit_rshift(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments and returns the number corresponding to the original number with some bits removed (`Shift` number of bits on the right side are removed.) (For example, `bit_rshift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` what corresponds to decimal `2`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- `long` **`bit_set`**`(<numeric type>` *`arg1`*`, <numeric type>` *`Index`*`, boolean` *`SetBitTo1`*`);`

  The `bit_set(<numeric type>, <numeric type>, boolean)` function accepts three arguments. The first two are of any numeric data type and the third is boolean. It takes integer parts of the first two arguments, sets the value of the bit of the first argument located on the `Index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as a long value. (For example, `bit_set(11,3,false)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` what corresponds to decimal `3`. And `bit_set(11,2,true)` would return `1111` what corresponds to decimal `15`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- `long` **`bit_xor`**`(<numeric type>` *`arg`*`, <numeric type>` *`arg`*`);`

  The `bit_xor(<numeric type>, <numeric type>)` function accepts two arguments of any numeric data type. It takes integer parts of both arguments and returns the number corresponding to the bitwise `exclusive or`. (For example, `bit_or(11,7)` returns `12`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` what corresponds to decimal `15`. Return data type is `long`, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- `number` **`e`**`();`

  The `e()` function accepts no argument and returns the Euler number.

- `number` **`exp`**`(<numeric type>` *`arg`*`);`

  The `exp(<numeric type>)` function takes one argument of any numeric data type and returns the result of the exponential function of this argument.

- `number` **`log`**`(<numeric type>` *`arg`*`);`

  The `log(<numeric type>)` takes one argument of any numeric data type and returns the result of the natural logarithm of this argument.

- `number` **`log10`**`(<numeric type>` *`arg`*`);`

  The `log10(<numeric type>)` function takes one argument of any numeric data type and returns the result of the logarithm of this argument to the base 10.

- `number` **`pi`**`();`

  The `pi()` function accepts no argument and returns the pi number.

- `number` **`pow`**`(<numeric type>` *`base`*`, <numeric type>` *`exp`*`);`

  The `pow(<numeric type>, <numeric type>)` function takes two arguments of any numeric data types (that do not need to be the same) and returns the exponential function of the first argument as the exponent with the second as the base.

- `number` **`random`**`();`

  The `random()` function accepts no argument and returns a random positive double greater than or equal to `0.0` and less than `1.0`.

- `number` **`random`**`(long` *`randomSeed`*`);`

  The `random(long)` function accepts one argument of long data type and returns a random positive double greater than or equal to `0.0` and less than `1.0`. The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `boolean **random_boolean**();`

  The `random_boolean()` function accepts no argument and generates at random boolean values `true` or `false`. If these values are sent to any numeric data type field, they are converted to their numeric representation automatically (`1` or `0`, respectively).

- `boolean **random_boolean**(long *randomSeed*);`

  The `random_boolean(long)` function accepts one argument of long data type and generates at random boolean values `true` or `false`. If these values are sent to any numeric data type field, they are converted to their numeric representation automatically (`1` or `0`, respectively). The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `<numeric type> **random_gaussian**();`

  The `random_gaussian()` function accepts no argument and generates at random both positive and negative values of return numeric data type in a Gaussian distribution.

- `<numeric type> **random_gaussian**(long *randomSeed*);`

  The `random_gaussian(long)` function accepts one argument of long data type and generates at random both positive and negative values of return numeric data type in a Gaussian distribution. The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `int **random_int**();`

  The `random_int()` function accepts no argument and generates at random both positive and negative integer values.

- `int **random_int**(long *randomSeed*);`

  The `random_int(long)` function accepts one argument of long data type and generates at random both positive and negative integer values. The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `int **random_int**(int *Minimum*, int *Maximum*);`

  The `random_int(int, int)` function accepts two argument of integer data types and returns a random integer value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `int **random_int**(int *Minimum*, int *Maximum*, long *randomSeed*);`

  The `random_int(int, int, long)` function accepts three arguments. The first two are of integer data types and the third is long. The function takes them and returns a random integer value greater than or equal to `Minimum` and less than or equal to `Maximum`. The third argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `long **random_long**();`

  The `random_long()` function accepts no argument and generates at random both positive and negative long values.

- `long **random_long**(long *randomSeed*);`

  The `random_long(long)` function accepts one argument of long data type and generates at random both positive and negative long values. The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `long` **`random_long`**`(long` *`Minimum`*`, long` *`Maximum`*`);`

  The `random_long(long, long)` function accepts two argument of long data types and returns a random long value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `long` **`random_long`**`(long` *`Minimum`*`, long` *`Maximum`*`, long` *`randomSeed`*`);`

  The `random_long(long, long, long)` function accepts three arguments of long data types and returns a random long value greater than or equal to `Minimum` and less than or equal to `Maximum`. The argument ensures that the generated values remain the same upon each run of the graph. The generated values can only be changed by changing the `randomSeed` value.

- `long` **`round`**`(<numeric type>` *`arg`*`);`

  The `round(<numeric type>)` function takes one argument of any numeric data type and returns the long that is closest to this argument.

- `number` **`sqrt`**`(<numeric type>` *`arg`*`);`

  The `sqrt(<numeric type>)` function takes one argument of any numeric data type and returns the square root of this argument.

String Functions

Some functions work with strings.

In the functions that work with strings, sometimes a format pattern of a date or any number must be defined.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 106).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of the functions:

- `string` **`char_at`**`(string` *`arg`*`, <numeric type>` *`index`*`);`

  The `char_at(string, <numeric type>)` function accepts two arguments: the first is string and the other is of any numeric data type. It takes the string and returns the character that is located at the position specified by the `index`.

- `string` **`chop`**`(string` *`arg`*`);`

  The `chop(string)` function accepts one string argument. The function takes this argument, removes the line feed and the carriage return characters from the end of the string specified as the argument and returns the new string without these characters.

- `string` **`chop`**`(string` *`arg1`*`, string` *`arg2`*`);`

  The `chop(string, string)` function accepts two string arguments. It takes the first argument, removes the string specified as the second argument from the end of the first argument and returns the first string argument without the string specified as the second argument.

- `string` **`concat`**`(<any type>` *`arg1`*`, ... ...,` `<any type>` *`argN`*`);`

  The `concat(<any type>, ..., <any type>)` function accepts unlimited number of arguments of any data type. But they do not need to be the same. It takes these arguments and returns their concatenation. If some arguments are not strings, they are converted to their string representation before the concatenation is done. You can also concatenate these arguments using plus signs, but this function is faster for more than two arguments.

- `int` **`count_char`**`(string` *`arg`*`, string` *`character`*`);`

  The `count_char(string, string)` function accepts two arguments: the first is string and the second is one character. It takes them and returns the number of occurrence of the character specified as the second argument in the string specified as the first argument.

- `list` **`cut`**`(string` *`arg`*`, list` *`list`*`);`

  The `cut(string, list)` function accepts two arguments: the first is string and the second is list of numbers. The function returns a list of strings. The number of elements of the list specified as the second argument must be even. The integer part of each pair of such adjacent numbers of the list argument serve as position (each number in the odd position) and length (each number in the even position). Substrings of the specified

length are taken from the string specified as the first argument starting from the specified position (excluding the character at the specified position). The resulting substrings are returned as list of strings. For example, `cut("somestringasanexample",[2,3,1,5])` returns `["mes","omest"]`.

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`);`

  The `edit_distance(string, string)` function accepts two string arguments. These strings will be compared to each other. The strength of comparison is 4 by default, the default value of locale for comparison is the system value and the maximum difference is 3 by default.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`);`

  The `edit_distance(string, string, string)` function accepts three arguments. The first two are strings that will be compared to each other and the third (string) is the locale that will be used for comparison. The default strength of comparison is 4. The maximum difference is 3 by default.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`);`

  The `edit_distance(string, string, int)` function accepts three arguments. The first two are strings that will be compared to each other and the third (integer) is the strength of comparison. The default locale that will be used for comparison is the system value. The maximum difference is 3 by default.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

- `int` **`edit_distance`**`(string` *`arg1`*`, string` *`arg2`*`, int` *`strength`*`, string` *`locale`*`);`

  The `edit_distance(string, string, int, string)` function accepts four arguments. The first two are strings that will be compared to each other, the third (integer) is the strength of comparison and the fourth (string) is the locale that will be used for comparison. The maximum difference is 3 by default.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `int edit_distance(string arg1, string arg2, string locale, int maxDifference);`

  The `edit_distance(string, string, string, int)` function accepts four arguments. The first two are strings that will be compared to each other, the third (string) is the locale that will be used for comparison and the fourth (integer) is the maximum difference. The strength of comparison is 4 by default.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `int edit_distance(string arg1, string arg2, int strength, int maxDifference);`

  The `edit_distance(string, string, int, int)` function accepts four arguments. The first two are strings that will be compared to each other and the two others are both integers. These are the strength of comparison (third argument) and the maximum difference (fourth argument). The locale is the default system value.

  (For more details, see another version of the `edit_distance()` function below - the `edit_distance(string, string, int, string, int)` function.)

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

- `int edit_distance(string arg1, string arg2, int strength, string locale, int maxDifference);`

  The `edit_distance(string, string, int, string, int)` function accepts five arguments. The first two are strings, the three others are integer, string and integer, respectively. The function takes the first two arguments and compares them to each other using the other three arguments.

  The third argument (integer number) specifies the strength of comparison. It can have any value from 1 to 4.

  If it is 4 (identical comparison), that means that only identical letters are considered equal. In case of 3 (tertiary comparison), that means that upper and lower cases are considered equal. If it is 2 (secondary comparison), that means that letters with diacritical marks are considered equal. And, if the strength of comparison is 1 (primary comparison), that means that even the letters with some specific signs are considered equal. In other versions of the `edit_distance()` function where this strength of comparison is not specified, the number 4 is used as the default strength (see above).

  The fourth argument is of string data type. It is the locale that serves for comparison. If no locale is specified in other versions of the `edit_distance()` function, its default value is the system value (see above).

  The fifth argument (integer number) means the number of letters that should be changed to transform one of the first two arguments to the other. If other version of the `edit_distance()` function does not specify this maximum difference, as the default maximum difference is accepted the number 3 (see above).

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be

changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

Actually the function is implemented for the following locales: CA, CZ, ES, DA, DE, ET, FI, FR, HR, HU, IS, IT, LT, LV, NL, NO, PL, PT, RO, SK, SL, SQ, SV, TR.

See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `list` **`find`**`(string` *`arg`*`, string` *`regex`*`);`

  The `find(string, string)` function accepts two string arguments. The second one is regular expression. The function takes them and returns a list of substrings corresponding to the regex pattern that are found in the string specified as the first argument.

- `string` **`get_alphanumeric_chars`**`(string` *`arg`*`);`

  The `get_alphanumeric_chars(string)` function takes one string argument and returns only letters and digits contained in the string argument in the order of their appearance in the string. The other characters are removed.

- `string` **`get_alphanumeric_chars`**`(string` *`arg`*`, boolean` *`takeAlpha`*`, boolean` *`takeNumeric`*`);`

  The `get_alphanumeric_chars(string, boolean, boolean)` function accepts three arguments: one string and two booleans. It takes them and returns letters and/or digits if the second and/or the third arguments, respectively, are set to true.

- `int` **`index_of`**`(string` *`arg`*`, string` *`substring`*`);`

  The `index_of(string, string)` function accepts two strings. It takes them and returns the index of the first appearance of `substring` in the string specified as the first argument.

- `int` **`index_of`**`(string` *`arg`*`, string` *`substring`*`, int` *`fromIndex`*`);`

  The `index_of(string, string, int)` function accepts three arguments: two strings and one integer. It takes them and returns the index of the first appearance of `substring` counted from the character located at the position specified by the third argument.

- `boolean` **`is_ascii`**`(string` *`arg`*`);`

  The `is_ascii(string)` function takes one string argument and returns a boolean value depending on whether the string can be encoded as an ASCII string (true) or not (false).

- `boolean` **`is_blank`**`(string` *`arg`*`);`

  The `is_blank(string)` function takes one string argument and returns a boolean value depending on whether the string contains only white space characters (true) or not (false).

- `boolean` **`is_date`**`(string` *`arg`*`, string` *`pattern`*`);`

  The `is_date(string, string)` function accepts two string arguments. It takes them, compares the first argument with the second as a pattern and, if the first string can be converted to a date which is valid within system value of `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `is_date()` function below - the `is_date(string, string, string, boolean)` function.)

  This function is a variant of the mentioned `is_date(string, string, string, boolean)` function in which the default value of the third argument is set to system value and the fourth argument is set to false by default.

- `boolean **is_date**(string *arg*, string *pattern*, string *locale*);`

  The `is_date(string, string, string)` function accepts three string arguments. It takes them, compares the first argument with the second as a pattern, use the third argument (`locale`) and, if the first string can be converted to a date which is valid within specified `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `is_date()` function below - the `is_date(string, string, string, boolean)` function.)

  This function is a variant of the mentioned `is_date(string, string, string, boolean)` function in which the default value of the fourth argument (`lenient`) is set to false by default.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `boolean **is_date**(string *arg*, string *pattern*, boolean *lenient*);`

  The `is_date(string, string, boolean)` function accepts two string arguments and one boolean.

  ### Note

  Since the version 2.8.1 of **CloverETL**, the `lenient` argument is ignored and is implicitly set to `false`.

  The function takes these arguments, compares the first argument with the second as a pattern and, if the first string can be converted to a date which is valid within system value of `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `is_date()` function below - the `is_date(string, string, string, boolean)` function.)

  This function is a variant of the mentioned `is_date(string, string, string, boolean)` function in which the default value of the third argument (`locale`) is set to system value.

- `boolean **is_date**(string *arg*, string *pattern*, string *locale*, boolean *lenient*);`

  The `is_date(string, string, string, boolean)` function accepts three string arguments and one boolean.

  ### Note

  Since the version 2.8.1 of **CloverETL**, the `lenient` argument is ignored and is implicitly set to `false`.

  The function takes these arguments, compares the first argument with the second as a pattern, use the third (`locale`) argument and, if the first string can be converted to a date which is valid within specified `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `boolean **is_integer**(string *arg*);`

  The `is_integer(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to an integer number (true) or not (false).

- `boolean **is_long**(string *arg*);`

  The `is_long(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a long number (true) or not (false).

- `boolean **is_number**(string *arg*);`

  The `is_number(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a double (true) or not (false).

- `string **join**(string *delimiter*, <any type> *arg1*, ... ..., <any type> *argN*);`

  The `join(string, <any type>, ..., <any type>)` function accepts unlimited number of arguments. The first is string, the others are of any data type. All data types do not need to be the same. The arguments that are not strings are converted to their string representation and put together with the first argument as delimiter.

Note

This function was already included in **CloverETL Engine** and **CloverETL Designer** that were release before 2.7.0 or 2.2.0, respectively. However, this older version of the `join()` function also added a terminal delimiter to the end of the sequence unlike the new version of the `join()` function in which the delimiter is only inserted between each pair of elements.

In older releases `join(";",argA,argB)` returned the following string: `"argA;argB;"`. Since **CloverETL Engine** 2.7.0 and **CloverETL Designer** 2.2.0, the result of the same expression is as follows: `"argA;argB"` (without the terminal `;`).

Thus, if you want to run an old graph created for old version of **CloverETL Engine** and **CloverETL Designer** that uses this function in **CloverETL Engine** 2.7.0 and **CloverETL Designer** 2.2.0 or higher, replace the old `join(delimiter,...)` expression in the graph by `join(delimiter,...) + delimiter`. In the case mentioned above, you should replace the older `join(";",argA,argB)` expression with the new `join(";",argA,argB) + ";"` expression.

- `string **join**(string *delimiter*, arraytype *arg*);`

  The `join(string, arraytype)` function accepts two arguments. The first is string, the other one is an array. The elements of the array argument are converted to their string representation and put together with the first argument as delimiter between each pair of them.

Note

Also in this case, older versions of **CloverETL Engine** and **CloverETL Designer** added a delimiter to the end of the resulting sequence, whereas the new versions (since **CloverETL Engine** 2.7.0 and **CloverETL Designer** 2.2.0) do not add any delimiter to its end.

- `string **left**(string *arg*, <numeric type> *length*);`

  The `left(string, <numeric type>)` function accepts two arguments: the first is string and the other is of any numeric data type. It takes them and returns the substring of the length specified as the second argument counted from the start of the string specified as the first argument.

- `int **length**(structuredtype *arg*);`

  The `length(structuredtype)` function accepts one argument of structured data type: `string`, `bytearray`, `list`, `map` or `record`. It takes this argument and returns the number of elements composing the argument.

- `string **lowercase**(string *arg*);`

  The `lowercase(string)` function takes one string argument and returns another string with cases converted to lower cases only.

- `string` **`metaphone`**`(string` *`arg`*`, int` *`maxLength`*`);`

  The `metaphone(string, int)` function accepts one string argument and one integer meaning the maximum length. The function takes these arguments and returns the metaphone code of the first argument of the specified maximum length. The default maximum length is 4. For more information, see the following site: www.lanw.com/java/phonetic/default.htm.

- `string` **`NYSIIS`**`(string` *`arg`*`);`

  The `NYSIIS(string)` function takes one string argument and returns the New York State Identification and Intelligence System Phonetic Code of the argument. For more information, see the following site: http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System.

- `string` **`random_string`**`(int` *`minLength`*`, int` *`maxLength`*`);`

  The `random_string(int, int)` function takes two integer arguments and returns strings composed of lowercase letters whose length varies between `minLength` and `maxLength`. These resulting strings are generated at random for records and fields. They can be different for both different records and different fields. Their length can also be equal to `minLength` or `maxLength`, however, they can be neither shorter than `minLength` nor longer than `maxLength`.

- `string` **`random_string`**`(int` *`minLength`*`, int` *`maxLength`*`, long` *`randomSeed`*`);`

  The `random_string(int, int, long)` function takes two integer arguments and one long argument and returns strings composed of lowercase letters whose length varies between `minLength` and `maxLength`. These resulting strings are generated at random for records and fields. They can be different for both different records and different fields. Their length can also be equal to `minLength` or `maxLength`, however, they can be neither shorter than `minLength` nor longer than `maxLength`. The argument ensures that the generated values remain the same upon each run of the graph.

- `string` **`remove_blank_space`**`(string` *`arg`*`);`

  The `remove_blank_space(string)` function takes one string argument and returns another string with white spaces removed.

- `string` **`remove_diacritic`**`(string` *`arg`*`);`

  The `remove_diacritic(string)` function takes one string argument and returns another string with diacritical marks removed.

- `string` **`remove_nonascii`**`(string` *`arg`*`);`

  The `remove_non ascii(string)` function takes one string argument and returns another string with non-ascii characters removed.

- `string` **`remove_nonprintable`**`(string` *`arg`*`);`

  The `remove_nonprintable(string)` function takes one string argument and returns another string with non-printable characters removed.

- `string` **`replace`**`(string` *`arg`*`, string` *`regex`*`, string` *`replacement`*`);`

  The `replace(string, string, string)` function takes three string arguments - a string, a regular expression, and a replacement - and replaces all regex matches inside the string with the replacement string you specified. All parts of the string that match the regex are replaced. You can also reference the matched text using a backreference in the replacement string. A backreference to the entire match is indicated as $0. If there are capturing parentheses, you can reference specifics groups as $1, $2, $3, etc.

  `replace("Hello","[Ll]","t")` returns `"Hetto"`

  `replace("Hello","[Ll]",$0)` returns `"HeHelloHelloo"`

- `string **right**(string *arg*, <numeric type> *length*);`

  The `right(string, <numeric type>)` function accepts two arguments: the first is string and the other is of any numeric data type. It takes them and returns the substring of the length specified as the second argument counted from the end of the string specified as the first argument.

- `string **soundex**(string *arg*);`

  The `soundex(string)` function takes one string argument and converts the string to another. The resulting string consists of the first letter of the string specified as the argument and three digits. The three digits are based on the consonants contained in the string when similar numbers correspond to similarly sounding consonants. Thus, `soundex("word")` returns `"w600"`.

- `list **split**(string *arg*, string *regex*);`

  The `split(string, string)` function accepts two string arguments. The second is some regular expression. It is searched in the first string argument and if it is found, the string is split into the parts located between the characters or substrings of such a regular expression. The resulting parts of the string are returned as a list. Thus, `split("abcdefg", "[ce]")` returns `["ab", "d", "fg"]`.

- `string **substring**(string *arg*, <numeric type> *fromIndex*, <numeric type> *length*);`

  The `substring(string, <numeric type>, <numeric type>)` function accepts three arguments: the first is string and the other two are of any numeric data type. The two numeric types do not need to be the same. The function takes the arguments and returns a substring of the defined length obtained from the original string by getting the `length` number of characters starting from the position defined by the second argument. If the second and third arguments are not integers, only the integer parts of them are used by the function. Thus, `substring("text", 1.3, 2.6)` returns `"ex"`.

- `string **translate**(string *arg*, string *searchingSet*, string *replaceSet*);`

  The `translate(string, string, string)` function accepts three string arguments. The number of characters must be equal in both the second and the third arguments. If some character from the string specified as the second argument is found in the string specified as the first argument, it is replaced by a character taken from the string specified as the third argument. The character from the third string must be at the same position as the character in the second string. Thus, `translate("hello", "leo", "pii")` returns `"hippi"`.

- `string **trim**(string *arg*);`

  The `trim(string)` function takes one string argument and returns another string with leading and trailing whitespace characters removed.

- `string **uppercase**(string *arg*);`

  The `uppercase(string)` function takes one string argument and returns another string with cases converted to upper cases only.

Container Functions

When you work with containers (`list`, `map`, `record`), you may use the following functions:

- `list` **`copy`**`(list` *`arg`*`, list` *`arg`*`);`

  The `copy(list, list)` function accepts two arguments, each of them is list. Elements of all lists must be of the same data type. The function takes the second argument, adds it to the end of the first list and returns the new resulting list. Thus, the resulting list is a sum of both strings specified as arguments. Remember that also the list specified as the first argument changes to this new value.

- `boolean` **`insert`**`(list` *`arg`*`, <numeric type>` *`position`*`, <element type>` *`newelement1`*`, ... ..., <element type>` *`newelementN`*`);`

  The `insert(list, <numeric type>, <element type>1, ..., <element type>N)` function accepts the following arguments: the first is a list, the second is of any numeric data type and the others are of any data type, which is the same for all of them. At the same time, this data type is equal to the that of the list elements. The function takes the elements that are contained in the function starting from the third argument (including the third argument) and inserts them one after another to the list starting from the position defined by the integer part of the second argument. The list specified as the first argument changes to this new value. The function returns `true` if it was successful, otherwise, it returns `false`. Remember that the list element are indexed starting from 0.

- `<element type>` **`poll`**`(list` *`arg`*`);`

  The `poll(list)` function accepts one argument of list data type. It takes this argument, removes the first element from the list and returns this element. Remember that the list specified as the argument changes to this new value (without the removed first element).

- `<element type>` **`pop`**`(list` *`arg`*`);`

  The `pop(list)` function accepts one argument of list data type. It takes this argument, removes the last element from the list and returns this element. Remember that the list specified as the argument changes to this new value (without the removed last element).

- `boolean` **`push`**`(list` *`arg`*`, <element type>` *`list_element`*`);`

  The `push(list, <element type>)` function accepts two arguments: the first is list and the second is of any data type. However, the second argument must be of the same data type as each element of the list. The function takes the second argument and adds it to the end of the first argument. Remember that the list specified as the first argument changes to this new value. The function returns `true` if it was successful, otherwise, it returns `false`.

- `list` **`remove`**`(list` *`arg`*`, <numeric type>` *`position`*`);`

  The `remove(list, <numeric type>)` function accepts two arguments: the first is list and the second is of any numeric data type. The function takes the integer part of the second argument and removes the list element at the specified position. Remember that the list specified as the first argument changes to this new value (without the removed element). And note that the function returns this new list. Remember that the list elements are indexed starting from 0.

- `boolean` **`remove_all`**`(list` *`arg`*`);`

  The `remove_all(list)` function accepts one list argument. The function takes this argument and empties the list. It returns a boolean value. Remember that the list specified as the argument changes to the empty list.

- `list` **`reverse`**`(list` *`arg`*`);`

  The `reverse(list)` function accepts one argument of list data type. It takes this argument, reverses the order of elements of the list and returns such new list. Remember that the list specified as the argument changes to this new value.

- `list` **`sort`**`(list` *`arg`*`);`

  The `sort(list)` function accepts one argument of list data type. It takes this argument, sorts the elements of the list in ascending order according to their values and returns such new list. Remember that the list specified as the argument changes to this new value.

Miscellaneous Functions

The rest of the functions can be denominated as miscellaneous. These are the following:

- `void **breakpoint**();`

  The `breakpoint()` function accepts no argument and prints out all global and local variables.

- `<any type> **iif**(boolean *con*, <any type> *iftruevalue*, <any type> *iffalsevalue*);`

  The `iif(boolean, <any type>, <any type>)` function accepts three arguments: one is boolean and two are of any data type. Both argument data types and return type are the same.

  The function takes the first argument and returns the second if the first is true or the third if the first is false.

- `boolean **isnull**(<any type> *arg*);`

  The `isnull(<any type>)` function takes one argument and returns a boolean value depending on whether the argument is null (true) or not (false). The argument may be of any data type.

Important

If you set the **Null value** property in metadata for any `string` data field to any non-empty string, the `isnull()` function will return `true` when applied on such string. And return `false` when applied on an empty field.

For example, if `field1` has **Null value** property set to `"<null>"`, `isnull($0.field1)` will return `true` on the records in which the value of `field1` is `"<null>"` and `false` on the others, even on those that are empty.

See Null value (p. 145) for detailed information.

- `<any type> **nvl**(<any type> *arg*, <any type> *default*);`

  The `nvl(<any type>, <any type>)` function accepts two arguments of any data type. Both arguments must be of the same type. If the first argument is not null, the function returns its value. If it is null, the function returns the default value specified as the second argument.

- `<any type> **nvl2**(<any type> *arg*, <any type> *arg_for_non_null*, <any type> *arg_for_null*);`

  The `nvl2(<any type>, <any type>, <any type>)` function accepts three arguments of any data type. This data type must be the same for all arguments and return value. If the first argument is not null, the function returns the value of the second argument. If the first argument is null, the function returns the value of the third argument.

- `void **print_err**(<any type> *message*);`

  The `print_err(<any type>)` function accepts one argument of any data type. It takes this argument and prints out the `message` on the error output.

Note

Remember that if you are using this function in any graph that runs on **CloverETL Server**, the `message` is saved to the log of **Server** (e.g., to the log of **Tomcat**). Use the `print_log()` function instead. It logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void print_err(<any type> message, boolean printLocation);`

  The `print_err(type, boolean)` function accepts two arguments: the first is of any data type and the second is boolean. It takes them and prints out the `message` and the location of the error (if the second argument is true).

  ### Note

  Remember that if you are using this function in any graph that runs on **CloverETL Server**, the `message` is saved to the log of **Server** (e.g., to the log of **Tomcat**). Use the `print_log()` function instead. It logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void print_log(level loglevel, <any type> message);`

  The `print_log(level, <any type>)` function accepts two arguments: the first is a log level of the `message` specified as the second argument, which is of any data type. The first argument is one of the following: `debug`, `info`, `warn`, `error`, `fatal`. The log level must be specified as a constant. It can be neither received through an edge nor set as variable. The function takes the arguments and sends out the `message` to a logger.

  ### Note

  Remember that you should use this function especially in any graph that would run on **CloverETL Server** instead of the `print_err()` function which logs error messages to the log of **Server** (e.g., to the log of **Tomcat**). Unlike `print_err()`, `print_log()` logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void print_stack();`

  The `print_stack()` function accepts no argument and prints out all variables from the stack.

- `void raise_error(string message);`

  The `raise_error(string)` function takes one string argument and throws out error with the `message` specified as the argument.

Dictionary Functions

CTL1 provides functions that allow to manipulate the dictionary entries of `string` data type.

Note

These functions allow to manipulate also the entries that are not defined in the graph.

You may write dictionary value to an entry first, and then access it using the functions for reading the dictionary.

- `string` **`read_dict`**`(string` *`name`*`);`

  This function takes the dictionary, selects the entry specified by the `name` and returns its value, which is of string data type.

- `string` **`dict_get_str`**`(string` *`name`*`);`

  This function takes the dictionary, selects the entry specified by the `name` and returns its value, which is of string data type.

- `void` **`write_dict`**`(string` *`name`*`, string` *`value`*`);`

  This function takes the dictionary and writes a new or updates the existing entry of string data type, specified as the first argument of the function, and assigns it the value specified as the second argument, which is also of string data type.

- `boolean` **`dict_put_str`**`(string` *`name`*`, string` *`value`*`);`

  This function takes the dictionary and writes a new or updates the existing entry of string data type, specified as the first argument of the function, and assigns it the value specified as the second argument, which is also of string data type.

- `void` **`delete_dict`**`(string` *`name`*`);`

  This function takes the dictionary and deletes the property with specified `name`.

Currently we are able to work just with string dictionary type. For this reason, to access the value of the `heightMin` property, following CTL code should be used:

```
value = read_dict("heightMin");
```

Lookup Table Functions

In your graphs you are also using lookup tables. You can use them in CTL by specifying ID of the lookup table and placing it as an argument in the `lookup()`, `lookup_next()`, `lookup_found()`, or `lookup_admin()` functions.

Note

The `lookup_admin()` functions do nothing since version 3.0 of **CloverETL** and can be removed.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

You have five options depending on what you want to do with the lookup table. You can create lookup table, get the value of the specified field name from the lookup table associated with the specified key, or get the next value of the specified field name from the lookup table, or (if the records are duplicated) count the number of the records with the same field name values, or you can destroy the lookup table.

Now, the key in the function below is a sequence of values of the field names separated by comma (not semicolon!). Thus, the `keyValue` is of the following form: `keyValuePart1,keyValuePart2,...,keyValuePartN`.

See the mentioned following five options:

- `lookup_admin(<lookup ID>,init)`[1)]

  This function initializes the specified lookup table.

- `lookup(<lookup ID>,keyValue).<field name>`

  This function searches the first record whose key value is equal to the value specified as the second argument in this function and returns the value of `<field name>`. Here, `<field name>` is a field of the lookup table metadata.

- `lookup_next(<lookup ID>).<field name>`

  After call the `lookup()` function, the `lookup_next()` function searches the next record whose key value is equal to the value specified as the second argument in the `lookup()` function and returns the value of `<field name>` value. Here, `<field name>` is a field of the lookup table.

- `lookup_found(<lookup ID>)`

  After call the `lookup()` function, the `lookup_found()` function returns the number of records whose key value is equal to the value specified as the second argument in the `lookup()` function.

- `lookup_admin(<lookup ID>,free)`[1)]

  This function destroys the specified lookup table.

Legend:

1) These functions do nothing since version 3.0 of **CloverETL** and can be removed from the code.

Warning

Remember that the usage of the `lookup_found(<lookup ID>)` function of CTL1 is not too recommended.

The reason is that such expression searches the records through the whole lookup table which may contain a great number of records.

You should better use a pair of two functions in a loop:

```
lookup(<lookup ID>,keyValue).<field name>
```

```
lookup_next(<lookup ID>).<field name>
```

Especially DB lookup tables may return -1 instead of real count of records with specified key value (if you do not set **Max cached size** to a non-zero value).

Sequence Functions

In your graphs you are also using sequences. You can use them in CTL by specifying ID of the sequence and placing it as an argument in the `sequence()` function.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

You have three options depending on what you want to do with the sequence. You can get the current number of the sequence, or get the next number of the sequence, or you may want to reset the sequence numbers to the initial number value.

See the mentioned following three options:

```
sequence(<sequence ID>).current

sequence(<sequence ID>).next

sequence(<sequence ID>).reset
```

Although these expressions return integer values, you may also want to get long or string values. This can be done in one of the following ways:

```
sequence(<sequence ID>,long).current

sequence(<sequence ID>,long).next

sequence(<sequence ID>,string).current

sequence(<sequence ID>,string).next
```

Custom CTL Functions

In addition to the prepared CTL functions, you can create your own CTL functions. To do that, you need to write your own code defining the custom CTL functions and specify its plugin.

Each custom CTL function library must be derived/inherited from:

`org.jetel.interpreter.extensions.TLFunctionLibrary` class.

Each custom CTL function must be derived/inhereted from:

`org.jetel.interpreter.extensions.TLFunctionPrototype` class.

These classes have some standard operations defined and several abstract methods which need to be defined so that the custom functions may be used. Within the custom functions code, an existing context must be used or some custom context must be defined. The context serves to store objects when function is to be executed repeatedly, in other words, on more records.

Along with the custom functions code, you also need to define the custom functions plugin. Both the library and the plugin will be used in **CloverETL**. For more information, see the following wiki page: wiki.cloveretl.org/doku.php?id=function_building.

Chapter 54. CTL2

This chapter describes the syntax and the use of CTL2. For detailed information on language reference or built-in functions see:

- Language Reference (p. 667)
- Functions Reference (p. 695)

Example 54.1. Example of CTL2 syntax (Rollup)

```
//#CTL2

string[] customers;
integer Length;


function void initGroup(VoidMetadata groupAccumulator) {
}

function boolean updateGroup(VoidMetadata groupAccumulator) {
     customers = split($0.customers," - ");
     Length = length(customers);

     return true;
}

function boolean finishGroup(VoidMetadata groupAccumulator) {
     return true;
}

function integer updateTransform(integer counter, VoidMetadata groupAccumulator) {
     if (counter >= Length) {
          clear(customers);

          return SKIP;
     }

     $0.customers = customers[counter];
     $0.EmployeeID = $0.EmployeeID;

     return ALL;
}

function integer transform(integer counter, VoidMetadata groupAccumulator) {
     return ALL;
}
```

Language Reference

Clover transformation language (CTL) is used to define transformations in many components. (in all **Joiners**, **DataGenerator**, **Partition**, **DataIntersection**, **Reformat**, **Denormalizer**, **Normalizer**, and **Rollup**)

This section describes the following areas:

- Program Structure (p. 668)
- Comments (p. 668)
- Import (p. 668)
- Data Types in CTL2 (p. 669)
- Literals (p. 672)
- Variables (p. 674)
- Dictionary in CTL2 (p. 675)
- Operators (p. 676)
- Simple Statement and Block of Statements (p. 681)
- Control Statements (p. 681)
- Functions (p. 686)
- Conditional Fail Expression (p. 687)
- Accessing Data Records and Fields (p. 688)
- Mapping (p. 690)
- Parameters (p. 694)

Program Structure

Each program written in CTL must contain the following parts:

```
ImportStatements
VariableDeclarations
FunctionDeclarations
Statements
Mappings
```

All of them may be interspersed, however, there are some principles that are valid for them:

- If an import statement is defined, it must be situated at the beginning of the code.
- Variables and functions must first be declared and only then they can be used.
- Declarations of variables and functions, statements and mappings may also be mutually interspersed.

Important

In CTL2 declaration of variables and functions may be in any place of the transformation code and may be preceded by other code. However, remember that each variable and each function must always be declared before it is used.

This is one of the differences between the two versions of **CloverETL** Transformation Language.

(In CTL1 the order of code parts was fixed and could not be changed.)

Comments

Throughout the program you can use comments. These comments are not processed, they only serve to describe what happens within the program.

The comments are of two types. They can be one-line comments or multiline comments. See the following two options:

```
// This is an one-line comment.

/* This is a multiline comment. */
```

Import

First of all, at the beginning of the program in CTL, you can import some of the existing programs in CTL. The way how you must do it is as follows:

```
import 'fileURL';

import "fileURL";
```

You must decide whether you want to use single or double quotes. Single quotes do not escape so called escape sequences. For more details see Literals (p. 672) below. For these `fileURL`, you must type the URL of some existing source code file.

But remember that you must import such files at the beginning before any other declaration(s) and/or statement(s).

Data Types in CTL2

For basic information about data types used in metadata see Data Types and Record Types (p. 97)

In any program, you can use some variables. Data types in CTL are the following:

boolean

Its declaration look like this: `boolean` *`identifier`*`;`

byte

This data type is an array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

Its declaration looks like this: `byte` *`identifier`*`;`

cbyte

This data type is a compressed array of bytes of a length that can be up to `Integer.MAX_VALUE` as a maximum. It behaves similarly to the list data type (see below).

Its declaration looks like this: `cbyte` *`identifier`*`;`

date

Its declaration look like this: `date` *`identifier`*`;`

decimal

Its declaration looks like this: `decimal` *`identifier`*`;`

By default, any decimal may have up to 32 significant digits. If you want to have different **Length** or **Scale**, you need to set these properties of `decimal` field in metadata.

Example 54.2. Example of usage of decimal data type in CTL2

If you assign 100.0 /3 to a decimal variable, its value might for example be `33.333333333333335701809119200333`. Assigning it to a decimal field (with default **Length** and **Scale**, which are 8 and 2, respectively), it will be converted to `33.33D`.

You can cast any float number to the decimal data type by apending the `d` letter to its end.

integer

Its declaration looks like this: `integer` *`identifier`*`;`

If you apend an `l` letter to the end of any integer number, you can cast it to the long data type

long

Its declaration looks like this: `long` *`identifier`*`;`

Any integer number can be cast to this data type by appending an `l` letter to its end.

number (double)

Its declaration looks like this: `number` *`identifier`*`;`

string

The declaration looks like this: `string` *`identifier`*`;`

list

Each `list` is a container of one the following data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`, `record`.

The list data type is indexed by integers starting from 0.

Its declaration can look like this: `string[]` *`identifier`*`;`

List cannot be created as a list of `lists` or `maps`.

The default list is an empty list.

Examples:

```
integer[] myIntegerList; myIntegerList[5] = 123;

Customer JohnSmith;

Customer PeterBrown;

Customer[] CompanyCustomers;

CompanyCustomers[0] = JohnSmith;

CompanyCustomers[1] = PeterBrown
```

Assignments:

- `myStringList[3] = "abc";`

  It means that the specified string is put to the fourth position in the string list. The other values are fille with `null` as follows:

  `myStringList` is `[null,null,null,"abc"]`

- `myList1 = myList2;`

  It means that both lists reference the same elements.

- `myList1 = myList1 + myList2;`

  It adds all elements of `myList2` to the end of `myList1`.

  Both list must be based on the same primitive data type.

- `myList1 = myList1 + "abc";`

  It adds the `"abc"`string to the `myList1` as its new last element.

  `myList1` must be based on string data type.

- `myList1 = null;`

It destroys the `myList1`.

Be careful when performing list operations (such as append). See Warning (p. 671).

map

This data type is a container of pairs of a key and a value.

Its declaration looks like this: `map[<type of key>, <type of value>]` *`identifier`*`;`

Both the `Key` and the `Value` can be of the following primitive data types: `boolean`, `byte`, `date`, `decimal`, `integer`, `long`, `number`, `string`. `Value` can be also of `record` type.

Map cannot be created as a map of `lists` or other `maps`.

The default map is an empty map.

Examples:

```
map[string, boolean] map1; map1["abc"]=true;

Customer JohnSmith;

Customer PeterBrown;

map[integer, Customer] CompanyCustomersMap;

CompanyCustomersMap[JohnSmith.ID] = JohnSmith;

CompanyCustomersMap[PeterBrown.ID] = PeterBrown
```

The assignments are similar to those valid for a list.

record

This data type is a set of fields of data.

The structure of record is based on metadata. Any metadata item represent a data type.

Declaration of a record looks like this: `<metadata name>` *`identifier`*`;`

Metadata names must be unique in a graph. Different metadata must have different names.

For more detailed information about possible expressions and records usage see Accessing Data Records and Fields (p. 688).

Record does not have a default value.

It can be indexed by both integer numbers and strings (field names). If indexed by numbers, fields are indexed starting from 0.

Warning

Be careful when a record is pushed|appended|inserted (`push()`, `append()`, `insert()` functions) to a list of records within the `transform()` or another function. If the record is declared as a global variable, the last item in the list will always reference the same record. To avoid that, declare your record as a local variable (within `transform()`). Calling `transform()`, a new reference will be created and a correct value will be put to the list.

Literals

Literals serve to write values of any data type.

Table 54.1. Literals

Literal	Description	Declaration syntax	Example
integer	digits representing integer number	[0-9]+	95623
long integer	digits representing integer number with absolute value even greater than 2^{31}, but less than 2^{63}	[0-9]+L?	257L, or 9562307813123123
hexadecimal integer	digits and letters representing integer number in hexadecimal form	0x[0-9A-F]+	0xA7B0
octal integer	digits representing integer number in octal form	0[0-7]*	0644
number (double)	floating point number represented by 64bits in double precision format	[0-9]+.[0-9]+	456.123
decimal	digits representing a decimal number	[0-9]+.[0-9]+D	123.456D
double quoted string	string value/literal enclosed in double quotes; escaped characters [\n,\r,\t, \\, \", \b] get translated into corresponding control chars	"...anything except ["]..."	"hello\tworld\n\r"
single quoted string	string value/literal enclosed in single quotes; only one escaped character [\'] gets translated into corresponding char [']	'...anything except [']...'	'hello\tworld\n\r'
list of literals	list of literals where individual literals can also be other lists/maps/records	[<any literal> (, <any literal>)*]	[10, 'hello', "world", 0x1A, 2008-01-01], [[1 , 2]] , [3 , 4]]
date	date value	this mask is expected: yyyy-MM-dd	2008-01-01
datetime	datetime value	this mask is expected: yyyy-MM-dd HH:mm:ss	2008-01-01 18:55:00

Important

You cannot use any literal for `byte` data type. If you want to write a `byte` value, you must use any of the conversion functions that return `byte` and aply it on an argument value.

For information on these conversion functions see Conversion Functions (p. 697)

Important

Remember that if you need to assign decimal value to a decimal field, you should use decimal literal. Otherwise, such number would not be decimal, it would be a double number!

For example:

1. **Decimal value to a decimal field (correct and accurate)**

```
// correct - assign decimal value to decimal field
myRecord.decimalField = 123.56d;
```

2. **Double value to a decimal field (possibly inaccurate)**

```
// possibly inaccurate - assign double value to decimal field
myRecord.decimalField = 123.56;
```

The latter might produce inaccurate results!

Variables

If you define some variable, you must do it by typing data type of the variable, white space, the name of the variable and semicolon.

Such variable can be initialized later, but it can also be initialized in the declaration itself. Of course, the value of the expression must be of the same data type as the variable.

Both cases of variable declaration and initialization are shown below:

```
dataType variable;

...

variable = expression;

dataType variable = expression;
```

Dictionary in CTL2

If you want to have a dictionary in your graph and acces an entry from CTL2, you must define it in the graph as shown in Chapter 28, Dictionary (p. 198).

To access the entries from CTL2, use the dot syntax as follows:

```
dictionary.<dictionary entry>
```

This expression can be used to

- define the value of the entry:

  ```
  dictionary.customer = "John Smith";
  ```

- get the value of the entry:

  ```
  myCustomer = dictionary.customer;
  ```

- map the value of the entry to an output field:

  ```
  $0.myCustomerField = dictionary.customer;
  ```

- serve as the argument of a function:

  ```
  myCustomerID = isInteger(dictionary.customer);
  ```

Operators

The operators serve to create more complicated expressions within the program. They can be arithmetic, relational and logical. The relational and logical operators serve to create expressions with resulting boolean value. The arithmetic operators can be used in all expressions, not only the logical ones.

All operators can be grouped into three categories:

- Arithmetic Operators (p. 676)
- Relational Operators (p. 678)
- Logical Operators (p. 680)

Arithmetic Operators

The following operators serve to put together values of different expressions (except those of boolean values). These signs can be used more times in one expression. In such a case, you can express priority of operations by parentheses. The result depends on the order of the expressions.

- Addition

+

The operator above serves to sum the values of two expressions.

But the addition of two boolean values or two date data types is not possible. To create a new value from two boolean values, you must use logical operators instead.

Nevertheless, if you want to add any data type to a string, the second data type is converted to a string automatically and it is concatenated with the first (string) summand. But remember that the string must be on the first place! Naturally, two strings can be summed in the same way. Note also that the `concat()` function is faster and you should use this function instead of adding any summand to a string.

You can also add any numeric data type to a date. The result is a date in which the number of days is increased by the whole part of the number. Again, here is also necessary to have the date on the first place.

The sum of two numeric data types depends on the order of the data types. The resulting data type is the same as that of the first summand. The second summand is converted to the first data type automatically.

- Subtraction and Unitary minus

-

The operator serves to subtract one numeric data type from another. Again the resulting data type is the same as that of the minuend. The subtrahend is converted to the minuend data type automatically.

But it can also serve to subtract numeric data type from a date data type. The result is a date in which the number of days is reduced by the whole part of the subtrahend.

- Multiplication

*

The operator serves only to multiplicate two numeric data types.

Remember that during multiplication the first multiplicand determines the resulting data type of the operation. If the first multiplicand is an integer number and the second is a decimal, the result will be an integer number. On the other hand, if the first multiplicand is a decimal and the second is an integer number, the result will be of decimal data type. In other words, order of multiplicands is of importance.

- Division

`/`

The operator serves only to divide two numeric data types. Remember that you must not divide by zero. Dividing by zero throws `TransformLangExecutorRuntimeException` or gives `Infinity` (in case of a number data type)

Remember that during division the numerator determines the resulting data type of the operation. If the nominator is an integer number and the denominator is a decimal, the result will be an integer number. On the other hand, if the nominator is a decimal and the denominator is an integer number, the result will be of decimal data type. In other words, data types of nominator and denominator are of importance.

- Modulus

`%`

The operator can be used for both floating-point data types and integer data types. It returns the remainder of division.

- Incrementing

`++`

The operator serves to increment numeric data type by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is incremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is incremented.

Important

Remember that the incrementing operator cannot be applied on literals, record fields, map, or list values of integer data type.

It can only be used with integer variables.

- Decrementing

`--`

The operator serves to decrement numeric data type by one. The operator can be used for both floating-point data types and integer data types.

If it is used as a prefix, the number is decremented first and then it is used in the expression.

If it is used as a postfix, first, the number is used in the expression and then it is decremented.

Important

Remember that the decrementing operator cannot be applied on literals, record fields, map, or list values of integer data type.

It can only be used with integer variables.

Relational Operators

The following operators serve to compare some subexpressions when you want to obtain a boolean value result. Each of the mentioned signs can be used. If you choose the `.operator.` signs, they must be surrounded by white spaces. These signs can be used more times in one expression. In such a case you can express priority of comparisons by parentheses.

- Greater than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`>`

`.gt.`

- Greater than or equal to

  Each of the three signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`>=`

`=>`

`.ge.`

- Less than

  Each of the two signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`<`

`.lt.`

- Less than or equal to

  Each of the three signs below can be used to compare expressions consisting of numeric, date and string data types. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`<=`

`=<`

`.le.`

- Equal to

  Each of the two signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`==`

`.eq.`

- Not equal to

  Each of the three signs below can be used to compare expressions of any data type. Both data types in the expressions must be comparable. The result can depend on the order of the two expressions if they are of different data type.

`!=`

`<>`

`.ne.`

- Matches regular expression

  The operator serves to compare string and some regular expression. The regular expression can look like this (for example): `"[^a-d].*"` It means that any character (it is expressed by the dot) except `a`, `b`, `c`, `d` (exception is expressed by the `^` sign) (`a-d` means - characters from `a` to `d`) can be contained zero or more times (expressed by `*`). Or, `'[p-s]{5}'` means that `p`, `r`, `s` must be contained exactly five times in the string. For more detailed explanation about how to use regular expressions see `java.util.regex.Pattern`.

`~=`

`.regex.`

- Contains regular expression

  The operator serves to compare string and some regular expression. It returns `true` if string contains regular expression, otherwise returns `false`.

  For more detailed explanation about how to use regular expressions see `java.util.regex.Pattern`.

`?=`

- Contained in

  This operator serves to specify whether some value is contained in the list or in the map of other values.

  The operator has double syntax. See following examples:

```
boolean myBool;
string myString;
string[] = myList;
...
myBool = myString.in(myList);
```

```
boolean myBool;
string myString;
string[] = myList;
...
myBool = in(myString,myList);
```

Logical Operators

If the expression whose value must be of boolean data type is complicated, it can consist of some subexpressions (see above) that are put together by logical conjunctions (AND, OR, NOT, .EQUAL TO, NOT EQUAL TO). If you want to express priority in such an expression, you can use parentheses. From the conjunctions mentioned below you can choose either form (for example, `&&` or `and`, etc.).

Every sign of the form `.operator.` must be surrounded by white space.

- Logical AND

`&&`

`and`

- Logical OR

`||`

`or`

- Logical NOT

`!`

`not`

- Logical EQUAL TO

`==`

`.eq.`

- Logical NOT EQUAL TO

`!=`

`<>`

`.ne.`

Simple Statement and Block of Statements

All statements can be divided into two groups:

- **Simple statement** is an expression terminated by semicolon.

  For example:

```
integer MyVariable;
```

- **Block of statements** is a series of simple statements (each of them is terminated by semicolon). The statements in a block can follow each other in one line or they can be written in more lines. They are surrounded by curled braces. No semicolon is used after the closing curled brace.

  For example:

```
while (MyInteger<100) {
    Sum = Sum + MyInteger;
    MyInteger++;
}
```

Control Statements

Some statements serve to control the process of the program.

All control statements can be grouped into the following categories:

- Conditional Statements (p. 681)
- Iteration Statements (p. 682)
- Jump Statements (p. 683)

Conditional Statements

These statements serve to branch out the process of the program.

If Statement

On the basis of the `Condition` value this statement decides whether the `Statement` should be executed. If the `Condition` is true, `Statement` is executed. If it is false, the `Statement` is ignored and process continues next after the `if` statement. `Statement` is either simple statement or a block of statements

```
if (Condition) Statement
```

Unlike the previous version of the `if` statement (in which the `Statement`is executed only if the `Condition` is true), other `Statements` that should be executed even if the `Condition` value is false can be added to the `if` statement. Thus, if the `Condition` is true, `Statement1` is executed, if it is false, `Statement2` is executed. See below:

```
if (Condition) Statement1 else Statement2
```

The `Statement2` can even be another `if` statement and also with `else` branch:

```
if (Condition1) Statement1
    else if (Condition2) Statement3
         else Statement4
```

Switch Statement

Sometimes you would have very complicated statement if you created the statement of more branched out `if` statement. In such a case, much more better is to use the `switch` statement.

Now, instead of the `Condition` as in the `if` statement with only two values (true or false), an `Expression` is evaluated and its value is compared with the `Constants` specified in the `switch` statement.

Only the `Constant` that equals to the value of the `Expression` decides which of the `Statements` is executed.

If the `Expression` value is `Constant1`, the `Statement1` will be executed, etc.

Important

Remember that literals must be unique in the `Switch statement`.

```
switch (Expression) {
    case Constant1 : Statement1 StatementA [break;]
    case Constant2 : Statement2 StatementB [break;]
    ...
    case ConstantN : StatementN StatementW [break;]
}
```

The optional `break;` statements ensure that only the statements correspoding to a constant will be executed. Otherwise, all below them would be executed as well.

In the following case, even if the value of the `Expression` does not equal to the values of the `Constant1,...,ConstantN`, the default statement (`StatementN+1`) is executed.

```
switch (Expression) {
    case Constant1 : Statement1 StatementA [break;]
    case Constant2 : Statement2 StatementB [break;]
    ...
    case ConstantN : StatementN StatementW [break;]
    default : StatementN+1 StatementZ
}
```

Iteration Statements

These iteration statements repeat some processes during which some inner `Statements` are executed cyclically until the `Condition` that limits the execution cycle becomes false or they are executed for all values of the same data type.

For Loop

First, the Initialization is set up, after that, the `Condition` is evaluated and if its value is true, the `Statement` is executed and finally the `Iteration` is made.

During the next cycle of the loop, the `Condition` is evaluated again and if it is true, `Statement` is executed and `Iteration` is made. This way the process repeats until the `Condition` becomes false. Then the loop is terminated and the process continues with the other part of the program.

If the `Condition` is false at the beginning, the process jumps over the `Statement` out of the loop.

```
for (Initialization;Condition;Iteration)
    Statement
```

Important

Remember that the `Initialization` part of the `For Loop` may also contain the declaration of the variable that is used in the loop.

`Initialization`, `Condition`, and `Iteration` are optional.

Do-While Loop

First, the `Statement` is executed, then the process depends on the value of `Condition`. If its value is true, the `Statement` is executed again and then the `Condition` is evaluated again and the subprocess either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the end of the loop, even if it is false at the beginning of the subprocess, the `Statement` is executed at least once.

```
do Statement while (Condition)
```

While Loop

This process depends on the value of `Condition`. If its value is true, the `Statements` is executed and then the `Condition` is evaluated again and the subprocess either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false). Since the `Condition` is at the start of the loop, if it is false at the beginning of the subprocess, the `Statements` is not executed at all and the loop is jumped over.

```
while (Condition) Statement
```

For-Each Loop

The `foreach` statement is executed on all fields of the same data type within a container. Its syntax is as follows:

```
foreach (<data type> myVariable : iterableVariable) Statement
```

All elements of the same data type (data type is declared in this statement) are searched in the `iterableVariable` container. The `iterableVariable` can be a list, a map, or a record. For each variable of the same data type, specified `Statement` is executed. It can be either a simple statement or a block of statements.

Thus, for example, the same `Statement` can be executed for all `string` fields of a record, etc.

Jump Statements

Sometimes you need to control the process in a different way than by decision based on the `Condition` value. To do that, you have the following options:

Break Statement

If you want to stop some subprocess, you can use the following statement in the program:

```
break;
```

The subprocess breaks and the process jumps to the higher level or to the next `Statements`.

Continue Statement

If you want to stop some iteration subprocess, you can use the following statement in the program:

```
continue;
```

The subprocess breaks and the process jumps to the next iteration step.

Return Statement

In the functions you can use the `return` word either alone or along with an `expression`. (See the following two options below.) The `return` statement can be in any place within the function. There may also be multiple `return` statements among which a specific one is executed depending on a condition, etc.

```
return;

return expression;
```

Error Handling

CTL2 also provides a simple mechanism for catching and handling possible errors.

However, CTL2 differs from CTL1 as regards handling errors. It does not use the `try-catch` statement.

It only uses a set of optional `OnError()` functions that exist to each required transformation function.

For example, for required functions (e.g., `append()`, `transform()`, etc.), there exist following optional functions:

`appendOnError()`, `transformOnError()`, etc.

Each of these required functions may have its (optional) couterpart whose name differs from the original (required) by adding the `OnError` suffix.

Moreover, every `<required ctl template function>OnError()` function returns the same values as the original required function.

This way, any exception that is thrown by the original required function causes call of its `<required ctl template function>OnError()` counterpart (e.g., `transform()` fail may call `transformOnError()`, etc.).

In this `transformOnError()`, any incorrect code can be fixed, error message can be printed to Console, etc.

Important

Remember that these `OnError()` functions are not called when the original required functions return **Error codes** (values less then -1)!

If you want that some `OnError()` function is called, you need to use a `raiseError(string arg)` function. Or (as has already been said) also any exception thrown by original required function calls its `OnError()` counterpart.

Functions

You can define your own functions in the following way:

```
function returnType functionName (type1 arg1, type2 arg2,..., typeN argN) {
    variableDeclarations
    otherFunctionDeclarations
    Statements
    Mappings
    return [expression];
}
```

You must put the return statement at the end. For more information about the return statement see Return Statement (p. 683). Inside some functions, there can be `Mappings`. These may be in any place inside the function.

In addition to any other data type mentioned above, the function can also return `void`.

Message Function

Since **CloverETL** version 2.8.0, you can also define a function for your own error messages.

```
function string getMessage() {
         return message;
}
```

This `message` variable should be declared as a global string variable and defined anywhere in the code so as to be used in the place where the `getMessage()` function is located. The `message` will be written to console.

Conditional Fail Expression

You can also use conditional fail expressions.

They look like this:

```
expression1 : expression2 : expression3 : ... : expressionN;
```

This conditional fail expression may be used for mapping, assignment to a variable, and as an argument of a function too.

The expressions are evaluated one by one, starting from the first expression and going from left to right.

1. As soon as one of these expressions may be successfully assigned to a variable, mapped to an output field, or used as the argument of the function, it is used and the other expressions are not evaluated.

2. If none of these expressions may be used (assigned to a variable, mapped to the output field, or used as an argument), graph fails.

Important

Remember that in CTL2 this expression may be used in multiple ways: for assigning to a variable, mapping to an output field, or as an argument of the function.

(In CTL1 it was only used for mapping to an output field.)

Remember also that this expression can only be used in interpreted mode of CTL2.

Accessing Data Records and Fields

This section describes the way how the record fields should be worked with. As you know, each component may have ports. Both input and output ports are numbered starting from 0.

Metadata of connected edges must be identified by their names. Different metadata must have different names.

Working with Records and Variables

Now we suppose that `Customers` is the ID of metadata, their name is `customers`, and their third field (field 2) is `firstname`.

Following expressions represent the value of the third field (field 2) of the specified metadata:

- `$<port number>.<field number>`

  Example: `$0.2`

  `$0.*` means all fields on the first port (port 0).

- `$<port number>.<field name>`

  Example: `$0.firstname`

- `$<metadata name>.<field number>`

  Example: `$customers.2`

  `$customers.*` means all fields on the first port (port 0).

- `$<metadata name>.<field name>`

  Example: `$customers.firstname`

You can also define records in CTL code. Such defitions can look like these:

- `<metadata name> MyCTLRecord;`

  Example: `customers myCustomers;`

- After that, you can use the following expressions:

  `<record variable name>.<field name>`

  Example: `myCustomers.firstname;`

Mapping of records to variables looks like this:

- `myVariable = $<port number>.<field number>;`

  Example: `FirstName = $0.2;`

- `myVariable = $<port number>.<field name>;`

  Example: `FirstName = $0.firstname;`

- `myVariable = $<metadata name>.<field number>;`

  Example: `FirstName = $customers.2;`

- `myVariable = $<metadata name>.<field name>;`

Example: `FirstName = $customers.firstname;`

- `myVariable = <record variable name>.<field name>;`

  Example: `FirstName = myCustomers.firstname;`

Mapping of variables to records can look like this:

- `$<port number>.<field number> = myVariable;`

  Example: `$0.2 = FirstName;`

- `$<port number>.<field name> = myVariable;`

  Example: `$0.firstname = FirstName;`

- `$<metadata name>.<field number> = myVariable;`

  Example: `$customers.2 = FirstName;`

- `$<metadata name>.<field name> = myVariable;`

  Example: `$customers.firstname = FirstName;`

- `<record variable name>.<field name> = myVariable;`

  Example: `myCustomers.firstname = FirstName;`

Important

Remember that if component has single input port or single output port, you can use the syntax as follows:

```
$firstname
```

Generally, the syntax is:

```
$<field name>
```

Important

You can assign input to an internal CTL record using following syntax:

```
MyCTLRecord.* = $0.*;
```

Also, you can map values of an internal record to the output using following syntax:

```
$0.* = MyCTLRecord.*;
```

Mapping

Mapping is a part of each transformation defined in some of the **CloverETL** components.

Calculated or generated values or values of input fields are assigned (mapped) to output fields.

1. Mapping assigns a value to an output field.
2. Mapping operator is the following:

   =

3. Mapping must always be defined inside a function.
4. Mapping may be defined in any place inside a function.

Important

In CTL2 mapping may be in any place of the transformation code and may be followed by any code. This is one of the differences between the two versions of **CloverETL** Transformation Language.

(In CTL1 mapping had to be at the end of the function and could only be followed by one `return` statement.)

In CTL2 mapping operator is simply the equal sign.

5. Remember that you can also wrap a mapping in a user-defined function which would be subsequently used inside another function.
6. You can also map different input metadata to different output metadata by field names or by field positions. See examples below.

Mapping of Different Metadata (by Name)

When you map input to output like this:

```
$0.* = $0.*;
```

input metadata may even differ from those on the output.

In the expression above, fields of the input are mapped to the fields of the output that have the same name and type as those of the input. The order in which they are contained in respective metadata and the number of all fields in either metadata is of no importance.

When you have input metadata in which the first two fields are `firstname` and `lastname`, each of these two fields is mapped to its counterpart on the output. Such output `firstname` field may even be the fifth and `lastname` field be the third, but those two fields of the input will be mapped to these two output fields .

Even if input metadata had more fields and output metadata had more fields, such fields would not be mapped to each other if there did not exist an output field with the same name as one of the input (independently on the mutual position of the fields in corresponding metadata).

In addition to the simple mapping as shown above (`$0.* = $0.*;`) you can also use the following function:

```
void copyByName(record to, record from);
```

Example 54.3. Mapping of Metadata by Name (using the copyByName() function)

```
recordName2 myOutputRecord;
copyByName(myOutputRecord.*,$0.*);
$0.* = myOutputRecord.*;
```

Important

Metadata fields are mapped from input to output by name and data type independently on their order and on the number of all fields!

Following syntax may also be used: `myOutputRecord.copyByName($0.*);`

Mapping of Different Metadata (by Position)

Sometimes you need to map input to ouput, but names of input fields are different from those of output fields. In such a case, you can map input to output by position.

To achieve this, you *must* to use the following function:

```
void copyByPosition(record to, record from);
```

Example 54.4. Mapping of Metadata by Position

```
recordName2 myOutputRecord;
copyByPosition(myOutputRecord,$0.*);
$0.* = myOutputRecord.*;
```

Important

Metadata fields may be mapped from input to output by position (as shown in the example above)!

Following syntax may also be used: `myOutputRecord.copyByPosition($0.*);`

Use Case 1 - One String Field to Upper Case

To show in more details how mapping works, we provide here a few examples of mappings.

We have a graph with a **Reformat** component. Metadata on its input and output are identical. First two fields (`field1` and `field2`) are of string data type, the third (`field3`) is of integer data type.

1. We want to change the letters of `field1` values to upper case while passing the other two fields unchanged to the output.

2. We also want to distribute records according to the value of `field3`. Those records in which the value of `field3` is less than 5 should be sent to the output port 0, the others to the output port 1.

Examples of Mapping

As the first possibility, we have the mapping for both ports and all fields defined inside the `transform()` function of CTL template.

Example 54.5. Example of Mapping with Individual Fields

Note that the mappings will be performed for all records. In other words, even when the record will go to the output port 1, also the mapping for output port 0 will be performed, and vice versa.

Moreover, mapping consists of individual fields, which may be complicated in case there are many fields in a record. In the next examples, we will see how this can be solved in a better way.

```
function integer transform() {

     // mapping input port records to output port records
     // each field is mapped separately
     $0.field1 = upperCase($0.field1);
     $0.field2 = $0.field2;
     $0.field3 = $0.field3;
     $1.field1 = upperCase($0.field1);
     $1.field2 = $0.field2;
     $1.field3 = $0.field3;

     // output port number returned
     if ($0.field3 < 5) return 0; else return 1;
}
```

Note

As CTL2 allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL2 mapping may be in any place of the transformation code and may be followed by any code!

As the second possibility, we also have the mapping for both ports and all fields defined inside the `transform()` function of CTL template. But now there are wild cards used in the mapping. These passes the records unchanged to the outputs and after this wildcard mapping the fields that should be changed are specified.

Example 54.6. Example of Mapping with Wild Cards

Note that mappings will be performed for all records. In other words, even when the record will go to the output port 1, also the mapping for output port 0 will be performed, and vice versa.

However, now the mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed *below* the mapping with wild cards.

This is useful when there are many unchanged fields and a few that will be changed.

```
function integer transform() {

     // mapping input port records to output port records
     // wild cards for mapping unchanged records
     // transformed records mapped aditionally
     $0.* = $0.*;
     $0.field1 = upperCase($0.field1);
     $1.* = $0.*;
     $1.field1 = upperCase($0.field1);

     // return the number of output port
     if ($0.field3 < 5) return 0; else return 1;
}
```

Note

As CTL2 allows to use any code *after* the mapping, here we have used the `if` statement with two `return` statements after the mapping.

In CTL2 mapping may be in any place of the transformation code and may be followed by any code!

As the third possibility, we have the mapping for both ports and all fields defined outside the `transform()` function of CTL template. Each output port has its own mapping.

Also here, wild cards are used.

The mapping that is defined in separate function for each output port allows the following improvements:

- Mapping is performed only for respective output port! In other words, now there is no need to map record to the port 1 when it will go to the port 0, and vice versa.

Example 54.7. Example of Mapping with Wild Cards in Separate User-Defined Functions

Moreover, mapping uses wild cards at first, which passes the records unchanged to the output, but the first field is changed below the mapping with wild card. This is of use when there are many unchanged fields and a few that will be changed.

```
// mapping input port records to output port records
// inside separate functions
// wild cards for mapping unchanged records
// transformed records mapped aditionally
function void mapToPort0 () {
     $0.* = $0.*;
     $0.field1 = upperCase($0.field1);
}

function void mapToPort1 () {
     $1.* = $0.*;
     $1.field1 = upperCase($0.field1);
}

// use mapping functions for all ports in the if statement
function integer transform() {
     if ($0.field3 < 5) {
          mapToPort0();
          return 0;
     }
     else {
          mapToPort1();
          return 1;
     }
}
```

Parameters

The parameters can be used in Clover transformation language in the following way: `${nameOfTheParameter}`. If you want such a parameter is considered a string data type, you must surround it by single or double quotes like this: `'${nameOfTheParameter}'` or `"${nameOfTheParameter}"`.

Important

1. Remember that escape sequences are always resolved as soon as they are assigned to parameters. For this reason, if you want that they are not resolved, type double backslashes in these strings instead of single ones.

2. Remember also that you can get the values of environment variables using parameters. To learn how to do it, see Environment Variables (p. 193).

Functions Reference

Clover transformation language has at its disposal a set of functions you can use. We describe them here.

All functions can be grouped into following categories:

- Conversion Functions (p. 697)
- Container Functions (p. 720)
- Date Functions (p. 705)
- Mathematical Functions (p. 707)
- String Functions (p. 711)
- Miscellaneous Functions (p. 722)
- Lookup Table Functions (p. 724)
- Sequence Functions (p. 727)
- Custom CTL Functions (p. 728)

Important

Remember that with CTL2 you can use both **CloverETL** built-in functions and your own functions in one of the ways listed below.

Built-in functions

- `substring(upperCase(getAplhanumericChars($0.field1))1,3)`
- `$0.field1.getAlphanumericChars().upperCase().substring(1,3)`

The two expressions above are equivalent. The second option with the first argument preceding the function itself is sometimes referred to as **object notation**. Do not forget to use the "$port.field.function()" syntax. Thus, `arg.substring(1,3)` is equal to `substring(arg,1,3).`

You can also declare your own function with a set of arguments of any data type, e.g.:

```
function integer myFunction(integer arg1, string arg2, boolean arg3) {
<function body>
}
```

User-defined functions

- `myFunction($0.integerField,$0.stringField,$0.booleanField)`
- `$0.integerField.myFunction($0.stringField,$0.booleanField)`

Warning

Remember that the object notation (<first argument>.function(<other arguments>) cannot be used in **Miscellaneous** functions. See Miscellaneous Functions (p. 722).

Important

Remember that if you set the **Null value** property in metadata for any `string` data field to any non-empty string, any function that accept `string` data field as an argument and throws NPE when applied on `null` (e.g., `length()`), it will throw NPE when applied on such specific string.

For example, if `field1` has **Null value** property set to `"<null>"`, `length($0.field1)` will fail on the records in which the value of `field1` is `"<null>"` and it will be 0 for empty field.

See Null value (p. 145) for detailed information.

Conversion Functions

Sometimes you need to convert values from one data type to another.

In the functions that convert one data type to another, sometimes a format pattern of a date or any number must be defined. Also locale can have an influence to their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 106).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of these functions:

- `byte` **`base64byte`**`(string` *`arg`*`);`

  The `base64byte(string)` function takes one string argument in `base64` representation and converts it to an array of bytes. Its counterpart is the `byte2base64(byte)` function.

- `string` **`bits2str`**`(byte` *`arg`*`);`

  The `bits2str(byte)` function takes an array of bytes and converts it to a string consisting of two characters: `"0"` or `"1"`. Each byte is represented by eight characters ("0" or "1"). For each byte, the lowest bit is at the beginning of these eight characters. The counterpart is the `str2bits(string)` function.

- `integer` **`bool2num`**`(boolean` *`arg`*`);`

  The `bool2num(boolean)` function takes one boolean argument and converts it to either integer `1` (if the argument is true) or integer `0` (if the argument is false). Its counterpart is the `num2bool(<numeric type>)` function.

- `string` **`byte2base64`**`(byte` *`arg`*`);`

  The `byte2base64(byte)` function takes an array of bytes and converts it to a string in `base64` representation. Its counterpart is the `base64byte(string)` function.

- `string` **`byte2hex`**`(byte` *`arg`*`);`

  The `byte2hex(byte)` function takes an array of bytes and converts it to a string in `hexadecimal` representation. Its counterpart is the `hex2byte(string)` function.

- `long` **`date2long`**`(date` *`arg`*`);`

  The `date2long(date)` function takes one date argument and converts it to a long type. Its value is equal to the number of milliseconds elapsed from `January 1, 1970, 00:00:00 GMT` to the date specified as the argument. Its counterpart is the `long2date(long)` function.

- `integer` **`date2num`**`(date` *`arg`*`, unit` *`timeunit`*`);`

  The `date2num(date, unit)` function accepts two arguments: the first is date and the other is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function

takes these two arguments and converts them to an integer using system locale. If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`. Remember that months are numbered starting from `1` unlike in CTL1. Thus, `date2num(2008-06-12, month)` returns `6`. And `date2num(2008-06-12, hour)` returns `0`.

- `integer` **`date2num`**`(date` *`arg`*`, unit` *`timeunit`*`, string` *`locale`*`);`

  The `date2num(date, unit, string)` function accepts three arguments: the first is date, the second is any time unit, the third is a locale. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes these two arguments and converts them to an integer using the specified locale. If the time unit is contained in the date, it is returned as an integer number. If it is not contained, the function returns `0`. Remember that months are numbered starting from `1` unlike in CTL1. Thus, `date2num(2008-06-12, month)` returns `6`. And `date2num(2008-06-12, hour)` returns `0`.

- `string` **`date2str`**`(date` *`arg`*`, string` *`pattern`*`);`

  The `date2str(date, string)` function accepts two arguments: date and string. The function takes them and converts the date according to the `pattern` specified as the second argument. Thus, `date2str(2008-06-12, "dd.MM.yyyy")` returns the following string: `"12.6.2008"`. Its counterpart is the `str2date(string, string)` function.

- `string` **`date2str`**`(date` *`arg`*`, string` *`pattern`*`, string` *`locale`*`);`

  Converts the `date` field type into a date of the `string` data type according to the `pattern` (describing the date and time format) and `locale` (defining what date format symbols should be used). Thus, `date2str(2009/01/04,"yyyy-MMM-d","fr.CA")` returns `2009-janv.-4`. See Locale (p. 112) for more info about locale settings.

- `number` **`decimal2double`**`(decimal` *`arg`*`);`

  The `decimal2double(decimal)` function takes one argument of decimal data type and converts it to a double value.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into a `double` (as the ranges of `double` data type do not cover all `decimal` values), the function throws exception. Thus, `decimal2double(92378352147483647.23)` returns `9.2378352147483648E16`.

  On the other hand, any `double` can be converted into a `decimal`. Both **Length** and **Scale** of a decimal can be adjusted for it.

- `integer` **`decimal2integer`**`(decimal` *`arg`*`);`

  The `decimal2integer(decimal)` function takes one argument of decimal data type and converts it to an integer.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into an `integer` (as the range of `integer` data type does not cover the range of `decimal` values), the function throws exception. Thus, `decimal2integer(352147483647.23)` throws exception, whereas `decimal2integer(25.95)` returns `25`.

  On the other hand, any `integer` can be converted into a `decimal` without loss of precision. **Length** of a decimal can be adjusted for it.

- `long` **`decimal2long`**`(decimal` *`arg`*`);`

  The `decimal2long(decimal)` function takes one argument of decimal data type and converts it to a long value.

  The conversion is narrowing. And, if a `decimal` value cannot be converted into a `long` (as the range of `long` data type does not cover all `decimal` values), the function throws

exception. Thus, `decimal2long(9759223372036854775807.25)` throws exception, whereas `decimal2long(72036854775807.79)` returns `72036854775807`.

On the other hand, any `long` can be converted into a `decimal` without loss of precision. **Length** of a decimal can be adjusted for it.

- `integer` **`double2integer`**`(number` *`arg`*`);`

  The `double2integer(number)` function takes one argument of double data type and converts it to an integer.

  The conversion is narrowing. And, if a `double` value cannot be converted into an `integer` (as the range of `double` data type does not cover all `integer` values), the function throws exception. Thus, `double2integer(352147483647.1)` throws exception, whereas `double2integer(25.757197)` returns `25`.

  On the other hand, any `integer` can be converted into a `double` without loss of precision.

- `long` **`double2long`**`(number` *`arg`*`);`

  The `double2long(number)` function takes one argument of double data type and converts it to a long.

  The conversion is narrowing. And, if a `double` value cannot be converted into a `long` (as the range of `double` data type does not cover all `long` values), the function throws exception. Thus, `double2long(1.3759739E23)` throws exception, whereas `double2long(25.8579)` returns `25`.

  On the other hand, any `long` can always be converted into a `double`, however, user should take into account that loss of precision may occur.

- `string` **`getFieldName`**`(record` *`argRecord`*`, integer` *`index`*`);`

  The `getFieldName(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the name of the field with the specified index. Fields are numbered starting from 0.

  ### Important

  The `argRecord` may have any of the following forms:

  - `$<port number>.*`

    E.g., `$0.*`

  - `$<metadata name>.*`

    E.g., `$customers.*`

  - `<record variable name>[.*]`

    E.g., `Customers` or `Customers.*` (both cases, if `Customers` was declared as record in CTL.)

  - `lookup(<lookup table name>).get(<key value>)[.*]`

    E.g., `lookup(Comp).get("JohnSmith")` or `lookup(Comp).get("JohnSmith").*`

  - `lookup(<lookup table name>).next()[.*]`

    E.g., `lookup(Comp).next()` or `lookup(Comp).next().*`

- `string` **`getFieldType`**`(record` *`argRecord`*`, integer` *`index`*`);`

The `getFieldType(record, integer)` function accepts two arguments: record and integer. The function takes them and returns the type of the field with the specified index. Fields are numbered starting from 0.

Important

Records as arguments look like the records for the `getFieldName()` function. See above.

- `byte` **`hex2byte`**`(string` *`arg`*`);`

  The `hex2byte(string)` function takes one string argument in `hexadecimal` representation and converts it to an array of bytes. Its counterpart is the `byte2hex(byte)` function.

- `string` **`json2xml`**`(string` *`arg`*`);`

  The `json2xml(string)` function takes one string argument that is `JSON` formatted and converts it to an `XML` formatted string. Its counterpart is the `xml2json(string)` function.

- `date` **`long2date`**`(long` *`arg`*`);`

  The `long2date(long)` function takes one long argument and converts it to a date. It adds the argument number of milliseconds to `January 1, 1970, 00:00:00 GMT` and returns the result as a date. Its counterpart is the `date2long(date)` function.

- `integer` **`long2integer`**`(long` *`arg`*`);`

  The `long2integer(decimal)` function takes one argument of long data type and converts it to an integer value. The conversion is successful only if it is possible without any loss of information, otherwise the function throws exception. Thus, `long2integer(352147483647)` throws exception, whereas `long2integer(25)` returns `25`.

  On the other hand, any `integer` value can be converted into a `long` number without loss of precision.

- `byte` **`long2packDecimal`**`(long` *`arg`*`);`

  The `long2packDecimal(long)` function takes one argument of long data type and returns its value in the representation of packed decimal number. It is the counterpart of the `packDecimal2long(byte)` function.

- `byte` **`md5`**`(byte` *`arg`*`);`

  The `md5(byte)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its `MD5` hash value.

- `byte` **`md5`**`(string` *`arg`*`);`

  The `md5(string)` function accepts one argument of string data type. It takes this argument and calculates its `MD5` hash value.

- `boolean` **`num2bool`**`(<numeric type>` *`arg`*`);`

  The `num2bool(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns boolean `false` for 0 and `true` for any other value.

- `string` **`num2str`**`(<numeric type>` *`arg`*`);`

  The `num2str(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and converts it to a string in decimal representation. Locale is system value. Thus, `num2str(20.52)` returns `"20.52"`.

- `string` **`num2str`**`(<numeric type>` *`arg`*`, integer` *`radix`*`);`

  The `num2str(<numeric type>,integer)` function accepts two arguments: the first is of any of three numeric data types (`integer`, `long`, `number`) and the second is integer. It takes these two arguments and

converts the first to its string representation in the `radix` based numeric system. Thus, `num2str(31, 16)` returns `"1F"`. Locale is system value.

For both `integer` and `long` data types, any integer number can be used as radix. For double (`number`) only 10 or 16 can be used as radix.

- `string` **`num2str`**`(<numeric type>` *`arg`*`, string` *`format`*`);`

  The `num2str(<numeric type>, string)` function accepts two arguments: the first is of any numeric data type (`integer`, `long`, `number`, or `decimal`) and the second is string. It takes these two arguments and converts the first to a string in decimal representation using the format specified as the second argument. Locale has system value.

- `string` **`num2str`**`(<numeric type>` *`arg`*`, string` *`format`*`, string` *`locale`*`);`

  The `num2str(<numeric type>, string, string)` function accepts three arguments: the first is of any numeric data type (`integer`, `long`, `number`, or `decimal`) and two are strings. It takes these arguments and converts the first to its string representation using the format specified as the second argument and the locale specified as the third argument.

- `long` **`packDecimal2long`**`(byte` *`arg`*`);`

  The `packDecimal2long(byte)` function takes one argument of an array of bytes whose meaning is the packed decimal representation of a long number. It returns its value as long data type. It is the counterpart of the `long2packDecimal(long)` function.

- `byte` **`sha`**`(byte` *`arg`*`);`

  The `sha(byte)` function accepts one argument consisting of an array of bytes. It takes this argument and calculates its SHA hash value.

- `byte` **`sha`**`(string` *`arg`*`);`

  The `sha(string)` function accepts one argument of string data type. It takes this argument and calculates its SHA hash value.

- `byte` **`str2bits`**`(string` *`arg`*`);`

  The `str2bits(string)` function takes one string argument and converts it to an array of bytes. Its counterpart is the `bits2str(byte)` function. The string consists of the following characters: Each of them can be either `"1"` or it can be any other character. In the string, each character `"1"` is converted to the bit `1`, all other characters (not only `"0"`, but also `"a"`, `"z"`, `"/"`, etc.) are converted to the bit `0`. If the number of characters in the string is not an integral multiple of eight, the string is completed by "0" characters from the right. Then, the string is converted to an array of bytes as if the number of its characters were integral multiple of eight.

  The first character represents the lowest bit.

- `boolean` **`str2bool`**`(string` *`arg`*`);`

  The `str2bool(string)` function takes one string argument and converts it to the corresponding boolean value. The string can be one of the following: `"TRUE"`, `"true"`, `"T"`, `"t"`, `"YES"`, `"yes"`, `"Y"`, `"y"`, `"1"`, `"FALSE"`, `"false"`, `"F"`, `"f"`, `"NO"`, `"no"`, `"N"`, `"n"`, `"0"`. The strings are converted to boolean `true` or boolean `false`.

- `date` **`str2date`**`(string` *`arg`*`, string` *`pattern`*`);`

  The `str2date(string, string)` function accepts two string arguments. It takes them and converts the first string to the date according to the `pattern` specified as the second argument. The `pattern` must correspond to the structure of the first argument. Thus, `str2date("12.6.2008", "dd.MM.yyyy")` returns the following date: `2008-06-12`.

- `date` **`str2date`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`);`

  The `str2date(string, string, string)` function accepts three string arguments and one boolean. It takes the arguments and converts the first string to the date according to the `pattern` and `locale` specified as the second and the third argument, respectively. The `pattern` must correspond to the structure of the

first argument. Thus, `str2date("12.6.2008", "dd.MM.yyyy",cs.CZ)` returns the following date: `2008-06-12` . The third argument defines the locale for the date.

- `decimal` **`str2decimal`**`(string` *`arg`*`);`

  The `str2decimal(string)` function takes one string argument and converts it to the corresponding decimal value.

- `decimal` **`str2decimal`**`(string` *`arg`*`, string` *`format`*`);`

  The `str2decimal(string, string)` function takes the first string argument and converts it to the corresponding decimal value according to the format specified as the second argument. Locale has system value.

- `decimal` **`str2decimal`**`(string` *`arg`*`, string` *`format`*`, string` *`locale`*`);`

  The `str2decimal(string, string, string)` function takes the first string argument and converts it to the corresponding decimal value according to the format specified as the second argument and the locale specified as the third argument.

- `number` **`str2double`**`(string` *`arg`*`);`

  The `str2double(string)` function takes one string argument and converts it to the corresponding double value.

- `number` **`str2double`**`(string` *`arg`*`, string` *`format`*`);`

  The `str2double(string, string)` function takes the first string argument and converts it to the corresponding double value according to the format specified as the second argument. Locale has system value.

- `number` **`str2double`**`(string` *`arg`*`, string` *`format`*`, string` *`locale`*`);`

  The `str2decimal(string, string, string)` function takes the first string argument and converts it to the corresponding double value according to the format specified as the second argument and the locale specified as the third argument.

- `integer` **`str2integer`**`(string` *`arg`*`);`

  The `str2integer(string)` function takes one string argument and converts it to the corresponding integer value.

- `integer` **`str2integer`**`(string` *`arg`*`, integer` *`radix`*`);`

  The `str2integer(string, integer)` function accepts two arguments: string and integer. It takes the first argument as if it were expressed in the `radix` based numeric system representation and returns its corresponding integer decimal value.

- `integer` **`str2integer`**`(string` *`arg`*`, string` *`format`*`);`

  The `str2integer(string, string)` function takes the first string argument as decimal string representation of an integer number corresponding to the format specified as the second argument and the system locale and converts it to the corresponding integer value.

- `integer` **`str2integer`**`(string` *`arg`*`, string` *`format`*`, string` *`locale`*`);`

  The `str2integer(string, string, string)` function takes the first string argument as decimal string representation of an integer number corresponding to the format specified as the second argument and the locale specified as the third argument and converts it to the corresponding integer value.

- `long` **`str2long`**`(string` *`arg`*`, integer` *`radix`*`);`

  The `str2long(string, integer)` function accepts two arguments: string and integer. It takes the first argument as if it were expressed in the `radix` based numeric system representation and returns its corresponding long decimal value.

- `long` **`str2long`**`(string` *`arg`*`, string` *`format`*`);`

  The `str2long(string, string)` function takes the first string argument as decimal string representation of a long number corresponding to the format specified as the second argument and the system locale and converts it to the corresponding long value.

- `long` **`str2long`**`(string` *`arg`*`, string` *`format`*`, string` *`locale`*`);`

  The `str2long(string, string, string)` function takes the first string argument as decimal string representation of a long number corresponding to the format specified as the second argument and the locale specified as the third argument and converts it to the corresponding long value.

- `string` **`toString`**`(<numeric|list|map type>` *`arg`*`);`

  The `toString(<numeric|list|map type>)` function takes one argument and converts it to its string representation. It accepts any numeric data type, list of any data type, as well as map of any data types.

- `string` **`xml2json`**`(string` *`arg`*`);`

  The `xml2josn(string)` function takes one string argument that is `XML` formatted and converts it to a `JSON` formatted string. Its counterpart is the `json2xml(string)` function.

Date Functions

When you work with date, you may use the functions that process dates.

In these functions, sometimes a format pattern of a date or any number must be defined. Also locale can have an influence to their formatting.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of the functions:

- `date` **`dateAdd`**`(date` *`arg`*`, long` *`amount`*`, unit` *`timeunit`*`);`

  The `dateAdd(date, long, unit)` function accepts three arguments: the first is date, the second is of long data type and the last is any time unit. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can neither be received through an edge nor set as variable. The function takes the first argument, adds the amount of time units to it and returns the result as a date. The `amount` and time `unit` are specified as the second and third arguments, respectively.

- `long` **`dateDiff`**`(date` *`later`*`, date` *`earlier`*`, unit` *`timeunit`*`);`

  The `dateDiff(date, date, unit)` function accepts three arguments: two dates and one time unit. It takes these arguments and subtracts the second argument from the first argument. The unit can be one of the following: `year`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisec`. The unit must be specified as a constant. It can be neither received through an edge nor set as variable. The function returns the resulting time difference expressed in time units specified as the third argument. Thus, the difference of two dates is expressed in defined time units. The result is expressed as an integer number. Thus, `dateDiff(2008-06-18, 2001-02-03, year)` returns `7`. But, `dateDiff(2001-02-03, 2008-06-18, year)` returns `-7`!

- `date` **`extractDate`**`(date` *`arg`*`);`

  The `extractDate(date)` function takes one date argument and returns only the information containing year, month, and day values.

- `date` **`extractTime`**`(date` *`arg`*`);`

  The `extractTime(date)` function takes one date argument and returns only the information containing hours, minutes, seconds, and milliseconds.

- `date` **`randomDate`**`(date` *`startDate`*`, date` *`endDate`*`);`

  The `randomDate(date, date)` function accepts two date arguments and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The default `format` is specified in the `defaultProperties` file.

- `date` **`randomDate`**`(long` *`startDate`*`, long` *`endDate`*`);`

  The `randomDate(long, long)` function accepts two arguments of long data type - each of them represents a date - and returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used. The default `format` is specified in the `defaultProperties` file.

- `date` **`randomDate`**`(string` *`startDate`*`, string` *`endDate`*`, string` *`format`*`);`

  The `randomDate(string, string, string)` function accepts three stringarguments. Two first represent dates, the third represents a format. The function returns a random date between `startDate` and `endDate` corresponding to the `format` specified by the third argument. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate`. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`. As `locale`, system value is used.

- `date` **`randomDate`**`(string` *`startDate`*`, string` *`endDate`*`, string` *`format`*`, string` *`locale`*`);`

  The `randomDate(string, string, string, string)` function accepts four string arguments. The first and the second argument represent dates. The third is a format and the fourt is locale. The function returns a random date between `startDate` and `endDate`. These resulting dates are generated at random for different records and different fields. They can be different for both records and fields. The return value can also be `startDate` or `endDate` corresponding to the `format` and the `locale` specified by the third and the fourth argument, respectively. However, it cannot be the date before `startDate` nor after `endDate`. Remember that dates represent 0 hours and 0 minutes and 0 seconds and 0 milliseconds of the specified day, thus, if you want that `endDate` could be returned, enter the next date as `endDate`.

- `date` **`today`**`();`

  The `today()` function accepts no argument and returns current date and time.

- `date` **`trunc`**`(date` *`arg`*`);`

  The `trunc(date)` function takes one date argument and returns the date with the same year, month and day, but hour, minute, second and millisecond are set to 0 values.

- `date` **`truncDate`**`(date` *`arg`*`);`

  The `truncDate(date)` function takes one date argument and returns the date with the same hour, minute, second and millisecond, but year, month and day are set to 0 values. The 0 date is 0001-01-01.

- `date` **`zeroDate`**`();`

  The `zeroDate()` function accepts no argument and returns 1.1.1970.

Mathematical Functions

You may also want to use some mathematical functions:

- `<numeric type> **abs**(<numeric type> *arg*);`

  The `abs(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns its absolute value in the same data type.

- `integer **bitAnd**(integer *arg1*, integer *arg2*);`

  The `bitAnd(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `and`. (For example, `bitAnd(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` what corresponds to decimal `3`.

- `long **bitAnd**(long *arg1*, long *arg2*);`

  The `bitAnd(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `and`. (For example, `bitAnd(11,7)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `11` what corresponds to decimal `3`.

- `boolean **bitIsSet**(integer *arg*, integer *Index*);`

  The `bitIsSet(integer, integer)` function accepts two arguments of integer data type. It takes them, determines the value of the bit of the first argument located on the `Index` and returns `true` or `false`, if the bit is `1` or `0`, respectively. (For example, `bitIsSet(11,3)` returns `true`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bitIsSet(11,2)` would return `false`.

- `boolean **bitIsSet**(long *arg*, integer *Index*);`

  The `bitIsSet(long, integer)` function accepts one argument of long data type and one integer. It takes these arguments, determines the value of the bit of the first argument located on the `Index` and returns `true` or `false`, if the bit is `1` or `0`, respectively. (For example, `bitIsSet(11,3)` returns `true`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is `1`, thus the result is `true`. And `bitIsSet(11,2)` would return `false`.

- `integer **bitLShift**(integer *arg*, integer *Shift*);`

  The `bitLShift(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the original number with some bits added (`Shift` number of bits on the left side are added and set to `0`.) (For example, `bitLShift(11,2)` returns `44`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`10`) are added and the result is `101100` which corresponds to decimal `44`.

- `long **bitLShift**(long *arg*, long *Shift*);`

  The `bitLShift(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the original number with some bits added (`Shift` number of bits on the left side are added and set to `0`.) (For example, `bitLShift(11,2)` returns `44`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side (`10`) are added and the result is `101100` which corresponds to decimal `44`.

- `integer **bitNegate**(integer *arg*);`

  The `bitNegate(integer)` function accepts one argument of integer data type. It returns the number corresponding to its bitwise `inverted number`. (For example, `bitNegate(11)` returns `-12`.) The function inverts all bits in an argument.

- `long` **`bitNegate`**`(long` *`arg`*`);`

  The `bitNegate(long)` function accepts one argument of long data type. It returns the number corresponding to its bitwise `inverted number`. (For example, `bitNegate(11)` returns `-12`.) The function inverts all bits in an argument.

- `integer` **`bitOr`**`(integer` *`arg1`*`, integer` *`arg2`*`);`

  The `bitOr(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `or`. (For example, `bitOr(11,7)` returns `15`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` what corresponds to decimal `15`.

- `long` **`bitOr`**`(long` *`arg1`*`, long` *`arg2`*`);`

  The `bitOr(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `or`. (For example, `bitOr(11,7)` returns `15`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1111` what corresponds to decimal `15`.

- `integer` **`bitRShift`**`(integer` *`arg`*`, integer` *`Shift`*`);`

  The `bitRShift(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the original number with some bits removed (`Shift` number of bits on the right side are removed.) (For example, `bitRShift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` what corresponds to decimal `2`.

- `long` **`bitRShift`**`(long` *`arg`*`, long` *`Shift`*`);`

  The `bitRShift(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the original number with some bits removed (`Shift` number of bits on the right side are removed.) (For example, `bitRShift(11,2)` returns `2`.) As decimal `11` can be expressed as bitwise `1011`, thus the two bits on the right side are removed and the result is `10` what corresponds to decimal `2`.

- `integer` **`bitSet`**`(integer` *`arg1`*`, integer` *`Index`*`, boolean` *`SetBitTo1`*`);`

  The `bitSet(integer, integer, boolean)` function accepts three arguments. The first two are of integer data type and the third is boolean. It takes them, sets the value of the bit of the first argument located on the `Index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as an integer. (For example, `bitSet(11,3,false)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` what corresponds to decimal `3`. And `bitSet(11,2,true)` would return `1111` what corresponds to decimal `15`.

- `long` **`bitSet`**`(long` *`arg1`*`, integer` *`Index`*`, boolean` *`SetBitTo1`*`);`

  The `bitSet(long, integer, boolean)` function accepts three arguments. The first is long, the second is integer, and the third is boolean. It takes them, sets the value of the bit of the first argument located on the `Index` specified as the second argument to `1` or `0`, if the third argument is `true` or `false`, respectively, and returns the result as an integer. (For example, `bitSet(11,3,false)` returns `3`.) As decimal `11` can be expressed as bitwise `1011`, the bit whose index is 3 (the fourth from the right) is set to `0`, thus the result is `11` what corresponds to decimal `3`. And `bitSet(11,2,true)` would return `1111` what corresponds to decimal `15`.

- `integer` **`bitXor`**`(integer` *`arg`*`, integer` *`arg`*`);`

  The `bitXor(integer, integer)` function accepts two arguments of integer data type. It takes them and returns the number corresponding to the bitwise `exclusive or`. (For example, `bitXor(11,7)` returns `12`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` what corresponds to decimal `15`.

- `long **bitXor**(long *arg*, long *arg*);`

  The `bitXor(long, long)` function accepts two arguments of long data type. It takes them and returns the number corresponding to the bitwise `exclusive or`. (For example, `bitXor(11,7)` returns `12`.) As decimal `11` can be expressed as bitwise `1011`, decimal `7` can be expressed as `111`, thus the result is `1100` what corresponds to decimal `15`.

- `number **ceil**(decimal *arg*);`

  The `ceil(decimal)` function takes one argument of decimal data type and returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.

- `number **ceil**(number *arg*);`

  The `ceil(number)` function takes one argument of double data type and returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.

- `number **e**();`

  The `e()` function accepts no argument and returns the Euler number.

- `number **exp**(<numeric type> *arg*);`

  The `exp(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the exponential function of this argument.

- `number **floor**(decimal *arg*);`

  The `floor(decimal)` function takes one argument of decimal data type and returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.

- `number **floor**(number *arg*);`

  The `floor(number)` function takes one argument of double data type and returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.

- `void **setRandomSeed**(long *arg*);`

  The `setRandomSeed(long)` takes one long argument and generates the seed for all functions that generate values at random.

  This function should be used in the `preExecute()` function or method.

  In such a case, all values generated at random do not change on different runs of the graph, they even remain the same after the graph is resetted.

- `number **log**(<numeric type> *arg*);`

  The `log(<numeric type>)` takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the natural logarithm of this argument.

- `number **log10**(<numeric type> *arg*);`

  The `log10(<numeric type>)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the result of the logarithm of this argument to the base 10.

- `number **pi**();`

  The `pi()` function accepts no argument and returns the pi number.

- `number` **`pow`**`(<numeric type>` *`base`*`, <numeric type>` *`exp`*`);`

  The `pow(<numeric type>, <numeric type>)` function takes two arguments of any numeric data types (that do not need to be the same, `integer`, `long`, `number`, or `decimal`) and returns the exponential function of the first argument as the exponent with the second as the base.

- `number` **`random`**`();`

  The `random()` function accepts no argument and returns a random positive double greater than or equal to `0.0` and less than `1.0`.

- `boolean` **`randomBoolean`**`();`

  The `randomBoolean()` function accepts no argument and generates at random boolean values `true` or `false`. If these values are sent to any numeric data type field, they are converted to their numeric representation automatically (`1` or `0`, respectively).

- `number` **`randomGaussian`**`();`

  The `randomGaussian()` function accepts no argument and generates at random both positive and negative values of number data type in a Gaussian distribution.

- `integer` **`randomInteger`**`();`

  The `randomInteger()` function accepts no argument and generates at random both positive and negative integer values.

- `integer` **`randomInteger`**`(integer` *`Minimum`*`, integer` *`Maximum`*`);`

  The `randomInteger(integer, integer)` function accepts two argument of integer data types and returns a random integer value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `long` **`randomLong`**`();`

  The `randomLong()` function accepts no argument and generates at random both positive and negative long values.

- `long` **`randomLong`**`(long` *`Minimum`*`, long` *`Maximum`*`);`

  The `randomLong(long, long)` function accepts two argument of long data types and returns a random long value greater than or equal to `Minimum` and less than or equal to `Maximum`.

- `long` **`round`**`(decimal` *`arg`*`);`

  The `round(decimal)` function takes one argument of decimal data type and returns the long that is closest to this argument. Decimal is converted to number prior to the operation.

- `long` **`round`**`(number` *`arg`*`);`

  The `round(number)` function takes one argument of number data type and returns the long that is closest to this argument.

- `number` **`sqrt`**`(<numeric type>` *`arg`*`);`

  The `sqrt(mumerictype)` function takes one argument of any numeric data type (`integer`, `long`, `number`, or `decimal`) and returns the square root of this argument.

String Functions

Some functions work with strings.

In the functions that work with strings, sometimes a format pattern of a date or any number must be defined.

- For detailed information about date formatting and/or parsing see Data and Time Format (p. 99).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 106).
- For detailed information about locale see Locale (p. 112).

Note

Remember that numeric and date formats are displayed using system value **Locale** or **Locale** specified in the `defaultProperties` file, unless other **Locale** is explicitly specified.

For more information on how **Locale** may be changed in the `defaultProperties` see Changing Default CloverETL Settings (p. 76).

Here we provide the list of the functions:

- `string` **`charAt`**`(string` *`arg`*`, integer` *`index`*`);`

  The `charAt(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes the string and returns the character that is located at the position specified by the `index`.

- `string` **`chop`**`(string` *`arg`*`);`

  The `chop(string)` function accepts one string argument. The function takes this argument, removes the line feed and the carriage return characters from the end of the string specified as the argument and returns the new string without these characters.

- `string` **`chop`**`(string` *`arg1`*`, string` *`arg2`*`);`

  The `chop(string, string)` function accepts two string arguments. It takes the first argument, removes the string specified as the second argument from the end of the first argument and returns the first string argument without the string specified as the second argument.

- `string` **`concat`**`(string` *`arg1`*`, string ..., string` *`argN`*`);`

  The `concat(string, ..., string)` function accepts unlimited number of arguments of string data type. It takes these arguments and returns their concatenation. You can also concatenate these arguments using plus signs, but this function is faster for more than two arguments.

- `integer` **`countChar`**`(string` *`arg`*`, string` *`character`*`);`

  The `countChar(string, string)` function accepts two arguments: the first is string and the second is one character. It takes them and returns the number of occurrences of the character specified as the second argument in the string specified as the first argument.

- `string[]` **`cut`**`(string` *`arg`*`, integer[]` *`indeces`*`);`

  The `cut(string, integer[])` function accepts two arguments: the first is string and the second is list of integers. The function returns a list of strings. The number of elements of the list specified as the second argument must be even. The integers in the list serve as position (each number in the odd position) and length (each number in the even position). Substrings of the specified length are taken from the string specified as the first argument starting from the specified position (excluding the character at the specified position).

The resulting substrings are returned as list of strings. For example, `cut("somestringasanexample", [2,3,1,5])` returns `["mes","omest"]`.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`);`

  The `editDistance(string, string)` function accepts two string arguments. These strings will be compared to each other. The strength of comparison is 4 by default, the default value of locale for comparison is the system value and the maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`);`

  The `editDistance(string, string, string)` function accepts three arguments. The first two are strings that will be compared to each other and the third (string) is the locale that will be used for comparison. The default strength of comparison is 4. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`);`

  The `editDistance(string, string, integer)` function accepts three arguments. The first two are strings that will be compared to each other and the third (integer) is the strength of comparison. The default locale that will be used for comparison is the system value. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, string` *`locale`*`);`

  The `editDistance(string, string, integer, string)` function accepts four arguments. The first two are strings that will be compared to each other, the third (integer) is the strength of comparison and the fourth (string) is the locale that will be used for comparison. The maximum difference is 3 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, string` *`locale`*`, integer` *`maxDifference`*`);`

  The `editDistance(string, string, string, integer)` function accepts four arguments. The first two are strings that will be compared to each other, the third (string) is the locale that will be used for comparison and the fourth (integer) is the maximum difference. The strength of comparison is 4 by default.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, integer` *`maxDifference`*`);`

  The `editDistance(string, string, integer, integer)` function accepts four arguments. The first two are strings that will be compared to each other and the two others are both integers. These are the strength of comparison (third argument) and the maximum difference (fourth argument). The locale is the default system value.

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

  For more details, see another version of the `editDistance()` function below - the editDistance (string, string, integer, string, integer) (p. 713) function.

- `integer` **`editDistance`**`(string` *`arg1`*`, string` *`arg2`*`, integer` *`strength`*`, string` *`locale`*`, integer` *`maxDifference`*`);`

  The `editDistance(string, string, integer, string, integer)` function accepts five arguments. The first two are strings, the three others are integer, string and integer, respectively. The function takes the first two arguments and compares them to each other using the other three arguments.

  The third argument (integer number) specifies the strength of comparison. It can have any value from 1 to 4.

  If it is 4 (identical comparison), that means that only identical letters are considered equal. In case of 3 (tertiary comparison), that means that upper and lower cases are considered equal. If it is 2 (secondary comparison), that means that letters with diacritical marks are considered equal. Last, if the strength of comparison is 1 (primary comparison), that means that even the letters with some specific signs are considered equal. In other versions of the `editDistance()` function where this strength of comparison is not specified, the number 4 is used as the default strength (see above).

  The fourth argument is the string data type. It is the locale that serves for comparison. If no locale is specified in other versions of the `editDistance()` function, its default value is the system value (see above).

  The fifth argument (integer number) means the number of letters that should be changed to transform one of the first two arguments to the other. If another version of the `editDistance()` function does not specify this maximum difference, the default maximum difference is number 3 (see above).

  The function returns the number of letters that should be changed to transform one of the first two arguments to the other. However, when the function is being executed, if it counts that the number of letters that should be

changed is at least the number specified as the maximum difference, the execution terminates and the function returns `maxDifference + 1` as the return value.

Actually the function is implemented for the following locales: CA, CZ, ES, DA, DE, ET, FI, FR, HR, HU, IS, IT, LT, LV, NL, NO, PL, PT, RO, SK, SL, SQ, SV, TR. These locales have one thing in common: they all contain language-specific characters. A complete list of these characters can be examined in CTL2 Appendix - List of National-specific Characters (p. 729)

See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `string` **`escapeUrl`**`(string` *`arg`*`);`

  The function escapes illegal characters within components of specified URL (see isUrl() CTL2 function (p. 716) for the URL component description). Illegal characters must be escaped by a percent character `%` symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character.

- `string[]` **`find`**`(string` *`arg`*`, string` *`regex`*`);`

  The `find(string, string)` function accepts two string arguments. The second one is regular expression. The function takes them and returns a list of substrings corresponding to the regex pattern that are found in the string specified as the first argument.

- `string` **`getAlphanumericChars`**`(string` *`arg`*`);`

  The `getAlphanumericChars(string)` function takes one string argument and returns only letters and digits contained in the string argument in the order of their appearance in the string. The other characters are removed.

- `string` **`getAlphanumericChars`**`(string` *`arg`*`, boolean` *`takeAlpha`*`, boolean` *`takeNumeric`*`);`

  The `getAlphanumericChars(string, boolean, boolean)` function accepts three arguments: one string and two booleans. It takes them and returns letters and/or digits if the second and/or the third arguments, respectively, are set to true.

- `string` **`getUrlHost`**`(string` *`arg`*`);`

  The function parses out host name from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the hostname part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `string` **`getUrlPath`**`(string` *`arg`*`);`

  The function parses out path from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the path part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `integer` **`getUrlPort`**`(string` *`arg`*`);`

  The function parses out port number from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the port part is not present in the URL argument, -1 is returned. If the URL has invalid syntax, -2 is returned.

- `string` **`getUrlProtocol`**`(string` *`arg`*`);`

  The function parses out protocol name from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the protocol part is not present in the URL argument, an empty string is returned. If the URL is not valid, `null` is returned.

- `string` **`getUrlQuery`**`(string` *`arg`*`);`

The function parses out query (parameters) from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the query part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `string` **`getUrlUserInfo`**`(string` *`arg`*`);`

  The function parses out username and password from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the userinfo part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `string` **`getUrlRef`**`(string` *`arg`*`);`

  The function parses out fragment after # character, also known as ref, reference or anchor, from specified URL (see isUrl() CTL2 function (p. 716) for the scheme). If the fragment part is not present in the URL argument, an empty string is returned. If the URL syntax is invalid, `null` is returned.

- `integer` **`indexOf`**`(string` *`arg`*`, string` *`substring`*`);`

  The `indexOf(string, string)` function accepts two strings. It takes them and returns the index of the first appearance of `substring` in the string specified as the first argument.

- `integer` **`indexOf`**`(string` *`arg`*`, string` *`substring`*`, integer` *`fromIndex`*`);`

  The `indexOf(string, string, integer)` function accepts three arguments: two strings and one integer. It takes them and returns the index of the first appearance of `substring` counted from the character located at the position specified by the third argument.

- `boolean` **`isAscii`**`(string` *`arg`*`);`

  The `isAscii(string)` function takes one string argument and returns a boolean value depending on whether the string can be encoded as an ASCII string (true) or not (false).

- `boolean` **`isBlank`**`(string` *`arg`*`);`

  The `isBlank(string)` function takes one string argument and returns a boolean value depending on whether the string contains only white space characters (true) or not (false).

- `boolean` **`isDate`**`(string` *`arg`*`, string` *`pattern`*`);`

  The `isDate(string, string)` function accepts two string arguments. It takes them, compares the first argument with the second as a pattern and, if the first string can be converted to a date which is valid within system value of `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `isDate()` function below - the `isDate(string, string, string, boolean)` function.)

  This function is a variant of the mentioned `isDate(string, string, string)` function in which the default value of the third argument is set to system value.

- `boolean` **`isDate`**`(string` *`arg`*`, string` *`pattern`*`, string` *`locale`*`);`

  The `isDate(string, string, string)` function accepts three string arguments. It takes them, compares the first argument with the second as a pattern, use the third argument (`locale`) and, if the first string can be converted to a date which is valid within specified `locale`, according to the specified `pattern`, the function returns true. If it is not possible, it returns false.

  (For more details, see another version of the `isDate()` function below - the `isDate(string, string, string, boolean)` function.)

  See http://java.sun.com/j2se/1.6.0/docs/api/java/util/Locale.html for details about **Locale**.

- `boolean` **`isInteger`**`(string` *`arg`*`);`

  The `isInteger(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to an integer number (true) or not (false).

- `boolean` **`isLong`**`(string` *`arg`*`);`

  The `isLong(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a long number (true) or not (false).

- `boolean` **`isNumber`**`(string` *`arg`*`);`

  The `isNumber(string)` function takes one string argument and returns a boolean value depending on whether the string can be converted to a double (true) or not (false).

- `boolean` **`isUrl`**`(string` *`arg`*`);`

  The function checks whether specified string is a valid URL of the following syntax

```
foo://username:passw@host.com:8042/there/index.dtb?type=animal;name=cat#nose
\_/   \____________/ \______/ \__/\_______________/ \____________________/ \__/
 |           |          |      |         |                   |              |
protocol  userinfo     host   port      path               query          ref
```

  See http://www.ietf.org/rfc/rfc2396.txt for more info about the URI standards.

- `string` **`join`**`(string` *`delimiter`*`, <element type>[]` *`arg`*`);`

  The `join(string, <element type>[])` function accepts two arguments. The first is string, the second is a list of elements of any data type. The elements that are not strings are converted to their string representation and put together with the first argument as delimiter.

- `string` **`join`**`(string` *`delimiter`*`, map[<type of key>,<type of value>]` *`arg`*`);`

  The `join(string, map[<type of key>,<type of value>])` function accepts two arguments. The first is string, the second is a map of any data types. The map elements are displayed as `key=value` strings. These are put together with the first argument as delimiter.

- `string` **`left`**`(string` *`arg`*`, integer` *`length`*`);`

  The `left(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes them and returns the substring of the length specified as the second argument counted from the start of the string specified as the first argument. If the input string is shorter than the *`length`* parameter, an exception is thrown and the graph fails. To avoid such failure, use the `left(string, integer, boolean)` function described below.

- `string` **`left`**`(string` *`arg`*`, integer` *`length`*`, boolean` *`spacePad`*`);`

The function returns prefix of the specified length. If the input string is longer or equally long as the *length* parameter, the function behaves the same way as the `left(string, integer)` function. There is different behaviour if the input string is shorter than the specified length. If the 3rd argument is `true`, the right side of the result is padded with blank spaces so that the result has specified length beeing left justified. Whereas if `false`, the input string is returned as the result with no space added.

- `integer` **`length`**`(structuredtype` *`arg`*`);`

  The `length(structuredtype)` function accepts a structured data type as its argument: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`. It takes the argument and returns a number of elements forming the structured data type.

- `string` **`lowerCase`**`(string` *`arg`*`);`

  The `lowerCase(string)` function takes one string argument and returns another string with cases converted to lower cases only.

- `boolean` **`matches`**`(string` *`arg`*`, string` *`regex`*`);`

  The `matches(string, string)` function takes two string arguments. The second argument is some regular expression. If the first argument can be expressed with such regular expression, the function returns `true`, otherwise it is `false`.

- `string` **`metaphone`**`(string` *`arg`*`, integer` *`maxLength`*`);`

  The `metaphone(string, integer)` function accepts one string argument and one integer meaning the maximum length. The function takes these arguments and returns the metaphone code of the first argument of the specified maximum length. The default maximum length is 4. For more information, see the following site: www.lanw.com/java/phonetic/default.htm.

- `string` **`NYSIIS`**`(string` *`arg`*`);`

  The `NYSIIS(string)` function takes one string argument and returns the New York State Identification and Intelligence System Phonetic Code of the argument. For more information, see the following site: http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System.

- `string` **`randomString`**`(integer` *`minLength`*`, integer` *`maxLength`*`);`

  The `randomString(integer, integer)` function takes two integer arguments and returns strings composed of lowercase letters whose length varies between `minLength` and `maxLength`. These resulting strings are generated at random for records and fields. They can be different for both different records and different fields. Their length can also be equal to `minLength` or `maxLength`, however, they can be neither shorter than `minLength` nor longer than `maxLength`.

- `string` **`removeBlankSpace`**`(string` *`arg`*`);`

  The `removeBlankSpace(string)` function takes one string argument and returns another string with white spaces removed.

- `string` **`removeDiacritic`**`(string` *`arg`*`);`

  The `removeDiacritic(string)` function takes one string argument and returns another string with diacritical marks removed.

- `string` **`removeNonAscii`**`(string` *`arg`*`);`

  The `removeNonAscii(string)` function takes one string argument and returns another string with non-ascii characters removed.

- `string` **`removeNonPrintable`**`(string` *`arg`*`);`

The `removeNonPrintable(string)` function takes one string argument and returns another string with non-printable characters removed.

- `string` **`replace`**`(string` *`arg`*`, string` *`regex`*`, string` *`replacement`*`);`

  The `replace(string, string, string)` function takes three string arguments - a string, a regular expression, and a replacement - and replaces all regex matches inside the string with the replacement string you specified. All parts of the string that match the regex are replaced. You can also reference the matched text using a backreference in the replacement string. A backreference to the entire match is indicated as $0. If there are capturing parentheses, you can reference specifics groups as $1, $2, $3, etc.

  `replace("Hello","[Ll]","t")` returns `"Hetto"`

  `replace("Hello", "e(l+)", "a$1")` returns `"Hallo"`

  Important - please beware of similar syntax of $0, $1 etc. While used inside the replacement string it refers to matching regular expression parenthesis (in order). If used outside a string, it means a reference to an input field. See other example:

  `replace("Hello", "e(l+)", $0.name)` returns `HJohno` if input field "name" on port 0 contains the name "John".

  You can also use modifier in the start of the regular expression: `(?i)` for case-insensitive search, `(?m)` for multiline mode or `(?s)` for "dotall" mode where a dot (".") matches even a newline character

  `replace("Hello", "(?i)L", "t")` will produce `Hetto` while `replace("Hello", "L", "t")` will just produce `Hello`

- `string` **`right`**`(string` *`arg`*`, integer` *`length`*`);`

  The `right(string, integer)` function accepts two arguments: the first is string and the second is integer. It takes them and returns the substring of the length specified as the second argument counted from the end of the string specified as the first argument. If the input string is shorter than the *`length`* parameter, an exception is thrown and the graph fails. To avoid such failure, use the `right(string, integer, boolean)` function described below.

- `string` **`right`**`(string` *`arg`*`, integer` *`length`*`, boolean` *`spacePad`*`);`

  The function returns suffix of the specified length. If the input string is longer or equally long as the *`length`* parameter, the function behaves the same way as the `right(string, integer)` function. There is different behaviour if the input string is shorter than the specified length. If the 3th argument is `true`, the left side of the result is padded with blank spaces so that the result has specified length beeing right justified. Whereas if `false`, the input string is returned as the result with no space added.

- `string` **`soundex`**`(string` *`arg`*`);`

  The `soundex(string)` function takes one string argument and converts the string to another. The resulting string consists of the first letter of the string specified as the argument and three digits. The three digits are based on the consonants contained in the string when similar numbers correspond to similarly sounding consonants. Thus, `soundex("word")` returns `"w600"`.

- `string[]` **`split`**`(string` *`arg`*`, string` *`regex`*`);`

  The `split(string, string)` function accepts two string arguments. The second is some regular expression. It is searched in the first string argument and if it is found, the string is split into the parts located between the characters or substrings of such a regular expression. The resulting parts of the string are returned as a list of strings. Thus, `split("abcdefg", "[ce]")` returns `["ab", "d", "fg"]`.

- `string` **`substring`**`(string` *`arg`*`, integer` *`fromIndex`*`, integer` *`length`*`);`

The `substring(string, integer, integer)` function accepts three arguments: the first is string and the other two are integers. The function takes the arguments and returns a substring of the defined length obtained from the original string by getting the `length` number of characters starting from the position defined by the second argument. Thus, `substring("text", 1, 2)` returns `"ex"`.

- `string` **`translate`**`(string` *`arg`*`, string` *`searchingSet`*`, string` *`replaceSet`*`);`

  The `translate(string, string, string)` function accepts three string arguments. The number of characters must be equal in both the second and the third arguments. If some character from the string specified as the second argument is found in the string specified as the first argument, it is replaced by a character taken from the string specified as the third argument. The character from the third string must be at the same position as the character in the second string. Thus, `translate("hello", "leo", "pii")` returns `"hippi"`.

- `string` **`trim`**`(string` *`arg`*`);`

  The `trim(string)` function takes one string argument and returns another string with leading and trailing white spaces removed.

- `string` **`unescapeUrl`**`(string` *`arg`*`);`

  The function decodes escape sequences of illegal characters within components of specified URL (see isUrl() CTL2 function (p. 716) for the URL component description). Escape sequences consist of a percent character `%` symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the ISO-Latin code point for the character, e.g., `%20` is the escaped encoding for the US-ASCII space character.

- `string` **`upperCase`**`(string` *`arg`*`);`

  The `upperCase(string)` function takes one string argument and returns another string with cases converted to upper cases only.

Container Functions

When you work with containers (`list`, `map`, `record`), you may use the following functions:

- `elemenettype[] **append**(<element type>[] *arg*, <element type> *list_element*);`

  The `append(<element type>[], <element type>)` function accepts two arguments: the first is a list of any element data type and the second is of the element data type. The function takes the second argument and adds it to the end of the first argument. The function returns the new value of list specified as the first argument.

  This function is alias of the `push(<element type>[], <element type>)` function. From the list point of view, `append()` is much more natural.

- `void **clear**(<element type>[] *arg*);`

  The `clear(<element type>[])` function accepts one list argument of any element data type. The function takes this argument and empties the list. It returns void.

- `void **clear**(map[<type of key>,<type of value>] *arg*);`

  The `clear(map[<type of key>,<type of value>])` function accepts one map argument. The function takes this argument and empties the map. It returns void.

- `<element type>[] **copy**(<element type>[] *arg*, <element type>[] *arg*);`

  The `copy(<element type>[], <element type>[])` function accepts two arguments, each of them is a list. Elements of both lists must be of the same data type. The function takes the second argument, adds it to the end of the first list and returns the new value of the list specified as the first argument.

- `void **copyByName**(record *to*, record *from*);`

  Copies data from the input record to the output record based on field names. Enables mapping of equally named fields only. This function can be used only for records in variables (of the input or output record type) or with wildcard syntax for input ports `$0.*`. There is no way how to pass an output record to this method.

- `void **copyByPosition**(record *to*, record *from*);`

  Copies data from the input record to the output record based on fields order. The number of fields in output metadata decides which input fields (beginning the first one) are mapped. This function can be used only for records in variables (of the input or output record type) or with wildcard syntax for input ports `$0.*`. There is no way how to pass an output record to this method.

- `map[<type of key>, <type of value>] **copy**(map[<type of key>, <type of value>] *arg*, map[<type of key>, <type of value>] *arg*);`

  The `copy(map[<type of key>, <type of value>], map[<type of key>, <type of value>])` function accepts two arguments, each of them is a map. Elements of both maps must be of the same data type. The function takes the second argument, adds it to the end of the first map replacing existing key mappings and returns the new value of the map specified as the first argument.

- `<element type>[] **insert**(<element type>[] *arg*, integer *position*, <element type> *newelement*);`

  The `insert(<element type>[], integer, <element type> )` function accepts the following arguments: the first is a list of any element data type, the second is integer, and the other is of the element data type. The function takes the third argument and inserts it to the list at the position defined by the second argument. The list specified as the first argument changes to this new value and it is returned by the function. Remember that the list element are indexed starting from 0.

- `boolean **isEmpty**(<element type>[] *arg*);`

The `isEmpty(<element type>[])` function accepts one argument of list of any element data type. It takes this argument, checks whether the list is empty and returns `true`, or `false`.

- `boolean` **`isEmpty`**`(map[<type of key>,<type of value>]` *`arg`*`);`

  The `isEmpty(map[<type of key>,<type of value>])` function accepts one argument of a map of any value data types. It takes this argument, checks whether the map is empty and returns `true`, or `false`.

- `integer` **`length`**`(structuredtype` *`arg`*`);`

  The `length(structuredtype)` function accepts a structured data type as its argument: `string`, `<element type>[]`, `map[<type of key>,<type of value>]` or `record`. It takes the argument and returns a number of elements forming the structured data type.

- `<element type>` **`poll`**`(<element type>[]` *`arg`*`);`

  The `poll(<element type>[])` function accepts one argument of list of any element data type. It takes this argument, removes the first element from the list and returns this element. The list specified as the argument changes to this new value (without the removed first element).

- `<element type>` **`pop`**`(<element type>[]` *`arg`*`);`

  The `pop(<element type>[])` function accepts one argument of list of any element data type. It takes this argument, removes the last element from the list and returns this element. The list specified as the argument changes to this new value (without the removed last element).

- `<element type>[]` **`push`**`(<element type>[]` *`arg`*`, <element type>` *`list_element`*`);`

  The `push(<element type>[], <element type>)` function accepts two arguments: the first is a list of any data type and the second is the data type of list element. The function takes the second argument and adds it to the end of the first argument. The function returns the new value of the list specified as the first argument.

  This function is alias of the `append(<element type>[], <element type>)` function. From the stack/queue point of view, `push()` is much more natural.

- `<element type>` **`remove`**`(<element type>[]` *`arg`*`, integer` *`position`*`);`

  The `remove(<element type>[], integer)` function accepts two arguments: the first is a list of any element data type and the second is integer. The function removes the element at the specified position and returns the removed element. The list specified as the first argument changes its value to the new one. (List elements are indexed starting from 0.)

- `<element type>[]` **`reverse`**`(<element type>[]` *`arg`*`);`

  The `reverse(<element type>[])` function accepts one argument of a list of any element data type. It takes this argument, reverses the order of elements of the list and returns such new value of the list specified as the first argument.

- `<element type>[]` **`sort`**`(<element type>[]` *`arg`*`);`

  The `sort(<element type>[])` function accepts one argument of a list of any element data type. It takes this argument, sorts the elements of the list in ascending order according to their values and returns such new value of the list specified as the first argument.

Miscellaneous Functions

The rest of the functions can be denominated as miscellaneous. They are the functions listed below.

Important

Remember that the object notation (e.g., `arg.isnull()`) cannot be used for these **Miscellaneous** functions!

For more information about object notation see Functions Reference (p. 695).

- `<any type> iif(boolean con, <any type> iftruevalue, <any type> iffalsevalue);`

  The `iif(boolean, <any type>, <any type>)` function accepts three arguments: one is boolean and two are of any data type. Both argument data types and return type are the same.

  The function takes the first argument and returns the second if the first is true or the third if the first is false.

- `boolean isnull(<any type> arg);`

  The `isnull(<any type>)` function takes one argument and returns a boolean value depending on whether the argument is null (true) or not (false). The argument may be of any data type.

Important

If you set the **Null value** property in metadata for any `string` data field to any non-empty string, the `isnull()` function will return `true` when applied on such string. And return `false` when applied on an empty field.

For example, if `field1` has **Null value** property set to `"<null>"`, `isnull($0.field1)` will return `true` on the records in which the value of `field1` is `"<null>"` and `false` on the others, even on those that are empty.

See Null value (p. 145) for detailed information.

- `<any type> nvl(<any type> arg, <any type> default);`

  The `nvl(<any type>, <any type>)` function accepts two arguments of any data type. Both arguments must be of the same type. If the first argument is not null, the function returns its value. If it is null, the function returns the default value specified as the second argument.

- `<any type> nvl2(<any type> arg, <any type> arg_for_non_null, <any type> arg_for_null);`

  The `nvl2(<any type>, <any type>, <any type>)` function accepts three arguments of any data type. This data type must be the same for all arguments and return value. If the first argument is not null, the function returns the value of the second argument. If the first argument is null, the function returns the value of the third argument.

- `void printErr(<any type> message);`

  The `printErr(<any type>)` function accepts one argument of any data type. It takes this argument and prints out the `message` on the error output.

  This function works as `void printErr(<any type> arg, boolean printLocation)` with `printLocation` set to `false`.

Note

Remember that if you are using this function in any graph that runs on **CloverETL Server**, the `message` is saved to the log of **Server** (e.g., to the log of **Tomcat**). Use the `printLog()` function instead. It logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void` **`printErr`**`(<any type>` *`message`*`, boolean` *`printLocation`*`);`

  The `printErr(type, boolean)` function accepts two arguments: the first is of any data type and the second is boolean. It takes them and prints out the `message` and the location of the error (if the second argument is `true`).

Note

Remember that if you are using this function in any graph that runs on **CloverETL Server**, the `message` is saved to the log of **Server** (e.g., to the log of **Tomcat**). Use the `printLog()` function instead. It logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void` **`printLog`**`(level` *`loglevel`*`, <any type>` *`message`*`);`

  The `printLog(level, <any type>)` function accepts two arguments: the first is a log level of the `message` specified as the second argument, which is of any data type. The first argument is one of the following: `debug`, `info`, `warn`, `error`, `fatal`. The log level must be specified as a constant. It can be neither received through an edge nor set as variable. The function takes the arguments and sends out the `message` to a logger.

Note

Remember that you should use this function especially in any graph that would run on **CloverETL Server** instead of the `printErr()` function which logs error messages to the log of **Server** (e.g., to the log of **Tomcat**). Unlike `printErr()`, `printLog()` logs error messages to the console even when the graph runs on **CloverETL Server**.

- `void` **`raiseError`**`(string` *`message`*`);`

  The `raiseError(string)` function takes one string argument and throws out error with the `message` specified as the argument.

- `void` **`sleep`**`(long` *`duration`*`);`

  The function pauses the execution for specified milliseconds.

Lookup Table Functions

In your graphs you are also using lookup tables. You need to use them in CTL by specifying the name of the lookup table and placing it as an argument in the `lookup()` function.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

Now, the key in the function below is a sequence of values of the field names separated by comma (not semicolon!). Thus, the key is of the following form: `keyValuePart1,keyValuePart2,...,keyValuePartN`.

See the following options:

- `lookup(<lookup name>).get(keyValue)[.<field name>|.*]`

  This function searches the first record whose key value is equal to the value specified in the `get(keyValue)` function.

  It returns the record of the lookup table. You can map it to other records in CTL2 (with the same metadata). If you want to get the value of the field, you can add the `.<field name>` part to the expression or `.*` to get the values of all fields.

- `lookup(<lookup name>).count(keyValue)`

  If you want to get the number of records whose key value equals to `keyValue`, use the syntax above.

- `lookup(<lookup name>).next()[.<field name>|.*]`

  After getting the number of duplicate records in lookup table using the `lookup().count()` function, and getting the first record with specified key value using the `lookup().get()` function, you can work (one by one) with all records of lookup table with the same key value.

  You need to use the syntax shown here in a loop and work with all records from lookup table. Each record will be processed in one loop step.

  The mentioned syntax returns the record of the lookup table. You can map it to other records in CTL2 (with the same metadata). If you want to get the value of the field, you can add the `.<field name>` part to the expression or `.*` to get the values of all fields.

Example 54.8. Usage of Lookup Table Functions

```
//#CTL2

// record with the same metadata as those of lookup table
recordName1 myRecord;

// variable for storing number of duplicates
integer count;

// Transforms input record into output record.
function integer transform() {

     // if lookup table contains duplicate records,
     // their number is returned by the following expression
     // and assigned to the count variable
     count = lookup(simpleLookup0).count($0.Field2);

     // getting the first record whose key value equals to $0.Field2
     myRecord = lookup(simpleLookup0).get($0.Field2);

     // loop for searching the last record in lookup table
     while ((count-1) > 0) {

     // searching the next record with the key specified above
     myRecord = lookup(simpleLookup0).next();

     // incrementing counter
     count--;
     }

     // mapping to the output

     // last record from lookup table
     $0.Field1 = myRecord.Field1;
     $0.Field2 = myRecord.Field2;

     // corresponding record from the edge
     $0.Field3 = $0.Field1;
     $0.Field4 = $0.Field2;
     return 0;
}
```

Warning

In the example above we have shown you the usage of all lookup table functions. However, we suggest you better use other syntax for lookup tables.

The reason is that the following expression of CTL2:

```
lookup(Lookup0).count($0.Field2);
```

searches the records through the whole lookup table which may contain a great number of records.

The syntax shown above may be replaced with the following loop:

```
myRecord = lookup(<name of lookup table>).get(<key value>);
while(myRecord != null) {
  process(myRecord);
```

```
  myRecord = lookup(<name of lookup table>).next();
}
```

Especially DB lookup tables can return -1 instead of real count of records with specified key value (if you do not set **Max cached size** to a non-zero value).

The `lookup_found(<lookup table ID>)` function for CTL1 is not too recommended either.

Important

Remember that DB lookup tables cannot be used in compiled mode! (code starts with the following header: `//#CTL2:COMPILE`)

You need to switch to interpreted mode (with the header: `//#CTL2`) to be able to access DB lookup tables from CTL2.

Sequence Functions

In your graphs you are also using sequences. You can use them in CTL by specifying the name of the sequence and placing it as an argument in the `sequence()` function.

Warning

Remember that you should not use the functions shown below in the `init()`, `preExecute()`, or `postExecute()` functions of CTL template.

You have three options depending on what you want to do with the sequence. You can get the current number of the sequence, or get the next number of the sequence, or you may want to reset the sequence numbers to the initial number value.

See the following options:

```
sequence(<sequence name>).current()

sequence(<sequence name>).next()

sequence(<sequence name>).reset()
```

Although these expressions return integer values, you may also want to get long or string values. This can be done in one of the following ways:

```
sequence(<sequence name>,long).current()

sequence(<sequence name>,long).next()

sequence(<sequence name>,string).current()

sequence(<sequence name>,string).next()
```

Custom CTL Functions

In addition to the prepared CTL functions, you can create your own CTL functions. To do that, you need to write your own code defining the custom CTL functions and specify its plugin.

Each custom CTL function library must be derived/inherited from:

`org.jetel.interpreter.extensions.TLFunctionLibrary` class.

Each custom CTL function must be derived/inhereted from:

`org.jetel.interpreter.extensions.TLFunctionPrototype` class.

These classes have some standard operations defined and several abstract methods which need to be defined so that the custom functions may be used. Within the custom functions code, an existing context must be used or some custom context must be defined. The context serves to store objects when function is to be executed repeatedly, in other words, on more records.

Along with the custom functions code, you also need to define the custom functions plugin. Both the library and the plugin will be used in **CloverETL**. For more information, see the following wiki page: wiki.cloveretl.org/doku.php?id=function_building.

CTL2 Appendix - List of National-specific Characters

Several functions, e.g. editDistance (string, string, integer, string, integer) (p. 713) need to work with special national characters. These are important especially when sorting items with a defined comparison strength.

The list below shows first the locale and then a list of its national-specific derivatives for each letter. These may be treated either as equal or different characters depending on the comparison stregth you define.

Table 54.2. National Characters

Locale	National Characters
CA - Catalan	"a=à=A=À", "e=è=é=E=È=É", "i=í=ï=I=Í=Ï", "o=ò=ó=O=Ò=Ó", "u=ú=ü=U=Ú=Ü", "c=ç=C=Ç"
CZ - Czech	"a=á=A=Á", "c=č=C=Č", "d=ď=D=Ď", "e=é=ě=E=É=Ě", "i=í=I=Í", "n=ň=N=Ň", "o=ó=O=Ó", "r=ř=R=Ř", "s=š=S=Š", "t=ť=T=Ť", "u=ů=ú=U=Ů=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
DA - Danish and Norwegian	"a=æ=å=A=Æ=Å", "o=ø=O=Ø"
DE - German	"a=ä=A=Ä", "o=ö=O=Ö", "u=ü=U=Ü"
ES - Spanish	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=O=Ó", "u=ú=ü=U=Ú=Ü", "n=ñ=N=Ñ"
ET - Estonian	"a=ä=A=Ä", "o=ö=õ=O=Ö=Õ", "u=ü=U=Ü", "s=š=S=Š", "z=ž=Z=Ž"
FI - Finnish	"a=ä=A=Ä", "o=ö=O=Ö"
FR - French	"a=à=â=A=À=Â", "e=è=é=ê=ë=E=È=É=Ê=Ë", "i=ï=I=Ï", "o=ô=O=Ô", "u=ù=û=ü=U=Ù=Û=Ü", "c=ç=C=Ç"
HR - Croatian	"c=ć=č=C=Ć=Č", "d=đ=D=Đ", "s=š=S=Š", "z=ž=Z=Ž"
HU - Hungarian	"a=á=A=Á", "e=é=E=É", "i=í=I=Í", "o=ó=ö=ő=O=Ó=Ö=Ő", "u=ú=ü=ű=U=Ú=Ü=Ű"
IS - Icelandic	"a=á=æ=A=Á=Æ", "e=é=E=É", "i=í=I=Í", "o=ó=ö=O=Ó=Ö", "u=ú=U=Ú", "y=ý=Y=Ý", "d=ð=D=Þ"

Locale	National Characters
IT - Italian	"a=à=A=À", "e=é=è=E=É=È", "i=ì=I=Ì", "o=ò=O=Ò", "u=ù=U=Ù"
LV - Latvian	"a=ā=A=Ā", "e=ē=E=Ē", "i=ī=I=Ī", "u=ū=U=Ū", "c=č=C=Č", "g=ģ=G=Ģ", "k=ķ=K=Ķ", "l=ļ=L=Ļ", "n=ņ=N=Ņ", "s=š=S=Š", "z=ž=Z=Ž"
PL - Polish	"a=ą=A=Ą", "c=ć=C=Ć", "e=ę=E=Ę", "l=ł=L=Ł", "n=ń=N=Ń", "o=ó=O=Ó", "s=ś=S=Ś", "z=ż=ź=Z=Ż=Ź"
PT - Portuguese	"a=ã=á=à=â=A=Ã=Á=À=Â", "e=é=ê=E=É=Ê", "i=í=I=Í", "o=õ=ó=ô=O=Õ=Ó=Ô", "u=ú=U=Ú", "c=ç=C=Ç"
RO - Romanian	"a=ă=â=A=Ă=Â", "i=î=I=Î", "s=ş=S=Ş", "t=ţ=T=Ţ"
RU - Russian	"Е=е=Ё=ё=Е́=е́", "И=и=Й=й=И́=и́", "О=о=Ó=ó", "А=а=А́=а́", "Ы=ы=Ы́=ы́", "Я=я=Я́=я́", "Ю=ю=Ю́=ю́", "У=у=У́=у́", "Э=э=Э́=э́"
SK - Slovak	"a=á=ä=A=Á=Ä", "c=č=C=Č", "d=ď=D=Ď", "e=é=E=É", "i=í=I=Í", "l=ĺ=I=L=Ľ=Ł", "n=ň=N=Ň", "o=ó=ô=O=Ó=Ô", "r=ŕ=R=Ŕ", "s=š=S=Š", "t=ť=T=Ť", "u=ú=U=Ú", "y=ý=Y=Ý", "z=ž=Z=Ž"
SL - Slovenian	"c=č=C=Č", "s=š=S=Š", "z=ž=Z=Ž"
SQ - Albanian	"e=ë=E=Ë", "c=ç=C=Ç"
SV - Swedish	"a=ä=å=A=Ä=Å", "o=ö=O=Ö"

List of Figures

1.1. Family of CloverETL Products 2
2.1. CloverETL Server Project Displayed after Opening CloverETL Designer 5
2.2. Prompt to Open CloverETL Server Project 5
2.3. Opening CloverETL Server Project 6
2.4. Network connections window 9
5.1. CloverETL Designer Splash Screen 15
5.2. Workspace Selection Dialog 15
5.3. CloverETL Designer Introductory Screen 16
6.1. Available Software 17
7.1. Giving a Name to a CloverETL Project 19
7.2. CloverETL Server Project Wizard 20
7.3. Giving a Name to the New CloverETL Server Project 20
7.4. CloverETL Examples Project Wizard 22
7.5. Renaming CloverETL Examples Projects 22
8.1. CloverETL Perspective with Highlighted Navigator Pane and the Project Folder Structure 24
8.2. Opening the Workspace.prm File 25
8.3. Workspace.prm File 25
8.4. Basic Eclipse Perspective 26
8.5. Selecting CloverETL Perspective 26
8.6. CloverETL Perspective 27
9.1. CloverETL Perspective 28
9.2. Graph Editor with an Opened Palette of Components 29
9.3. Closing the Graphs 30
9.4. Rulers in the Graph Editor 30
9.5. Grid in the Graph Editor 31
9.6. A Graph before Selecting Auto-Layout. 31
9.7. A Graph after Selecting Auto-Layout. 32
9.8. Six New Buttons in the Tool Bar Appear Highlighted (Align Middle is shown) 32
9.9. Alignments from the Context Menu 33
9.10. Navigator Pane 33
9.11. Outline Pane 34
9.12. Another Representation of the Outline Pane 34
9.13. Properties Tab 35
9.14. Console Tab 35
9.15. Problems Tab 36
9.16. Clover - Regex Tester Tab 36
9.17. Clover - Graph Tracking Tab 37
9.18. Clover - Log Tab 37
10.1. Creating a New Graph 38
10.2. Giving a Name to a New CloverETL Graph 39
10.3. Selecting the Parent Folder for the Graph 39
10.4. CloverETL Perspective with Highlighted Graph Editor 40
10.5. Graph Editor with a New Graph and the Palette of Components 40
10.6. Components Selected from the Palette 41
10.7. Components are Connected by Edges 42
10.8. Creating an Input File 42
10.9. Creating the Contents of the Input File 43
10.10. Metadata Editor with Default Names of the Fields 43
10.11. Metadata Editor with New Names of the Fields 44
10.12. Edge Has Been Assigned Metadata 44
10.13. Metadata Have Been Propagated through the Component 45
10.14. Opening the Attribute Row 45
10.15. Selecting the Input File 46
10.16. Input File URL Attribute Has Been Set 46
10.17. Output File URL without a File 47

10.18. Output File URL with a File 47
10.19. Defining a Sort Key 48
10.20. Sort Key Has Been Defined 48
10.21. Running the Graph 49
10.22. Result of Successful Run of the Graph 49
10.23. Contents of the Output File 50
11.1. Running a Graph from the Main Menu 51
11.2. Running a Graph from the Context Menu 52
11.3. Running a Graph from the Upper Tool Bar 52
11.4. Successful Graph Execution 53
11.5. Console Tab with an Overview of the Graph Execution 53
11.6. Counting Parsed Data 54
11.7. Run Configurations Dialog 54
12.1. Selecting Cheat Sheets 56
12.2. The Cheat Sheet Selection Wizard 56
12.3. CloverETL and Standard Eclipse Commands (Collapsed) 57
12.4. CloverETL and Standard Eclipse Commands (Expanded) 57
12.5. CloverETL Designer Reference Cheat Sheet 58
12.6. Locating a Custom Cheat Sheet 58
13.1. URL File Dialog 59
13.2. Edit Value Dialog 60
13.3. Find Wizard 60
13.4. Go to Line Wizard 61
13.5. Open Type Dialog 61
14.1. Import (Main Menu) 62
14.2. Import (Context Menu) 62
14.3. Import Options 63
14.4. Import Projects 63
14.5. Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server) 64
14.6. Import from CloverETL Server Sandbox Wizard (List of Files) 64
14.7. Import Graphs 65
14.8. Import Metadata from XSD 66
14.9. Import Metadata from DDL 67
15.1. Export Options 68
15.2. Export Graphs 68
15.3. Export Graphs to HTML 69
15.4. Export metadata to XSD 70
15.5. Export to CloverETL Server Sandbox 71
15.6. Export Image 72
16.1. Setting Up Memory Size 75
16.2. Custom Clover Settings 76
16.3. Enlarging the Font of Numbers 78
16.4. Setting the Font Size 78
16.5. Setting The Java Runtime Environment 79
16.6. Preferences Wizard 80
16.7. Installed JREs Wizard 80
16.8. Adding a Java Development Kit 81
16.9. Searching for JDK Jars 82
16.10. Adding JDK Jars 82
18.1. Selecting the Edge Type 87
18.2. Creating Metadata on an empty Edge 88
18.3. Assigning Metadata to an Edge 89
18.4. Metadata in the Tooltip 90
18.5. Properties of an Edge 91
18.6. Filter Editor Wizard 91
18.7. Debug Properties Wizard 93
18.8. View Data Dialog 93
18.9. Viewing Data 93

18.10. Hide/Show Columns when Viewing Data 94
18.11. View Record Dialog 94
18.12. Find Dialog 95
18.13. Copy Dialog 95
19.1. Creating Internal Metadata in the Outline Pane 119
19.2. Creating Internal Metadata in the Graph Editor 120
19.3. Externalizing and/or Exporting Internal Metadata 121
19.4. Selecting a Location for a New Externalized and/or Exported Internal Metadata 121
19.5. Creating External (Shared) Metadata in the Main Menu and/or in the Navigator Pane 122
19.6. Internalizing External (Shared) Metadata 123
19.7. Extracting Metadata from Delimited Flat File 124
19.8. Extracting Metadata from Fixed Length Flat File 125
19.9. Setting Up Delimited Metadata 126
19.10. Setting Up Fixed Length Metadata 128
19.11. Extracting Metadata from XLS File 129
19.12. Extracting Internal Metadata from a Database 130
19.13. Database Connection Wizard 131
19.14. Selecting Columns for Metadata 131
19.15. Generating a Query 132
19.16. DBF Metadata Editor 134
19.17. Specifying Lotus Notes connection for metadata extraction 135
19.18. Lotus Notes metadata extraction wizard, page 2 136
19.19. Creating Database Table from Metadata and Database Connection 138
19.20. Metadata Editor for a Delimited File 141
19.21. Metadata Editor for a Fixed Length File 142
20.1. Creating Internal Database Connection 151
20.2. Externalizing Internal Database Connection 152
20.3. Internalizing External (Shared) Database Connection 154
20.4. Database Connection Wizard 155
20.5. Adding a New JDBC Driver into the List of Available Drivers 155
20.6. Running a Graph with the Password Encrypted 158
21.1. Edit JMS Connection Wizard 162
22.1. QuickBase Connection Dialog 164
23.1. Lotus Notes Connection Dialog 165
24.1. Creating Internal Lookup Table 167
24.2. Externalizing Wizard 168
24.3. Selecting Lookup Table Item 170
24.4. Lookup Table Internalization Wizard 171
24.5. Lookup Table Wizard 172
24.6. Simple Lookup Table Wizard 172
24.7. Edit Key Wizard 173
24.8. Simple Lookup Table Wizard with File URL 173
24.9. Simple Lookup Table Wizard with Data 174
24.10. Changing Data 174
24.11. Database Lookup Table Wizard 175
24.12. Appropriate Data for Range Lookup Table 176
24.13. Range Lookup Table Wizard 176
24.14. Persistent Lookup Table Wizard 178
24.15. Aspell Lookup Table Wizard 180
25.1. Creating a Sequence 182
25.2. Editing a Sequence 185
25.3. A New Run of the Graph with the Previous Start Value of the Sequence 185
26.1. Creating Internal Parameters 188
26.2. Externalizing Internal Parameters 189
26.3. Internalizing External (Shared) Parameter 191
26.4. Example of a Parameter-Value Pairs 192
28.1. Dictionary Dialog with Defined Entries 199
29.1. Pasting a Note to the Graph Editor Pane 202

29.2. Enlarging the Note 202
29.3. Highlighted Margins of the Note Have Disappeared 203
29.4. Changing the Note Label 203
29.5. Writing a New Description in the Note 204
29.6. A New Note with a New Description 204
29.7. Folding the Note 205
29.8. Properties of a Note 205
30.1. CloverETL Search Tab 206
30.2. Search Results 207
32.1. Valid selections 210
32.2. How to select the FTL wizard 210
32.3. Select a wizard (new wizard selection window) 211
32.4. Deselection 211
32.5. New Graph Name Page 212
32.6. Output Page 212
32.7. File Selection Page 213
32.8. URL Dialog 213
32.9. Database Connection Page 214
32.10. Fact Table Selection Page 214
32.11. Dimension Table Selection Page 215
32.12. Order Table Page 215
32.13. Mapping Page 216
32.14. Fact Mapping Page 216
32.15. Summary Page 217
32.16. Created Graph 217
32.17. Graph Parameters 218
34.1. Selecting Components 221
34.2. Components in Palette 221
34.3. Removing Components from the Palette 222
35.1. Edit Component Dialog (Properties Tab) 223
35.2. Edit Component Dialog (Ports Tab) 223
35.3. Simple Renaming Components 227
35.4. Running a Graph with Various Phases 228
35.5. Setting the Phases for More Components 228
35.6. Running a Graph with Disabled Component 229
35.7. Running a Graph with Component in PassThrough Mode 230
36.1. Defining Group Key 231
36.2. Defining Sort Key and Sort Order 232
36.3. Define Error Actions Dialog 240
36.4. Transformations Tab of the Transform Editor 241
36.5. Copying the Input Field to the Output 242
36.6. Transformation Definition in CTL (Transformations Tab) 242
36.7. Mapping of Inputs to Outputs (Connecting Lines) 243
36.8. Editor with Fields and Functions 243
36.9. Transformation Definition in CTL (Source Tab) 244
36.10. Java Transform Wizard Dialog 244
36.11. Confirmation Message 245
36.12. Transformation Definition in CTL (Transform Tab of the Graph Editor) 245
36.13. Outline Pane Displaying Variables and Functions 246
36.14. Content Assist (Record and Field Names) 246
36.15. Content Assist (List of CTL Functions) 247
36.16. Error in Transformation 247
36.17. Converting Transformation to Java 247
36.18. Transformation Definition in Java 248
37.1. Viewing Data in Components 255
37.2. Viewing Data as Plain Text 256
37.3. Viewing Data as Grid 256
37.4. Plain Text Data Viewing 256

37.5. Grid Data Viewing 257
37.6. XML Features Dialog 261
38.1. Viewing Data on Components 267
38.2. Viewing Data as Plain Text 267
38.3. Viewing Data as Grid 267
38.4. Plain Text Data Viewing 268
38.5. Grid Data Viewing 268
40.1. Source Tab of the Transform Editor in Joiners 279
44.1. Sequences Dialog 300
44.2. A Sequence Assigned 300
44.3. Edit Key Dialog 301
44.4. Source Tab of the Transform Editor in DataGenerator 302
44.5. Generated Query with Question Marks 309
44.6. Generated Query with Output Fields 310
44.7. XLS Mapping Dialog 340
44.8. XLS Fields Mapped to Clover Fields 341
44.9. The Mapping Dialog for XMLExtract 348
44.10. Parent Elements Visible in XML Fields 349
45.1. Generated Query with Question Marks 375
45.2. Generated Query with Input Fields 375
45.3. Generated Query with Returned Fields 376
45.4. EmailSender Message Wizard 378
45.5. Edit Attachments Wizard 379
45.6. Attachment Wizard 380
45.7. Create Mask Dialog 412
45.8. Mapping Editor 425
45.9. Adding Child to Root Element. 426
45.10. Wildcard attribute and its properties. 427
45.11. Attribute and its properties. 428
45.12. Element and its properties. 428
45.13. Mapping editor toolbar. 431
45.14. Binding of Port and Element. 433
45.15. Generating XML from XSD root element. 435
45.16. Source tab in Mapping editor. 435
45.17. Content Assist inside element. 436
45.18. Content Assist for ports and fields. 437
46.1. DataBase Configuration 443
46.2. Input mapping 443
46.3. Output mapping 444
46.4. Source Tab of the Transform Editor in the Denormalizer Component (I) 459
46.5. Source Tab of the Transform Editor in the Denormalizer Component (II) 459
46.6. Example MetaPivot Input 483
46.7. Example MetaPivot Output 483
46.8. Source Tab of the Transform Editor in the Normalizer Component (I) 486
46.9. Source Tab of the Transform Editor in the Normalizer Component (II) 486
46.10. Source Tab of the Transform Editor in the Partitioning Component 494
46.11. Source Tab of the Transform Editor in the Rollup Component (I) 507
46.12. Source Tab of the Transform Editor in the Rollup Component (II) 507
46.13. Source Tab of the Transform Editor in the Rollup Component (III) 508
46.14. XSLT Mapping 522
46.15. An Example of Mapping 522
47.1. Matching Key Wizard (Master Key Tab) 527
47.2. Matching Key Wizard (Slave Key Tab) 527
47.3. Join Key Wizard (Master Key Tab) 529
47.4. Join Key Wizard (Slave Key Tab) 529
47.5. An Example of the Join Key Attribute in ApproximativeJoin Component 530
47.6. An Example of the Join Key Attribute in ExtHashJoin Component 538
47.7. Hash Join Key Wizard 539

47.8. An Example of the Join Key Attribute in ExtMergeJoin Component 543
47.9. Join Key Wizard (Master Key Tab) 543
47.10. Join Key Wizard (Slave Key Tab) 544
47.11. Edit Key Wizard 548
47.12. An Example of the Join Key Attribute in the RelationalJoin Component 550
47.13. Join Key Wizard (Master Key Tab) 551
47.14. Join Key Wizard (Slave Key Tab) 551
49.1. Foreign Key Definition Wizard (Foreign Key Tab) 562
49.2. Foreign Key Definition Wizard (Primary Key Tab) 562
49.3. Foreign Key Definition Wizard (Foreign and Primary Keys Assigned) 563

List of Tables

6.1. Sites with CloverETL 17
8.1. Standard Folders and Parameters 24
19.1. Data Types in Metadata 97
19.2. Available date engines 99
19.3. Date Format Pattern Syntax (Java) 100
19.4. Rules for Date Format Usage (Java) 101
19.5. Date and Time Format Patterns and Results (Java) 102
19.6. Date Format Pattern Syntax (Joda) 103
19.7. Rules for Date Format Usage (Joda) 103
19.8. Numeric Format Pattern Syntax 106
19.9. BNF Diagram 107
19.10. Used Notation 107
19.11. Locale-Sensitive Formatting 107
19.12. Numeric Format Patterns and Results 108
19.13. Available Binary Formats 109
19.14. List of all Locale 112
19.15. CloverETL-to-SQL Data Types Transformation Table (Part I) 139
19.16. CloverETL-to-SQL Data Types Transformation Table (Part II) 139
36.1. Transformations Overview 237
37.1. Readers Comparison 251
38.1. Writers Comparison 263
39.1. Transformers Comparison 273
40.1. Joiners Comparison 276
40.2. Functions in Joiners, DataIntersection, and Reformat 279
41.1. Cluster Components Comparison 283
42.1. Others Comparison 284
44.1. Functions in DataGenerator 302
44.2. Error Metadata for Parallel Reader 327
44.3. Error Metadata for QuickBaseRecordReader 330
44.4. Error Metadata for UniversalDataReader 334
45.1. Error Metadata for DB2DataWriter 365
45.2. Error Fields for InformixDataWriter 384
45.3. Error Fields for MSSQLDataWriter 394
45.4. Error Metadata for MySQLDataWriter 397
45.5. Error Fields for QuickBaseImportCSV 407
45.6. Error Fields for QuickBaseRecordWriter 409
46.1. Functions in Denormalizer 459
46.2. Error Fields for EmailFilter 466
46.3. Functions in Normalizer 487
46.4. Functions in Partition (or ClusterPartitioner) 494
46.5. Functions in Rollup 508
49.1. Input Metadata for RunGraph (In-Out Mode) 576
49.2. Output Metadata for RunGraph 576
51.1. CTL Version Comparison 591
51.2. CTL Version Differences 592
53.1. Literals 610
54.1. Literals 672
54.2. National Characters 729

List of Examples

19.1. String Format 111
19.2. Examples of Locale 112
26.1. Canonizing File Paths 194
32.1. Example of usage 209
36.1. Sorting 233
36.2. Example of the Error Actions Attribute 240
44.1. Example State Function 296
44.2. Field Mapping in XLSDataReader 341
44.3. Mapping in XMLExtract 344
44.4. From XML Structure to Mapping Structure 345
44.5. Mapping in XMLXPathReader 356
45.1. Internal Structure of Archived Output File(s) 363
45.2. Examples of Queries 371
45.3. Example of a Control script 401
45.4. Using Expressions in Ports and Fields 426
45.5. Include and Exclude property examples 427
45.6. Attribute value examples 427
45.7. Writing null attribute 429
45.8. Omitting Null Attribute 429
45.9. Hide Element 430
45.10. Partitioning According to Any Element 430
45.11. Writing and omitting blank elements 431
45.12. Binding that serves as JOIN 434
45.13. Insert Wildcard attributes in Source tab 437
46.1. Aggregation Mapping 447
46.2. Join Key for DataIntersection 451
46.3. Key for Denormalizer 457
46.4. Example MetaPivot Transformation 483
46.5. Data Transformation with Pivot - Using Key 500
47.1. Matching Key 527
47.2. Join Key for ApproximativeJoin 530
47.3. Join Key for DBJoin 534
47.4. Slave Part of Join Key for ExtHashJoin 538
47.5. Join Key for ExtHashJoin 539
47.6. Join Key for ExtMergeJoin 544
47.7. Join Key for LookupJoin 548
47.8. Join Key for RelationalJoin 552
49.1. Working with Quoted Command Line Arguments 578
52.1. Example of dictionary usage 598
53.1. Example of CTL1 syntax (Rollup) 605
53.2. Eval() Function Examples 624
53.3. Mapping of Metadata by Name 629
53.4. Example of Mapping with Individual Fields 630
53.5. Example of Mapping with Wild Cards 631
53.6. Example of Mapping with Wild Cards in Separate User-Defined Functions 632
53.7. Example of Successive Mapping in Separate User-Defined Functions 634
54.1. Example of CTL2 syntax (Rollup) 666
54.2. Example of usage of decimal data type in CTL2 669
54.3. Mapping of Metadata by Name (using the copyByName() function) 690
54.4. Mapping of Metadata by Position 691
54.5. Example of Mapping with Individual Fields 692
54.6. Example of Mapping with Wild Cards 692
54.7. Example of Mapping with Wild Cards in Separate User-Defined Functions 693
54.8. Usage of Lookup Table Functions 725