Copyright and disclaimer

Copyright © 2003, 2013, 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.
# Table of Contents

## I. CloverETL Overview
- 1. Family of CloverETL Products ................................................................. 1
  - 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. License Manager .............................................................................. 19
  - CloverETL License dialog ............................................................ 19
  - CloverETL License wizard ............................................................ 20
  - Activation using License key .......................................................... 21
  - Activation online ........................................................................ 23
- 8. Creating CloverETL Projects ............................................................... 26
  - CloverETL Project ....................................................................... 26
  - CloverETL Server Project ............................................................. 27
  - CloverETL Examples Project .......................................................... 29
- 9. Structure of CloverETL Projects ........................................................... 30
  - Standard Structure of All CloverETL Projects .................................. 31
  - Workspace.prm File .................................................................... 32
  - Opening the CloverETL Perspective ................................................. 33
- 10. Appearance of CloverETL Perspective .................................................. 35
  - CloverETL Designer Panes ............................................................ 35
    - Graph Editor with Palette of Components ...................................... 36
    - Navigator Pane ........................................................................ 40
    - Outline Pane ........................................................................... 40
    - Tabs Pane ................................................................................ 42
- 11. Creating CloverETL Graphs ................................................................ 46
  - Creating Empty Graphs .................................................................. 46
  - Creating a Simple Graph in a Few Simple Steps .................................. 50
- 12. Running CloverETL Graphs ................................................................ 60
  - Successful Graph Execution .......................................................... 61
  - Using the Run Configurations Dialog .............................................. 63

## IV. Working with CloverETL Designer ....................................................... 64
- 13. Using Cheat Sheets ........................................................................... 65
- 14. Common Dialogs ................................................................................ 68
  - URL File Dialog .......................................................................... 68
  - Edit Value Dialog ........................................................................ 69
  - Open Type Dialog ........................................................................ 70
Extracting Metadata from a Flat File ................................................................. 135
  Extracting Metadata from Delimited Files ..................................................... 137
  Extracting Metadata from Fixed Length Files ............................................... 139
Extracting Metadata from an XLS(X) File ......................................................... 140
Extracting Metadata from a Database .............................................................. 142
Extracting Metadata from a DBase File ........................................................... 146
Creating Metadata by User ............................................................................... 146
Extracting Metadata from Lotus Notes .............................................................. 146
Merging existing metadata .................................................................................. 148
Dynamic Metadata .............................................................................................. 149
Reading Metadata from Special Sources ............................................................ 150
Creating Database Table from Metadata and Database Connection .................. 151
Metadata Editor .................................................................................................. 153
  Basics of Metadata Editor ................................................................................. 153
  Record Pane ....................................................................................................... 155
  Field Name vs. Label vs. Description ................................................................. 156
  Details Pane ....................................................................................................... 156
Changing and Defining Delimiters .................................................................... 159
  Changing Record Delimiter ............................................................................... 161
  Changing Default Delimiter ............................................................................... 162
  Defining Non-Default Delimiter for a Field .................................................... 162
Editing Metadata in the Source Code ................................................................ 163
Multivalue Fields .................................................................................................. 163
  Lists and Maps Support in Components .......................................................... 164
  Joining on Lists and Maps (Comparison Rules) ............................................... 165
21. Database Connections .................................................................................... 167
  Internal Database Connections ........................................................................ 167
  Creating Internal Database Connections ......................................................... 167
  Externalizing Internal Database Connections ............................................... 168
  Exporting Internal Database Connections ...................................................... 169
  External (Shared) Database Connections ....................................................... 170
  Creating External (Shared) Database Connections ......................................... 170
  Linking External (Shared) Database Connections .......................................... 170
  Internalizing External (Shared) Database Connections .................................. 170
Database Connection Wizard .............................................................................. 171
Encrypting the Access Password ........................................................................ 175
Browsing Database and Extracting Metadata from Database Tables ................ 176
Windows Authentication on Microsoft SQL Server .......................................... 176
Getting the Native Library .................................................................................. 177
Installation ........................................................................................................... 177
22. JMS Connections ........................................................................................... 179
  Internal JMS Connections ............................................................................... 179
  Creating Internal JMS Connections ................................................................. 179
  Externalizing Internal JMS Connections ......................................................... 179
  Exporting Internal JMS Connections ............................................................... 180
  External (Shared) JMS Connections ............................................................... 181
  Creating External (Shared) JMS Connections ............................................... 181
  Linking External (Shared) JMS Connection .................................................... 181
  Internalizing External (Shared) JMS Connections ......................................... 181
Edit JMS Connection Wizard ............................................................................... 182
Encrypting the Authentication Password ......................................................... 183
23. QuickBase Connections ................................................................................. 184
24. Lotus Connections .......................................................................................... 185
25. Lookup Tables ................................................................................................ 186
  Internal Lookup Tables .................................................................................... 187
  Creating Internal Lookup Tables ...................................................................... 187
  Externalizing Internal Lookup Tables ............................................................. 187
  Exporting Internal Lookup Tables ................................................................... 189
<table>
<thead>
<tr>
<th>Component Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VII. Components Overview</td>
<td>250</td>
</tr>
<tr>
<td>35. Jobflow Design Patterns</td>
<td>247</td>
</tr>
<tr>
<td>36. Introduction to Components</td>
<td>251</td>
</tr>
<tr>
<td>37. Palette of Components</td>
<td>252</td>
</tr>
<tr>
<td>38. Find / Add Components</td>
<td>254</td>
</tr>
<tr>
<td>Finding Components</td>
<td>254</td>
</tr>
<tr>
<td>Adding Components</td>
<td>254</td>
</tr>
<tr>
<td>39. Common Properties of All Components</td>
<td>256</td>
</tr>
<tr>
<td>Edit Component Dialog</td>
<td>257</td>
</tr>
<tr>
<td>Component Name</td>
<td>260</td>
</tr>
<tr>
<td>Phases</td>
<td>261</td>
</tr>
<tr>
<td>Enable/Disable Component</td>
<td>262</td>
</tr>
<tr>
<td>PassThrough Mode</td>
<td>263</td>
</tr>
<tr>
<td>40. Common Properties of Most Components</td>
<td>264</td>
</tr>
<tr>
<td>Metadata Templates</td>
<td>264</td>
</tr>
<tr>
<td>Group Key</td>
<td>264</td>
</tr>
<tr>
<td>Sort Key</td>
<td>265</td>
</tr>
<tr>
<td>Defining Transformations</td>
<td>268</td>
</tr>
<tr>
<td>Return Values of Transformations</td>
<td>272</td>
</tr>
<tr>
<td>Error Actions and Error Log (deprecated since 3.0)</td>
<td>274</td>
</tr>
<tr>
<td>Transform Editor</td>
<td>275</td>
</tr>
<tr>
<td>Common Java Interfaces</td>
<td>284</td>
</tr>
<tr>
<td>41. Common Properties of Readers</td>
<td>285</td>
</tr>
<tr>
<td>Supported File URL Formats for Readers</td>
<td>286</td>
</tr>
<tr>
<td>Viewing Data on Readers</td>
<td>290</td>
</tr>
<tr>
<td>Input Port Reading</td>
<td>292</td>
</tr>
<tr>
<td>Incremental Reading</td>
<td>293</td>
</tr>
<tr>
<td>Selecting Input Records</td>
<td>294</td>
</tr>
<tr>
<td>Data Policy</td>
<td>295</td>
</tr>
<tr>
<td>XML Features</td>
<td>296</td>
</tr>
<tr>
<td>CTL Templates for Readers</td>
<td>296</td>
</tr>
<tr>
<td>Java Interfaces for Readers</td>
<td>296</td>
</tr>
<tr>
<td>42. Common Properties of Writers</td>
<td>298</td>
</tr>
<tr>
<td>Supported File URL Formats for Writers</td>
<td>299</td>
</tr>
<tr>
<td>Viewing Data on Writers</td>
<td>303</td>
</tr>
<tr>
<td>Output Port Writing</td>
<td>305</td>
</tr>
<tr>
<td>How and Where Data Should Be Written</td>
<td>305</td>
</tr>
<tr>
<td>Selecting Output Records</td>
<td>306</td>
</tr>
<tr>
<td>Partitioning Output into Different Output Files</td>
<td>306</td>
</tr>
<tr>
<td>Java Interfaces for Writers</td>
<td>308</td>
</tr>
<tr>
<td>43. Common Properties of Transformers</td>
<td>309</td>
</tr>
<tr>
<td>CTL Templates for Transformers</td>
<td>310</td>
</tr>
<tr>
<td>Java Interfaces for Transformers</td>
<td>311</td>
</tr>
<tr>
<td>44. Common Properties of Joiners</td>
<td>312</td>
</tr>
<tr>
<td>Join Types</td>
<td>312</td>
</tr>
<tr>
<td>Slave Duplicates</td>
<td>313</td>
</tr>
<tr>
<td>CTL Templates for Joiners</td>
<td>314</td>
</tr>
<tr>
<td>Java Interfaces for Joiners</td>
<td>317</td>
</tr>
<tr>
<td>45. Common Properties of Cluster Components</td>
<td>319</td>
</tr>
<tr>
<td>CTL Templates for Cluster Components</td>
<td>319</td>
</tr>
<tr>
<td>Java Interfaces for Cluster Components</td>
<td>319</td>
</tr>
<tr>
<td>46. Common Properties of Others</td>
<td>320</td>
</tr>
<tr>
<td>47. Common Properties of Data Quality</td>
<td>321</td>
</tr>
<tr>
<td>48. Common Properties of Job Control</td>
<td>322</td>
</tr>
<tr>
<td>49. Common Properties of File Operations</td>
<td>323</td>
</tr>
<tr>
<td>Supported URL Formats for File Operations</td>
<td>324</td>
</tr>
<tr>
<td>50. Custom Components</td>
<td>326</td>
</tr>
<tr>
<td>VIII. Component Reference</td>
<td>327</td>
</tr>
</tbody>
</table>
51. Readers .................................................................................................................. 328
   CloverDataReader .............................................................................................. 330
   ComplexDataReader ......................................................................................... 332
   DataGenerator ................................................................................................. 340
   DBFDataReader .................................................................................................. 347
   DBInputTable ..................................................................................................... 349
   EmailReader ......................................................................................................... 353
   JavaBeanReader ................................................................................................. 356
   JMSReader ........................................................................................................... 362
   JSONReader ....................................................................................................... 365
   LDAPReader ....................................................................................................... 371
   LotusReader ......................................................................................................... 374
   MultiLevelReader ............................................................................................... 376
   ParallelReader ..................................................................................................... 380
   QuickBaseRecordReader ..................................................................................... 383
   QuickBaseQueryReader ...................................................................................... 385
   SpreadsheetDataReader ..................................................................................... 387
   UniversalDataReader ......................................................................................... 397
   XLSDataReader .................................................................................................... 402
   XMLExtract ........................................................................................................ 406
   XMLReader .......................................................................................................... 423
   XMLXPathReader ............................................................................................... 431

52. Writers .................................................................................................................. 438
   CloverDataWriter .............................................................................................. 440
   DB2DataWriter .................................................................................................. 442
   DBFDataWriter .................................................................................................. 448
   DBOutputTable ................................................................................................... 451
   EmailSender ........................................................................................................ 459
   InfobrightDataWriter ......................................................................................... 463
   InformixDataWriter ......................................................................................... 465
   JavaBeanWriter ................................................................................................. 468
   JavaMapWriter .................................................................................................... 472
   JMSWriter ........................................................................................................... 477
   JSONWriter ......................................................................................................... 480
   LDAPWriter ....................................................................................................... 485
   LotusWriter ......................................................................................................... 487
   MSSQLDataWriter ............................................................................................. 489
   MySQLDataWriter ............................................................................................. 492
   OracleDataWriter ............................................................................................... 495
   PostgreSQLDataWriter ..................................................................................... 499
   QuickBaseImportCSV ....................................................................................... 502
   QuickBaseRecordWriter .................................................................................... 504
   SpreadsheetDataWriter ...................................................................................... 506
   StructuredDataWriter ......................................................................................... 520
   Trash .................................................................................................................. 524
   UniversalDataWriter .......................................................................................... 526
   XLSDataWriter ................................................................................................... 529
   XMLWriter .......................................................................................................... 532

53. Transformers ........................................................................................................ 550
   Aggregate .......................................................................................................... 552
   Concatenate ....................................................................................................... 555
   DataIntersection ................................................................................................. 556
   DataSampler ....................................................................................................... 559
   Dedup ................................................................................................................... 561
   Denormalizer ...................................................................................................... 563
   ExtFilter .............................................................................................................. 572
   ExtSort ................................................................................................................ 574
   FastSort .............................................................................................................. 576
<table>
<thead>
<tr>
<th>54. Joiners</th>
<th>624</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApproximativeJoin</td>
<td>625</td>
</tr>
<tr>
<td>Combine</td>
<td>633</td>
</tr>
<tr>
<td>DBJoin</td>
<td>635</td>
</tr>
<tr>
<td>ExtHashJoin</td>
<td>638</td>
</tr>
<tr>
<td>ExtMergeJoin</td>
<td>644</td>
</tr>
<tr>
<td>LookupJoin</td>
<td>649</td>
</tr>
<tr>
<td>RelationalJoin</td>
<td>652</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>55. Job Control</td>
<td>656</td>
</tr>
<tr>
<td>Barrier</td>
<td>657</td>
</tr>
<tr>
<td>Condition</td>
<td>660</td>
</tr>
<tr>
<td>ExecuteGraph</td>
<td>663</td>
</tr>
<tr>
<td>ExecuteJobflow</td>
<td>669</td>
</tr>
<tr>
<td>ExecuteScript</td>
<td>671</td>
</tr>
<tr>
<td>Fail</td>
<td>677</td>
</tr>
<tr>
<td>GetJobInput</td>
<td>680</td>
</tr>
<tr>
<td>KillGraph</td>
<td>682</td>
</tr>
<tr>
<td>KillJobflow</td>
<td>686</td>
</tr>
<tr>
<td>MonitorGraph</td>
<td>688</td>
</tr>
<tr>
<td>MonitorJobflow</td>
<td>692</td>
</tr>
<tr>
<td>SetJobOutput</td>
<td>694</td>
</tr>
<tr>
<td>Success</td>
<td>696</td>
</tr>
<tr>
<td>TokenGather</td>
<td>698</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>56. File Operations</td>
<td>700</td>
</tr>
<tr>
<td>CopyFiles</td>
<td>701</td>
</tr>
<tr>
<td>CreateFiles</td>
<td>705</td>
</tr>
<tr>
<td>DeleteFiles</td>
<td>708</td>
</tr>
<tr>
<td>ListFiles</td>
<td>711</td>
</tr>
<tr>
<td>MoveFiles</td>
<td>714</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>57. Cluster Components</td>
<td>717</td>
</tr>
<tr>
<td>ClusterGather</td>
<td>718</td>
</tr>
<tr>
<td>ClusterPartitioner</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>58. Data Quality</td>
<td>723</td>
</tr>
<tr>
<td>Address Doctor 5</td>
<td>724</td>
</tr>
<tr>
<td>EmailFilter</td>
<td>728</td>
</tr>
<tr>
<td>ProfilerProbe</td>
<td>733</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>59. Others</td>
<td>738</td>
</tr>
<tr>
<td>CheckForeignKey</td>
<td>739</td>
</tr>
<tr>
<td>DBExecute</td>
<td>743</td>
</tr>
<tr>
<td>HTTPConnector</td>
<td>747</td>
</tr>
<tr>
<td>JavaExecute</td>
<td>752</td>
</tr>
<tr>
<td>LookupTableReaderWriter</td>
<td>754</td>
</tr>
<tr>
<td>RunGraph</td>
<td>756</td>
</tr>
<tr>
<td>SequenceChecker</td>
<td>760</td>
</tr>
<tr>
<td>SpeedLimiter</td>
<td>762</td>
</tr>
<tr>
<td>SystemExecute</td>
<td>764</td>
</tr>
<tr>
<td>WebServiceClient</td>
<td>767</td>
</tr>
<tr>
<td>IX. CTL - CloverETL Transformation Language</td>
<td>771</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>60. Overview</td>
<td>772</td>
</tr>
<tr>
<td>61. CTL1 vs. CTL2 Comparison</td>
<td>774</td>
</tr>
<tr>
<td>Typed Language</td>
<td>774</td>
</tr>
<tr>
<td>Arbitrary Order of Code Parts</td>
<td>774</td>
</tr>
<tr>
<td>Compiled Mode</td>
<td>774</td>
</tr>
<tr>
<td>Access to Graph Elements (Lookups, Sequences, ...)</td>
<td>774</td>
</tr>
<tr>
<td>Metadata</td>
<td>774</td>
</tr>
<tr>
<td>62. Migrating CTL1 to CTL2</td>
<td>778</td>
</tr>
<tr>
<td>63. CTL1 Language Reference</td>
<td>788</td>
</tr>
<tr>
<td>Language Reference</td>
<td>789</td>
</tr>
<tr>
<td>Program Structure</td>
<td>790</td>
</tr>
<tr>
<td>Comments</td>
<td>790</td>
</tr>
<tr>
<td>Import</td>
<td>790</td>
</tr>
<tr>
<td>Data Types in CTL</td>
<td>791</td>
</tr>
<tr>
<td>Literals</td>
<td>793</td>
</tr>
<tr>
<td>Variables</td>
<td>795</td>
</tr>
<tr>
<td>Operators</td>
<td>796</td>
</tr>
<tr>
<td>Simple Statement and Block of Statements</td>
<td>801</td>
</tr>
<tr>
<td>Control Statements</td>
<td>801</td>
</tr>
<tr>
<td>Error Handling</td>
<td>805</td>
</tr>
<tr>
<td>Functions</td>
<td>806</td>
</tr>
<tr>
<td>Eval</td>
<td>807</td>
</tr>
<tr>
<td>Conditional Fail Expression</td>
<td>808</td>
</tr>
<tr>
<td>Accessing Data Records and Fields</td>
<td>809</td>
</tr>
<tr>
<td>Mapping</td>
<td>812</td>
</tr>
<tr>
<td>Parameters</td>
<td>818</td>
</tr>
<tr>
<td>Functions Reference</td>
<td>819</td>
</tr>
<tr>
<td>Conversion Functions</td>
<td>820</td>
</tr>
<tr>
<td>Date Functions</td>
<td>825</td>
</tr>
<tr>
<td>Mathematical Functions</td>
<td>828</td>
</tr>
<tr>
<td>String Functions</td>
<td>832</td>
</tr>
<tr>
<td>Container Functions</td>
<td>840</td>
</tr>
<tr>
<td>Miscellaneous Functions</td>
<td>842</td>
</tr>
<tr>
<td>Dictionary Functions</td>
<td>844</td>
</tr>
<tr>
<td>Lookup Table Functions</td>
<td>845</td>
</tr>
<tr>
<td>Sequence Functions</td>
<td>847</td>
</tr>
<tr>
<td>Custom CTL Functions</td>
<td>848</td>
</tr>
<tr>
<td>64. CTL2 Language Reference</td>
<td>849</td>
</tr>
<tr>
<td>Language Reference</td>
<td>850</td>
</tr>
<tr>
<td>Program Structure</td>
<td>851</td>
</tr>
<tr>
<td>Comments</td>
<td>851</td>
</tr>
<tr>
<td>Import</td>
<td>851</td>
</tr>
<tr>
<td>Data Types in CTL2</td>
<td>852</td>
</tr>
<tr>
<td>Literals</td>
<td>855</td>
</tr>
<tr>
<td>Variables</td>
<td>857</td>
</tr>
<tr>
<td>Dictionary in CTL2</td>
<td>858</td>
</tr>
<tr>
<td>Operators</td>
<td>859</td>
</tr>
<tr>
<td>Simple Statement and Block of Statements</td>
<td>864</td>
</tr>
<tr>
<td>Control Statements</td>
<td>864</td>
</tr>
<tr>
<td>Error Handling</td>
<td>868</td>
</tr>
<tr>
<td>Functions</td>
<td>869</td>
</tr>
<tr>
<td>Conditional Fail Expression</td>
<td>870</td>
</tr>
<tr>
<td>Accessing Data Records and Fields</td>
<td>871</td>
</tr>
<tr>
<td>Mapping</td>
<td>873</td>
</tr>
<tr>
<td>Parameters</td>
<td>877</td>
</tr>
<tr>
<td>Functions Reference</td>
<td>878</td>
</tr>
<tr>
<td>Conversion Functions</td>
<td>880</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Date Functions</td>
<td>887</td>
</tr>
<tr>
<td>Mathematical Functions</td>
<td>889</td>
</tr>
<tr>
<td>String Functions</td>
<td>893</td>
</tr>
<tr>
<td>Container Functions</td>
<td>902</td>
</tr>
<tr>
<td>Record functions (dynamic field access)</td>
<td>906</td>
</tr>
<tr>
<td>Miscellaneous Functions</td>
<td>910</td>
</tr>
<tr>
<td>Lookup Table Functions</td>
<td>914</td>
</tr>
<tr>
<td>Sequence Functions</td>
<td>917</td>
</tr>
<tr>
<td>Custom CTL Functions</td>
<td>918</td>
</tr>
<tr>
<td>CTL2 Appendix - List of National-specific Characters</td>
<td>919</td>
</tr>
<tr>
<td>65. Regular Expressions</td>
<td>921</td>
</tr>
<tr>
<td>List of Figures</td>
<td>922</td>
</tr>
<tr>
<td>List of Tables</td>
<td>929</td>
</tr>
<tr>
<td>List of Examples</td>
<td>930</td>
</tr>
</tbody>
</table>
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**.

![CloverETL Products Diagram](image)

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](http://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.
Chapter 1. Family of CloverETL Products

**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** explaining briefly the basics of **CloverETL Designer** usage.
  
  [www.cloveretl.com/documentation/quickstart](http://www.cloveretl.com/documentation/quickstart)

- **FAQ** concerning various areas of **CloverETL** products.
  
  [www.cloveretl.com/faq](http://www.cloveretl.com/faq)

- **Forum** about details of CloverETL features.
  
  [http://forum.cloveretl.com](http://forum.cloveretl.com)

- **Blog** describing interesting solutions based on **CloverETL** products.
  
  [http://blog.cloveretl.com](http://blog.cloveretl.com)

- **Wiki** page containing the information about **CloverETL** tools.
  
  [http://wiki.cloveretl.com](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](http://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+.

Remember 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. 27) 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. 73)
- Export to CloverETL Server Sandbox (p. 80)


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 a 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:
Chapter 2. Integrating CloverETL Designer with CloverETL Server

- 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:

![CloverETL Server Project Displayed after Opening CloverETL Designer](image)

**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 described 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.

![Prompt to Open CloverETL Server Project](image)

**Figure 2.2. Prompt to Open CloverETL Server Project**

After that, click OK and the CloverETL Server project.
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 \\n  -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
```
Chapter 2. Integrating CloverETL Designer with CloverETL Server

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:

```xml
<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"
    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:

```ini
-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"
    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](image)

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:**
  - Microsoft Windows - none, the installer includes Eclipse Platform 3.6.2 for Java developers with RSE + GEF + Eclipse Web Tools Platform.
  - Mac OS X, Linux - Java 6 Runtime Environment or newer (Java 7 Development Kit is recommended)

**Plugin installation**

- **Software requirements:**
  - Recommended: Eclipse Platform 3.6.2 with RSE + GEF + Eclipse Web Tools Platform. Java 7 Development Kit
  - Eclipse 3.7 is fully supported.

**Important**

For Mac OS users:

Please make sure your default Java system is set to 1.7 or newer. Go to Finder → Applications → Utilities → Java → Java preferences and reorder the available Java installations so that Java 7 is at the top of the list.

**Related Links**

- Eclipse Classic download page - choose a proper version for your OS
  
  http://www.eclipse.org/downloads
Chapter 3. System Requirements
for CloverETL Designer

• JDK download page - Clover supports Java 1.6+
  

• Useful links for understanding the difference between Java Development Kit (JDK) and Java SE Runtime Environment (JRE)
  
  http://docs.oracle.com/javase/7/docs/

  http://www.oracle.com/technetwork/java/javase/webnotes-136672.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 obliged 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

<table>
<thead>
<tr>
<th>OS</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 32 bit</td>
<td>designer.cloveretl.com/update/cloveretl-designer-win32.exe</td>
</tr>
<tr>
<td>Windows 64 bit</td>
<td>designer.cloveretl.com/update/cloveretl-designer-win32-x86_64.exe</td>
</tr>
<tr>
<td>Linux 32 bit</td>
<td>designer.cloveretl.com/update/cloveretl-designer-linux-gtk.tar.gz</td>
</tr>
<tr>
<td>Linux 64 bit</td>
<td>designer.cloveretl.com/update/cloveretl-designer-linux-gtk-x86_64.tar.gz</td>
</tr>
<tr>
<td>Mac OS Cocoa</td>
<td>designer.cloveretl.com/update/cloveretl-designer-macosx-cocoa-x86_64.dmg.zip</td>
</tr>
</tbody>
</table>

Get the Eclipse Plugin

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>online</td>
<td>designer.cloveretl.com/update</td>
</tr>
<tr>
<td>offline</td>
<td>designer.cloveretl.com/update/cloveretl-designer.zip</td>
</tr>
</tbody>
</table>

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
Chapter 4. Downloading CloverETL

<table>
<thead>
<tr>
<th>OS</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 32bit</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32.exe</td>
</tr>
<tr>
<td>Windows 64 bit</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval-win32-x86_64.exe</td>
</tr>
<tr>
<td>Linux 32bit</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk.tar.gz</td>
</tr>
<tr>
<td>Linux 64bit</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval-linux-gtk-x86_64.tar.gz</td>
</tr>
<tr>
<td>Mac OS Cocoa</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval-macosx-cocoa-x86_64.dmg.zip</td>
</tr>
</tbody>
</table>

**Get the Eclipse Plugin**

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>online</td>
<td>designer.cloveretl.com/eval-update</td>
</tr>
<tr>
<td>offline</td>
<td>designer.cloveretl.com/eval-update/cloveretl-designer-eval.zip</td>
</tr>
</tbody>
</table>

See Chapter 6, *Installing Designer as Eclipse plug*in *(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](http://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**

<table>
<thead>
<tr>
<th>OS</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 64 bit</td>
<td>designer.cloveretl.com/community-update/cloveretl-designer-community-win32-x86_64.exe</td>
</tr>
<tr>
<td>Linux 32bit</td>
<td>designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk.tar.gz</td>
</tr>
<tr>
<td>Linux 64bit</td>
<td>designer.cloveretl.com/community-update/cloveretl-designer-community-linux-gtk-x86_64.tar.gz</td>
</tr>
<tr>
<td>Mac OS Cocoa</td>
<td>designer.cloveretl.com/community-update/cloveretl-designer-community-macosx-cocoa-x86_64.dmg.zip</td>
</tr>
</tbody>
</table>

**Get the Eclipse Plugin**

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Download Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>online</td>
<td>designer.cloveretl.com/community-update</td>
</tr>
<tr>
<td>offline</td>
<td>designer.cloveretl.com/community-update/cloveretl-designer-community.zip</td>
</tr>
</tbody>
</table>

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:

![CloverETL Designer Splash Screen](image)

**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)

![Workspace Selection Dialog](image)

**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.
Chapter 5. Starting CloverETL Designer

The first steps with CloverETL Designer are described in Chapter 8, Creating CloverETL Projects (p. 26).

Sooner or later, you might want to get access to product resources online or to manage your Clover licenses. The easiest way is to navigate to help menu as implied on the figure below:
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.

![Available Software](image)

**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**

<table>
<thead>
<tr>
<th>CloverETL Product</th>
<th>Update Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloverETL Desktop Edition</td>
<td>designer.cloveretl.com/update</td>
</tr>
<tr>
<td>CloverETL Desktop Trial Edition</td>
<td>designer.cloveretl.com/eval-update</td>
</tr>
<tr>
<td>CloverETL Community Edition</td>
<td>designer.cloveretl.com/community-update</td>
</tr>
</tbody>
</table>

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. License Manager

This chapter describes how you can add or remove licenses at CloverETL Designer.

License Manager is designed to easily add new licenses and remove or view existing licenses. The manager is accessible in the main menu - select Help → CloverETL → License Manager.

![License Manager](image)

**Important**

License Manager is not accessible in CloverETL Designer Community.

License manager allows you to:

- Browse all available licenses and its details:
  - License number - number of installed license.
  - Company
  - Products - list of licensed products.
  - Expiration - expiration date of the license.
- Open CloverETL License dialog (p. 19) to view all available information about the license.
- Check available license sources. The license sources are shown after clicking on License Manager Info.
- Open CloverETL License wizard (p. 20). New license can be added with the help of this wizard. Click Add New License button to start the process of license activation.
- Delete existing license. Remove button is shown if it is possible to remove activated license. Confirmation is required when deleting license.

**CloverETL License dialog**

CloverETL License dialog shows all available information about the license. License terms are available from this place. It can be opened from License Manager (Chapter 7, License Manager (p. 19))
Note

License Terms needn't to be accessible for some licenses.

CloverETL License wizard

CloverETL License wizard guides you through the process of license activation.
Chapter 7. License Manager

There are two ways how to activate new license:

- **Activation using License key** (p. 21)
  
  Offline activation if you have license key.

- **Activation online** (p. 23)
  
  Online activation if you have license number and password.

**Activation using License key**

The license can be activated using license key. Internet connection isn't necessary for this choice. Following pictures illustrates the process of new license activation:
Figure 7.4. Select Activate using license key radio button and click Next.

Figure 7.5. Enter the path to the license file or copy and paste the license text.
Figure 7.6. Confirm you accept the license agreement and click Finish button.

After these steps the information dialog about successful license activation is shown. Confirm dialog by pressing OK button to finish the process of activation.

Note

The process of new license activation can be terminated whenever before pressing Finish button. Already activated license can be deleted with the help of License Manager.

Activation online

License number and password can be used for online activation of new license. Internet connection is necessary in this case. Following pictures illustrates the process of new license activation:
Chapter 7. License Manager

Figure 7.7. Select Activate online radio button, enter your license number and password and click Next.

**Note**

Error message is shown if entered password or license number is not correct.

Figure 7.8. Confirm you accept the license agreement and click Finish button.

After these steps the information dialog about successful license activation is shown. Confirm dialog by pressing OK button to finish the process of activation.
Note

The process of new license activation can be terminated whenever before pressing Finish button. Already activated license can be deleted with the help of License Manager.
Chapter 8. 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. 26)
  
  It is a local CloverETL project. The whole project structure is on your local computer.

- **CloverETL Server Project** (p. 27)
  
  It is a CloverETL project corresponding to a CloverETL Server sandbox. The whole project structure is on CloverETL Server.

- **CloverETL Examples Project** (p. 29)
  
  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 perspective. 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 8.1. Giving a Name to a CloverETL Project](image)

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:

![CloverETL Server Project Wizard](image)

*Figure 8.2. 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 sandboxes, 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:

![Giving a Name to the New CloverETL Server Project](image)

*Figure 8.3. 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:

![CloverETL Examples Project Wizard](image1)

**Figure 8.4. CloverETL Examples Project Wizard**

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

After clicking **Finish**, the selected local CloverETL Examples 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**.

![Renaming CloverETL Examples Projects](image2)

**Figure 8.5. Renaming CloverETL Examples Projects**
Chapter 9. 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. 26) and CloverETL Examples Project (p. 29), but also for CloverETL Server Project (p. 27).

1. **Standard Structure of All CloverETL Projects** (p. 31)

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

2. **Workspace.prm File** (p. 32)

   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. 33)

   • 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 9.1. Standard Folders and Parameters

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Standard folder</th>
<th>Standard parameter</th>
<th>Parameter usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>all connections</td>
<td>conn</td>
<td>CONN_DIR</td>
<td>${CONN_DIR}</td>
</tr>
<tr>
<td>input data</td>
<td>data-in</td>
<td>DATAIN_DIR</td>
<td>${DATAIN_DIR}</td>
</tr>
<tr>
<td>output data</td>
<td>data-out</td>
<td>DATAOUT_DIR</td>
<td>${DATAOUT_DIR}</td>
</tr>
<tr>
<td>temporary data</td>
<td>data-tmp</td>
<td>DATATMP_DIR</td>
<td>${DATATMP_DIR}</td>
</tr>
<tr>
<td>graphs</td>
<td>graph</td>
<td>GRAPH_DIR</td>
<td>${GRAPH_DIR}</td>
</tr>
<tr>
<td>jobflows (*.jbf)</td>
<td>jobflow</td>
<td>JOBFLOW_DIR</td>
<td>${JOBFLOW_DIR}</td>
</tr>
<tr>
<td>lookup tables</td>
<td>lookup</td>
<td>LOOKUP_DIR</td>
<td>${LOOKUP_DIR}</td>
</tr>
<tr>
<td>profiling jobs (*.cpj)</td>
<td>profile</td>
<td>PROFILE_DIR</td>
<td>${PROFILE_DIR}</td>
</tr>
<tr>
<td>sequences</td>
<td>seq</td>
<td>SEQ_DIR</td>
<td>${SEQ_DIR}</td>
</tr>
<tr>
<td>transformation definitions (both source files and classes)</td>
<td>trans</td>
<td>TRANS_DIR</td>
<td>${TRANS_DIR}</td>
</tr>
</tbody>
</table>

Legend:

1): For more information about parameters, see Chapter 27, Parameters (p. 207), and about their usage, see Using Parameters (p. 215).

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 9.1. Project Folder Structure inside Navigator Pane
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 9.2. 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 9.3. 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 screen, 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 9.4. Basic Eclipse Perspective](image)

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

![Figure 9.5. Selecting CloverETL Perspective](image)
CloverETL perspective will open:

![CloverETL Perspective](image)

*Figure 9.6. CloverETL Perspective*
Chapter 10. Appearance of CloverETL Perspective

The CloverETL perspective consists of 4 panes:

- **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. 36).)

- **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. 40).)

- **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. 40).)

- **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. 42).)

**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. 36)

- **Navigator Pane** (p. 40)
• **Outline Pane** (p. 40)

• **Tabs Pane** (p. 42)

**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**.

![Graph Editor with an Opened Palette of Components](image)

**Figure 10.2. 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:
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.

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 10.5. 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 10.6. A Graph before Selecting Auto-Layout.

Once you have selected the mentioned item, graph could look like this:
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:
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 10.9. 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 collapse 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 10.10. 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.

**Tip**

Activate the Link with Editor yellow icon in the top right corner and every time you select a component in the graph editor Clover will select it in the Outline as well. Although this is convenient for smaller graphs, turn this off for complex graphs to prevent the outline from expanding the big list of components again and again when as you are working in the graph.

Figure 10.11. 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 10.12. 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.

**Locking Elements**

In Outline, you can lock any of the shared graph elements. A lock is just a flag (with an optional text message) that you deliberately assign to an element so that others attempting to modify the element know it might not be a good idea. These graph elements can be locked:

- **Metadata**
- **Connections**
- **Parameters** - only external
- **Sequences**
- **Lookups**

To lock any of these elements, right click it in Outline and click **Lock**.

**Note**

Locks are by no means a security tool - anyone can perform unlock and locks are not owned by users.

In various places (such as the Transform Editor (p. 275)), you are warned if you are accessing a locked element, e.g. modifying locked metadata.

![Metadata locked](image)

*Figure 10.13. Accessing a locked graph element - you can add any text you like to describe the lock.*

**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.
Chapter 10. Appearance of CloverETL Perspective

Figure 10.14. 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 10.15. Console Tab
Chapter 10. Appearance of CloverETL Perspective

- **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 10.16. Problems Tab](image)

- **Clover - Regex Tester tab**

In this tab, you can work with regular expressions (p. 921). 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://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html](http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html)

![Figure 10.17. Clover - Regex Tester Tab](image)
**Clover - Graph tracking** tab

In this tab, you can see a brief description of the graph that ran 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**, **Average waiting records** and **Used memory** 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 10.18. 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 10.19. Clover - Log Tab**
Chapter 11. 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. 46).

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. 50) 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.

When creating a pure ETL graph, mind these two options in the File → New menu:

• Jobflow - creates a *.jbf file similar to an ETL graph. You can fill it with Job Control (p. 656) components. These are meant for executing, monitoring and aborting other graphs and complex workflows. Further reading also at Chapter 48, Common Properties of Job Control (p. 322).

• Profiler Job - creates a new *.cpj file that lets you perform statistical analyses of your data. See the ProfilerProbe (p. 733) component.

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.

Note

Creating a new Jobflow works in a similar way. For Profiler Job, see ProfilerProbe (p. 733).
Chapter 11. Creating CloverETL Graphs

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.

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.
Chapter 11. Creating
CloverETL Graphs

Figure 11.3. 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 11.4. 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 VII, Components Overview (p. 250) for information.
Figure 11.5. Graph Editor with a New Graph and the Palette of Components
Chapter 11. Creating
CloverETL Graphs

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. 93) 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.

![Components Selected from the Palette](image)

Figure 11.6. 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 19, Edges (p. 96) for more information about Edges.)
Figure 11.7. 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.

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:
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 20, Metadata (p. 107) for more information about creating Metadata.)

After doing that, the names of the two fields will be Firstname and Surname, respectively.
Figure 11.11. 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 11.12. 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 11.13. 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. 397) for more information about **UniversalDataReader**.)

Figure 11.14. Opening the Attribute Row

After that, **URL File Dialog** (p. 68) 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 11.15. Selecting the Input File

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

![Image of File URL set in UniversalDataReader](image1)

**Figure 11.16. Input File URL Attribute Has Been Set**

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

Then, double click **UniversalDataWriter**.

(You can see **UniversalDataWriter** (p. 526) 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. 68) will open. Double-click **data-out** folder. Then click **OK**. The **File URL** attribute row will look like this:
Chapter 11. Creating CloverETL Graphs

Figure 11.17. 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 11.18. 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. 574) 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**.
Chapter 11. Creating CloverETL Graphs

Figure 11.19. Defining a Sort Key

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

Figure 11.20. 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 12, Running CloverETL Graphs (p. 60).)
Figure 11.21. 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 11.22. 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 11.23. 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 12. 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 simplest 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.

Tip

To execute a Jobflow (p. 656), follow the same instructions and choose CloverETL Jobflow as the final step. Note that for some job control components, you need to be in the Clover Server environment. Thus, exporting your project to a server sandbox (p. 80) might be necessary.
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. 42) for detailed information.)
Chapter 12. Running
CloverETL Graphs

Figure 12.4. Successful Graph Execution

Figure 12.5. Console Tab with an Overview of the Graph Execution

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

More details about using the Run Configurations dialog can be found in Chapter 17, Advanced Topics (p. 82).
Part IV. Working with CloverETL Designer
Chapter 13. 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 13.1. Selecting Cheat Sheets

After that, the following window will open:

Figure 13.2. The Cheat Sheet Selection Wizard

The two cheat sheets of CloverETL Designer are as follows:
Chapter 13. Using Cheat Sheets

- 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 13.3. CloverETL and Standard Eclipse Commands (Collapsed)](image)

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

![Figure 13.4. CloverETL and Standard Eclipse Commands (Expanded)](image)
Chapter 13. Using Cheat Sheets

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

![CloveETL Designer](image)

**Figure 13.5. CloveETL 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 **CloveETL 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 **CloveETL Designer** window.

![Select a cheat sheet](image)

**Figure 13.6. Locating a Custom Cheat Sheet**
Chapter 14. Common Dialogs

Here we provide the list of the most common dialogs:

- **URL File Dialog** (p. 68)
- **Edit Value Dialog** (p. 69)
- **Open Type Dialog** (p. 70)

**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 14.1. URL File Dialog](image)

- **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 URLs for each of the tabs described above is provided in sections

- **Supported File URL Formats for Readers** (p. 286)
- **Supported File URL Formats for Writers** (p. 299)

---

**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.

![Edit Value Dialog](image)

**Figure 14.2. 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**.

![Find Wizard](image)

**Figure 14.3. 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.

![Go to Line Wizard](image)

**Figure 14.4. 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.

![Open Type Dialog](image)

**Figure 14.5. Open Type Dialog**
Chapter 15. 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.

![Image of the main menu](image1.png)

Figure 15.1. Import (Main Menu)

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

![Image of the context menu](image2.png)

Figure 15.2. Import (Context Menu)

After that, the following window opens. When you expand the Clover ETL category, the window will look like this:
Chapter 15. Import

Figure 15.3. 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:

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.
Chapter 15. Import

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:

![Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server)](image)

**Figure 15.5. 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:

![Import from CloverETL Server Sandbox Wizard (List of Files)](image)

**Figure 15.6. 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:

![Import Graphs Window]

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**.
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 20, Metadata (p. 107) 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 15.8. Import Metadata from XSD](image)

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:

![Image of Import Metadata from DDL window]

Figure 15.9. 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 16. 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 16.1. Export Options](image)

Export Graphs

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

![Figure 16.2. Export Graphs](image)
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:

![Export Graphs to HTML window](image)

Figure 16.3. 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 20, **Metadata** (p. 107) for more information.

![Figure 16.4. Export metadata to XSD](image)

You must select the metadata and specify to which output directory the selected metadata should be exported.
Chapter 16. Export

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:

![Wizard screen](image)

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.

**Note**

Exporting to a partitioned sandbox is not supported. You will get errors because the sandbox location to be affected is not known.
Export Image

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

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.

Figure 16.6. Export Image
Chapter 17. Advanced Topics

As was already described in Using the Run Configurations Dialog (p. 63), 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. 82)
- **Changing Default CloverETL Settings** (p. 85)

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

- **Enlarging the Font of Displayed Numbers** (p. 88)

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. 89)
  - **Setting Java Runtime Environment** (p. 89)
  - **Installing Java Development Kit** (p. 91)

**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.

- **Delete obsolete temp files**
  **NOTE:** Jobflow (p. 656) only.
  Before your jobflow is executed, temp files from older jobflow runs on Clover Server will be deleted. When you execute a graph/jobflow from Designer, the DEBUG mode is always invoked, which is why the temp files are kept on server.

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.

Sensitive data - such as usernames and passwords when reading/writing via (S)FTP - are **not** printed to the log by default. If you need to switch this kind of logging on for a reason, supply a custom `log4j` file which enables `log4j.logger.sensitive`.

---

**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 17.1. Setting Up Memory Size](image)
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 17.2. Custom Clover Settings

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

• Record.RECORD_LIMIT_SIZE = 32 MB
  It limits the maximum size of a record. Theoretically, the limit is tens of MBs, but you should keep it as low as possible for an easier error detection. See Edge Memory Allocation (p. 105) for more details on memory demands.

• Record.FIELD_LIMIT_SIZE = 64 kB
  It limits the maximum size of one field within a record. See Edge Memory Allocation (p. 105) for more details on memory demands.

• Record.RECORD_INITIAL_SIZE
  Sets the initial amount of memory allocated to each record. The memory can grow dynamically up to Record.RECORD_LIMIT_SIZE, depending on how memory-greedy an edge is. See Edge Memory Allocation (p. 105).

• Record.FIELD_INITIAL_SIZE
Sets the initial amount of memory allocated to each field within a record. The memory can grow dynamically up to \texttt{Record.FIELD\_LIMIT\_SIZE}, depending on how memory-greedy an edge is. See \texttt{Edge Memory Allocation} (p. 105).

- \texttt{Record.DEFAULT\_COMPRESSION\_LEVEL=5}

  This sets the compression level for compressed data fields (\texttt{cbyte}).

- \texttt{DEFAULT\_INTERNAL\_IO\_BUFFER\_SIZE = 32768}

  It determines the internal buffer size the components allocate for I/O operations. Increasing this value affects performance negligibly.

- \texttt{USE\_DIRECT\_MEMORY = true}

  The clover engine intensively uses direct memory for data records manipulation. For example underlying memory of CloverBuffer (container for serialised data records) is allocated outside of the Java heap space in direct memory. This attribute is by default \texttt{true} in order to better performance. However, direct memory is out of control java virtual machine, so try to turn off usage of direct memory in case an OutOfMemory exception occurs.

- \texttt{DEFAULT\_DATE\_FORMAT = yyyy-MM-dd}

- \texttt{DEFAULT\_TIME\_FORMAT = HH:mm:ss}

- \texttt{DEFAULT\_DATETIME\_FORMAT = yyyy-MM-dd HH:mm:ss}

- \texttt{DEFAULT\_REGEXP\_TRUE\_STRING = true|T|TRUE|YES|Y|t|1|yes|y}

- \texttt{DEFAULT\_REGEXP\_FALSE\_STRING = false|F|FALSE|NO|N|f|0|no|n}

- \texttt{DataParser.DEFAULT\_CHARSET\_DECODER = ISO-8859-1}

- \texttt{DataFormatter.DEFAULT\_CHARSET\_ENCODER = ISO-8859-1}

- \texttt{Lookup.LOOKUP\_INITIAL\_CAPACITY = 512}

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

- \texttt{DataFieldMetadata.DECIMAL\_LENGTH = 12}

  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.

- \texttt{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.

- \texttt{Record.MAX\_RECORD\_SIZE = 32\ MB}

  \textbf{Note}

  This is a deprecated property. Nowadays, you should use \texttt{Record.RECORD\_LIMIT\_SIZE}.

  It limits the maximum size of a record. Theoretically, the limit is tens of MBs, but you should keep it as low as possible for an easier error detection.

  \textbf{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. 123)
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 17.3. Enlarging the Font of Numbers](image)

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 17.4. Setting the Font Size](image)
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. 89).
- For detailed information about installing JDK see Installing Java Development Kit (p. 91).

### Setting Java Runtime Environment

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

![Setting Java Runtime Environment](image)

*Figure 17.5. Setting The Java Runtime Environment*

After clicking the option, you see the following window:
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 17.6. Preferences Wizard**

**Figure 17.7. 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+.

![Adding a Java Development Kit](image)

*Figure 17.8. 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.
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.
Part V. Graph Elements, Structures and Tools
Chapter 18. 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. 328)
  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. 438)
  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. 550)
  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. 624)
  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.

- **Job Control** (p. 656)
  **Job control** is a group of components focused on execution and monitoring of various job types. These components allow running ETL graphs, jobflows and any interpreted scripts. Graphs and jobflows can be monitored and optionally aborted.

  **Tip**

  Note if you cannot see this component category, navigate to Window → Preferences → CloverETL → Components in Palette and tick both checkboxes next to Job Control.

- **File Operations** (p. 700)
  **File Operations** are components meant for manipulating files on the file system - either local or remote (via FTP). They can also access files in Clover Server sandboxes.

  **Tip**

  Note if you cannot see this component category, navigate to Window → Preferences → CloverETL → Components in Palette and tick both checkboxes next to File Operations.

- **Cluster Components** (p. 717)
  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.

• **Data Quality** (p. 723)

The **Data Quality** is a group of components performing various tasks related to quality of your data - determining information about the data, finding and fixing problems etc.

• **Others** (p. 738)

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. 256)

• Some are common to most of them.

  **Common Properties of Most Components** (p. 264)

• Other properties are common to each of the groups:

  • **Common Properties of Readers** (p. 285)
  • **Common Properties of Writers** (p. 298)
  • **Common Properties of Transformers** (p. 309)
  • **Common Properties of Joiners** (p. 312)
  • **Common Properties of Cluster Components** (p. 319)
  • **Common Properties of Others** (p. 320)
  • **Common Properties of Data Quality** (p. 321)

For information about these common properties see Part VII, **Components Overview** (p. 250).

For information about individual components see Part VIII, **Component Reference** (p. 327).
Chapter 19. 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. 96)
  Each edge must connect two components.

- **Types of Edges** (p. 97)
  Each edge is of one of the four types.

- **Assigning Metadata to the Edges** (p. 98)
  Metadata must be assigned to each edge, describing the data flowing through the edge.

- **Propagating Metadata through the Edges** (p. 99)
  Metadata can be propagated through some components from their input port to their output ports.

- **Colors of the Edges** (p. 99)
  Each edge changes its color upon metadata assignment, edge selection, etc.

- **Debugging the Edges** (p. 100)
  Each edge can be debugged.

- **Edge Memory Allocation** (p. 105)
  Some edges are more memory-greedy than others. This section contains the explanation.

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.

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 for ETL graphs.

• **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. This is the default edge type for jobflows.

• **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 19.2. 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 19.3. Assigning Metadata to an Edge

You can also select a metadata to 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.)
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. 100).
3. View debug data. See Viewing Debug Data (p. 103).
4. Turn off the debug. See Turning Off Debug (p. 105).

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.

- **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.
Chapter 19. Edges

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 19.7. 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**.

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. You 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]**.

Figure 19.8. View Data Dialog

Figure 19.9. Viewing Data on Debugged Edge
Chapter 19. Edges

Note

If records are too big, you will see the [... ] mark indicating some data could not be displayed.

If there are too many records, you will see the Load more... blue text below the grid. Clicking it, a new chunk of records is added behind the currently displayed ones. This is especially useful when observing records while your graph is still running - they are loaded on your click as they are produced by graph's transformations.

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.

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.

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 (p. 921). 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.

![Find Dialog](image)

**Figure 19.12. 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.

![Copy Dialog](image)

**Figure 19.13. 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.

**Edge Memory Allocation**

Manipulating large volumes of data in a single record is always an issue. In **CloverETL Designer**, sending big data along graph edges means this:

• Whenever there is a need to carry many MBs of data between two components in a single record, the edge connecting them expands its capacity. This is referred to as dynamic memory allocation.

• If you have a complicated ETL scenario with some sections transferring huge data then only the edges in these sections will use dynamic memory allocation. The other edges retain low memory requirements.
• An edge which has carried a big record before and allocated more memory for itself will not 'shrink' back again. It consumes bigger amount of memory till your graph execution is finished.

By default, the maximum size of a record sent along an edge is 32 MB. This value can be increased, theoretically, up to tens of MBs by setting the `Record.RECORD_LIMIT_SIZE` property (Changing Default CloverETL Settings (p. 85)). `Record.FIELD_LIMIT_SIZE` can also be 32 MB by default. Naturally, all fields in total cannot use more memory than `Record.RECORD_LIMIT_SIZE`.

There is no harm in increasing `Record.RECORD_LIMIT_SIZE` to whatever size you want. The only reason for keeping it smaller is an early error detection. For instance, if you start appending to a string field and forget to reset record (after each record), the field size can break the limits.

**Note**

Let us look a little deeper into what happens in the memory. Initially, a record starts with 64k of memory allocated to it. If there is a need to transfer huge data, its size can dynamically grow up to the value of `Record.RECORD_LIMIT_SIZE`. If you ever wondered how much memory a record could consume, then the answer is `<64k; Record.RECORD_LIMIT_SIZE>`.

In your ETL graph, edges which are more ‘memory greedy’ look like regular edges. They have no special visual effects.

**Measuring and Estimating Edge Memory Demands**

You can turn on a real-time overview of how much memory your edges and even components consume while a graph is running. To do this, navigate to Window → Preferences. Then you need to expand CloverETL and select Tracking. In the pane, click New..., select Used Memory and confirm. You will get a new column of the Clover - Graph Tracking tab, see Figure 10.18, Clover - Graph Tracking Tab (p. 45).

To estimate how memory-greedy your graph is even before executing it, consult the table below (note: computations are simplified). In general, a graph's memory demands depend on the input data, components used and edge types. In this place, we contribute to understanding the last one. See how much memory approx. your graph takes before its execution and to what extent memory demands can rise:

**Table 19.1. Memory Demands per Edge Type**

<table>
<thead>
<tr>
<th>Edge type</th>
<th>Initial</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>576 kB</td>
<td>9 RIS</td>
</tr>
<tr>
<td>Buffered</td>
<td>1344 kB</td>
<td>21 RIS</td>
</tr>
<tr>
<td>Phase</td>
<td>128 kB</td>
<td>2 RIS</td>
</tr>
<tr>
<td>Direct Fast Propagate</td>
<td>256 kB</td>
<td>4 RIS *</td>
</tr>
</tbody>
</table>

* ... 4 is the number of buffers and it can be changed. In general, buffers' memory can rise up to RLS * (number of buffers)

**Legend:**

RIS = `Record.RECORD_INITIAL_SIZE` = 64 kB (by default)

RLS = `Record.RECORD_LIMIT_SIZE` = 32 MB (by default)
Chapter 20. 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. 108).

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

- Date and Time Format (p. 110)
- Numeric Format (p. 117)
- Locale (p. 123)

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

See Autofilling Functions (p. 128).

Each metadata can be created as:

- **Internal**: See Internal Metadata (p. 130).

  Internal metadata can be:

  - **Externalized**: See Externalizing Internal Metadata (p. 131).
  - **Exported**: See Exporting Internal Metadata (p. 132).

- **External (shared)**: See External (Shared) Metadata (p. 133).

  External (shared) metadata can be:

  - **Linked to the graph**: See Linking External (Shared) Metadata (p. 133).
  - **Internalized**: See Internalizing External (Shared) Metadata (p. 134).

Metadata can be created from:

- **Flat file**: See Extracting Metadata from a Flat File (p. 135).
- **XLS(X) file**: See Extracting Metadata from an XLS(X) File (p. 140).
- **DBase file**: See Extracting Metadata from a DBase File (p. 146).
- **Database**: See Extracting Metadata from a Database (p. 142).
- **By user**: See Creating Metadata by User (p. 146).
- **Lotus Notes**: See Extracting Metadata from Lotus Notes (p. 146).
- **Cobol Copybook**

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

  - **Dynamic metadata**: See Dynamic Metadata (p. 149).
  - **Read from special sources**: See Reading Metadata from Special Sources (p. 150).
**Metadata editor** is described in [Metadata Editor](p. 153).

For detailed information about changing or defining delimiters in delimited or mixed record types see [Changing and Defining Delimiters](p. 159).

Metadata can also be edited in its source code. See [Editing Metadata in the Source Code](p. 163).

Metadata can serve as a source for creating a database table. See [Create Database Table from Metadata](p. 151).

---

### 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. 108)
- [Data Types in CTL](p. 791) for CTL1
- [Data Types in CTL2](p. 852) for CTL2

### Data Types in Metadata

Following are the types of record fields used in metadata:

**Table 20.1. Data Types in Metadata**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size</th>
<th>Range or values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td></td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td>from -128 to 127</td>
<td>null</td>
</tr>
<tr>
<td>cbyte</td>
<td></td>
<td>from -128 to 127</td>
<td>null</td>
</tr>
<tr>
<td>date</td>
<td>64 bits</td>
<td>starts January 1, 1970, 00:00:00 GMT and is incremented by 1 ms.</td>
<td>current date and time</td>
</tr>
<tr>
<td>decimal</td>
<td></td>
<td>decimal(6,2) (They can have values from -9999.99 to 9999.99, length and scale can only be defined in CTL1)</td>
<td>0.00</td>
</tr>
<tr>
<td>integer</td>
<td>32 bits</td>
<td>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.</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>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.</td>
<td>0</td>
</tr>
</tbody>
</table>
### Data type | Size | Range or values | Default value |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number</strong></td>
<td>64 bits</td>
<td>Negative values are from (-2^{-52}, 2^{1023}) to (-2^{-1074}), another value is 0, and positive values are from (2^{-1074}) to (2^{-2^{-52}}). Three special values: NaN, -Infinity, and Infinity are defined.</td>
<td>0.0</td>
</tr>
</tbody>
</table>
| **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 \(\text{MAX\_RECORD\_SIZE}\) of bytes, see Chapter 17, Advanced Topics (p. 82). | \(\text{null}^{4})\)

**Legend:**

1): Any date can be parsed and formatted using date and time format pattern. See Date and Time Format (p. 110). Parsing and formatting can also be influenced by locale. See Locale (p. 123).

2): Any numeric data type can be parsed and formatted using numeric format pattern. See Numeric Format (p. 117). Parsing and formatting may also be influenced by locale. See Locale (p. 123).

3): The default \(\text{length}\) and \(\text{scale}\) of a decimal are \(12\) 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 \(\text{null}\) 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 \(\text{MAX\_RECORD\_SIZE}\) property (the maximum size of a record in bytes, see Chapter 17, Advanced Topics (p. 82)). 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. 791) for CTL1 or Data Types in CTL2 (p. 852) 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 Date and Time Format (p. 110).

   Parsing and formatting can also be influenced by locale (names of months, order of day or month information, etc.). See Locale (p. 123).

2. Any numeric data type (decimal, integer, long, number) can be parsed and/or formatted using numeric format pattern. See Numeric Format (p. 117).

   Parsing and formatting can also be influenced by locale (e.g., decimal dot or decimal comma, etc.). See Locale (p. 123).

3. Any boolean data type can be parsed and formatted using boolean format pattern. See Boolean Format (p. 121).

4. Any string data type can be parsed using string format pattern. See String Format (p. 122).

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. 85).

Date 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 20.2. Available date engines

<table>
<thead>
<tr>
<th>Date engine</th>
<th>Prefix</th>
<th>Default</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>java:</td>
<td>yes - when no prefix is given</td>
<td>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.</td>
<td>java:yyyy-MM-dd HH:mm:ss</td>
</tr>
</tbody>
</table>
Chapter 20. Metadata

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

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 20.3. Date Format Pattern Syntax (Java)*

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

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

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

Examples of date format patterns and resulting dates follow:

**Table 20.5. Date and Time Format Patterns and Results (Java)**

<table>
<thead>
<tr>
<th>Date and Time Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;yyyy.MM.dd G 'at' HH:mm:ss z&quot;</td>
<td>2001.07.04 AD at 12:08:56 PDT</td>
</tr>
<tr>
<td>&quot;EEE, MMM d, &quot;yy&quot;</td>
<td>Wed, Jul 4, '01</td>
</tr>
<tr>
<td>&quot;h:mm a&quot;</td>
<td>12:08 PM</td>
</tr>
<tr>
<td>&quot;hh 'o''clock' a, zzzz&quot;</td>
<td>12 o'clock PM, Pacific Daylight Time</td>
</tr>
<tr>
<td>&quot;K:mm a, z&quot;</td>
<td>0:08 PM, PDT</td>
</tr>
<tr>
<td>&quot;yyyyy.MMMMM.dd GGG hh:mm aaa&quot;</td>
<td>02001.July.04 AD 12:08 PM</td>
</tr>
<tr>
<td>&quot;EEE, d MMM yyyy HH:mm:ss Z&quot;</td>
<td>Wed, 4 Jul 2001 12:08:56 –0700</td>
</tr>
<tr>
<td>&quot;yyMMddHHmmssz&quot;</td>
<td>010704120856-0700</td>
</tr>
<tr>
<td>&quot;yyyy-MM-dd'T'HH:mm:ss.SSSZ&quot;</td>
<td>2001-07-04T12:08:56.235-0700</td>
</tr>
</tbody>
</table>

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 20.6. Date Format Pattern Syntax (Joda)**

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

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

**Table 20.7. Rules for Date Format Usage (Joda)**

<table>
<thead>
<tr>
<th>Presentation</th>
<th>Processing</th>
<th>Number of Pattern Letters</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Formatting</td>
<td>1 - 3</td>
<td>short or abbreviated form, if one exists</td>
</tr>
<tr>
<td>Text</td>
<td>Formatting</td>
<td>&gt;= 4</td>
<td>full form</td>
</tr>
<tr>
<td>Text</td>
<td>Parsing</td>
<td>&gt;= 1</td>
<td>both forms</td>
</tr>
<tr>
<td>Year</td>
<td>Formatting</td>
<td>2</td>
<td>truncated to 2 digits</td>
</tr>
<tr>
<td>Year</td>
<td>Formatting</td>
<td>1 or &gt;= 3</td>
<td>interpreted as Number.</td>
</tr>
<tr>
<td>Year</td>
<td>Parsing</td>
<td>&gt;= 1</td>
<td>interpreted literally</td>
</tr>
<tr>
<td>Presentation</td>
<td>Processing</td>
<td>Number of Pattern Letters</td>
<td>Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>---------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>Month</td>
<td>Both</td>
<td>1-2</td>
<td>interpreted as a Number</td>
</tr>
<tr>
<td>Month</td>
<td>Parsing</td>
<td>&gt;= 3</td>
<td>interpreted as Text (using Roman numbers, abbreviated month name - if exists, or full month name)</td>
</tr>
<tr>
<td>Month</td>
<td>Formatting</td>
<td>3</td>
<td>interpreted as Text (using Roman numbers, or abbreviated month name - if exists)</td>
</tr>
<tr>
<td>Month</td>
<td>Formatting</td>
<td>&gt;= 4</td>
<td>interpreted as Text (full month name)</td>
</tr>
<tr>
<td>Number</td>
<td>Formatting</td>
<td>minimum number of required digits</td>
<td>shorter numbers are padded with zeros</td>
</tr>
<tr>
<td>Number</td>
<td>Parsing</td>
<td>&gt;= 1</td>
<td>any form</td>
</tr>
<tr>
<td>Zone name</td>
<td>Formatting</td>
<td>1-3</td>
<td>short or abbreviated form</td>
</tr>
<tr>
<td>Zone name</td>
<td>Formatting</td>
<td>&gt;= 4</td>
<td>full form</td>
</tr>
<tr>
<td>Time zone offset/id</td>
<td>Formatting</td>
<td>1</td>
<td>Offset without a colon between hours and minutes</td>
</tr>
<tr>
<td>Time zone offset/id</td>
<td>Formatting</td>
<td>2</td>
<td>Offset with a colon between hours and minutes</td>
</tr>
<tr>
<td>Time zone offset/id</td>
<td>Formatting</td>
<td>&gt;= 3</td>
<td>Full textual form like this: &quot;Continent/City&quot;</td>
</tr>
<tr>
<td>Time zone offset/id</td>
<td>Parsing</td>
<td>1</td>
<td>Offset without a colon between hours and minutes</td>
</tr>
<tr>
<td>Time zone offset/id</td>
<td>Parsing</td>
<td>2</td>
<td>Offset with a colon between hours and minutes</td>
</tr>
</tbody>
</table>

**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. 108)
- **For CTL1:**  
  [Data Types in CTL](p. 791)
- **For CTL2:**  
  [Data Types in CTL2](p. 852)

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

**For CTL1:**

- [Conversion Functions](p. 820)
- [Date Functions](p. 825)
• **String Functions** (p. 832)

**For CTL2:**

• **Conversion Functions** (p. 880)
• **Date Functions** (p. 887)
• **String Functions** (p. 893)
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 20.8. Numeric Format Pattern Syntax

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Location</th>
<th>Localized?</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Number</td>
<td>Yes</td>
<td>Digit, zero shows as absent</td>
</tr>
<tr>
<td>0</td>
<td>Number</td>
<td>Yes</td>
<td>Digit</td>
</tr>
<tr>
<td>.</td>
<td>Number</td>
<td>Yes</td>
<td>Decimal separator or monetary decimal separator</td>
</tr>
<tr>
<td>-</td>
<td>Number</td>
<td>Yes</td>
<td>Minus sign</td>
</tr>
<tr>
<td>,</td>
<td>Number</td>
<td>Yes</td>
<td>Grouping separator</td>
</tr>
<tr>
<td>E</td>
<td>Number</td>
<td>Yes</td>
<td>Separates mantissa and exponent in scientific notation. Need not be quoted in prefix or suffix.</td>
</tr>
<tr>
<td>;</td>
<td>Subpattern boundary</td>
<td>Yes</td>
<td>Separates positive and negative subpatterns</td>
</tr>
<tr>
<td>%</td>
<td>Prefix or suffix</td>
<td>Yes</td>
<td>Multiply by 100 and show as percentage</td>
</tr>
<tr>
<td>%e (\u2030)</td>
<td>Prefix or suffix</td>
<td>Yes</td>
<td>Multiply by 1000 and show as per mille value</td>
</tr>
<tr>
<td>\u00A4 (\u00A4)</td>
<td>Prefix or suffix</td>
<td>No</td>
<td>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.</td>
</tr>
<tr>
<td>'</td>
<td>Prefix or suffix</td>
<td>No</td>
<td>Used to quote special characters in a prefix or suffix, for example, &quot;#&quot; formats 123 to &quot;#123&quot;. To create a single quote itself, use two in a row: &quot;:#&quot;</td>
</tr>
</tbody>
</table>

• 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 20.9. BNF Diagram

<table>
<thead>
<tr>
<th>Format</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>subpattern{subpattern}</td>
</tr>
<tr>
<td>subpattern</td>
<td>(prefix)integer{.fraction}{suffix}</td>
</tr>
<tr>
<td>prefix</td>
<td>\u0000..'\uFFFD' - specialCharacters</td>
</tr>
<tr>
<td>suffix</td>
<td>\u0000..'\uFFFD' - specialCharacters</td>
</tr>
<tr>
<td>integer</td>
<td>#<em>'0'</em> '0'</td>
</tr>
<tr>
<td>fraction</td>
<td>'0'* '#*'</td>
</tr>
</tbody>
</table>

Explanation of these symbols follow:

Table 20.10. Used Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X*</td>
<td>0 or more instances of X</td>
</tr>
<tr>
<td>(X</td>
<td>Y)</td>
</tr>
<tr>
<td>X..Y</td>
<td>any character from X up to Y, inclusive</td>
</tr>
<tr>
<td>S - T</td>
<td>characters in S, except those in T</td>
</tr>
<tr>
<td>{X}</td>
<td>X is optional</td>
</tr>
</tbody>
</table>

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 sensitive. See the following table in which results are different for different locales:

Table 20.11. Locale-Sensitive Formatting

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Locale</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>###,###,###</td>
<td>en.US</td>
<td>123,456,789</td>
</tr>
<tr>
<td>###,###,###</td>
<td>de.DE</td>
<td>123,456,789</td>
</tr>
<tr>
<td>###,###,###</td>
<td>fr.FR</td>
<td>123,456,789</td>
</tr>
</tbody>
</table>

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 $1.234 \times 10^3$.

The mantissa is often in the range $1.0 \leq x < 10.0$, 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 20.12. Numeric Format Patterns and Results

<table>
<thead>
<tr>
<th>Value</th>
<th>Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>0.###E0</td>
<td>1.234E3</td>
</tr>
<tr>
<td>12345</td>
<td>###0.#######E0$^1$</td>
<td>12.345E3</td>
</tr>
<tr>
<td>123456</td>
<td>###0.#######E0$^1$</td>
<td>123.456E3</td>
</tr>
<tr>
<td>1234567</td>
<td>###0.#######E0$^1$</td>
<td>1.234567E6</td>
</tr>
<tr>
<td>1234</td>
<td>#0.######E0$^2$</td>
<td>1.2345E4</td>
</tr>
<tr>
<td>123456</td>
<td>#0.######E0$^2$</td>
<td>12.3456E4</td>
</tr>
<tr>
<td>1234567</td>
<td>#0.######E0$^2$</td>
<td>1.234567E6</td>
</tr>
<tr>
<td>0.00123</td>
<td>00.#####E0$^3$</td>
<td>12.3E-4</td>
</tr>
<tr>
<td>123456</td>
<td>###0.###E0$^4$</td>
<td>12.346E3</td>
</tr>
</tbody>
</table>

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 multiple 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 multiple 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 20.13. Available Binary Formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Format</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>BIG_ENDIAN</td>
<td>two's-complement, big-endian</td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td>LITTLE_ENDIAN</td>
<td>two's-complement, little-endian</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PACKED_DECIMAL</td>
<td>packed decimal</td>
<td></td>
</tr>
<tr>
<td>floating-point</td>
<td>DOUBLE_BIG_ENDIAN</td>
<td>IEEE 754, big-endian</td>
<td>8 bytes</td>
</tr>
<tr>
<td></td>
<td>DOUBLE_LITTLE_ENDIAN</td>
<td>IEEE 754, little-endian</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT_BIG_ENDIAN</td>
<td>IEEE 754, big-endian</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>FLOAT_LITTLE_ENDIAN</td>
<td>IEEE 754, little-endian</td>
<td></td>
</tr>
</tbody>
</table>

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. 332) 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/

the 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 (p. 921) that allows or prohibits parsing of a string.

Example 20.1. 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. 123) for more information.

Strings can also be influenced by **Locale sensitivity**. See **Locale Sensitivity** (p. 127).

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. 85)

- 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 20.2. 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. 118).

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 20.14. List of all Locale**

<table>
<thead>
<tr>
<th>Locale code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[system default]</td>
<td>Locale determined by your OS</td>
</tr>
<tr>
<td>ar</td>
<td>Arabic language</td>
</tr>
<tr>
<td>ar.AE</td>
<td>Arabic - United Arab Emirates</td>
</tr>
<tr>
<td>ar.BH</td>
<td>Arabic - Bahrain</td>
</tr>
<tr>
<td>ar.DZ</td>
<td>Arabic - Algeria</td>
</tr>
<tr>
<td>Locale code</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>ar.EG</td>
<td>Arabic - Egypt</td>
</tr>
<tr>
<td>ar.IQ</td>
<td>Arabic - Iraq</td>
</tr>
<tr>
<td>ar.JO</td>
<td>Arabic - Jordan</td>
</tr>
<tr>
<td>ar.KW</td>
<td>Arabic - Kuwait</td>
</tr>
<tr>
<td>ar.LB</td>
<td>Arabic - Lebanon</td>
</tr>
<tr>
<td>ar.LY</td>
<td>Arabic - Lybia</td>
</tr>
<tr>
<td>ar.MA</td>
<td>Arabic - Morrocco</td>
</tr>
<tr>
<td>ar.OM</td>
<td>Arabic - Oman</td>
</tr>
<tr>
<td>ar.QA</td>
<td>Arabic - Qatar</td>
</tr>
<tr>
<td>ar.SA</td>
<td>Arabic - Saudi Arabia</td>
</tr>
<tr>
<td>ar.SD</td>
<td>Arabic - Sudan</td>
</tr>
<tr>
<td>ar.SY</td>
<td>Arabic - Syrian Arab Republic</td>
</tr>
<tr>
<td>ar.TN</td>
<td>Arabic - Tunisia</td>
</tr>
<tr>
<td>ar.YE</td>
<td>Arabic - Yemen</td>
</tr>
<tr>
<td>be</td>
<td>Byelorussian language</td>
</tr>
<tr>
<td>be.BY</td>
<td>Byelorussian - Belarus</td>
</tr>
<tr>
<td>bg</td>
<td>Bulgarian language</td>
</tr>
<tr>
<td>bg.BG</td>
<td>Bulgarian - Bulgaria</td>
</tr>
<tr>
<td>ca</td>
<td>Catalan language</td>
</tr>
<tr>
<td>ca.ES</td>
<td>Catalan - Spain</td>
</tr>
<tr>
<td>cs</td>
<td>Czech language</td>
</tr>
<tr>
<td>cs.CZ</td>
<td>Czech - Czech Republic</td>
</tr>
<tr>
<td>da</td>
<td>Danish language</td>
</tr>
<tr>
<td>da.DK</td>
<td>Danish - Denmark</td>
</tr>
<tr>
<td>de</td>
<td>German language</td>
</tr>
<tr>
<td>de.AT</td>
<td>German - Austria</td>
</tr>
<tr>
<td>de.CH</td>
<td>German - Switzerland</td>
</tr>
<tr>
<td>de.DE</td>
<td>German - Germany</td>
</tr>
<tr>
<td>de.LU</td>
<td>German - Luxembourg</td>
</tr>
<tr>
<td>el</td>
<td>Greek language</td>
</tr>
<tr>
<td>el.CY</td>
<td>Greek - Cyprus</td>
</tr>
<tr>
<td>el.GR</td>
<td>Greek - Greece</td>
</tr>
<tr>
<td>en</td>
<td>English language</td>
</tr>
<tr>
<td>en.AU</td>
<td>English - Australia</td>
</tr>
<tr>
<td>en.CA</td>
<td>English - Canada</td>
</tr>
<tr>
<td>en.GB</td>
<td>English - Great Britain</td>
</tr>
<tr>
<td>en.IE</td>
<td>English - Ireland</td>
</tr>
<tr>
<td>en.IN</td>
<td>English - India</td>
</tr>
<tr>
<td>en.MT</td>
<td>English - Malta</td>
</tr>
<tr>
<td>en.NZ</td>
<td>English - New Zealand</td>
</tr>
<tr>
<td>Locale code</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>en.PH</td>
<td>English - Philippines</td>
</tr>
<tr>
<td>en.SG</td>
<td>English - Singapore</td>
</tr>
<tr>
<td>en.US</td>
<td>English - United States</td>
</tr>
<tr>
<td>en.ZA</td>
<td>English - South Africa</td>
</tr>
<tr>
<td>es</td>
<td>Spanish language</td>
</tr>
<tr>
<td>es.AR</td>
<td>Spanish - Argentina</td>
</tr>
<tr>
<td>es.BO</td>
<td>Spanish - Bolivia</td>
</tr>
<tr>
<td>es.CL</td>
<td>Spanish - Chile</td>
</tr>
<tr>
<td>es.CO</td>
<td>Spanish - Colombia</td>
</tr>
<tr>
<td>es.CR</td>
<td>Spanish - Costa Rica</td>
</tr>
<tr>
<td>es.DO</td>
<td>Spanish - Dominican Republic</td>
</tr>
<tr>
<td>es.EC</td>
<td>Spanish - Ecuador</td>
</tr>
<tr>
<td>es.ES</td>
<td>Spanish - Spain</td>
</tr>
<tr>
<td>es.GT</td>
<td>Spanish - Guatemala</td>
</tr>
<tr>
<td>es.HN</td>
<td>Spanish - Honduras</td>
</tr>
<tr>
<td>es.MX</td>
<td>Spanish - Mexico</td>
</tr>
<tr>
<td>es.NI</td>
<td>Spanish - Nicaragua</td>
</tr>
<tr>
<td>es.PA</td>
<td>Spanish - Panama</td>
</tr>
<tr>
<td>es.PR</td>
<td>Spanish - Puerto Rico</td>
</tr>
<tr>
<td>es.PY</td>
<td>Spanish - Paraguay</td>
</tr>
<tr>
<td>es.US</td>
<td>Spanish - United States</td>
</tr>
<tr>
<td>es.UY</td>
<td>Spanish - Uruguay</td>
</tr>
<tr>
<td>es.VE</td>
<td>Spanish - Venezuela</td>
</tr>
<tr>
<td>et</td>
<td>Estonian language</td>
</tr>
<tr>
<td>et.EE</td>
<td>Estonian - Estonia</td>
</tr>
<tr>
<td>fi</td>
<td>Finnish language</td>
</tr>
<tr>
<td>fi.FI</td>
<td>Finnish - Finland</td>
</tr>
<tr>
<td>fr</td>
<td>French language</td>
</tr>
<tr>
<td>fr.BE</td>
<td>French - Belgium</td>
</tr>
<tr>
<td>fr.CA</td>
<td>French - Canada</td>
</tr>
<tr>
<td>fr.CH</td>
<td>French - Switzerland</td>
</tr>
<tr>
<td>fr.FR</td>
<td>French - France</td>
</tr>
<tr>
<td>fr.LU</td>
<td>French - Luxembourg</td>
</tr>
<tr>
<td>ga</td>
<td>Irish language</td>
</tr>
<tr>
<td>ga.IE</td>
<td>Irish - Ireland</td>
</tr>
<tr>
<td>he</td>
<td>Hebrew language</td>
</tr>
<tr>
<td>he.IL</td>
<td>Hebrew - Israel</td>
</tr>
<tr>
<td>hi.IN</td>
<td>Hindi - India</td>
</tr>
<tr>
<td>hr</td>
<td>Croatian language</td>
</tr>
<tr>
<td>hr.HR</td>
<td>Croatian - Croatia</td>
</tr>
<tr>
<td>Locale code</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>id</td>
<td>Indonesian language</td>
</tr>
<tr>
<td>id.ID</td>
<td>Indonesian - Indonesia</td>
</tr>
<tr>
<td>is</td>
<td>Icelandic language</td>
</tr>
<tr>
<td>is.IS</td>
<td>Icelandic - Iceland</td>
</tr>
<tr>
<td>it</td>
<td>Italian language</td>
</tr>
<tr>
<td>it.CH</td>
<td>Italian - Switzerland</td>
</tr>
<tr>
<td>it.IT</td>
<td>Italian - Italy</td>
</tr>
<tr>
<td>iw</td>
<td>Hebrew language</td>
</tr>
<tr>
<td>iw.IL</td>
<td>Hebrew - Israel</td>
</tr>
<tr>
<td>ja</td>
<td>Japanese language</td>
</tr>
<tr>
<td>ja.JP</td>
<td>Japanese - Japan</td>
</tr>
<tr>
<td>ko</td>
<td>Korean language</td>
</tr>
<tr>
<td>ko.KR</td>
<td>Korean - Republic of Korea</td>
</tr>
<tr>
<td>lt</td>
<td>Lithuanian language</td>
</tr>
<tr>
<td>lt.LT</td>
<td>Lithuanian language - Lithuania</td>
</tr>
<tr>
<td>lv</td>
<td>Latvian language</td>
</tr>
<tr>
<td>lv.LV</td>
<td>Latvian language - Latvia</td>
</tr>
<tr>
<td>mk</td>
<td>Macedonian language</td>
</tr>
<tr>
<td>mk.MK</td>
<td>Macedonian - The Former Yugoslav Republic of Macedonia</td>
</tr>
<tr>
<td>ms</td>
<td>Malay language</td>
</tr>
<tr>
<td>ms.MY</td>
<td>Malay - Burmese</td>
</tr>
<tr>
<td>mt</td>
<td>Maltese language</td>
</tr>
<tr>
<td>mt.MT</td>
<td>Maltese - Malta</td>
</tr>
<tr>
<td>nl</td>
<td>Dutch language</td>
</tr>
<tr>
<td>nl.BE</td>
<td>Dutch - Belgium</td>
</tr>
<tr>
<td>nl.NL</td>
<td>Dutch - Netherlands</td>
</tr>
<tr>
<td>no</td>
<td>Norwegian language</td>
</tr>
<tr>
<td>no.NO</td>
<td>Norwegian - Norway</td>
</tr>
<tr>
<td>pl</td>
<td>Polish language</td>
</tr>
<tr>
<td>pl.PL</td>
<td>Polish - Poland</td>
</tr>
<tr>
<td>pt</td>
<td>Portuguese language</td>
</tr>
<tr>
<td>pt.BR</td>
<td>Portuguese - Brazil</td>
</tr>
<tr>
<td>pt.PT</td>
<td>Portuguese - Portugal</td>
</tr>
<tr>
<td>ro</td>
<td>Romanian language</td>
</tr>
<tr>
<td>ro.RO</td>
<td>Romanian - Romany</td>
</tr>
<tr>
<td>ru</td>
<td>Russian language</td>
</tr>
<tr>
<td>ru.RU</td>
<td>Russian - Russian Federation</td>
</tr>
<tr>
<td>sk</td>
<td>Slovak language</td>
</tr>
<tr>
<td>sk.SK</td>
<td>Slovak - Slovakia</td>
</tr>
<tr>
<td>sl</td>
<td>Slovenian language</td>
</tr>
<tr>
<td>Locale code</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>sl.SI</td>
<td>Slovenian - Slovenia</td>
</tr>
<tr>
<td>sq</td>
<td>Albanian language</td>
</tr>
<tr>
<td>sq.AL</td>
<td>Albanian - Albania</td>
</tr>
<tr>
<td>sr</td>
<td>Serbian language</td>
</tr>
<tr>
<td>sr.BA</td>
<td>Serbian - Bosnia and Herzegovina</td>
</tr>
<tr>
<td>sr.CS</td>
<td>Serbian - Serbia and Montenegro</td>
</tr>
<tr>
<td>sr.ME</td>
<td>Serbian - Serbia (Cyrillic, Montenegro)</td>
</tr>
<tr>
<td>sr.RS</td>
<td>Serbian - Serbia (Latin, Serbia)</td>
</tr>
<tr>
<td>sv</td>
<td>Swedish language</td>
</tr>
<tr>
<td>sv.SE</td>
<td>Swedish - Sweden</td>
</tr>
<tr>
<td>th</td>
<td>Thai language</td>
</tr>
<tr>
<td>th.TH</td>
<td>Thai - Thailand</td>
</tr>
<tr>
<td>tr</td>
<td>Turkish language</td>
</tr>
<tr>
<td>tr.TR</td>
<td>Turkish - Turkey</td>
</tr>
<tr>
<td>uk</td>
<td>Ukrainian language</td>
</tr>
<tr>
<td>uk.UA</td>
<td>Ukrainian - Ukraine</td>
</tr>
<tr>
<td>vi.VN</td>
<td>Vietnamese - Vietnam</td>
</tr>
<tr>
<td>zh</td>
<td>Chinese language</td>
</tr>
<tr>
<td>zh.CN</td>
<td>Chinese - China</td>
</tr>
<tr>
<td>zh.HK</td>
<td>Chinese - Hong Kong</td>
</tr>
<tr>
<td>zh.SG</td>
<td>Chinese - Singapore</td>
</tr>
<tr>
<td>zh.TW</td>
<td>Chinese - Taiwan</td>
</tr>
</tbody>
</table>

**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

There is a set of functions you can use to fill records with some special, pre-defined values (e.g. name of the file you are reading, size of the data source etc.). These functions are available in Metadata editor → Details pane → Advanced properties

The following functions are supported by most Readers, 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.

- **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(X) file from which data records are read. It can be used by **SpreadsheetDataReader** and **XLSDataReader** components 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 20.1. Creating Internal Metadata in the Outline Pane
Figure 20.2. 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.
Chapter 20. Metadata

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 20.3. Externalizing and/or Exporting Internal Metadata

Figure 20.4. 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 → 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 20.5. 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 metadata 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 20.6. Internalizing External (Shared) Metadata](image-url)
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 20.7. Extracting Metadata from Delimited 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. 153). 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, 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. 153).

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. 108).

This third pane is different between Delimited and Fixed Length files. See: Chapter 20. Metadata


- **Extracting Metadata from Delimited Files** (p. 137)
- **Extracting Metadata from Fixed Length Files** (p. 139)

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 20.9. 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 get 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" - 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 20.10. Setting Up Fixed Length Metadata](image)

You must specify the sizes of each field (Resize). You may also want to split any column, merge columns, add one or more columns, remove columns. You can change the sizes by moving the borders of the columns.
Chapter 20. Metadata

Extracting Metadata from an XLS(X) File

If you want to extract metadata from an XLS(X) file, right-click Metadata (in Outline) and select New → Extract from XLS(X) file.

Tip

Equally, you can drag an XLS file from the Navigator area and drop it on Metadata in the Outline. This will also bring the extracting wizard described below.

![Extract metadata from an Excel XLS(X) spreadsheet](image)

**Figure 20.11. Extract Metadata from Excel Spreadsheet Wizard**

In this wizard:

- **Browse** for the desired XLS file and click OK.

- Decide about the orientation of the source data. In Properties → Global → Orientation you can switch between Vertical processing (row by row) or Horizontal processing (column by column).

- Select cells representing the header of your data. You can do that by clicking a whole Excel row/column, clicking and drawing a selection area, Ctrl-clicking or Shift-clicking cells just like you would do in Excel. By default, the first row is selected.

- Click Mark selection as fields. Cells you have selected will change colour and will be considered metadata fields from now on. If you change your mind, click a selected cell and click Clear to not extract metadata from it.

- For each field, you need to specify a cell providing a sample value. The wizard then derives the corresponding metadata type from it. By default, a cell just underneath a marked cell is selected (notice its dashed border), see below. In the figure, ‘Percent’ will become the field name while ‘10,00%’ determines the field type (which would be Long in this case). To change the area where sample values are taken from, adjust Data offset (more on that below).

As for colours: orange cells form the header, yellow ones indicate the beginning of the area data is taken from.

Optional tasks you can do in this dialog:
• Type in Encryption password if the source file is locked. Be sure to type the password exactly as it should be, including correct letter case or special characters.

• Data contains headers - cells marked for field extraction will be considered headers. Data type and format is extracted from cells below the marked ones - with respect to the current Data offset.

• Extract formats - for each field, its Format property will get populated with a pattern corresponding to the sample data. This format pattern will appear in the next step of the wizard, in Property → Advanced → Format as e.g. #0.00%. See Numeric Format (p. 117) for more information.

Figure 20.12. Format Extracted from Spreadsheet Cell

Caution

The format extracted from metadata has nothing to do with Format field in the section called “SpreadsheetDataReader” (p. 387) Format field is an extra metadata field holding the Excel format of a particular cell (as a string).

• Adjust Data offset (in the right-hand Properties pane, Selected cells tab). In metadata, data offset determines where data types are guessed from. Basically, its value represents ‘a number of rows (in vertical mode) or columns (in horizontal mode) to be omitted’. By default, data offset is 1 (‘data beginning in the following row’). Click the spinner in the Value field to adjust data offset smoothly. Notice how modifying data offset is visualised in the sheet preview - you can see the 'omitted' rows change colour.

As a final step, click either OK (when creating internal metadata), or Next, select location (meta, by default) and type a name (when creating external/shared metadata). The .fmt extension will be added to the metadata name automatically.
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 20.13. Extracting Internal Metadata from a Database

After each of these three options, a Database Connection wizard opens.
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.
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 20.16. Generating a Query](image)

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. 151)*

- 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:

![DBF Metadata Editor](image)

**Figure 20.17. 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. 153).

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. 374) LotusWriter (p. 487) 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. 130) and External (Shared) Metadata (p. 133) to learn how to create internal and external metadata.
Figure 20.18. 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 24, Lotus Connections (p. 185) 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 20.19. 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 attempt 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.

Merging existing metadata

You can create new metadata by combining two or more existing metadata into one new metadata object. Fields and their settings are copied from the selected sources into the new metadata.

Conflicting field names are resolved either:

• automatically - two options: only the first field is taken; or duplicates are renamed (in a way like field_1, field_2 etc.)

• manually, which is the second step of this wizard

The Merge metadata dialog lets you choose which metadata and which fields will go into the result. You can invoke the dialog:

1. In Outline, right-click two or more existing metadata.

OR

Metadata → New Metadata → Merge existing
OR

Right click an edge and click New metadata.

2. Click Merge metadata... (Merge existing)

3. You will continue in a two-step wizard. In its first step, you manage all fields of the metadata you have selected. Select only those you want to include in the final merger (they are highlighted in bold):

![Image of Merge metadata dialog]

*Figure 20.20. Merging two metadata - conflicts can be resolved in one of the three ways (notice radio buttons at the bottom).*

4. Click Next to review merged metadata or Finish to create it instantly.

**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:

```xml
<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.

**Note**

It is highly recommended you skip the `checkConfig` method whenever dynamic metadata is used. To do that, add `-skipcheckconfig` among program arguments. See Program and VM Arguments (p. 82).

---

**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:

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

or:

```xml
<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 20.21. Creating Database Table from Metadata and Database Connection](image)

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

When you select some connection to a database. For more details see Chapter 21, **Database Connections** (p. 167). 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 20.15. CloverETL-to-SQL Data Types Transformation Table (Part I)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>SMALLINT</td>
<td>BOOLEAN</td>
<td>TINYINT(1)</td>
<td>BIT</td>
<td>BIT</td>
</tr>
<tr>
<td>byte</td>
<td>VARCHAR(80)</td>
<td>BYTE</td>
<td>VARBINARY(80)</td>
<td>VARBINARY(80)</td>
<td>VARBINARY(80)</td>
</tr>
<tr>
<td></td>
<td>FOR BIT DATA</td>
<td></td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>cbyte</td>
<td>VARCHAR(80)</td>
<td>BYTE</td>
<td>VARBINARY(80)</td>
<td>VARBINARY(80)</td>
<td>VARBINARY(80)</td>
</tr>
<tr>
<td></td>
<td>FOR BIT DATA</td>
<td></td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>date</td>
<td>DATETIME</td>
<td>DATE</td>
<td>DATETIME</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>YEAR TO SECOND</td>
<td>DATE TIME</td>
<td></td>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p)</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p)</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>integer</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INT</td>
<td>INT</td>
<td>INT</td>
</tr>
<tr>
<td>long</td>
<td>BIGINT</td>
<td>INT8</td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>number</td>
<td>BIGINT</td>
<td>INT8</td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>string</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td></td>
</tr>
</tbody>
</table>

### Table 20.16. CloverETL-to-SQL Data Types Transformation Table (Part II)

<table>
<thead>
<tr>
<th>DB type</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>Sybase</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>RAW(80)</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>BIT</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>byte</td>
<td>RAW(n)</td>
<td>BYTEA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbyte</td>
<td>RAW(80)</td>
<td>BYTEA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
<td>TIME</td>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
<td>TIME</td>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p)</td>
<td>DECIMAL(p)</td>
<td>DECIMAL(p)</td>
<td>DECIMAL(p)</td>
<td>DECIMAL(p)</td>
</tr>
<tr>
<td></td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
</tbody>
</table>
**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. 153).

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:

<table>
<thead>
<tr>
<th>DB type</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>Sybase</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clover type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>NUMBER(11,0)</td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>number</td>
<td>FLOAT</td>
<td>REAL</td>
<td>NUMERIC</td>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>string</td>
<td>VARCHAR2(80)</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
<td>VARCHAR(80)</td>
</tr>
<tr>
<td></td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
</tr>
</tbody>
</table>
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. 155) for more detailed information.

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

See [Details Pane](p. 156) 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. 157) for more detailed information.

- If another row is selected, details about selected field are displayed in the **Details** pane.
  
  See [Field Details](p. 158) for more detailed information.

**Note**

Default values of some properties are printed in gray text.

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 20.22. Metadata Editor for a Delimited File](image)
In a Jobflow (p. 240) the values of selected fields can be tracked (p. 241). The fields can be selected using the Log field with token button, as shown below:

**Figure 20.24. Trackable Fields Selection in Metadata Editor**

**Trackable Fields Selection**

**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 may display respectively: the default delimiter separating each field from the following one (except for the last one) or the size of the whole record (in fixed-length metadata).

The last column is always the label. It is similar to the field name, but there are no restrictions relating to it. See **Field Name vs. Label vs. Description** (p. 156).

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. 108) 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. 159).

### Field Name vs. Label vs. Description

The section should help you understand these basic differences.

**Field name** is an internal Clover denotation used when e.g. metadata are extracted from a file. Field names are not arbitrary - you cannot use spaces, diacritics nor accents in them.

**Field label** is automatically copied from the field name and you can change it without any restrictions - accents, diacritics etc. are all allowed. What is more, labels inside one record can be duplicate. Normally, when extracting metadata from e.g. a CSV file, you will get field names in a "machine" format. You can then change them to neat labels using any characters you want. At last, writing to an Excel file, you let those labels become spreadsheet headers. (Write field names attribute in some writers, see Chapter 52, **Writers** (p. 438))

**Description** is a pure comment. Using it, you give advice to yourself or other users who are going to work with your metadata. It produces no outputs.

### 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. 157).

- If you select another row, details about the selected field are displayed.
Chapter 20. Metadata

See Field Details (p. 158).

Note

Default values of some properties are printed in gray text.

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. 109) 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. 159) 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. 159) 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:

- **Quoted strings -** fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If Quoted strings is true, special characters are are not treated as delimiters and are:
  - removed - when reading input by a Reader
  - written out - output fields will be enclosed in Quoted strings (see UniversalDataWriter Attributes (p. 527))

  If a component has this attribute (e.g. ParallelReader, ComplexDataReader, UniversalDataReader, UniversalDataWriter), its value is set according to the settings of Quoted strings in metadata on input/output port. The true/false value in a component, however, has a higher priority than the one in metadata - you can override it.

  Example (e.g. for ParallelReader): To read input data "25" | "John", switch Quoted strings to true and set Quote character to ". This will produce two fields: 25 | John.
• **Quote character** - specifies which kind of quotes will be used in *Quoted strings*. If a component has this attribute (e.g. ParallelReader, ComplexDataReader, UniversalDataReader, UniversalDataWriter), its value is set according to the settings of **Quote character** in metadata on input/output port. The value in a component, however, has a higher priority than the one in metadata - you can override it.

• **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 overridden by the **Locale** specified for individual field.

See [Locale](p. 123) for detailed information.

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

See [Locale Sensitivity](p. 127) 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 overridden 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. 108) for more detailed information.

• **Container type** - determines whether a field can store multiple values (of the same type). There are two options: *list* and *map*. Switching back to *single* makes it a common single-value field again.

For more information, see the section called “**Multivalue Fields**” (p. 163).

• **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. 159) 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. 128) 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 12. See Data Types and Record Types (p. 108) 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. 108) 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. 110) 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. 123) 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. 127) 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. 128) 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. 161) for more information.

- **change default delimiter**
  
  See Changing Default Delimiter (p. 162) for more information.

- **define other, non-default delimiter**
  
  See Defining Non-Default Delimiter for a Field (p. 162) 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.

Multivalue Fields

Each metadata field commonly stores only one value (e.g. one integer, one string, one date). However, you can also set one field to carry more values of the same type.

Note

Multivalue fields is a new feature available as of Clover v. 3.3.

Example 20.3. Example situations when you could take advantage of multivalue fields

- A record containing an employee's ID, Name and Address. Since employees move from time to time, you might need to keep track of all their addresses, both current and past. Instead of creating new metadata fields each time an employee moves to a new house, you can store a list of all addresses into one field.

- You are processing an input stream of CSV files, each containing a different column count. Normally, that would imply creating new metadata for each file (each column count). Instead, you can define a generic map in metadata and append fields to it each time they occur.

As implied above, there are two types of structures:

**list** - is a set containing elements of a given data type (any you want). In source code, lists are marked by the [] brackets. Example:

```
integer[] list1 = [1, 367, -1, 20, 5, 0, -79]; // a list of integer elements
boolean[] list2 = [true, false, randomBoolean()]; // a list of three boolean elements
string[] list3; // a just-declared empty list to be filled by strings
```

**map** - is a pair of keys and their values. A key is always a string while a value can be any data type - but you cannot mix them (remember a map holds values of the same type). Example:

```
map[string,date] dateMap; // declaration

// filling the map with values
dateMap["a"] = 2011-01-01;
dateMap["b"] = 2012-12-31;
dateMap["c"] = randomDate(2011-01-01,2012-12-31);
```

You will find out more about maps and lists if you go to Data Types in CTL2 (p. 852).

Important

To change a field from single-value to multi-value:
1. Go to Metadata Editor.

2. Click a field or create a new one.

3. In Property → Basic, switch Container Type either to list or map. (You will see an icon appears next to the field Type in the left hand record pane.)

### Lists and Maps Support in Components

A list of components which you can use multivalue fields in:

<table>
<thead>
<tr>
<th>Component</th>
<th>List</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denormalizer (p. 563)</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>ExtFilter (p. 572)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ExtSort (p. 574)</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Merge (p. 580)</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Normalizer (p. 585)</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Partition (p. 592)</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Reformat (p. 603)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Rollup (p. 606)</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>SimpleCopy (p. 618)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SimpleGather (p. 619)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CloverDataReader (p. 330)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CloverDataWriter (p. 440)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>DataGenerator (p. 340)</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JavaBeanReader</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>JavaBeanWriter</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>JSONReader</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>JSONWriter</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>
### Component List

<table>
<thead>
<tr>
<th>Component</th>
<th>List</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLReader</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>XMLWriter (p. 532)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JavaMapWriter</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Concatenate (p. 555)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DataIntersection (p. 556)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Dedup (p. 561)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SortWithinGroups (p. 620)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ApproximativeJoin (p. 625)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DBJoin (p. 635)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ExtHashJoin (p. 638)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ExtMergeJoin (p. 644)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LookupJoin (p. 649)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RelationalJoin (p. 652)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ClusterGather (p. 718)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ClusterPartitioner (p. 720)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LookupTableReaderWriter (p. 754)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SequenceChecker (p. 760)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SpeedLimiter (p. 762)</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

At the moment, neither **map** nor **list** structures can be extracted as metadata from flat files.

### Joining on Lists and Maps (Comparison Rules)

You can specify fields that are lists or maps as **Join keys** (see **Join Types** (p. 313)) just like any other fields. The only question is when two maps (lists) equal.
First of all, let us clarify this. A list/map can:

- be **null** - it is not specified

```java
map[string,date] myMap; // a just-declared map - no keys, no values
```

- contain empty elements

```java
string[] myList = ["hello", ""]; // a list whose second element is empty
```

- contain \(n\) elements - an ordinary case described e.g. in Example 20.3, “Example situations when you could take advantage of multivalue fields” (p. 163)

Two maps (lists) are equal if both of them are not null, they have the same data type, element count and all element values (keys-values in maps) are equal.

Two maps (lists) are not equal if either of them is null.

**Important**

When comparing two lists, the order of their elements has to match, too. In maps, there is no 'order' of elements and therefore you cannot use them in Sort key.

**Example 20.4. Integer lists which are (not) equal - symoblic notation**

```java
[1,2] == [1,2]
>null] != [1,2]
[1] != [1,2]
null != null // two unspecified lists
[null] == [null] // an extra case: lists which are not empty but whose elements are null
```

Note: Maps are implemented as LinkedHashMap and thus their properties derive from it.
Chapter 21. 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. 167).
  
  Internal database connection can be:
  
  - **Externalized**: See Externalizing Internal Database Connections (p. 168).
  
  - **Exported**: See Exporting Internal Database Connections (p. 169).
  
  - **External (shared)**: See External (Shared) Database Connections (p. 170).
  
  External (shared) database connection can be:
  
  - **Linked to the graph**: See Linking External (Shared) Database Connections (p. 170).
  
  - **Internalized**: See Internalizing External (Shared) Database Connections (p. 170).

**Database Connection Wizard** is described in Database Connection Wizard (p. 171).

Access password can be encrypted. See Encrypting the Access Password (p. 175).

Database connection can serve as resource for creating metadata. See Browsing Database and Extracting Metadata from Database Tables (p. 176).

Remember that you can also create database table directly from metadata. See Create Database Table from Metadata (p. 151).

---

**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.
Chapter 21. Database Connections

Figure 21.1. 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. 171) 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 21.2. Externalizing Internal Database Connection](image)

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. 171) 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. 171)) 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 21.3. 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 cooperation 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

If you need to connect to ODBC resources, use the Generic ODBC driver. Choose it, however, 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.
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:
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 cooperation with the given type of database.

**Generic ODBC**

The driver serves for reading data sources which are not directly listed in Available drivers, e.g. DBF. To connect to ODBC resources:

- Click the Generic ODBC driver.

- **URL** - specify the dsn_source. In Windows, this is what you can see in ODBC Data Source Administrator → User DSN as Name.

- **User** and **Password** - leave these blank.

Notes on using Generic ODBC driver:

- In DBOutputTable (p. 451), mapping of metadata fields to SQL fields cannot be checked. It is up to you to design the mapping correctly. If your mapping is invalid, the graph fails.

- You cannot set any transaction isolation level (a warning about it is written to the log).

**Important**

Choose Generic ODBC only if other direct JDBC drivers do not work. Even if the ODBC driver exists it does not necessarily have to work in Clover (which was successfully tested with MySQL ODBC driver). Moreover, mind using a proper ODBC version which suits your Clover - either 32 or 64 bit.

**MS Access**

The driver supposes you have default MS Access drivers installed (check if there is MS Access Database in ODBC Data Source Administrator → User DSN). Next steps:
• Click Microsoft Access in Available drivers.

• URL - replace database_file with absolute path to your MDB file.

Notes on using MS Access driver:

• In DBOutputTable (p. 451) long and decimal types cannot be used in input metadata. Consider using Reformat (p. 603) in your graph to convert these to other metadata types.

• In DBOutputTable (p. 451), mapping of metadata fields to SQL fields cannot be checked. It is up to you to design the mapping correctly. If your mapping is invalid, the graph fails.

• You cannot set any transaction isolation level (a warning about it is written to the log).

• boolean fields that are null will be actually written as false (null value is not supported)

• binary fields - you cannot write null into them either

Advanced Properties

In addition to the Basic properties tab described 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://docs.oracle.com/javase/6/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://docs.oracle.com/javase/6/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 visible 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.
Chapter 21. Database Connections

Figure 21.6. 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. 168) and Internalizing External (Shared) Database Connections (p. 170)), 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. 151).

Windows Authentication on Microsoft SQL Server

Windows authentication means creating a database connection to Microsoft SQL Server while leaving User and Password blank (see figure below). Accessing the MS SQL database, the JTDS driver uses your Windows account to log in. To enable this all, follow the steps described in this section.
Figure 21.7. Connecting to MS SQL with Windows authentication. Setting-up a database connection like this is not sufficient. Additional steps explained below this figure need to be performed.

Clover comes with a JDBC driver from JDTS. However, it does not provide native libraries which are required for JDTS to work with Windows authentication on Microsoft SQL Server. Thus, it is necessary to download that native dll (ntlmauth.dll) and perform some additional settings.

**Getting the Native Library**

Clover supports **JTDS v. 1.2.4**. The download instructions are:

1. Get the dist package.
2. Extract the contents and go to folder `x64\SSO`, or `x86\SSO`.
3. ntlmauth.dll is located there. Copy the file (for 64b or 32b version of Clover, respectively) to a folder, e.g. `C:\jtds_dll`

**Installation**

Now there are two ways how to make the dll work. The first one involves changing Windows PATH variables. If you do not want to do that, go for the second option.

1. Add the absolute path to the dll file (C:\jtds_dll) to the Windows PATH variable. Alternatively, you can put the dll file to some folder which is already included in PATH, e.g C:\WINDOWS\system32.
2. Modify the `java.library.path` property for all members of the CloverETL Family of products:
   - **Designer**
     
     Modify CloverETLDesigner.ini and add a new line setting the java library path to the location of the dll:

     ```
     -Djava.library.path=C:\jtds_dll
     ```
Next, modify Program and VM Arguments (p. 82) in the graph's Run Configurations screen (see figure below). Add this line to VM arguments:

-Djava.library.path=C:\jtds_dll

**Note**

It is required you modify VM arguments for every graph whose components want to use Windows authentication.

![Figure 21.8. Adding path to the native dll to VM arguments.](image)

- **Clover Server**

  In the script that starts Tomcat, add the `-Djava.library.path=C:\jtds_dll` option to JAVA_OPT. For example, add the following line at the beginning of `catalina.bat`:

  ```
  set JAVA_OPTS=%JAVA_OPTS% -Djava.library.path=C:\jtds_dll
  ```

3. **MS SQL Server** - make sure you have:

   - **TCP/IP Enabled** in SQL Server Network Configuration → Protocols
   - **TCP Port** set to 1433 in TCP/IP Properties → IP Adresses → IPAll
Chapter 22. 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. 179).

  Internal JMS connection can be:

  - **Externalized**: See [Externalizing Internal JMS Connections](#) (p. 179).
  - **Exported**: See [Exporting Internal JMS Connections](#) (p. 180).

- **External (shared)**: See [External (Shared) JMS Connections](#) (p. 181).

  External (shared) JMS connection can be:

  - **Linked to the graph**: See [Linking External (Shared) JMS Connection](#) (p. 181).
  - **Internalized**: See [Internalizing External (Shared) JMS Connections](#) (p. 181).

**Edit JMS Connection Wizard** is described in [Edit JMS Connection Wizard](#) (p. 182).

Authentication password can be encrypted. See [Encrypting the Authentication Password](#) (p. 183).

---

**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. 182).

**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. 182).

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. 182)), 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**.)

![Edit JMS Connection Wizard](image)

**Figure 22.1. 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 23. QuickBase Connections

To work with a QuickBase database, use the QuickBase connection wizard to define connection parameters first.

![QuickBase connection dialog](image)

Figure 23.1. 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 24. 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. 167) and Creating External (Shared) Database Connections (p. 170) to learn how to create them. The process for Lotus Domino connections is very similar to other Database connections.

![Lotus Notes Connection Dialog](image)

**Figure 24.1. 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. 175).

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 25. 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. 187).
  
  Internal lookup tables can be:
  
  - **Externalized**: See Externalizing Internal Lookup Tables (p. 187).
  
  - **Exported**: See Exporting Internal Lookup Tables (p. 189).

- **External (shared)**: See External (Shared) Lookup Tables (p. 190).
  
  External (shared) lookup tables can be:
  
  - **Linked to the graph**: See Linking External (Shared) Lookup Tables (p. 190).
  
  - **Internalized**: See Internalizing External (Shared) Lookup Tables (p. 191).

Types of lookup tables are the following:

- **Simple lookup table**: See Simple Lookup Table (p. 192).

- **Database lookup table**: See Database Lookup Table (p. 195).

- **Range lookup table**: See Range Lookup Table (p. 196).

- **Persistent lookup table**: See Persistent Lookup Table (p. 198).

- **Aspell lookup table**: See Aspell Lookup Table (p. 199).
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 Lookups → Create lookup table. A Lookup table wizard opens. See Types of Lookup Tables (p. 192). 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 25.1. 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.
Chapter 25. Lookup Tables

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.

![Image showing the process of creating a lookup table]

Figure 25.3. Selecting Lookup Table Item

After that, the New lookup table wizard opens. See Types of Lookup Tables (p. 192). 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).

![Lookup Table Internalization Wizard]

Figure 25.4. 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 25.5. Lookup Table Wizard](image)

**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 well. You can change the default value by changing the Lookup.LOOKUP_INITIAL_CAPACITY value in `defaultProperties`.

![Figure 25.6. Simple Lookup Table Wizard](image)
Chapter 25. Lookup Tables

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.

![Edit Key Wizard](image)

**Figure 25.7. 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.

![Simple Lookup Table Wizard with File URL](image)

**Figure 25.8. Simple Lookup Table Wizard with File URL**
Chapter 25. Lookup Tables

Figure 25.9. Simple Lookup Table Wizard with Data

You can set or edit the data after clicking the Edit data button.

Figure 25.10. 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 table is the "where fieldName = ? [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 25.5, Lookup Table Wizard (p. 192).)

![Database Lookup Table Wizard](image)

Figure 25.11. 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.

**Important**

To specify a lookup table key, add a "where fieldName = ? [and ...]" statement at the end of the query, fieldName being e.g. "EMPLOYEE_ID". Records matching the given key replace the question mark character in the query.

Then, you can click OK and then Finish. See Extracting Metadata from a Database (p. 142) 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 25.12. Appropriate Data for Range Lookup Table](image1)

When you create a **Range lookup table**, you must check the **Range lookup** radio button and click **Next**. (See Figure 25.5, [Lookup Table Wizard](p. 192).)

![Figure 25.13. Range Lookup Table Wizard](image2)

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.
Chapter 25. Lookup Tables

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 25.14. Persistent Lookup Table Wizard
Chapter 25. Lookup Tables

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 25.15. 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 26. 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. 202).
  
  Internal sequences can be:
  
  - **Externalized**: See [Externalizing Internal Sequences](p. 202).
  - **Exported**: See [Exporting Internal Sequences](p. 203).

- **External (shared)**: See [External (Shared) Sequences](p. 204).
  
  External (shared) sequences can be:
  
  - **Linked to the graph**: See [Linking External (Shared) Sequences](p. 204).
  - **Internalized**: See [Internalizing External (Shared) Sequences](p. 204).

**Editing Sequence Wizard** is described in [Editing a Sequence](p. 205).
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. 205).

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. 205).

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. 205)), 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.

And when the graph has been run once again, the same sequence started from 1001:
You can also see how the sequence numbers fill one of the record fields.
Chapter 27. 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. 207).

  Internal parameters can be:

  • **Externalized**: See Externalizing Internal Parameters (p. 208).

  • **Exported**: See Exporting Internal Parameters (p. 209).

• **External (shared)**: See External (Shared) Parameters (p. 210).

  External (shared) parameters can be:

  • **Linked to the graph**: See Linking External (Shared) Parameters (p. 210).

  • **Internalized**: See Internalizing External (Shared) Parameters (p. 210).

Parameters Wizard is described in Parameters Wizard (p. 212).

---

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. 212).
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.
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, right click Parameters in Outline and select Parameters → Graph parameter file.

Graph parameters wizard opens. See Parameters Wizard (p. 212). 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. 212)), 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 27.3. 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 27.4. 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. 213) 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. 213) for details.

2. java.io.tmdir, MyPATH

   These parameters resolve to environment variables. See Environment Variables (p. 213) for details.

3. database, db_table
These are standard parameters.

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 the operating 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`

  **Important**

  Use single quotes when referring to path environment variables, especially on Windows. This is necessary to avoid conflicts between double quotes delimiting the string value of the variable, and possible double quotes contained within the value itself.

- 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 27.1. 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.
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 28. Internal/External Graph Elements

This chapter applies for Metadata (p. 107), Database Connections (p. 167), JMS Connections (p. 179), QuickBase Connections (p. 184), Lookup Tables (p. 186), Sequences (p. 201), and Parameters (p. 207).

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 29. 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. 844)

- **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. 858)

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. 218)

- Using the Dictionary in a Graph (p. 220)

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.
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. 858) for detailed information.

   There are three other data types of dictionary entry (available in Java):

   • object - [CloverETL](https://cloveretl.com) data type available with [CloverETL Engine](https://cloveretl.com).
   
   • 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. **Content Type**

   This specifies the content type of the output entry. This content type will be used, for example, when the graph is launched via Launch Service to send the results back to user.

6. **Initial Value**
Default value of an entry - useful when executing your graph without actually populating the dictionary with external data. Note that not you cannot edit this field for all data types (e.g. object). As you set a new Initial Value, a corresponding name-value pair is created in the right-hand Properties pane. Initial value is therefore the same as the first value you have created in that pane.

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. 289) for information about URL in Readers.

2. **For Writers**
   
   - source
     
     This processing type is preselect by default.
   
   - stream
     
     If no processing type is specified, stream is used.
   
   - discrete
     
     See also Writing to Dictionary (p. 302) 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:

```java
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. 858).

**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. 844).
Chapter 30. 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 30.1. Pasting a Note to the Graph Editor Pane](image)

You can also enlarge the Note by clicking it and by dragging any of its margins that have been highlighted after this click.

![Figure 30.2. Enlarging the Note](image)
At the end, click anywhere outside the **Note** so that the highlighting disappears.

![Image of Highlighted Margins of the Note Have Disappeared](image1)

**Figure 30.3. 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.

![Image of Changing the Note Label](image2)

**Figure 30.4. 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.
Chapter 30. Notes in the Graphs

Figure 30.5. Writing a New Description in the Note

The resulting Note can be as follows:

Figure 30.6. 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:
Chapter 30. Notes in the Graphs

Figure 30.7. 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 30.8. Properties of a Note
Chapter 31. Search Functionality

If you select Search → Search... from the main menu of CloverETL Designer, a window with following tabs opens:

![CloverETL Search Tab](image)

**Figure 31.1. 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 (p. 921) 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 31.2. Search Results
Chapter 32. 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. 271).

- 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 IX, CTL - CloverETL Transformation Language (p. 771) for details about Clover Transformation Language and each of its versions.

  See Defining Transformations (p. 268) for more detailed information about transformations.
Chapter 33. 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 33.1. 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. 229).

2. How should be worked with Fact Table Loader wizard. See Working with Fact Table Loader Wizard (p. 231).

3. How a created graph looks like. See Created graph (p. 237).

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. 229)

• Wizard with the project parameter file disabled (p. 231)

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.
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**.

From options in the displayed window select Fact Table Load.
Chapter 33. Fact table loader

Figure 33.3. 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 Ctrl button and clicking on the selected items.

Figure 33.4. 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. 229).

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 33.5. 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 33.6. 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 33.7. 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 33.8. 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 33.9. 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 33.10. 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 33.11. 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 33.12. Order Table Page

In this window the user is able to see selected dimension tables and databases, 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.
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.

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.
Chapter 33. Fact table loader

Figure 33.15. 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 instead 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 33.16. Created Graph

This graph also contains created parameters.
Figure 33.17. Graph Parameters
Part VI. Jobflow
Chapter 34. Jobflow Overview

Introduction

What is CloverETL Jobflow?

CloverETL Jobflow module allows combining ETL graphs together with other activities into complex processes - providing orchestration, conditional job execution, and error handling. Actions that can participate in jobflow include:

- CloverETL ETL graphs
- Native OS applications and scripts
- Web services (REST/SOAP) calls
- Operations with local and remote files

Besides the above mentioned actions available as dedicated jobflow components, the jobflow may also include ETL components. This allows additional flexibility in composing the jobflow logic and presents additional options for passing configuration to jobflow from outer environment.

Tip

You can use DBInputTable ETL component in jobflow to read a list of jobs and their parameters from a database, then use ExecuteGraph, a jobflow component, to execute each job on the list with desired parameters. Finally, the EmailSender component can be attached to the flow to notify about any errors in execution.

When editing jobflow, there are some visual modifications to the editor: components have different background colour, rounded shapes and above all Palette content changes. To change which components you can drag from Palette, go to Window → Preferences → CloverETL → Components in Palette

Design and execution

Execution of jobflow requires CloverETL Server environment. It is, however, possible to design the jobflow offline using a standalone CloverETL Designer. Developers can compose, deploy, and execute jobflows on the Server interactively using Server Integration module available in CloverETL Designer. To automate execution of jobflow processes in the Server environment, the jobflows are fully integrated with existing automation, such as the Scheduler or File Triggers, and with Server API including SOAP and REST services.

Anatomy of the Jobflow module

The CloverETL Designer contains the design-time functional elements of the Jobflow module while the CloverETL Server contains the necessary runtime support and automation features.

Jobflow elements in CloverETL Designer:

- **Jobflow editors**: Dedicated UI for designing jobflows (*.jbf). Open visual editor, there are some visual modifications to the editor. Components have different background colour, rounded shapes and above all Palette content changes.

- **Jobflow components in palette**: The jobflow-related components are available under sections Chapter 55, Job Control (p. 656) and Chapter 56, File Operations (p. 700). Additional ETL components can be used include WebServiceClient and HTTPConnector from Chapter 59, Others (p. 738) category. Some of the Chapter 55, Job Control (p. 656) components are also available in the ETL perspective.
Chapter 34. Jobflow Overview

- **ProfilerProbe component**: Available in the „Data Quality“ palette category; this component allows execution of CloverETL Profiler jobs as part of ETL graph or jobflow.

- **Predefined metadata**: Designer contains predefined metadata templates describing expected inputs or outputs provided by jobflow components. The templates generate metadata instances in which developers may decide to modify. The templates are available from edge context menu in graph editor, „New metadata from template“.

- **Trackable fields**: Metadata editor in jobflow perspective allows flagging selected fields (p. 245) in token metadata as „trackable“. Values of trackable fields are automatically logged by jobflow components at runtime (see description of token-centric logging below). The aim is to simplify tracking of the execution flow during debugging or post-mortem job analysis.

- **CTL functions**: A set of CTL functions allowing interaction with outer environment - access to environment variables (getEnvironmentVariables()), graph parameters (getParamValues()) and Java system properties (getJavaProperties()).

- **Log console**: Console view contains execution logs in the jobflow as well as any errors.

Jobflow elements in CloverETL Server:

- **Execution history**: Hierarchical view of overall execution as well as individual steps in the jobflow together with their parent-child relationships. Includes status, performance statistics as well as listing of actual parameter and dictionary values passed between jobflow steps.

- **Automated Logging**: Token-centric logging can track a „token“ that triggers execution of particular steps. Tokens are uniquely ordered for easy identification; however, developers may opt to log additional information (e.g. file name, message identification or session identifiers) together with the default numeric token identifier.

- **Integration with automation modules**: All CloverETL Server modules include Scheduler, Triggers, and SOAP API to allow launching of jobflows and passing user-defined parameters for individual executions.

### Important concepts

#### Dynamic attribute setting

Besides static configuration of component’ attributes, the jobflow components (as well as WebServiceClient and HTTPConnector components) allow dynamic attribute configuration during runtime from a connected input port.

Values are read from incoming tokens from connected input port and mapped to component’ attributes via mapping defined by “Input Mapping” property. Dynamically set attributes are merged with any static component configuration; in case of a conflict, the dynamic setting overrides static configuration. The combined configurations of a component are finally used for execution triggered by token.

**Tip**

The dynamic configuration can be used for implementation of a “for-loop” by having a static configuration job in ExecuteGraph/ExecuteJobflow while passing the job parameters dynamically via “Input Mapping”.

The dynamic setting of parameters can also be used with HTTPConnector or WebServiceClient to dynamically construct the target URL, include HTTP GET parameters into URL or redirect the connection.

#### Parameter passing

The ExecuteGraph and ExecuteJobflow components allow passing of graph parameters and dictionary values from parent to child. With dictionary it is also possible for a parent to retrieve values from a child’s dictionary. This is only possible AFTER a child has finished its execution.
In order to pass dictionary between two steps in jobflow, it is necessary to:

1. Declare desired dictionary entries in child’s dictionary
2. Tag the entry as “input” (entry value is set by parent) or “output” (parent to retrieve value from a child)
3. Define mapping for each entry in parent’s ExecuteGraph/ExecuteJobflow component’s “Input Mapping” or “Output Mapping” properties.
4. For a child to pass an entry to the parent, value can be set during child execution using the Success, Fail, or SetJobOutput component, but it is also possible via CTL code.
5. Parameters declared in the child graph (local or from parametric file) can be set in the “Input Mapping” of ExecuteGraph/ExecuteJobflow in the parent graph. It is NOT possible for the child to change the parameter value during its runtime or the parent to retrieve parameter value from a child.

Both parameters and any input/output dictionary entries declared in the child graph are automatically displayed in the “Input Mapping” or “Output Mapping” of ExecuteGraph/ExecuteJobflow accordingly.

When to use parameters vs dictionary:

- **Parameters:** for configuration of the child graph (changing component settings, graph layout etc)
- **Dictionary:** for passing data (or values) to be processed by the child, to return values to the parent

**Tip**

Parameters can appear anywhere in component attributes and as textual macros expanded before graph execution; they can be used to significantly change the runtime graph layout. Use them to create reusable logic by disabling processing branches in graphs, routing output to particular destination, or passing dataset restrictions/filter (e.g. business day, product type, active user). They can also be used to pass environment information in a centralized fashion.

Use dictionary to pass result “variables” back to parent or for a child to receive initial “variable” values from parent. You can highlight the process of “receiving” or “setting” the dictionary entries with GetJobInput and SetJobOutput components.

**Pass-through mapping**

Any field can be passed through the jobflow components. The “Output mapping” property can be used in mapping the incoming token fields to output combining with other component output values.

**Tip**

With webservises, the pass-through mapping can be used to perform login operation only once then pass a session token through multiple HTTPConnector or WebServiceClient components.

**Execution status reporting**

The jobflow components report the success or failure of the executed activity via two standardized output ports. The “success” output port (0 - zero) by default carries information about all successful executions while the output “error” port (1 - one) carries information about all failed executions.

Developers may choose to redirect even failed executions to the success-output using the “Redirect error output” attribute available in all jobflow components.

**Error handling**
Chapter 34. Jobflow Overview

The ExecuteGraph, ExecuteScript, ExecuteJobflow, and file operations components behave as natural Java try/catch block. If the executed activity finishes successfully, the result is routed to the 0-th “success” port – this case is analogous to situation where no exception was thrown.

When activity started by the component fails, the error is routed to the 1-st “error” port where a compensating logic can be attached. This behavior resembles the exception being handled by a “catch” block in code.

In case there is no edge connected to the “error” port, the component throws a regular Java exception and terminates its processing. In case the job in error was started by a parent job, the exception causes a failure in parent’s Execute in which it may choose to handle or throw the exception further.

Tip

Using the “try/catch” approach, you may construct logic handling of particular errors in processing while deliberately leaving errors requiring human interaction unhandled. Uncaught errors will abort the processing and show the job as failed in Server Execution History and can be handled by production support teams.

You can use the Fail component in jobflow or ETL graph to highlight that an error is being “thrown”; it can be used to compose a meaningful error message from input data or to set dictionary values indicating error status to the parent job.

Jobflow execution model: Single token

Activities in the jobflow are by default executed sequentially in the order of edge direction. The first component (having no input) in the flow starts executing, and after the action finishes, it produces an output token that is passed to the next connected component in the flow. The token serves as a triggering event for the next component and the next job is started.

This is different to ETL graph execution where the first component produces data records for the second component but both run together in parallel.

In case where a jobflow component output is forked (e.g. via SimpleCopy) and connected to input of two other jobflow components, the token triggers both of these components to start executing in parallel and at the same time.

The concept of phases available in ETL graphs can also be used in jobflow.

Tip

Use the branching to “fork” the processing into multiple parallel branches of execution. If you need to “join” the branches after execution, use Combiner, Barrier, or TokenGather component; the selection depends on how you want to process the execution results.

Jobflow execution model: Multiple tokens

In a basic scenario, only one token passes through the jobflow; this means each action in the jobflow is started exactly once or not started at all. A more advanced use of jobflows involves multiple tokens entering the jobflow to trigger execution of certain actions repeatedly.

The concept of multiple tokens allows developers to implement for-loop iterations, and more specifically, to support CloverETL Jobflow parallel for-loop pattern.

In the parallel for-loop pattern, as opposed to a traditional for-loop, each new iteration of the loop is started independently in parallel with any previous iterations. In jobflow terms, this means when two tokens enter the jobflow, actions triggered by the first token may potentially execute together with actions triggered by the second token arriving later.

As the parallel execution might be undesirable at times, it is important to understand how to enforce sequential execution of the loop body or actions immediately following the “loop”:
• **Sequence the loop body:** force the sequential execution of multiple steps forming the loop body, essentially means converting the parallel for loop into a traditional for loop.

To sequence the loop body and make it behave as a traditional for loop, wrap actions in the loop body into an ExecuteJobflow component running in synchronous execution mode. This causes the component to wait for the completion of the underlying logic before starting a new “iteration”.

**Tip**

Imagine a data warehousing scenario where we receive two files with data to be loaded into a dimension table and a fact table respectively (e.g. every day we receive two files - dim-20120801.txt and fact-20120801.txt). The data must be first inserted into a dimension table, only then the fact table can be loaded (so that the dimension keys are found). Additionally, the data from previous day must be loaded before loading data (dimension+fact) for the current day. This is necessary as a data warehouse keeps track of changes of data over time.

To implement this using jobflow, we would use a DataGenerator component to generate a week’s worth of data and feed that to ExecuteJobflow implementing the body of the loop – loading of the warehouse for a single day. Specifically, the ExecuteJobflow would contain two ExecuteGraph components: LoadDimensionForDay and LoadFactTableForDay.

• **Sequence the execution of actions following the loop:** instead of having actions immediately following the loop being triggered by each new iteration, we want the actions to be triggered only once – after all iterations have completed.

To have the actions following the loop execute once all iterations have finished, prefix the actions with Barrier component with “All tokens form one group” option enabled. In this mode, the Barrier first collects all incoming tokens (waits for all iterations), then produces a single output token (control flow is passed to actions immediately following the for loop).

**Tip**

After loading a week’s worth of data into the data warehouse from previous example, we need to rebuild/enable indexes. This can only happen after all files have been processed. In order to do that, we can add a Barrier component followed by another ExecuteGraph into the joflow.

The Barrier would cause all loading to finished and only then the final ExecuteGraph step would launch one or more DBExecute components with necessary SQL statements to create the indexes.

**Stopping on error**

When multiple tokens trigger a single ExecuteGraph/ExecuteJobflow component and an unhandled error occurs in the underlying job, the default behavior of the component is not to process any further tokens to avoid starting new jobs while an exception is being thrown.

If there is a desire to continue processing regardless of failures, the component’s behavior can be changed using the “Stop processing on fail” attribute on ExecuteGraph/ExecuteJobflow.

**Synchronous vs. asynchronous execution**

ExecuteGraph and ExecuteJobflow by default execute their child jobs synchronously; this means that they wait for the underlying job to finish. Only then they produce an output token to trigger the next component in line. While the component is waiting for its job to finish, it does not start new jobs even if more triggering tokens are available on the input.

For advanced use cases, the components also support asynchronous execution; this is controlled by the “Execution type” property. In asynchronous mode of execution, the component starts the child job as soon as a new input token is available and does not wait for the child job to finish. In such case, the ExecuteGraph/ExecuteJobflow components only output job run id as job statistics might not be available.
Developers can use the MonitorGraph or MonitorJobflow component to wait for asynchronously started graphs or jobflows.

**Tip**

The asynchronous execution mode is useful to start jobs in parallel when the total number of jobs is unknown.

### Logging

Log messages produced by jobflow components are token-centric. A token represents basic triggering mechanism in jobflow and one or multiple tokens can be present in a single running jobflow instance. To allow easy identification of the activities triggered by token, the tokens are automatically numbered.

**Example 34.1. Example jobflow log - token starting a subgraph**

```
```

Format of the log is obvious: `date time RunID [COMPONENT_NAME] Token [#number] message`.

Every jobflow component automatically logs information on token lifecycle. The important token lifecycle messages are:

- **Token created**: a new token entered jobflow
- **Token finalized**: a token lifecycle finished in a component; either in a terminating component (e.g. Fail, Success) or when multiple tokens resulted from the terminating one (e.g. in SimpleCopy)
- **Token sent**: a token was sent through a connected output port to the next following component
- **Token link**: logs relationships between incoming (terminating) tokens and new tokens being created as a result (e.g. in Barrier, Combine)
- **Token received**: a token was read from a connected input port from the previous component
- **Job started**: token triggered an execution of job in ExecuteGraph/ExecuteJobflow/ExecuteScript.
- **Job finished**: a child job finished execution and the component will continue processing the next one
- **Token message**: component produced a (user-defined) log message triggered by the token

### Metadata fields tracking

Additionally, the developers may enable logging of additional information stored in token fields through the concept of trackable fields (p. 155). The tracked field will be displayed in the log like this (example for field `fileName`):

```
2012-10-08 14:00:29,948 INFO 28 [EXECUTE_JOBFLOW0_28] Token [#1 (fileURL=${DATA_IN_DIR}/inputData.txt)] received from input port 0.
```

**Tip**

File names, directories, and session/user ids serve as useful trackable fields as they are often iterated upon.

### Advanced Concepts
Daemon jobs

CloverETL Jobflow allows child jobs to outlive their parents. By default, this behavior is disabled meaning when the parent job finishes processing or is aborted, all its child jobs are terminated as well. The behavior can be changed by the ExecuteGraph/ExecuteJobflow property “Execute as daemon”.

Killing jobs

While using the try/catch to control job behavior on errors, the jobflows also allow developers to forcibly terminate jobs using the KillGraph and KillJobflow components. A job execution can be killed by using its unique run id; for mass termination of jobs, the concept of execution group can be used. A job can be tagged in an execution group using the “Execution group” property in ExecuteGraph/ExecuteJobflow components.
Chapter 35. Jobflow Design Patterns

Try/Catch block

All execution components simply allow to react to success and failure. In case of job success, token is send to the first output port. In case of job failure, token is send to the second output port.

Try/Finally block

All execution components allow to redirect the error token to the first output port. Use ‘Redirect error output’ attribute for uniform reaction to job success and failure.

Sequential execution

Sequential jobs execution is performed by simple execution components chaining. Failure of any job causes jobflow termination.

Sequential execution with error handling

Sequential jobs execution can be extended by common job failure handling. Component TokenGather is suitable for gathering all tokens representing job failures.

Parallel execution

Parallel jobs execution is simply allowed by set of independent executors. Reaction to success and failure is available for each individual job.

Parallel execution with common success/error handling
Barrier component allows to react to success or failure of parallel running jobs. By default, group of parallel running jobs is considered successful if all jobs finished successfully. Barrier component has various settings to satisfy all your needs in this manner.

Conditional processing

Conditional processing is allowed by token routing. Based on results of Job A you can decide using Condition component which branch of processing will be used afterwards.

Dictionary driven jobflow

Parent jobflow can pass some data to child job using input dictionary entries, these job parameters can be read by GetJobInput component and can be used in further processing. On the other side jobflow results can be written to output dictionary entries using SetJobOutput component. These results are available in parent jobflow.

Fail control

You can intentionally stop processing of jobflow using Fail component. User-specified message can be reported by Fail component.

Asynchronous graphs execution

Parallel processing of variable number of jobs is allowed using asynchronous job processing. The example bellow shows how to process all csv files in parallel way. First, all file names are listed by ListFiles component. Single graph for each file name is asynchronously executed by ExecuteGraph component. Graph run identifications (runId) are sent to MonitorGraph component which waits for all graph results.

Asynchronous execution is available only for graphs and jobflows.
File operations

Jobflow provides set of file operations component - list files, create, copy, move and delete files. This use-case shows how to use file operation components to process set of remote files. The files are downloaded from remote FTP server, each file is processed by a job, results are copied to a final destination and possible temporary files are deleted.

Aborting Graphs

Graphs and jobflows can be explicitly aborted by KillGraph respectively by KillJobflow components. The example bellow shows how to process a list of tasks in parallel way and jobs which reached user-specified timeout are automatically aborted.
Part VII. Components Overview
Chapter 36. Introduction to Components

For basic information about components see Chapter 18, Components (p. 94).

In the palette of components of the Graph Editor, all components are divided into following groups: Readers (p. 328) Writers (p. 438) Transformers (p. 550) Joiners (p. 624) Cluster Components (p. 717), and Others (p. 738). 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 37. 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 37.1. Selecting Components

After that, you must expand the CloverETL item and choose Components in Palette.

Figure 37.2. 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.
Then you only need to close and open graph and the components will be removed from the Palette.
Chapter 38. Find / Add Components

Besides Palette (p. 252), you can use smart dialogs to find/add components.

Finding Components

If you built a complex graph and cannot find components easily, press Ctrl+O. That opens the Find component dialog:

![Find Components dialog](image)

*Figure 38.1. Find Components dialog - the searched text is highlighted both in component names and description.*

As you type, the components are searched by their:

- name - i.e. if you rename e.g. UniversalDataReader to 'read customers from text file', you can search the component by typing 'customers', 'text file' etc.

- description - again, both the default description or a custom one you added to a component yourself

1. Click a component in the search results.
2. Press Enter
3. The component will be selected and focused in your graph layout.

Adding Components

If you need to quickly add a component without navigating to Palette OR you do not know which component you should use, press Shift Space. This brings the Add component dialog.

A great feature: components are searched by their name and description (p. 254) again.
Example 38.1. Finding a sort component

You need to sort your data, but CloverETL offers so many sort components. A quick solution: press `Shift+Space` and type 'sort'. You will see all available sorters (with a description).

![Add Components dialog - finding a sorter.](image)

Figure 38.2. Add Components dialog - finding a sorter.

Note

Recently searched/added components get cummulated at the top of the dialog for a quicker access to them.
Chapter 39. 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 37, Palette of Components (p. 252)).

- Each component can be set up using Edit Component Dialog (Edit Component Dialog (p. 257)).

![Figure 39.1. Edit Component Dialog (Properties Tab)](image1)

![Figure 39.2. Edit Component Dialog (Ports Tab)](image2)
Chapter 39. Common Properties of All Components

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. 260)).
- Each graph can be processed in phases ([Phases](p. 261)).
- Components can be disabled ([Enable/Disable Component](p. 262)).
- Components can be switched to **PassThrough mode** ([PassThrough Mode](p. 263)).

## 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.
Chapter 39. Common Properties of All Components

See Component Name (p. 260) 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.** ID identifies the component among all other components of the same type. If you check Generate component ID from its name in Window → Preferences → CloverETL and your component is called e.g. 'Write employees to XML', then it automatically gets this ID: 'WRITE_EMPLOYEES_TO_XML'. While the option is checked, the ID changes every time you rename the component.

  - **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. 261) 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. 262) 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. 263) 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. 263) 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 inherited 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:<sandboxID>`
Here sandbox is a partitioned sandbox.

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 sandbox 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.
- 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.

Figure 39.3. Simple Renaming Components
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.

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 39.4. Running a Graph with Various Phases

Figure 39.5. 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 39.6. 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. 262)), 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.

![Diagram of a graph with component in PassThrough mode](image)

**Figure 39.7. 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. 257).
Chapter 40. 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. 68).

- Some components use a specific metadata structure on their ports. The connected edges can be easily assigned metadata from predefined templates. See Metadata Templates (p. 264).

- 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. 264).

- 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. 265).

- In many components from different groups of components, a transformation can be or must be defined. See Defining Transformations (p. 268).

Metadata Templates

Some components require metadata on their ports have a specific structure. For example, see Error Metadata for UniversalDataReader (p. 398) For some other components, such as Chapter 56, File Operations (p. 700) the metadata structure is not required, but recommended. In both those cases, it is possible to make use of pre-defined metadata templates.

In order to create a new metadata with the recommended structure, right-click an edge connected to a port which has a template defined, select New metadata from template from the context menu and then pick a template from the submenu.

![Figure 40.1. Creating Metadata from a Template](image)

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.)
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 fields descends down from top in the dialog pane and to the right from the left in the attribute row. See Sort Key (p. 265) 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. 620)
- Merge key in Merge (p. 580)
- Partition key in Partition (p. 592), and ClusterPartitioner (p. 720)
- Aggregate key in Aggregate (p. 552)
- Key in Denormalizer (p. 563)
- Group key in Rollup (p. 606)
- Matching key in ApproximativeJoin (p. 625)
- 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. 306)

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.
Chapter 40. Common Properties of Most Components

Figure 40.3. 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**

**ASCII**betical vs. **alpha**betical 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 40.1. 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. 574)
• Sort key in FastSort (p. 576)
• Sort key in SortWithinGroups (p. 620)
• Dedup key in Dedup (p. 561)
• Sort key in SequenceChecker (p. 760)
Defining Transformations

For basic information about transformations see Chapter 32, Transformations (p. 228).

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. 271).

Below we can learn the following:

1. What components allow transformations.
   Components Allowing Transformation (p. 268)

2. What language can be used to write transformations.
   Java or CTL (p. 269)

3. Whether definition can be internal or external.
   Internal or External Definition (p. 269)

4. What the return values of transformations are.
   Return Values of Transformations (p. 272)

5. What can be done when error occurs.
   Error Actions and Error Log (deprecated since 3.0) (p. 274)

6. The Transform editor and how to work with it.
   Transform Editor (p. 275)

7. What interfaces are common for many of the transformation-allowing components.
   Common Java Interfaces (p. 284)

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. 275) or **Edit Value Dialog** (p. 69).

• **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. 68). 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. 68).

• **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. 70) 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 40.1. Transformations Overview

<table>
<thead>
<tr>
<th>Component</th>
<th>Transformation required</th>
<th>Java</th>
<th>CTL</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>CTU template</th>
<th>Java interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataGenerator (p. 340)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>☐</td>
<td>(p. 343)</td>
<td>(p. 346)</td>
</tr>
<tr>
<td>JMSReader (p. 362)</td>
<td>☐</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>-</td>
<td>-</td>
<td>(p. 364)</td>
</tr>
<tr>
<td>MultiLevelReader (p. 376)</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>☐</td>
<td>-</td>
<td>-</td>
<td>(p. 378)</td>
</tr>
<tr>
<td>Writers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMSWriter (p. 477)</td>
<td>☐</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(p. 479)</td>
</tr>
<tr>
<td>Transformers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition (p. 592)</td>
<td>☐</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>-</td>
<td>(p. 594)</td>
<td>(p. 598)</td>
</tr>
<tr>
<td>DataIntersection (p. 556)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 558)</td>
<td>(p. 558)</td>
</tr>
<tr>
<td>Reformat (p. 603)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>-</td>
<td>(p. 604)</td>
<td>(p. 605)</td>
</tr>
<tr>
<td>Denormalizer (p. 563)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 565)</td>
<td>(p. 571)</td>
</tr>
<tr>
<td>Pivot (p. 599)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 602)</td>
<td>(p. 602)</td>
</tr>
<tr>
<td>Normalizer (p. 585)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 586)</td>
<td>(p. 591)</td>
</tr>
<tr>
<td>MetaPivot (p. 582)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Rollup (p. 606)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>-</td>
<td>(p. 608)</td>
<td>(p. 616)</td>
</tr>
<tr>
<td>DataSource (p. 559)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Joiners</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApproximativeJoin (p. 625)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>ExtHashJoin (p. 638)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>ExtMergeJoin (p. 644)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>LookupJoin (p. 649)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>DBJoin (p. 635)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>RelationalJoin (p. 652)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(p. 314)</td>
<td>(p. 317)</td>
</tr>
<tr>
<td>Cluster Components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClusterPartitioner (p. 720)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>☐</td>
<td>-</td>
<td>(p. 722)</td>
<td>(p. 722)</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaExecute (p. 752)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(p. 753)</td>
</tr>
</tbody>
</table>

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. 272) for more information.
Return Values of Transformations

In those components in which 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. 274) 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.
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.

![Define error actions dialog](image)

Figure 40.4. 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 40.2. 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. 68).
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: Transformations Source and Regex tester.

Transformations

The Transformations tab can look like this:

![Figure 40.5. Transformations Tab of the Transform Editor](image)

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 input fields of all input ports and their data types in the left pane. Output fields of all output ports and their data types display in the right pane. You can see the following tabs in the middle bottom area: Functions, 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. Transformations defined here can be adjusted by a left mouse click drag and drop or via toolbar buttons in the upper left hand corner.

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.

**Tip**

To design the transformation in a much easier way, you can simply drag fields from the left hand pane to the right hand pane. The transformation stub in the central pane will be prepared for you automatically. Be careful when dropping the field, though. If you drop it onto an output field, you will create a mapping. If you drop it into a blank space of the right hand pane (between two fields), you will just copy input metadata to the output. Metadata copying is a feature which works only within a single port.

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 40.6. Copying the Input Field to the Output](image)

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.

**Note**

Fields of output metadata can be rearranged by a simple drag and drop with the left mouse button.

The resulting simple mapping can look like this:
Figure 40.7. 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 40.8. Mapping of Inputs to Outputs (Connecting Lines)

You can write the desired transformation:

- Into individual rows of the Transformations pane - optionally, drag any function you need from the bottom Functions tab (the same counts for Variables, Sequences or Parameters) and drop them into the pane. Use Filter to quickly jump to the function you are looking for.

- By clicking the '...' button which appears after selecting a row inside the Transformations pane. This opens an editor for defining the transformation. It contains a list of fields, functions and operators and also provides hints. See below:
Transform editor supports wildcards in mapping. If you right click a record or one of its fields, click **Map record to** and select a record, you will produce a transformation like this (as observed in the **Source** tab): $\text{out.}_0.* = \text{in.}_1.*$; meaning "all output fields of record no 0 are mapped to all input fields of record no 1". In **Transformations**, wildcard mapping looks like this:

**Figure 40.9. Editor with Fields and Functions**

**Source**

Some of your transformations may be too complicated to define in the **Transformations** tab. You can use the **Source** tab instead.

**(Source** tabs of individual components are shown in corresponding sections describing these components.)

Below you can see the **Source** tab with the transformation defined above. It is written in Clover transformation language (Chapter 64, **CTL2** (p. 849)).
Chapter 40. Common Properties of Most Components

Figure 40.11. 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 40.12. 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. 271). 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 40.13. Confirmation Message**

The tab can look like this:

**Figure 40.14. 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:

![Outline pane](image)

**Figure 40.15. 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.

![Content assist](image)

**Figure 40.16. 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.
Chapter 40. Common Properties of Most Components

Figure 40.17. Content Assist (List of CTL Functions)

Tip

Press Shift+Space to bring the Available CTL functions dialog.

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 40.18. 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 40.19. Converting Transformation to Java

After selecting and clicking OK, the transformation converts into the following form:
Figure 40.20. 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 described.

**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.

**Regex Tester**

This is the last tab of the Transform Editor and it is described here: Tabs Pane (p. 42).
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 updateTransform() 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 41. 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. 103). 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. 286)
- Viewing Data on Readers (p. 290)
- Input Port Reading (p. 292)
- Incremental Reading (p. 293)
- Selecting Input Records (p. 294)
- Data Policy (p. 295)
- XML Features (p. 296)
- As has been shown in Defining Transformations (p. 268), 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. 296)
- As has been shown in Defining Transformations (p. 268), 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. 296)

Here we present an overview of all Readers:
Table 41.1. Readers Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs (^1)</th>
<th>Different to different outputs (^2)</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataGenerator (p. 340)</td>
<td>none</td>
<td>0</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>yes (^3)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>UniversalDataReader (p. 397)</td>
<td>flat file</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ParallelReader (p. 380)</td>
<td>flat file</td>
<td>0</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CloverDataReader (p. 330)</td>
<td>clover binary file</td>
<td>0</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SpreadsheetDataReader (p. 387)</td>
<td>XLS(X) file</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>XLSDataReader (p. 402)</td>
<td>XLS(X) file</td>
<td>0-1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DBFDataReader (p. 347)</td>
<td>dBase file</td>
<td>0-1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DBInputTable (p. 349)</td>
<td>database</td>
<td>0</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>XMLExtract (p. 406)</td>
<td>XML file</td>
<td>0-1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>XMLXPathReader (p. 431)</td>
<td>XML file</td>
<td>0-1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IMSReader (p. 362)</td>
<td>jms messages</td>
<td>0</td>
<td>1</td>
<td>✗</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>EmailReader (p. 353)</td>
<td>email messages</td>
<td>0</td>
<td>1</td>
<td>✗</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LDAPReader (p. 371)</td>
<td>LDAP directory tree</td>
<td>0</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MultiLevelReader (p. 376)</td>
<td>flat file</td>
<td>1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ComplexDataReader (p. 332)</td>
<td>flat file</td>
<td>1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>QuickBaseRecordReader (p. 383)</td>
<td>QuickBase</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>QuickBaseQueryReader (p. 385)</td>
<td>QuickBase</td>
<td>0</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LotusReader (p. 374)</td>
<td>Lotus Notes</td>
<td>0</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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. 272) 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. 68).
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.gz file, writing to .tar.gz files is not supported.

- `tar: (gzip:/path/file???.tar.gz)#innerfolder?/filename*.txt`
Chapter 41. Common Properties of Readers

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.

- \texttt{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

- \texttt{ftp://username:password@server/path/filename.txt}
  
  Reads specified \texttt{filename.txt} file on remote server connected via ftp protocol using username and password.

- \texttt{sftp://username:password@server/path/filename.txt}
  
  Reads specified \texttt{filename.txt} file on remote server connected via ftp protocol using username and password.

- \texttt{http://server/path/filename.txt}
  
  Reads specified \texttt{filename.txt} file on remote server connected via http protocol.

- \texttt{https://server/path/filename.txt}
  
  Reads specified \texttt{filename.txt} file on remote server connected via https protocol.

- \texttt{zip:(ftp://username:password@server/path/file.zip)#innerfolder/filename.txt}
  
  Reads specified \texttt{filename.txt} file compressed in the \texttt{file.zip} file on remote server connected via ftp protocol using username and password.

- \texttt{zip:(http://server/path/file.zip)#innerfolder/filename.txt}
  
  Reads specified \texttt{filename.txt} file compressed in the \texttt{file.zip} file on remote server connected via http protocol.

- \texttt{tar:(ftp://username:password@server/path/file.tar)#innerfolder/filename.txt}
  
  Reads specified \texttt{filename.txt} file archived in the \texttt{file.tar} file on remote server connected via ftp protocol using username and password.

- \texttt{zip:(zip:(ftp://username:password@server/path/name.zip)#innerfolder/file.zip)#innermostfolder/filename.txt}
  
  Reads specified \texttt{filename.txt} file compressed in the \texttt{file.zip} file that is also compressed in the \texttt{name.zip} file on remote server connected via ftp protocol using username and password.

- \texttt{gzip:(http://server/path/file.gz)}
  
  Reads the first file compressed in the \texttt{file.gz} file on remote server connected via http protocol.

- \texttt{http://server/filename*.dat}
  
  Reads all files from WebDAV server which satisfy specified mask (only * is supported.)
Chapter 41. Common Properties of Readers

- http://access_key_id:secret_access_key@bucketname.s3.amazonaws.com/*

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 \texttt{eof}.

Reading from Console

- 
  Reads data from \texttt{stdin} after start of the graph. When you want to stop reading, press \texttt{Ctrl}+\texttt{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\textsuperscript{1)}
  Reads data from dictionary.

- dict:KeyName:source\textsuperscript{1)}
  Reads data from dictionary in the same way like the \texttt{discrete} processing type, but expects that the dictionary values are input file URLs. The data from this input passes to the \texttt{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: FileInputStream, 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 minimize 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 41.1. Viewing Data in Components](image)

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. You 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 41.2. Viewing Data as Plain Text

On the other hand, if you select the **Grid** option, you can select **Filter expression**, but no **Charset**.

Figure 41.3. Viewing Data as Grid

The result can be seen as follows in **Plain text** mode:

Figure 41.4. Plain Text Data Viewing

Or in the **Grid** mode, it can be like the following:
Figure 41.5. Grid Data Viewing

Note

If there are too many records to be displayed, you will see the Load more... blue text below the view. Clicking it, a new chunk of records is added behind the currently displayed ones. In Plain View, you can also scroll down to the bottom of the view (alternatively, by pressing Page Down) to have the records loaded.

The same can be done in some of the Writers. See Viewing Data on Writers (p. 303). 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 Readers:

• UniversalDataReader
• XLSDDataReader
• SpreadsheetDataReader
• 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 Readers 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]`.

*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. 68) 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. The **Reader** starts parsing data as soon as first bytes come by the port and process it progressively until **eof** comes. See **Output Port Writing** (p. 305) 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 multiplicatted 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 multiplicatted 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
- SpreadsheetDataReader
- 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)**

- **SpreadsheetDataReader (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** and **SpreadsheetDataReader** enable 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. 146) for detailed information about how metadata should be created by user. See SpreadsheetDataReader documentation for its format of error port metadata.

- **Lenient.** This data policy means that incorrect records are only skipped and data parsing continues.

### XML Features

In XMLExtract (p. 406) and XMLXPathReader (p. 431) you can configure the validation your input XML files by specifying the **Xml features** attribute. The Xml features configure validation of the XML in more detail by enabling or disabling specific checks, see Parser Features. It is expressed as a sequence of individual expressions of one of the following form: \texttt{nameM:=true} or \texttt{nameN:=false}, where each \texttt{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:

![XML Features Dialog](image)

*Figure 41.6. 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. 340) requires a transformation which can be written in both CTL and Java.

  See CTL Templates for DataGenerator (p. 343) 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. 272)). Mapping must be defined for such port.

### Java Interfaces for Readers

- **DataGenerator** (p. 340) requires a transformation which can be written in both CTL and Java.
See *Java Interfaces for DataGenerator* (p. 346) 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. 272)). Mapping must be defined for such port.

- **JMSReader** (p. 362) allows optionally a transformation which can be written in Java only.

  See *Java Interfaces for JMSReader* (p. 364) 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. 376) requires a transformation which can only be written in Java.

  See *Java Interfaces for MultiLevelReader* (p. 378) for more information.
Chapter 42. 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. 103). 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. 299)

- Viewing Data on Writers (p. 303)

- Output Port Writing (p. 305)

- How and Where Data Should Be Written (p. 305)

- Selecting Output Records (p. 306)

- Partitioning Output into Different Output Files (p. 306)

- As has been shown in Defining Transformations (p. 268), 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. 308)

Here we present an overview of all Writers:
### Table 42.1. Writers Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trash (p. 524)</td>
<td>none</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UniversalDataWriter (p. 526)</td>
<td>flat file</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloverDataWriter (p. 440)</td>
<td>Clover binary</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpreadsheetDataWriter (p. 506)</td>
<td>XLS(X) file</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLSDataWriter (p. 529)</td>
<td>XLS(X) file</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StructuredDataWriter (p. 520)</td>
<td>structured flat file</td>
<td>1-3</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EmailSender (p. 459)</td>
<td>e-mails</td>
<td>1</td>
<td>0-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBOutputTable (p. 451)</td>
<td>database</td>
<td>1</td>
<td>0-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB2DataWriter (p. 442)</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InfobrightDataWriter (p. 463)</td>
<td>database</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InformixDataWriter (p. 465)</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSSQLDataWriter (p. 489)</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQLDataWriter (p. 492)</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OracleDataWriter (p. 495)</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQLDataWriter (p. 499)</td>
<td>database</td>
<td>0-1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLWriter (p. 532)</td>
<td>XML file</td>
<td>1-n</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMSWriter (p. 477)</td>
<td>jms messages</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDAPWriter (p. 485)</td>
<td>LDAP directory</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickBaseRecordWriter (p. 504)</td>
<td>QuickBase</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickBaseImportCSV (p. 502)</td>
<td>QuickBase</td>
<td>1</td>
<td>0-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LotusWriter (p. 487)</td>
<td>Lotus Notes</td>
<td>1</td>
<td>0-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBFDataWriter (p. 448)</td>
<td>.dbf file</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Supported File URL Formats for Writers

The **File URL** attribute may be defined using the **URL File Dialog** (p. 68).

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.
Chapter 42. Common Properties of Writers

- **Hash sign** should be used when each of multiple output files only contains records corresponding 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 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**
Chapter 42. Common Properties of Writers

- http://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.
  Proxy setting for ftp protocol.
- sftp://66.11.122.193:443//user:password@server/path/file.dat
  Proxy setting for sftp protocol.

**Writing to Dictionary**

- dict: keyName: source
  Writes data to a file URL specified in dictionary. Target file URL is retrieved from specified dictionary entry.
- dict: keyName: discrete
  Writes data to dictionary. Creates ArrayList<byte[]>
- dict: keyName: stream
  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.
Chapter 42. Common Properties of Writers

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 42.1. 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. You 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 42.2. Viewing Data as Plain Text

On the other hand, if you select the Grid option, you can select Filter expression, but no Charset.
Chapter 42. Common Properties of Writers

Figure 42.3. Viewing Data as Grid

The result can be as follows in the Plain text mode:

Figure 42.4. Plain Text Data Viewing

Or in the Grid mode, it can be like the following:

Figure 42.5. Grid Data Viewing

Note

If there are too many records to be displayed, you will see the Load more... blue text below the view. Clicking it, a new chunk of records is added behind the currently displayed ones. In Plain View, you can also scroll down to the bottom of the view (alternatively, by pressing Page Down) to have the records loaded.
The same can be done in some of the Readers. See Viewing Data on Readers (p. 290).

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. 68).

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. 292) 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 corresponding 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).
Chapter 42. Common Properties of Writers

• 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 UniversalDataReader 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. 477) allows optionally a transformation, which can only be written in Java.

See Java Interfaces for JMSWriter (p. 479) for more information about the interface.
Chapter 43. 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. 268) 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. 310)

- 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. 311)

Here we present an overview of all Transformers:

Table 43.1. Transformers Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleCopy (p. 618)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ExtSort (p. 574)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FastSort (p. 576)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SortWithinGroups (p. 620)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dedup (p. 561)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ExtFilter (p. 572)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Concatenate (p. 555)</td>
<td>✗</td>
<td>✗</td>
<td>1-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SimpleGather (p. 619)</td>
<td>✗</td>
<td>✗</td>
<td>1-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Merge (p. 580)</td>
<td>✗</td>
<td>✗</td>
<td>2-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Partition (p. 592)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>yes/no¹</td>
<td>yes/no¹</td>
</tr>
<tr>
<td>DataIntersection (p. 556)</td>
<td>✗</td>
<td>✗</td>
<td>2</td>
<td>3</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Aggregate (p. 552)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Reformat (p. 603)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Denormalizer (p. 563)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Chapter 43. Common Properties of Transformers

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivot (p. 599)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Normalizer (p. 585)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MetaPivot (p. 582)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Rollup (p. 606)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DataSampler (p. 559)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XSLTransformer (p. 622)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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. 592) 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. 598) 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. 272)). Mapping does not need to be done, records are mapped automatically.

- **DataIntersection** (p. 556) requires a transformation which can be written in both CTL and Java.

  See [CTL Templates for DataIntersection](p. 558) for more information about the transformation template.

- **Reformat** (p. 603) requires a transformation which can be written in both CTL and Java.

  See [CTL Templates for Reformat](p. 604) 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. 272)). Mapping must be defined for such port.

- **Denormalizer** (p. 563) requires a transformation which can be written in both CTL and Java.

  See [CTL Templates for Denormalizer](p. 565) for more information about the transformation template.

- **Normalizer** (p. 585) requires a transformation which can be written in both CTL and Java.

  See [CTL Templates for Normalizer](p. 586) for more information about the transformation template.

- **Rollup** (p. 606) requires a transformation which can be written in both CTL and Java.

  See [CTL Templates for Rollup](p. 608) 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. 272)). Mapping must be defined for such port.
Java Interfaces for Transformers

- **Partition** (p. 592) 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. 598) 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. 272)). Mapping does not need to be done, records are mapped automatically.

- **DataIntersection** (p. 556) requires a transformation which can be written in both CTL and Java.

  See [Java Interfaces for DataIntersection](p. 558) for more information about the interface.

- **Reformat** (p. 603) requires a transformation which can be written in both CTL and Java.

  See [Java Interfaces for Reformat](p. 605) 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. 272)). Mapping must be defined for such port.

- **Denormalizer** (p. 563) requires a transformation which can be written in both CTL and Java.

  See [Java Interfaces for Denormalizer](p. 571) for more information about the interface.

- **Normalizer** (p. 585) requires a transformation which can be written in both CTL and Java.

  See [Java Interfaces for Normalizer](p. 591) for more information about the interface.

- **Rollup** (p. 606) requires a transformation which can be written in both CTL and Java.

  See [Java Interfaces for Rollup](p. 616) 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. 272)). Mapping must be defined for such port.
Chapter 44. 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. 268) for detailed information about how transformation should be defined. All of the transformations in Joiners use common transformation template (CTL Templates for Joiners (p. 314)) and common Java interface (Java Interfaces for Joiners (p. 317)).

Here we present a brief overview of links to these options:

- Join Types (p. 313)
- Slave Duplicates (p. 313)
- CTL Templates for Joiners (p. 314)
- Java Interfaces for Joiners (p. 317)

Here we present an overview of all Joiners:

Table 44.1. Joiners Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApproximativeJoin</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>2-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtHashJoin</td>
<td>*</td>
<td>*</td>
<td>1-n</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExtMergeJoin</td>
<td>*</td>
<td>*</td>
<td>1-n</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LookupJoin</td>
<td>*</td>
<td>*</td>
<td>1 (virtual)</td>
<td>1-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBJoin</td>
<td>*</td>
<td>*</td>
<td>1 (virtual)</td>
<td>1-2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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 \textit{Join key} fields equal to the values of their slave counterparts are processed and sent out through the output port for joined records.

  For \textit{ApproximativeJoin}, the name of this attribute is \textit{Matching key}.

  The unmatched master records can be sent out through the optional output port for master records without a slave (in \textit{ApproximativeJoin}, \textit{LookupJoin} or \textit{DBJoin} only).

  The unmatched slave records can be sent out through the optional output port for slave records without a master (in \textit{ApproximativeJoin} only).

- **Left Outer Join**

  In this processing mode, only the master records in which the values of \textit{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 \textit{ApproximativeJoin}, the name of this attribute is \textit{Matching key}.

  The unmatched slave records can be sent out through the optional output port for slave records without a master (in \textit{ApproximativeJoin} only).

- **Full Outer Join**

  In this processing mode, all records, both the masters and the slaves, regardless of whether the values of \textit{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 \textit{ApproximativeJoin}, the name of this attribute is \textit{Matching key}.

### Important

- **Full outer join** mode is not allowed in \textit{LookupJoin} and \textit{DBJoin}.

### Note

Remember that \textit{Joiners} parse each pair of records (master and slave) in which the same fields of the \textit{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 \textit{Joiners}, sometimes more slave records have the same values of corresponding fields of \textit{Join key} \textit{(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.
Chapter 44. Common Properties of Joiners

Figure 44.1. Source Tab of the Transform Editor in Joiners

Table 44.2. Functions in Joiners, DataIntersection, and Reformat

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th>boolean init()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>integer transform()</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
</tr>
<tr>
<td><strong>Input Parameters</strong></td>
</tr>
<tr>
<td><strong>Returns</strong></td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
</tr>
</tbody>
</table>
### 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. 272) 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 |
**Chapter 44. Common Properties of Joiners**

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th>void preExecute()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>No</td>
</tr>
<tr>
<td>Input parameters</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called during each graph run before the transform is executed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void postExecute()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
</tr>
<tr>
<td>Input parameters</td>
</tr>
<tr>
<td>Returns</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Invocation</td>
</tr>
</tbody>
</table>

---

**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. 284).

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. 272) 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. 272) 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 45. 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 45.1. Cluster Components Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterGather (p. 718)</td>
<td>✔</td>
<td>✗</td>
<td>1-n (virtual)</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ClusterPartitioner (p. 720)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n (virtual)</td>
<td>yes/no¹</td>
<td>yes/no¹</td>
</tr>
</tbody>
</table>

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. 720) 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. 722) 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. 272)). 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. 720) 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. 722) 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. 272)). Mapping does not need to be done, records are mapped automatically.

  Records are sent through these virtual ports to Cluster nodes.
Chapter 46. Common Properties of Others

These components serve to fulfil some tasks that have not been mentioned already. We will describe them now. They have no common properties as they are heterogeneous group.

Only JavaExecute (p. 752) requires that a transformation is defined in this component. It must implement the following interface:

Java Interfaces for JavaExecute (p. 753)

Below we will present an overview of all Others:

Table 46.1. Others Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs?</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemExecute (p. 764)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-1</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JavaExecute (p. 752)</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DBExecute (p. 743)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>RunGraph (p. 756)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>1-2</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>HTTPConnector (p. 747)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-1</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WebServiceClient (p. 767)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-N</td>
<td>no²)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CheckForeignKey (p. 739)</td>
<td>✗</td>
<td>✗</td>
<td>2</td>
<td>1-2</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SequenceChecker (p. 760)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LookupTableReaderWriter (p. 754)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-n</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SpeedLimiter (p. 762)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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 59, Others (p. 738).
Chapter 47. Common Properties of Data Quality

The Data Quality is a group of components performing various tasks related to quality of your data - determining information about the data, finding and fixing problems etc. These components have no common properties as they perform a wide range of tasks.

Below we will present an overview of all Data Quality components:

Table 47.1. Data Quality Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Doctor 5 (p. 724)</td>
<td>-</td>
<td>❌</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EmailFilter (p. 728)</td>
<td>-</td>
<td>❌</td>
<td>1</td>
<td>0-2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ProfilerProbe (p. 733)</td>
<td>-</td>
<td>❌</td>
<td>1</td>
<td>1-n</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Legend

1) Component sends each data record to all connected output ports.

Go now to Chapter 58, Data Quality (p. 723).
Chapter 48. Common Properties of Job Control

The Job Control is a group of components managing various job types - executing, monitoring and optionally aborting ETL graphs, jobflows and interpreted scripts. Most of these components are tightly bound with jobflow (p. 240).

All execution components ExecuteGraph, ExecuteJobflow, ExecuteScript and few other Job Control components have similar approach to job execution management. Each of them has an optional input port. Each incoming token from this port is interpreted by an execution component and a respective job is started. Default execution settings are specified directly in various component attributes. This default settings can be overridden by values from incoming token - Input mapping attribute specifies the override. Results of successful jobs are sent to the first output port and unsuccessful job runs are sent to the second output port. Content of these output tokens is defined in Output mapping and Error mapping.

In case no input port is attached, only single job is started with execution settings specified directly in component attributes. In case the first output port is not connected, job results are printed out to log. And finally in case the second output port is not connected, first unsuccessful job causes failure of whole jobflow.

Redirect error output attribute can be used to route all successful and even unsuccessful job results to the first output port - Output mapping is used for all job executions.

Below we will present an overview of all Job control components:

Table 48.1. Job control Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier (p. 657)</td>
<td>✗</td>
<td>✗</td>
<td>1-n</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Condition (p. 660)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ExecuteGraph (p. 663)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ExecuteJobflow (p. 669)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ExecuteScript (p. 671)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Fail (p. 677)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>GetJobInput (p. 680)</td>
<td>-</td>
<td>✗</td>
<td>0</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>KillGraph (p. 682)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>KillJobflow (p. 686)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MonitorGraph (p. 688)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MonitorJobflow (p. 692)</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SetJobOutput (p. 694)</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>0</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Success (p. 696)</td>
<td>✗</td>
<td>✗</td>
<td>0-n</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TokenGather (p. 698)</td>
<td>✗</td>
<td>✗</td>
<td>1-n</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Go now to Chapter 55, Job Control (p. 656).
Chapter 49. Common Properties of File Operations

The File Operation components manipulate with files and directories.

An overview of all File Operation components is presented below:

Table 49.1. File Operations Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyFiles (p. 701)</td>
<td>0-1</td>
<td>0-2</td>
</tr>
<tr>
<td>CreateFiles (p. 705)</td>
<td>0-1</td>
<td>0-2</td>
</tr>
<tr>
<td>DeleteFiles (p. 708)</td>
<td>0-1</td>
<td>0-2</td>
</tr>
<tr>
<td>MoveFiles (p. 714)</td>
<td>0-1</td>
<td>0-2</td>
</tr>
<tr>
<td>ListFiles (p. 711)</td>
<td>0-1</td>
<td>1-2</td>
</tr>
</tbody>
</table>

Legend

1) Component sends each data record to all connected output ports.

Common attributes of File Operation components

For an overview of URL formats supported by File Operations, see Supported URL Formats for File Operations (p. 324).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input mapping</td>
<td>1)</td>
<td>defines mapping of input records to component attributes</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>1)</td>
<td>defines mapping of results to standard output port</td>
<td></td>
</tr>
<tr>
<td>Error mapping</td>
<td>1)</td>
<td>defines mapping of errors to error output port</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>if enabled, errors will be sent to the output port instead of the error port</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

Legend

1) If the mapping is omitted, default mapping based on identical names will be used.

Input Mapping

The operation will be executed for each input record. If the input edge is not connected, the operation will be performed exactly once.

The attributes of the components may be overridden by the values read from the input port, as specified by the Input mapping.

Output Mapping

It is essential to understand the meaning of records on the left-hand side of the Output mapping and Error mapping editor. There may be one or two records displayed.
The first record is only displayed if the component has an input edge connected, because it is the real input record which has been read from the edge. This record has Port 0 displayed in the Type column.

The other record on the left-hand side named Result is displayed always and is the result record generated by the component.

**Error Handling**

By default, the component will cause the graph to fail if it fails to perform the operation. This can be prevented by connecting the error port. If the error port is connected, the failures will be send to the error port and the component will continue. The standard output port may be also be used for error handling, if the Redirect error output option is enabled.

In case of a failure, the component will not execute subsequent operations unless the Stop processing on fail option is disabled. The information about skipped operations will be sent to the error output port.

Go now to Chapter 56, File Operations (p. 700).

---

**Supported URL Formats for File Operations**

The URL attributes may be defined using the URL File Dialog (p. 68).

Unless explicitly stated otherwise, URL attributes of File Operation components accept multiple URLs separated with a semicolon (';').

**Important**

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

Most protocols support wildcards: ? (question mark) matches one arbitrary character; * (asterisk) matches any number of arbitrary characters. Note that wildcard support and their syntax is protocol-dependent.

Here we present some examples of possible URL for File Operations:

**Local Files**

- /path/filename.txt
  
  One specified file.

- /path1/filename1.txt;/path2/filename2.txt
  
  Two specified files.

- /path/?.txt
  
  All files satisfying the mask.

- /path/*
  
  All files in the specified directory.

- /path/*/?.txt
  
  All .txt files in directories that satisfy the path? mask.

**Remote Files**
Common Properties of File Operations

- ftp://username:password@server/path/filename.txt
  Denotes filename.txt file on remote server connected via ftp protocol using username and password.

- ftp://username:password@server/dir/*\.txt
  Denotes all files satisfying the mask on remote server connected via ftp protocol using username and password.

- sftp://username:password@server/path/filename.txt
  Denotes filename.txt file on remote server connected via sftp protocol using username and password.

- sftp://username:password@server/path\?/filename\.txt
  Denotes all files filename.txt in directories satisfying the mask on remote server connected via sftp protocol using username and password.

- http://server/path/filename.txt
  Denotes filename.txt file on remote server connected via http protocol.

- https://server/path/filename.txt
  Denotes filename.txt file on remote server connected via https protocol.

Using Proxy in File Operations

- 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@protocolserver: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.

Sandbox Resources

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. A graph does not have to run on the node which has local access to the resource.
Chapter 50. 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 VIII. Component Reference
Chapter 51. Readers

We assume that you already know what components are. See Chapter 18, Components (p. 94) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

We can distinguish Readers according to what they can read:

- One component only generates data:
  - DataGenerator (p. 340) generates data.

Other Readers read data from files.

- Flat files:
  - UniversalDataReader (p. 397) reads data from flat files (delimited or fixed length).
  - ParallelReader (p. 380) reads data from delimited flat files using more threads.
  - ComplexDataReader (p. 332) 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. 376) reads data from flat files with a heterogeneous structure.

- Other files:
  - CloverDataReader (p. 330) reads data from files in Clover binary format.
  - SpreadsheetDataReader (p. 387) reads data from XLS or XLSX files.
  - XLSDataReader (p. 402) reads data from XLS or XLSX files.
  - DBFDataReader (p. 347) reads data from dBase files.
  - XMLExtract (p. 406) reads data from XML files using SAX technology.
  - XMLXPathReader (p. 431) reads data from XML files using XPath queries.

Other Readers unload data from databases.

- Databases:
  - DBInputTable (p. 349) unloads data from database using JDBC driver.
  - QuickBaseRecordWriter (p. 504) reads data from the QuickBase online database.
  - QuickBaseImportCSV (p. 502) reads data from the QuickBase online database using queries.
• **LotusReader** (p. 374) reads data from **Lotus Notes** or **Lotus Domino** database.

Other **Readers** receive JMS messages or read directory structure.

• JMS messages:
  • **JMSReader** (p. 362) converts JMS messages into data records.

• Directory structure:
  • **LDAPReader** (p. 371) converts directory structure into data records.

• Email messages:
  • **EmailReader** (p. 353) Reads email messages.
CloverDataReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

CloverDataReader reads data stored in our internal binary Clover data format files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloverDataReader</td>
<td>clover binary file</td>
<td>0</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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.
Chapter 51. Readers

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For correct data records</td>
<td>Output 0</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can use Autofilling Functions (p. 128).

CloverDataReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying what data source(s) will be read (flat file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 286).</td>
<td></td>
</tr>
<tr>
<td>Index file URL&lt;sup&gt;1)&lt;/sup&gt;</td>
<td></td>
<td>Name of the index file, including path. See Supported File URL Formats for Readers (p. 286). See also Output File Structure (p. 441) for more information about index file names.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be read. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start record</td>
<td></td>
<td>Has exclusive meaning: Last record before the first that is already read. Has lower priority than Number of skipped records.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Final record</td>
<td></td>
<td>Has inclusive meaning: Last record to be read. Has lower priority than Max number of records.</td>
<td>all (default)</td>
</tr>
</tbody>
</table>

Legend:

1) Please note this is a deprecated attribute. If it is not specified, all records must be read.
ComplexDataReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purpose, see Readers Comparison (p. 286).

Short Summary

ComplexDataReader reads non-homogeneous data from files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexDataReader</td>
<td>flat file</td>
<td>1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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. 295).

Icon
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See Reading from Input Port (p. 289).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any (Out0)</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>no</td>
<td>For correct data records</td>
<td>Any (Out1-OutN)</td>
</tr>
</tbody>
</table>

### Legend:

1): Metadata on output ports can use Autofilling Functions (p. 128). Note: source_timestamp and source_size functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

### ComplexDataReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>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. 286).</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Transform</td>
<td></td>
<td>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. 334).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>The encoding of records that are read.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines steps that are done when an error occurs. See Data Policy (p. 295) 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).</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Trim strings</td>
<td></td>
<td>Specifies whether leading and trailing whitespaces should be removed from strings before inserting them to data fields. See Trimming Data (p. 399).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Quoted strings</td>
<td></td>
<td>Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If true, special characters are removed when read by the component (they are not treated as delimiters).</td>
<td>false</td>
</tr>
<tr>
<td>Example: To read input data &quot;25&quot;</td>
<td>&quot;John&quot;., switch Quoted strings to true and set Quote character to &quot;. This will produce two fields: 25</td>
<td>John.</td>
<td></td>
</tr>
<tr>
<td>By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 157).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quote character</td>
<td></td>
<td>Specifies which kind of quotes will be permitted in Quoted strings. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 157).</td>
<td>both</td>
</tr>
</tbody>
</table>

**Advanced**
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip leading blanks</td>
<td></td>
<td>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 <strong>Trim strings</strong> is used. See <strong>Trimming Data</strong> (p. 399).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Skip trailing blanks</td>
<td></td>
<td>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 <strong>Trim strings</strong> is used. See <strong>Trimming Data</strong> (p. 399).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>The maximum number of tolerated error records on the input. The attribute is applicable only if <strong>Controlled Data Policy</strong> is being used.</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Treat multiple delimiters as one</td>
<td></td>
<td>If a field is delimited by a multiplied delimiter character, it will be interpreted as a single delimiter if this attribute is <strong>true</strong>.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Verbose</td>
<td></td>
<td>By default, not so complex error notification is provided and the performance is fairly high. However, if switched to <strong>true</strong>, more detailed information with lower performance will be provided.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Selector code</td>
<td>1)</td>
<td>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 <strong>Selector</strong> (p. 338).</td>
<td></td>
</tr>
<tr>
<td>Selector URL</td>
<td>1)</td>
<td>The name and path to an external file containing a selector code written in Java. To learn more about the Selector, see <strong>Advanced Description</strong> (p. 334).</td>
<td></td>
</tr>
<tr>
<td>Selector class</td>
<td>1)</td>
<td>The name of an external class containing the Selector. To learn more about the Selector, see <strong>Advanced Description</strong> (p. 334).</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td></td>
<td>The path to an external file which defines state transitions in the state machine.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td></td>
<td>The name of a Java class that defines state transitions in the state machine.</td>
<td></td>
</tr>
<tr>
<td>Selector properties</td>
<td></td>
<td>Allows you to instantly edit the current Selector in the <strong>State transitions</strong> window.</td>
<td></td>
</tr>
<tr>
<td>State metadata</td>
<td></td>
<td>Allows you to instantly edit the metadata and states assigned to them in the <strong>State transitions</strong> window.</td>
<td></td>
</tr>
</tbody>
</table>

**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 each other. 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. 337) 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. 337)

**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 51.1. Example State Function**

```java
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
    } else {
        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 
acording to the current state and the value of $seq$.

**Selector**

By default, the selector takes the initial part of the data being read (a *prefix*) to decide about the next state of the 
state machine. This is implemented as `PrefixInputMetadataSelector`.

![Figure 51.1. Configuring prefix selector in ComplexDataReader. Rules are defined in the Selector 
properties pane. Notice the two extra attributes for regular expressions.](image)

The selector can be configured by creating a list of *rules*. Every rule consists of:

1. a state in which it is applied *(From state)*

2. specification of *Prefix* and its *Type*. A prefix may be specified as a plain text, a sequence of bytes written in 
hexadecimal code, or using a regular expression. These are the *Types* of the rules. The prefix can be empty 
meaning the rule will be always applied no matter the input.

3. the next state of the automaton *(Go to state)*

As the selector is invoked, it goes through the list of rules (top to bottom) and searches for the first applicable rule. 
If successful, the automaton switches to the target state of the selected rule.
Caution

Be very careful: the remaining rules are not checked at all. Therefore you have to think thoroughly over the order of rules. If a rule with an empty prefix appears in the list, the selector will not get to the rules below it. Generally, the least specific rules should be at the end of the list. See example below.

Example 51.2.

Let us have two rules and assume both are applicable in any state:

- `{1, 3}`PATH (a regular expression)
- APATHY

If rules are defined in this exact order, the latter will never be applied because the regular expression also matches the first five letters of "APATHY".

Note

Because some regular expressions can match sequences of characters of arbitrary length, two new attributes were introduced to prevent PrefixInputMetadataSelector from reading the whole input. These attributes are optional, but it is strongly recommended to use at least one of them, otherwise the selector always reads the whole input whenever there is a rule with a regular expression. The two attributes: Maximum prefix length and Record delimiter. When matching regular expressions, the selector reads ahead at most Maximum prefix length of characters (0 meaning "unlimited"). The reading is also terminated if a Record delimiter is encountered.
DataGenerator

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

**Short Summary**

DataGenerator generates data.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataGenerator</td>
<td>generated</td>
<td>0</td>
<td>1-N</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>1)</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**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. 272) 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. 272).

**Icon**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For generated data records</td>
<td>Any&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>no</td>
<td>For generated data records</td>
<td>Output 0</td>
</tr>
</tbody>
</table>

**Legend:**

1): Metadata on all output ports can use Autofilling Functions (p. 128).
## DataGenerator Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generator</td>
<td>1)</td>
<td>Definition of records should be generated written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Generator URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the way how records should be generated written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Generator class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be generated.</td>
<td></td>
</tr>
<tr>
<td>Number of records to generate</td>
<td>yes</td>
<td>Number of records to be generated.</td>
<td></td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record pattern</td>
<td>2)</td>
<td>String consisting of all fields of generated records that are constant. It does not contain values of random or sequence fields. User should define random and sequence fields first. See Record Pattern (p. 342) for more information.</td>
<td></td>
</tr>
<tr>
<td>Random fields</td>
<td>2)</td>
<td>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. 343) for more information. Example of one individual field range: $salary:=random(&quot;0&quot;,&quot;20000&quot;).</td>
<td></td>
</tr>
<tr>
<td>Sequence fields</td>
<td>2)</td>
<td>Fields generated by sequence. They are defined as the sequence of individual field mappings ($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. 342) for more information.</td>
<td></td>
</tr>
<tr>
<td>Random seed</td>
<td>2)</td>
<td>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.</td>
<td>0-N</td>
</tr>
</tbody>
</table>

**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. 343) or Java Interfaces for DataGenerator (p. 346) for more information.

See also Defining Transformations (p. 268) 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 51.2. Sequences Dialog](image1.png)

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 51.3. A Sequence Assigned](image2.png)
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.

![Edit Key Dialog](image)

**Figure 51.4. 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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, [CTL - CloverETL Transformation Language](p. 771). (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 51.5. Source Tab of the Transform Editor in DataGenerator

### Table 51.1. Functions in DataGenerator

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th>boolean init()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>No</td>
</tr>
<tr>
<td>Description</td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td>Returns</td>
<td>true</td>
</tr>
</tbody>
</table>

#### integer generate()

<table>
<thead>
<tr>
<th>Required</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameters</td>
<td>none</td>
</tr>
<tr>
<td>Returns</td>
<td>Integer numbers. See Return Values of Transformations (p. 272) for detailed information.</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called repeatedly for each output record</td>
</tr>
<tr>
<td>Description</td>
<td>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().</td>
</tr>
</tbody>
</table>
### CTL Template Functions

**Example**

```plaintext
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. 272) 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**

```plaintext
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

<table>
<thead>
<tr>
<th>Input parameters</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>Description</td>
<td>Should be used to free any resources allocated within the <code>preExecute()</code> function.</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called during each graph run after the entire transform was executed.</td>
</tr>
</tbody>
</table>

#### 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. 284)

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. 272) 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.
Chapter 51. Readers

DBFDataReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

DBFDataReader reads data from fixed-length dbase files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBFDataReader</td>
<td>dBase file</td>
<td>0-1</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See Reading from Input Port (p. 289).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any¹</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For correct data records</td>
<td>Output 0</td>
</tr>
</tbody>
</table>
Legend:

1): Metadata on output ports can use **Autofilling Functions** (p. 128). Note: **source_timestamp** and **source_size** functions only work when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

**DBFDataReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying what data source(s) will be read (dbase file, console, input port, dictionary). See <strong>Supported File URL Formats for Readers</strong> (p. 286).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read.</td>
<td>IBM850 (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See <strong>Data Policy</strong> (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
</tbody>
</table>

| **Advanced**                |     |                                                                             |                                      |
| Number of skipped records   |     | Number of records to be skipped continuously throughout all source files. See **Selecting Input Records** (p. 294). | 0-N                                  |
| Max number of records       |     | Maximum number of records to be read continuously throughout all source files. See **Selecting Input Records** (p. 294). | 0-N                                  |
| Number of skipped records per source |     | Number of records to be skipped from each source file. See **Selecting Input Records** (p. 294). | 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. 294). | 0-N                                  |
| Incremental file            | 1)  | Name of the file storing the incremental key, including path. See **Incremental Reading** (p. 293). |                                      |
| Incremental key             | 1)  | Variable storing the position of the last read record. See **Incremental Reading** (p. 293). |                                      |

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

DBInputTable unloads data from database using JDBC driver.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBInputTable</td>
<td>database</td>
<td>0-1</td>
<td>1-n</td>
<td>✓</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

1) Sending each data record to every connected output port
2) Sending data records to output ports according to Return Values of Transformations (p. 272)

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-1</td>
<td>❌</td>
<td>Incoming queries to be used in the SQL query attribute. When the input port is connected, Query URL should be specified as e.g. port:$0.fieldName:discrete. See Reading from Input Port (p. 289).</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✓</td>
<td>for correct data records</td>
<td>equal metadata 1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>❌</td>
<td>for correct data records</td>
<td></td>
</tr>
</tbody>
</table>

1) Output metadata can use Autofilling Functions (p. 128)

DBInputTable Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB connection</td>
<td>✚</td>
<td>ID of the database connection to be used to access the database</td>
<td></td>
</tr>
<tr>
<td>Query URL</td>
<td>1)</td>
<td>Name of external file, including path, defining SQL query.</td>
<td></td>
</tr>
<tr>
<td>SQL query</td>
<td>1)</td>
<td>SQL query defined in the graph. See SQL Query Editor (p. 351) for detailed information.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Query source charset</td>
<td></td>
<td>Encoding of external file defining SQL query.</td>
<td></td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See Data Policy (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Fetch size</td>
<td></td>
<td>Specifies the number of records that should be fetched from the database at once.</td>
<td>20</td>
</tr>
<tr>
<td>Incremental file</td>
<td>2)</td>
<td>Name of the file storing the incremental key, including path. See Incremental Reading (p. 293).</td>
<td></td>
</tr>
<tr>
<td>Incremental key</td>
<td>2)</td>
<td>Variable storing the position of the last read record. See Incremental Reading (p. 293).</td>
<td></td>
</tr>
<tr>
<td>Auto commit</td>
<td></td>
<td>By default, your SQL queries are committed immediately. If you need to perform more operations inside one transaction, switch this attribute to false.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

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**

  When order of CloverETL metadata fields and database columns in select statement is same and data types are compatible, implicit mapping can be used which performs positional mapping. Standard SQL query syntax should be used:

  - select * from table [where dbfieldJ = ? and dbfieldK = somevalue]
  - select column3, column1, column2, ... from table [where dbfieldJ = ? and dbfieldK = somevalue]

  See SQL Query Editor (p. 351) 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:

  ```
  ```

  See SQL Query Editor (p. 351) 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](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 51.6. Generated Query with Question Marks](image)

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:

![Generated Query with Output Fields](image)

**Figure 51.7. 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

EmailReader reads a store of email messages, either locally from a delimited flat file, or on an external server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmailReader</td>
<td>✗</td>
<td>✗</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Abstract

EmailReader is a component that enables 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 delimiter, and each message will be passed one by one into the EmailReader for parsing and processing.
Case Two: If an edge is not connected to 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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>For inputting email messages from a flat file</td>
<td>String field</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>The content port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>The attachment port</td>
<td>Any</td>
</tr>
</tbody>
</table>

### EmailReader Attributes

Whether many of the attributes are required or not depends solely on the configuration of the component. See Ports (p. 353): 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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server Type</td>
<td></td>
<td>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.</td>
<td>POP3, IMAP</td>
</tr>
<tr>
<td>Server Name</td>
<td></td>
<td>The hostname of the server.</td>
<td>e.g. imap.google.com</td>
</tr>
<tr>
<td>Server Port</td>
<td></td>
<td>Specifies the port used to connect to an external server. If left blank, a default port will be used.</td>
<td>Integers</td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td>Specifies the security protocol used to connect to the server.</td>
<td>NONE, SSL, STARTTLS, SSL +STARTTLS</td>
</tr>
<tr>
<td>User Name</td>
<td></td>
<td>Username to connect to the server (if authorization is required)</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password to connect to server (if authorization is required)</td>
<td></td>
</tr>
<tr>
<td>Fetch Messages</td>
<td></td>
<td>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.</td>
<td>NEW, ALL</td>
</tr>
<tr>
<td>Field Mapping</td>
<td>Yes</td>
<td>Defines how parts of the email (standard and user-defined) will be mapped to Clover fields. See Mapping Fields (p. 355).</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP3 Cache File</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
Advanced Description

Mapping Fields

If you edit the Field Mapping attribute, you will get the following simple dialog:

![Figure 51.8. Mapping to Clover fields in EmailReader](image)

In its two tabs - Message and Attachments - you map incoming email fields to Clover fields by a simple drag and drop. Notice the buttons on the right hand side allowing you to Cancel all mappings. Auto mapping is automatically performed when you first open this window. Finally, remember you will only see metadata fields in Attachments if you are using the second output port (see Ports (p. 353) to learn why).

**Note**

User-defined Fields let you handle all fields that can occur besides the Standard ones. Example: custom fields in the email header.

**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.
JavaBeanReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

JavaBeanReader reads a JavaBeans hierarchical structure which is stored in a dictionary. That allows dynamic data interchange between Clover graphs and external environment, such as cloud. The dictionary you are reading to serves as an interface between the outer (e.g. cloud) and inner worlds (Clover).

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaBeanReader</td>
<td>dictionary</td>
<td>0</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

JavaBeanReader reads data from JavaBeans through a dictionary. It maps Java attributes / collection elements to output records based on a mapping you define. You do not have to map the whole input file - you use XPath expressions to select only the data you need. The component sends data to different connected output records as defined by your mapping.

The mapping process is similar to the one in XMLXPathReader (p. 431).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Successfully read records.</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td></td>
<td>Connect other output ports if your mapping requires that.</td>
<td>Any. Each port can have different metadata.</td>
</tr>
</tbody>
</table>
## JavaBeanReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary source</td>
<td>yes</td>
<td>The dictionary you want to read JavaBeans from.</td>
<td>Name of a dictionary you have previously defined.</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error on reading occurs. See Data Policy (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Mapping the input JavaBeans structure to output ports. See JavaBeanReader Mapping Definition (p. 358) for more information.</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition.</td>
<td></td>
</tr>
<tr>
<td>Implicit mapping</td>
<td></td>
<td>By default, you have to manually map input elements even to Clover fields of the same name. If you switch to true, JSON-to-Clover mapping on matching names will be performed automatically. That can save you a lot of effort in long and well-structured JSON files. See JSON Mapping - Specifics (p. 367).</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.
Advanced Description

JavaBeanReader Mapping Definition

1. Every **Mapping** definition consists of `<Context>` tags which contain also some attributes and allow mapping of element names to Clover fields. Nested structure of `<Context>` tags is similar to the nested structure of elements in input JavaBeans.

2. Each `<Context>` tag can surround a serie of nested `<Mapping>` tags. These allow to rename JavaBeans elements to Clover fields. However, **Mapping** does not need to copy the whole input structure, it can start at an arbitrary depth in the tree.

3. Each of these `<Context>` and `<Mapping>` tags contains some [JavaBeanReader Context Tag Attributes](p. 359) and [JavaBeanReader Mapping Tag Attributes](p. 360), respectively.

Example 51.3. Example Mapping in JavaBeanReader

```xml
<Context xpath="/employees" outPort="0" sequenceId="empSeq" sequenceField="id">
  <Mapping xpath="firstName" cloverField="firstName"/>
  <Mapping xpath="lastName" cloverField="lastName"/>
  <Mapping xpath="salary" cloverField="salary"/>
  <Mapping xpath="jobTitle" cloverField="jobTitle"/>
  <Context xpath="children" outPort="1" parentKey="id" generatedKey="empID">
    <Mapping xpath="name" cloverField="chname"/>
    <Mapping xpath="age" cloverField="age"/>
  </Context>
  <Context xpath="benefits" outPort="2" parentKey="id" generatedKey="empID">
    <Mapping xpath="car" cloverField="car"/>
    <Mapping xpath="cellPhone" cloverField="mobilephone"/>
    <Mapping xpath="monthlyBonus" cloverField="monthlyBonus"/>
    <Mapping xpath="yearlyBonus" cloverField="yearlyBonus"/>
  </Context>
  <Context xpath="projects" outPort="3" parentKey="id" generatedKey="empID">
    <Mapping xpath="name" cloverField="projName"/>
    <Mapping xpath="manager" cloverField="projManager"/>
    <Mapping xpath="start" cloverField="Start"/>
    <Mapping xpath="end" cloverField="End"/>
    <Mapping xpath="customers" cloverField="customers"/>
  </Context>
</Context>
```

**Important**

If you switch **Implicit mapping** to `true`, elements (e.g. `salary`) will be automatically mapped onto fields of the same name (`salary`) and you do **not** have to write:

```xml
<Mapping xpath="salary" cloverField="salary"/>
```

and you map explicitly only to populate fields with data from distinct elements.

4. **JavaBeanReader Context Tags and Mapping Tags**

   - **Empty Context Tag (Without a Child)**

```xml
<Context xpath="xpathexpression" JavaBeanReader Context Tag Attributes (p. 359) />
```
• Non-Empty Context Tag (Parent with a Child)

```xml
<Context xpath="xpathexpression" JavaBeanReader Context Tag Attributes (p. 359) >

(nested Context and Mapping elements (only children, parents with one or more children, etc.)
</Context>
```

• Empty Mapping Tag (Renaming Tag)

  • xpath is used:

```xml
<Mapping xpath="xpathexpression" JavaBeanReader Mapping Tag Attributes (p. 360) />
```

  • nodeName is used:

```xml
<Mapping nodeName="elementname" JavaBeanReader Mapping Tag Attributes (p. 360) />
```

5. JavaBeanReader Context Tag and Mapping Tag Attributes

1) JavaBeanReader 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"`

2) **JavaBeanReader 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.

JavaBeans node that should be mapped to Clover field.

Example: `nodeName="salary"`

* cloverField

Required

Clover field to which JavaBeans node should be mapped.

Name of the field in the corresponding level.

Example: `cloverFields="SALARY"`

**Reading Mutlivalue Fields**

As of Clover 3.3, reading multivalue fields is supported - you can read only lists, however (see Multivalue Fields (p. 163)).

**Note**

Reading maps is handled as reading pure `string` (for all data types as map's values).
Example 51.4. Reading lists with JavaBeanReader

An example input file containing e.g. a list of three elements: John, Vicky, Brian
can be read back by the component with this mapping:

```xml
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

where `attendanceList` is a field of your metadata. The metadata has to be assigned to the component's output edge. After you run the graph, the field will get populated by data like this (that what you will see in View data):

[John, Vicky, Brian]
JMSReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

JMSReader converts JMS messages into Clover data records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs 1)</th>
<th>Different to different outputs 2)</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSReader</td>
<td>jms messages</td>
<td>0</td>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any 1)</td>
</tr>
</tbody>
</table>
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. 128).

**JMSReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS connection</td>
<td>yes</td>
<td>ID of the JMS connection to be used.</td>
<td></td>
</tr>
<tr>
<td>Processor code</td>
<td>1)</td>
<td>Transformation of JMS messages to records written in the graph in Java.</td>
<td></td>
</tr>
<tr>
<td>Processor URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the transformation of JMS messages to records written in Java.</td>
<td></td>
</tr>
<tr>
<td>Processor class</td>
<td>1)</td>
<td>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.</td>
<td>JmsMsg2DataRecordProperties (default)</td>
</tr>
<tr>
<td>JMS message selector</td>
<td></td>
<td>Standard JMX &quot;query&quot; 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 <a href="http://docs.oracle.com/javaee/1.4/api/javax/jms/Message.html">http://docs.oracle.com/javaee/1.4/api/javax/jms/Message.html</a> for more information.</td>
<td></td>
</tr>
<tr>
<td>Processor source charset</td>
<td></td>
<td>Encoding of external file containing the transformation in Java.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Message charset</td>
<td></td>
<td>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.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Max msg count</td>
<td></td>
<td>Maximum number of messages to be received. 0 means without limitation. See Limit of Run (p. 364) for more information.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Timeout</td>
<td></td>
<td>Maximum time to receive messages in milliseconds. 0 means without limitation. See Limit of Run (p. 364) for more information.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Message body field</td>
<td></td>
<td>Name of the field to which message body should be written. This attribute is used by the default processor implementation (JmsMsg2DataRecordProperties). If no <strong>Message body field</strong> 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.</td>
<td>bodyField (default)</td>
</tr>
</tbody>
</table>

**Legend:**
1) One of these may be set. Any of these transformation attributes implements a JmsMsg2DataRecord interface.

See Java Interfaces for JMSReader (p. 364) for more information.

See also Defining Transformations (p. 268) 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. 284).

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. 364) 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.
Chapter 51. Readers

JSONReader

Commercial Component

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)

• Chapter 40, Common Properties of Most Components (p. 264)

• Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

JSONReader reads data in the Java Script Object Notation - JSON format, typically stored in a *.json file. JSON is a hierarchical text format where values you want to read are stored either in name-value pairs or arrays. Arrays are just the caveat in mapping - see Handling arrays (p. 369). JSON objects are often repeated - that is why you usually map to more than one output port.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Exact to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSONReader</td>
<td>JSON file</td>
<td>0-1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

JSONReader takes the input JSON and internally converts it to DOM. Afterwards, you use XPath expressions to traverse the DOM tree and select which JSON data structures will be mapped to Clover records.

DOM contains elements only, not attributes. As a consequence, remember that you XPath expressions will never contain @.

Note that the whole input is stored in memory and therefore the component can be memory-greedy.

Icon

Ports
<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Optional. For port reading.</td>
<td>Only one field (byte or cbyte or string) is used. The field name is used in File URL to govern how the input records are processed - one of these modes: discrete, source or stream. See Reading from Input Port (p. 289).</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Successfully read records.</td>
<td>Any.</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>Successfully read records.</td>
<td>Any. Each output port can have different metadata.</td>
</tr>
</tbody>
</table>

**JSONReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Specifies which data source(s) will be read (a JSON file, dictionary or port). See Supported File URL Formats for Readers (p. 286) and Notes and Limitations (p. 370) .</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read. JSON automatically recognizes the family of UTF-* encodings (Auto). If your input uses another charset, explicitly specify it in this attribute yourself.</td>
<td>Auto (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See Data Policy (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition.</td>
<td></td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Mapping the input JSON structure to output ports. See Advanced Description (p. 367).</td>
<td></td>
</tr>
<tr>
<td>Implicit mapping</td>
<td></td>
<td>By default, you have to manually map JSON elements even to Clover fields of the same name. If you switch to true, JSON-to-Clover mapping on matching names will be performed automatically. That can save you a lot of effort in long and well-structured JSON files. See JSON Mapping - Specifics (p. 367).</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these has to be specified. If both are specified, Mapping URL has a higher priority.
Advanced Description

JSON is a representation for tree data as every JSON object can contain other nested JSON objects. Thus, the way you create `JSONReader` mapping is similar to reading XML and other tree formats. JSONReader configuration resembles `XMLXPathReader` configuration. The basics of mapping are:

- `<Context>` element chooses elements in the JSON structure you want to map.
- `<Mapping>` element maps those JSON elements (selected by `<Context>`) to Clover fields.
- Both use XPath expressions (p. 368).

You will see mapping instructions and examples when you edit the `Mapping` attribute for the first time.

JSON Mapping - Specifics

- **Important**
  
  The first `<Context>` element of your mapping has a fixed format. There are only two ways how to set its `xpath` for the component to work properly:

  - `xpath="/root/object"` (if root in JSON structure is an object)
  - `xpath="/root/array"` (if root in JSON structure is an array)

  Example JSON:

  ```
  [{ "value" : 1},
   { "value" : 2}]
  ```

  **JSONReader** mapping:

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <Context outPort="0" xpath="/root/array">
    <Mapping cloverField="cloverValue" xpath="value"/>
  </Context>
  ```

  (considering `cloverValue` is a field in metadata assigned to the output edge)

  - To read data from regular name-value pairs, first remember to set your position in the JSON structure to a correct depth - e.g. `<Context xpath="zoo/animals/tiger">`.

    Optionally, you can map the subtree of `<Context>` to the output port - e.g. `<Context xpath="childObjects" outPort="2">`.

    Do the `<Mapping>`: select a name-value pair in `xpath`. Next, send the value to Clover using `cloverField`; e.g. `<Mapping cloverField="id" xpath="nestedObject">`.

    Example JSON:
Chapter 51. Readers

```
{
  "property" : 1,
  "innerObject" : {
    "property" : 2
  }
}
```

**JSONReader** mapping:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Context outPort="0" xpath="/root/object">
  <Mapping cloverField="property" xpath="property"/>
  <Context xpath="innerObject">
    <Mapping cloverField="propertyOfInnerObject" xpath="property"/>
  </Context>
</Context>
```

- XPath expressions - remember that you do not use the @ symbol to access 'attributes' as there are none. In order to select objects with specific values you will write mapping in a way like:

```
<Context xpath="//website[uri='http://www.w3.org/']" outPort="1">
  <Mapping cloverField="dateUpdated" xpath="dateUpdated" />
  <Mapping cloverField="title" xpath="title"/>
</Context>
```

The XPath in the example selects all elements *website* (no matter how deep in the JSON they are) whose *uri* matches the given string. Next, it sends its two elements (*dateUpdated* and *title*) to respective metadata fields on port 1.

As has already been mentioned, JSON is internally converted into a XML DOM. Since not all JSON names are valid XML element names, the names are encoded. Invalid characters are replaced with with escape sequences of the form \_xHHHH where HHHH is a hexadecimal Unicode code point. These sequences must therefore also be used in JSONReader’s XPath expressions. Note that underscore (\_) has become a special escape character and must itself be escaped with \_x005f.

The XPath `name()` function can be used to read the names of properties of JSON objects (for a description of XPath functions on nodes, see http://www.w3schools.com/xpath/xpath_functions.asp#node). However, the names may contain escape sequences, as described above. JSONReader offers two functions to deal with them, the functions are available from http://www.cloveretl.com/ns/TagNameEncoder namespace which has to be declared using the `namespacePaths` attribute, as will be shown below. These functions are the `decode(string)` function, which can be used to decode \_xHHHH escape sequences, and its counterpart, the `encode(string)` function, which escapes invalid characters.

For example, let’s try to process the following structure:

```
{"map" : { "0" : 2 , "7" : 1 , "16" : 1 , "26" : 3 , "38" : 1 }}
```
A suitable mapping could look like this:

```xml
<Context xpath="/root/object/map/*" outPort="0" namespacePaths='tag="http://www.cloveretl.com/ns/TagNameEncoder"'>
  <Mapping cloverField="key" xpath="tag:decode(name())" />
  <Mapping cloverField="value" xpath="." />
</Context>
```

The mapping maps the names of properties of "map" ("0", "7", "16", "26" and "38") to the field "key" and their values (2, 1, 1, 3 and 1, resp.) to the field "value".

• **Implicit mapping** - if you switch the component's attribute to `true`, you can save a lot of space because mapping of JSON structures to fields of the same name:

```xml
<Mapping cloverField="salary" xpath="salary"/>
```

will be performed automatically (i.e. you do not write the mapping code above).

**Handling arrays**

• Once again, remember that JSON structures are wrapped either by objects or arrays. Thus, your mapping has to start in one of the two ways (see JSON Mapping - Specifics (p. 367)):

```xml
<Context xpath="/root/object">
<Context xpath="/root/array">
```

• Nested arrays - if you have two or more arrays inside each other, you can reach values of the inner ones by repeatedly using a single name (of the array on the very top). Therefore in XPath, you will write constructs like: `arrayName/arrayName/.../arrayName` depending on how many arrays are nested. Example:

**JSON:**

```json
{
  "commonArray" : [ "hello", "hi", "howdy" ],
  "arrayOfArrays" : [ [ "val1", "val2", "val3" ] , [""], [ "val5", "val6" ] ]
}
```

**JSONReader** mapping:
Notice the usage of dot in mapping (p. 417). This is the only mapping which produces results you expect, i.e. on port 1:

![Figure 51.9. Example mapping of nested arrays - the result.](image)

- Null and empty elements in arrays - in Figure 51.9, "Example mapping of nested arrays - the result" (p. 370), you could notice that an empty string inside an array (i.e. `""`) populates a field with an empty string (record 4 in the figure).

Null values (i.e. `[ ]`), on the other hand, are completely skipped. **JSONReader** treats them as if they were not in the source.

### Notes and Limitations

- **JSONReader** reads data from JSON contained in a file, dictionary or port. If you are reading from a port or dictionary, always set **Charset** explicitly (otherwise you will get errors). There is no autodetection.

- If your metadata contains the underscore `_'`, you will be warned. Underscore is an illegal character in **JSONReader** mapping. You should either:
  a) Remove the character.
  b) Replace it e.g. by the dash `-`.
  c) Replace the underscore by its Unicode representation: `_x005f`.

---

Chapter 51. Readers
LDAPReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

**Short Summary**

*LDAPReader* reads information from an LDAP directory.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPReader</td>
<td>LDAP directory tree</td>
<td>0</td>
<td>1-n</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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. 272) 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**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any(^1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For correct data records</td>
<td>Output 0</td>
</tr>
</tbody>
</table>
Legend:

1): Metadata on the output must precisely describe the structure of the read object. Metadata can use Autofilling Functions (p. 128).

**LDAPReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDAP URL</td>
<td>yes</td>
<td>LDAP URL of the directory.</td>
<td>ldap://host:port/</td>
</tr>
<tr>
<td>Base DN</td>
<td>yes</td>
<td>Base <em>Distinguished Name</em> (the root of your LDAP tree). It is a comma separated list of <em>attribute=value</em> pairs referring 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.</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>yes</td>
<td><em>attribute=value</em> 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.</td>
<td></td>
</tr>
<tr>
<td>Scope</td>
<td></td>
<td>Scope of the search request. By default, only one object is searched. If <em>onelevel</em>, the level immediately below the distinguished name, if <em>subtree</em>, the whole subtree below the distinguished name is searched.</td>
<td>object (default)</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td>User DN to be used when connecting to the LDAP directory. Similar to the following: cn=john.smith,dc=example,dc=com</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password to be used when connecting to the LDAP directory.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-value separator</td>
<td></td>
<td><strong>LDAPReader</strong> can handle keys with multiple values. These are delimited by this string or character. &lt;none&gt; 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.</td>
<td>&quot;&quot; (default)</td>
</tr>
<tr>
<td>Alias handling</td>
<td></td>
<td>to control how aliases (leaf entries pointing to another object in the namespace) are dereferenced</td>
<td>always</td>
</tr>
<tr>
<td>Referral handling</td>
<td></td>
<td>By default, links to other servers are ignored. If <em>follow</em>, the referrals are processed.</td>
<td>ignore (default)</td>
</tr>
</tbody>
</table>

**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.
Chapter 51. Readers

LotusReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Chapter 51, Readers (p. 328).

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LotusReader</td>
<td>Lotus Domino</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

1) Sending each data record to every connected output port
2) Sending data records to output ports according to Return Values of Transformations (p. 272)

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 library 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 24, Lotus Connections (p. 185)).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>✔️</td>
<td>for read data records</td>
<td></td>
</tr>
</tbody>
</table>
## LotusReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domino connection</td>
<td>✓</td>
<td>ID of the connection to the Lotus Domino database.</td>
<td></td>
</tr>
<tr>
<td>View</td>
<td>✓</td>
<td>The name of the View in Lotus database from which the data records will be read.</td>
<td></td>
</tr>
<tr>
<td>Multi-value read mode</td>
<td>✗</td>
<td>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.</td>
<td>Read all values (default)</td>
</tr>
<tr>
<td>Multi-value separator</td>
<td>✗</td>
<td>A string that will be used to separate values from multi-value Lotus fields.</td>
<td>&quot;,&quot; (default)</td>
</tr>
</tbody>
</table>
MultiLevelReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

MultiLevelReader reads data from flat files with a heterogeneous structure.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiLevelReader</td>
<td>flat file</td>
<td>1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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. 295) for more detailed information.

Icon


## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See [Reading from Input Port](p. 289).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any(Out0)¹</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>no</td>
<td>For correct data records</td>
<td>Any(Out1-OutN)¹</td>
</tr>
</tbody>
</table>

### Legend:

1) Metadata on all output ports can use [Autofilling Functions](p. 128). Note: `source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

## MultiLevelReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying what data source(s) will be read (flat file, console, input port, dictionary). See [Supported File URL Formats for Readers](p. 286).</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See [Data Policy](p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Selector code</td>
<td>1)</td>
<td>Transformation of rows of input data file to data records written in the graph in Java.</td>
<td></td>
</tr>
<tr>
<td>Selector URL</td>
<td>1)</td>
<td>Name of external file, including path, defining the transformation of rows of input data file to data records written in Java.</td>
<td></td>
</tr>
<tr>
<td>Selector class</td>
<td>1)</td>
<td>Name of external class defining the transformation of rows of input data file to data records.</td>
<td>PrefixMultiLevel (default)</td>
</tr>
<tr>
<td>Selector properties</td>
<td></td>
<td>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 series of characters from the beginning of the row contained in the flat file that enable differentiate groups of records.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped continuously throughout all source files. See [Selecting Input Records](p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be read continuously throughout all source files. See [Selecting Input Records](p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Number of skipped records per source</td>
<td></td>
<td>Number of records to be skipped from each source file. See [Selecting Input Records](p. 294).</td>
<td>Same as in Metadata (default)</td>
</tr>
</tbody>
</table>
### 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)**

---

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. 378) for more information.

See also [Defining Transformations](#) (p. 268) for detailed information about transformations.
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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

ParallelReader reads data from flat files using multiple threads.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParallelReader</td>
<td>flat file</td>
<td>0</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. 272) 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. 295) 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
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>✔️</td>
<td>for correct data records</td>
<td>any 1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>for incorrect data records</td>
<td>specific structure, see table below</td>
</tr>
</tbody>
</table>

1) Metadata on output port can use Autofilling Functions (p. 128)

#### Table 51.2. Error Metadata for Parallel Reader

<table>
<thead>
<tr>
<th>Field Number</th>
<th>Field Content</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>record number</td>
<td>integer</td>
<td>position of the erroneous record in the dataset (record numbering starts at 1)</td>
</tr>
<tr>
<td>1</td>
<td>field number</td>
<td>integer</td>
<td>position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)</td>
</tr>
<tr>
<td>2</td>
<td>raw record</td>
<td>string</td>
<td>erroneous record in raw form (including delimiters)</td>
</tr>
<tr>
<td>3</td>
<td>error message</td>
<td>string</td>
<td>error message - detailed information about this error</td>
</tr>
<tr>
<td>4</td>
<td>first record offset</td>
<td>long</td>
<td>indicates the initial file offset of the parsing thread</td>
</tr>
</tbody>
</table>

### ParallelReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>✔️</td>
<td>Attribute specifying what data source(s) will be read. See Supported File URL Formats for Readers (p. 286).</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read in.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See Data Policy (p. 295) for more information.</td>
<td>False (default)</td>
</tr>
<tr>
<td>Trim strings</td>
<td></td>
<td>specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 399). If true, the use of the robust parser is forced.</td>
<td>False (default)</td>
</tr>
<tr>
<td>Quoted strings</td>
<td></td>
<td>Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If true, special characters are removed when read by the component (they are not treated as delimiters).</td>
<td>False</td>
</tr>
<tr>
<td>Quote character</td>
<td></td>
<td>Specifies which kind of quotes will be permitted in Quoted strings. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 157).</td>
<td>Both</td>
</tr>
</tbody>
</table>
Chapter 51. Readers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip leading blanks</td>
<td></td>
<td>specifies whether to skip leading whitespace (blanks e.g.) before setting</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input strings to data fields. If not explicitly set (i.e., having the default</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value), the value of <strong>Trim strings</strong> attribute is used. See <strong>Trimming Data</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p. 399) If true, the use of the robust parser is enforced.</td>
<td></td>
</tr>
<tr>
<td>Skip trailing blanks</td>
<td></td>
<td>specifies whether to skip trailing whitespace (blanks e.g.) before setting</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input strings to data fields. If not explicitly set (i.e., having the default</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value), the value of <strong>Trim strings</strong> attribute is used. See <strong>Trimming Data</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p. 399) If true, the use of the robust parser is enforced.</td>
<td></td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>maximum number of tolerated error records in input file(s); applicable only</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if <strong>Controlled Data Policy</strong> is set</td>
<td></td>
</tr>
<tr>
<td>Treat multiple delimiters as</td>
<td></td>
<td>If a field is delimited by a multiplied delimiter char, it will be</td>
<td>false (default)</td>
</tr>
<tr>
<td>one</td>
<td></td>
<td>interpreted as a single delimiter when setting to true.</td>
<td></td>
</tr>
<tr>
<td>Verbose</td>
<td></td>
<td>By default, less comprehensive error notification is provided and the</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>performance is slightly higher. However, if switched to true, more detailed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>information with less performance is provided.</td>
<td></td>
</tr>
<tr>
<td>Level of parallelism</td>
<td></td>
<td>Number of threads used to read input data files. The order of records is</td>
<td>2 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not preserved if it is 2 or higher. If the file is too small, this value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>will be switched to 1 automatically.</td>
<td></td>
</tr>
<tr>
<td>Distributed file segment</td>
<td></td>
<td>In case a graph is running in a <strong>CloverETL Server</strong> environment, component</td>
<td>false (default)</td>
</tr>
<tr>
<td>reading</td>
<td></td>
<td>can only process the appropriate part of the file. The whole file is</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>divided into segments by <strong>CloverETL Server</strong> and each cluster worker</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>processes only one proper part of file. By default, this option is turned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>off.</td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td></td>
<td>By default, the most appropriate parser is applied. Besides, the parser</td>
<td>auto (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for processing data may be set explicitly. If an improper one is set, an</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>exception is thrown and the graph fails. See <strong>Data Parsers</strong> (p. 400)</td>
<td></td>
</tr>
</tbody>
</table>

**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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

**Short Summary**

QuickBaseRecordReader reads data from QuickBase online database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBaseRecordReader</td>
<td>QuickBase</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

1) Sending each data record to every connected output port
2) Sending data records to output ports according to Return Values of Transformations (p. 272)

**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 be sent out through the optional second port if it is connected.

This component wraps the API_GetRecordInfo HTTP interaction (http://www.quickbase.com/api-guide/getrecordinfo.html).

**Icon**
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>for getting application table record IDs to be read</td>
<td>first field: integer</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✔️</td>
<td>for correct data records</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>information about rejected records</td>
<td>Error Metadata for QuickBaseRecordReader (p. 384)</td>
</tr>
</tbody>
</table>

1 Only source_row_count autofilling function returning the record ID can be used.
2 Error metadata cannot use Autofilling Functions (p. 128).

### Table 51.3. Error Metadata for QuickBaseRecordReader

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;any_name1&gt;</td>
<td>integer</td>
<td>long</td>
</tr>
<tr>
<td>1</td>
<td>&lt;any_name2&gt;</td>
<td>integer</td>
<td>long</td>
</tr>
<tr>
<td>2</td>
<td>&lt;any_name3&gt;</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

### QuickBaseRecordReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBase connection</td>
<td>✔️</td>
<td>ID of the connection to the QuickBase online database, see Chapter 23, QuickBase Connections (p. 184)</td>
<td></td>
</tr>
<tr>
<td>Table ID</td>
<td>✔️</td>
<td>ID of the table in the QuickBase application data records are to be read from (see the application_stats for getting the table ID)</td>
<td></td>
</tr>
<tr>
<td>Records list</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 51. Readers

QuickBaseQueryReader

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

QuickBaseQueryReader gets records fulfilling given conditions from the QuickBase online database table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBaseQueryReader</td>
<td>QuickBase</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

1) Sending each data record to every connected output port
2) Sending data records to output ports according to Return Values of Transformations (p. 272)

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 wraps the API_DoQuery HTTP interaction (http://www.quickbase.com/api-guide/do_query.html).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for correct data records</td>
<td>any 1</td>
</tr>
</tbody>
</table>

1 Metadata cannot use Autofilling Functions (p. 128)

QuickBaseQueryReader Attributes
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickBase connection</td>
<td>✔</td>
<td>ID of the connection to the QuickBase online database, see Chapter 23, QuickBase Connections (p. 184)</td>
<td></td>
</tr>
<tr>
<td>Table ID</td>
<td>✔</td>
<td>ID of the table in the QuickBase application data records are to be get from (see the application_stats for getting the table ID)</td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td></td>
<td>Determines which records are returned (all, by default) using the form {&lt;field_id&gt;.&lt;operator&gt;.&lt;matching_value&gt;}</td>
<td></td>
</tr>
<tr>
<td>CList</td>
<td></td>
<td>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_ids separated by a period.</td>
<td></td>
</tr>
<tr>
<td>SList</td>
<td></td>
<td>The sort list determines the order in which the returned records are displayed. Use field_id separated by a period.</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td>Options used for data records that are read. See Options (p. 386) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced Description

**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-\(D\)**
  
  Specifies 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.
SpreadsheetDataReader

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

SpreadsheetDataReader reads data from Excel spreadsheets (XLS or XLSX files). SpreadsheetDataReader supersedes the original XLSDataReader with a lot more new features, read modes and improved performance. (XLSDataReader is still available for backwards compatibility and in the Community edition)

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpreadsheetDataReader</td>
<td>XLS(X) file</td>
<td>0–1</td>
<td>1–2</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

SpreadsheetDataReader reads data from a specified sheet(s) of XLS or XLSX files. Complex data mapping is possible (forms, tables, multirow records, etc.). All standard input options are available as in other readers: local and remote files, zip archives, an input port or a dictionary.

Supported file formats:

- XLS: only Excel 97/2003 XLS files are supported (BIFF8)
- XLSX: Open Document Format, Microsoft Excel 2007 and newer

In XLSX, even files with more than 1,048,576 rows can be read although the XLSX format does not officially support it (Excel will show no more than $2^{20}$ rows).

Icon

Ports
Port type | Number | Required | Description | Metadata
---|---|---|---|---
Input | 0 | no | For optional port reading. See Reading from Input Port (p. 289). | One field (byte, cbyte, string).
Output | 0 | yes | Successfully read records | Any
1 | no | Error records | Fixed default fields + optional fields from port 0

This component has Metadata Templates (p. 264) available.

**Legend**

1) Metadata can use Autofilling Functions (p. 128). Note: source_timestamp and source_size functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

2) Records which could not be read correctly are sent to output port 1 if Data Policy (p. 295) is set to Controlled and the port has an edge connected (without the edge, messages are logged to the console). There is a fixed set of fields describing the reason and position of the error which caused the record to fail. Additionally, you can map any field from port 0 as well. Please note: for each error in the input there is one error record generated. That is for multiple errors in one record you get multiple error records – you can group them e.g. by the very first integer field.

**Table 51.4. Error Port Metadata - first ten fields have mandatory types, names can be arbitrary**

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>recordNo</td>
<td>integer</td>
<td>index of the incorrectly read record (record numbering starts at 1)</td>
</tr>
<tr>
<td>1</td>
<td>fileName</td>
<td>string</td>
<td>name of the file (if available) the error occurred in</td>
</tr>
<tr>
<td>2</td>
<td>sheetName</td>
<td>string</td>
<td>name of the sheet the error occurred in</td>
</tr>
<tr>
<td>3</td>
<td>fieldNo</td>
<td>integer</td>
<td>index (zero-based) of the field data could not be read into</td>
</tr>
<tr>
<td>4</td>
<td>fieldName</td>
<td>string</td>
<td>name of the field data could not be read into; example: &quot;CustomerID&quot;</td>
</tr>
<tr>
<td>5</td>
<td>cellCoords</td>
<td>string</td>
<td>coordinates of the cell in the source spreadsheet which caused errors on reading; example: &quot;D7&quot;</td>
</tr>
<tr>
<td>6</td>
<td>cellValue</td>
<td>string</td>
<td>value of the cell which caused errors on reading, example: &quot;.5,12&quot;</td>
</tr>
<tr>
<td>7</td>
<td>cellType</td>
<td>string</td>
<td>Excel type of the cell which caused reading errors, example: &quot;String&quot;</td>
</tr>
<tr>
<td>8</td>
<td>cellFormat</td>
<td>string</td>
<td>Excel format string of the cell which caused reading errors, example: &quot;:#,##0&quot;</td>
</tr>
<tr>
<td>9</td>
<td>errMsg</td>
<td>string</td>
<td>error message in a human readable format, example: &quot;Cannot get Date value from cell of type String in C1&quot;</td>
</tr>
</tbody>
</table>

**SpreadsheetDataReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Specifies the data source(s) that will be read</td>
<td>See Supported File URL Formats for Readers (p. 286).</td>
</tr>
</tbody>
</table>
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sheet</td>
<td></td>
<td>Name or number (zero-based) of the sheet to be read. You can specify multiple sheets separated by a semicolon &quot;;&quot;. You can also use the ? and * wildcards to specify multiple sheets. Sheets are then read sequentially one after another using the same mapping.</td>
<td>0 (read the first sheet)</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Maps spreadsheet cells to Clover fields in a visual mapping editor. See the section called “Advanced Description” (p. 390).</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>Path to an XML file containing your Mapping definition. Put your mapping to an external file if you want to share a single mapping among multiple graphs.</td>
<td></td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what is done when an error occurs. See Data Policy (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
</tbody>
</table>

#### Advanced

| Read mode           |     | Determines how data is read from the input file. In-memory mode stores the whole input file in memory allowing for faster reading. Suitable for smaller files. In "streaming" mode the file is being read directly without storing in memory. Streaming should thus allow you to read bigger files without running out of memory. Streaming supports both XLS and XLSX. | In memory (default) | Stream           |
|---------------------|-----|-----------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| Number of skipped records |     | Total number of records throughout all source files that will be skipped. See Selecting Input Records (p. 294). | 0–N                      |
| Max number of records |     | Total number of records throughout all source files that will be read. See Selecting Input Records (p. 294). | 0–N                      |
| Number of skipped records per source |     | Number of records to be skipped in each source file. See Selecting Input Records (p. 294). | 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. 294). | 0–N                      |
| Number of skipped records per spreadsheet |     | Number of records to be skipped in each Excel sheet.                                                                                           |                          |
| Max number of records per spreadsheet |     | Maximum number of records to be read from each Excel sheet.                                                                                     |                          |
| Max error count      |     | Maximum number of allowed errors before the graph fails. Applies for the Controlled value of Data Policy.                                      | 0 (default) | 1–N              |
| Incremental file     | 2)  | Name of a file storing the incremental key (including path). See Incremental Reading (p. 293).                                                  |                          |
| Incremental key      | 2)  | Stores the position of the last record read. See Incremental Reading (p. 293).                                                                 |                          |
| Encryption password  |     | If data are encrypted in the source spreadsheet, type password in here. Mind typing all characters precisely, including the letter case, special characters, accented letters etc. |                          |

**Legend:**

1) One of these two has to be specified to define the mapping.
2) Either both or none of these attributes has to be specified.

**Advanced Description**

**Introduction to Spreadsheet Mapping**

A mapping is a universal pattern guiding the component how to read an Excel spreadsheet. The mapping editor always previews spreadsheets of one file but the mapping can be applied to a whole group of similar files.

Each cell can be mapped to a Clover field in one of the following modes:

- **Map by order** is the simplest approach – spreadsheet cells are mapped one by one to output fields in the same order as on the input. If you select another metadata, the cells will be remapped automatically to the new fields.

- **Map by name** – for each mapped leading cell, the component reads its contents (string) and tries to find a matching field with the same name or label (see [Field Name vs. Label vs. Description](p. 156)). Fields that could not be mapped to the current file are marked as unresolved. You can either map these explicitly, unmap them or modify output metadata. Note that unresolved cells are not a bad thing – you might read say a group of similar input files, each containing just a subset of possible columns. Mappings with unresolved cells do not result in your graph failing on execution.

**Note**

Both **Map by order** and **Map by name** modes try to automatically map the contents of the input file to the output metadata. Thus these modes are useful in cases when you read multiple source files and you want to design a single “one-fits-all” generic mapping.

- **Explicit** – at any time, you may decide to map a cell to a field of your preference. This way, you can have e.g. a whole sheet mapped by order with only one cell, which does not fit the mapping, mapped explicitly to a correct field. Simply go to **Selected cells** and fill in **Field name or index** with the target field. If a cell is not mapped yet, you might need to switch **Mapping mode** to **Explicit**, first. You can also explicitly map a cell to a field by dragging the field from metadata viewer onto the cell. Opposite direction also works (dragging a cell to a field), but you have to first click the cell to select it, because only selected cell can be dragged. Note that you can drag-and-drop more fields/cells at once.

- **Implicit** – special case if you leave all the **Mapping** component property completely blank. The component will not fail but instead the first spreadsheet row will be whole mapped by name with data offset equal to 1. Another type of implicit mapping is created when you map no cell in the mapping editor, but confirm the mapping by clicking OK button. Then, only basic mapping properties will be stored in the **Mapping** attribute. This way you can change default **Rows per record** or **Data offset** used by the basic implicit mapping mentioned above (if default offset is set to 0, mapping by name is used instead of mapping by name). Alternatively, by switching the reading **Orientation** property, the first column gets implicitly mapped instead of the first row.

**Colours in spreadsheet mapping editor**

- Orange cells are called **leading cells** and they form the header. They are a place where a number of mapping settings can be made, see [Advanced mapping options](p. 392).

- Yellow cells indicate the beginning of the first record.

- Cells in dashed border, which appear after a leading cell is selected, indicate the area data is taken from.

**Mapping Editor**

To start mapping, fill in the **File URL** and **Sheet** attributes with the file and sheet name containing data, respectively. After that, edit **Mapping** to open a visual mapping editor. It will preview the sheet you have selected like this:
Figure 51.10. SpreadsheetDataReader Mapping Editor

As you can see, the editor consists of these elements:

- Toolbar – buttons controlling how you Map your Excel data (either by order, or by name) and global data offset control (see Advanced mapping options (p. 392) for an explanation of data offsets).
- Sheet preview area – this is where you will do and see all the mapping of the source file.
- Output metadata – Clover fields you will map Excel cells to.
- Properties – either for the whole source file (Global) or just the ones concerning Selected cells
- Summary tab – a place where you can neatly review the whole spreadsheet-to-clover mapping you have made.

**Metadata**

Before you start reading a spreadsheet, you might need to extract its metadata as Clover fields (see Extracting Metadata from an XLS(X) File (p. 140)). Note that the extracting wizard resembles the spreadsheet Mapping editor introduced here and it uses the same principle.

**Note**

You can use the mapping editor to extract metadata right in place without needing to jump to the metadata extract wizard (which is suitable if you need to get just the spreadsheet metadata).

Metadata assigned to the outgoing edge can be edited in the Output metadata area. You can create and manipulate metadata right from the mapping editor, even if you have not connected an output edge (it is created automatically once you create some fields). Available operations include:

- Select existing metadata in the graph using the Output metadata combo.
- Create new metadata using the <new metadata> option in the Output metadata combo.
- Double click a Field to rename it.
• Change data Type via combo-boxes.

• For more operations on the output metadata use the Edit button.

• To create metadata, drag cells from the spreadsheet preview area and drop them between output metadata fields.

**Basic Mapping Example**

Typically, your Excel data contains headers in the first row and, thus, can be easily mapped. This section describes how to do that.

• First, make sure you have set **Vertical** mode in Properties → Global → Orientation. This makes SpreadsheetDataReader process the input by rows (opposite to **Horizontal** orientation, where reading advances by columns).

• Optional (in case you have not extracted metadata as in Extracting Metadata from an XLS(X) File (p. 140)): select the first row and drag its fields to the **Output metadata** pane. This will create fields for all cells in the selection. Types will be guessed automatically, but it is worth checking them yourself afterwards.

• Select the whole first row (by clicking the "1" row header) and click either Map by order or Map by name (for explanation, see Introduction to Spreadsheet Mapping (p. 390)).

![Spreadsheet Mapping Editor](image)

*Figure 51.11. Basic Mapping – notice leading cells and dashed borders marking the area data will be taken from*

**Advanced mapping options**

This section provides an explanation of some more concepts extending the Basic Mapping Example (p. 392)

• **Data offsets (global)** – determines where data is taken from. Basically, its value represents 'a number of rows (in vertical mode) or columns (in horizontal mode) to be omitted - relative to the leading cell (the orange one)'.
Click the arrow buttons in the top right corner to adjust data offsets for the whole spreadsheet. Additionally, you can click the spinner in the Selected cells area of each leading cell (the orange one) to adjust data offset locally, i.e. for a particular column only. Notice how modifying data offset is visualised in the sheet preview – the ‘omitted’ rows change colour. By following dashed cells, which appear when you click a leading cell, you can quickly state where your record will start at.

**Tip**

The arrow buttons in Data offsets (global) only shift the data offset property of each cell either up or down. So mixed offsets are retained, just shifted as desired. To set all data offsets to a single value, enter the value into the number field of Data offsets (global). Note that if there are some mixed offsets, the value is displayed in gray.

---

**Figure 51.12.** The difference between global data offsets set to 1 (default) and 3. In the right hand figure, reading would start at row 4 (ignoring data in rows 2 and 3).

**Figure 51.13.** Global data offset is set to 1 to all columns. In the third column, it is locally changed to 3.

- **Rows per record** – a Global property specifying how many rows form one record. Best imagined if you look at the figure below:

**Figure 51.14.** Rows per record is set to 4. This makes SpreadsheetsDataReader take 4 Excel rows and create one record out of their cells. Cells actually becoming fields of a record are marked by a dashed border, therefore the record is not populated by all data. Which cells populate a record is also determined by the data offsets setting, see the following bullet point.

- Combination of Data offsets (global and local) and **Rows per record** – you can put the settings described in preceding bullet points together. See example:
Figure 51.15. Rows per record is set to 3. The first and third columns ‘contribute’ to the record by their first row (because of the global data offset being 1). The second and fourth columns have (local) data offsets 2 and 4, respectively. The first record will, thus, be formed by ‘zig-zagged’ cells (the yellow ones – follow them to make sure you understand this concept clearly).

- **Max number of records** – a Global property which you can specify via component attributes, too (see SpreadsheetDataReader Attributes (p. 388)). If you reduce it, you will notice the number of dashed cells in the spreadsheet preview reduces as well (highlighting only the cells which will be mapped to records in fact).

- **Format Field** – Excel format (as in Excel’s right-click menu – Format Cells) can be retrieved from read cells. Select a leading cell and specify the Format Field property (in Selected cells) as a target field to which the format patterns will be read. Keep in mind the target field has to be string. You can use this approach even if read data cells have various formats (e.g. various currencies).

  **Note**

  If an Excel cell has the General format, which is a common case, the format cannot be transferred to Clover due to an internal Excel formatting. Instead, the target field will bear a string “General”.

Figure 51.16. Retrieving format from a date field. Format Field was set to the “Special” field as target.

Formats can also be extracted during the one-time metadata extraction process. In metadata, format is taken from a single cell which you supply as a sample value to the metadata extraction wizard. See Extracting Metadata from an XLS(X) File (p. 140).

If a cell has its format specified by the Excel format string (excel:), SpreadsheetDataReader can read it back. Other readers would ignore it. For further reading on format strings, see Formatting cells (Format Field) (p. 514).

- **Multiple leading cells per column** – in some spreadsheets, data in one column gets mixed, but you still need to process it all into one record. For example, imagine a column containing first names in odd rows and surnames in even rows one after another. In that case, you will create two leading cells above each other to be able to read both the first names and surnames. Remember to set Rows per record to an appropriate value (2 in this example) not to read same data in all leading cells. Also, mind raising Data offset in the upper leading cell to start reading data where it truly begins. Look at the figure below:
Chapter 51. Readers

Figure 51.17. Reading mixed data using two leading cells per column. Rows per record is 2, Data offset needed to be raised to 2 – looking at the first leading cell which has to start reading on the third row.

Notes and Limitations

- **Invalid mapping** - It is possible to create invalid mapping using the mapping editor. Invalid mapping causes SpreadsheetDataReader to fail. Such a mapping arises when, for example, one metadata field is mapped to more than one cell, or an autofilled field is mapped (see Autofilling Functions (p. 128)). Another invalid mapping would be caused by an attempt to read a cell (at least one) into more than one metadata field.

When you change mapping in any way, the validation process is automatically run and you will see the warning icon with cell(s) and/or metadata field(s) which cause the mapping to be invalid. When you mouse over such a cell or field, a tooltip with information about the validation problem will be displayed. Also, one of the warning validation messages is displayed at the top of the editor (the white header area).

Note that warnings caused by cells mapped by name/order will not necessarily lead to the copomonet's failure (as mentioned earlier).

- **Reading date as string** - SpreadsheetDataReader cannot guarantee that dates read into string fields will be displayed identically to how they appear in MS Excel. The reason is Clover interprets the format string stored in a cell otherwise than Excel - it depends on your locale.

**Important**

It is recommend you read dates into date fields and convert them to string using a CTL (p. 849) transformation.

**Built-in** Excel formats are interpreted according to the following table:

*Table 51.5. Format strings*

<table>
<thead>
<tr>
<th>Format index stored in Excel cell</th>
<th>Format string</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;General&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;0.00&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;:.##0&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;:.##0.00&quot;</td>
</tr>
<tr>
<td>5</td>
<td>&quot;$#,.##0_);($#,##0)&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;$#,.##0_);<a href="$#,##0">Red</a>&quot;</td>
</tr>
<tr>
<td>7</td>
<td>&quot;$#,.##0.00);($#,##0.00)&quot;</td>
</tr>
<tr>
<td>Format index stored in Excel cell</td>
<td>Format string</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>8</td>
<td>&quot;$#,##0.00_;[Red]$(#,##0.00)&quot;</td>
</tr>
<tr>
<td>9</td>
<td>&quot;0%&quot;</td>
</tr>
<tr>
<td>0xa</td>
<td>&quot;0.00%&quot;</td>
</tr>
<tr>
<td>0xb</td>
<td>&quot;0.00E+00&quot;</td>
</tr>
<tr>
<td>0xc</td>
<td>&quot;# ??&quot;</td>
</tr>
<tr>
<td>0xd</td>
<td>&quot;# ?????&quot;</td>
</tr>
<tr>
<td>0xe</td>
<td>&quot;m/d/yy&quot;</td>
</tr>
<tr>
<td>0xf</td>
<td>&quot;d-mmm-yy&quot;</td>
</tr>
<tr>
<td>0x10</td>
<td>&quot;d-mmm&quot;</td>
</tr>
<tr>
<td>0x11</td>
<td>&quot;mmm-yy&quot;</td>
</tr>
<tr>
<td>0x12</td>
<td>&quot;h:mm AM/PM&quot;</td>
</tr>
<tr>
<td>0x13</td>
<td>&quot;h:mm:ss AM/PM&quot;</td>
</tr>
<tr>
<td>0x14</td>
<td>&quot;h:mm&quot;</td>
</tr>
<tr>
<td>0x15</td>
<td>&quot;h:mm:ss&quot;</td>
</tr>
<tr>
<td>0x16</td>
<td>&quot;m/d/yy h:mm&quot;</td>
</tr>
<tr>
<td>0x25</td>
<td>&quot;#,##0_:;(#,##0)&quot;</td>
</tr>
<tr>
<td>0x26</td>
<td>&quot;#,##0_:;[Red]$(#,##0)&quot;</td>
</tr>
<tr>
<td>0x27</td>
<td>&quot;#,##0.00_:;(#,##0.00)&quot;</td>
</tr>
<tr>
<td>0x28</td>
<td>&quot;#,##0.00_:;[Red]$(#,##0.00)&quot;</td>
</tr>
</tbody>
</table>
| 0x29                              | "_(*#,##0_):_(*$(#,##0)_;_(*"\"-\"_;_;(@ _)"
| 0x2a                              | "_$(#,##0_):_$(#,##0);_$("\"-\"_;_;(@ _"
| 0x2b                              | "_(*#,##0.00_):_(*$(#,##0.00);_(*"\"-\"??_;_;(@ _"
| 0x2c                              | "_$(#,##0.00_):_$(#,##0.00);_$("\"-\"??_;_;(@ _"
| 0x2d                              | "mm:ss"       |
| 0x2e                              | "[h]:mm:ss"   |
| 0x2f                              | "mm:ss.0"     |
| 0x30                              | "##0.0E+0"    |
| 0x31                              | "," (this is text format) |

**Custom** format strings are read as they are defined in Excel. Decimal point is modified according to your locale. Special characters such as double quotes are not interpreted at all.

In both cases (built-in and custom formats), the result may vary from how Excel displays it.
Chapter 51. Readers

UniversalDataReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

UniversalDataReader reads data from flat files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniversalDataReader</td>
<td>flat file</td>
<td>0-1</td>
<td>1-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

1) Sending each data record to every connected output port
2) Sending data records to output ports according to Return Values of Transformations (p. 272)

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. 295) 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>for Input Port Reading (p. 292)</td>
<td>include specific byte/ cbyte/ string field</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✔️</td>
<td>for correct data records</td>
<td>any 1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>for incorrect data records</td>
<td>specific structure, see table below</td>
</tr>
</tbody>
</table>

1) Metadata on output port 0 can use Autofilling Functions (p. 128) Note: source_timestamp and source_size functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).
This component has Metadata Templates (p. 264) available.

The optional logging port for incorrect records has to define the following metadata structure - the record contains exactly five fields (named arbitrarily) of given types in the following order:

Table 51.6. Error Metadata for UniversalDataReader

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>recordNo</td>
<td>integer</td>
<td>position of the erroneous record in the dataset (record numbering starts at 1)</td>
</tr>
<tr>
<td>1</td>
<td>fieldNo</td>
<td>integer</td>
<td>position of the erroneous field in the record (1 stands for the first field, i.e., that of index 0)</td>
</tr>
<tr>
<td>2</td>
<td>data</td>
<td>string</td>
<td>byte</td>
</tr>
<tr>
<td>3</td>
<td>errorMsg</td>
<td>string</td>
<td>byte</td>
</tr>
<tr>
<td>4</td>
<td>srcFile</td>
<td>string</td>
<td>source file in which the error occurred</td>
</tr>
</tbody>
</table>

UniversalDataReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>✓</td>
<td>path to data source (flat file, console, input port, dictionary) to be read specified, see Supported File URL Formats for Readers (p. 286).</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>character encoding of input records (character encoding does not apply on byte fields if the record type is fixed)</td>
<td>strict (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>specifies how to handle misformatted or incorrect data, see Data Policy (p. 295)</td>
<td>strict</td>
</tr>
<tr>
<td>Trim strings</td>
<td></td>
<td>specifies whether leading and trailing whitespace should be removed from strings before setting them to data fields, see Trimming Data (p. 399) below</td>
<td>default</td>
</tr>
<tr>
<td>Quoted strings</td>
<td></td>
<td>Fields containing a special character (comma, newline, or double quote) have to be enclosed in quotes. Only single/double quote is accepted as the quote character. If true, special characters are removed when read by the component (they are not treated as delimiters). Example: To read input data &quot;25&quot;</td>
<td>&quot;John&quot;, switch Quoted strings to true and set Quote character to &quot;. This will produce two fields: 25</td>
</tr>
<tr>
<td>Quote character</td>
<td></td>
<td>Specifies which kind of quotes will be permitted in Quoted strings. By default, the value of this attribute is inherited from metadata on output port 0. See also Record Details (p. 157).</td>
<td>both</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Skip leading blanks</td>
<td></td>
<td>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 <strong>Trim strings</strong> attribute is used. See <strong>Trimming Data</strong> (p. 399).</td>
<td>default</td>
</tr>
<tr>
<td>Skip trailing blanks</td>
<td></td>
<td>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 <strong>Trim strings</strong> attribute is used. See <strong>Trimming Data</strong> (p. 399).</td>
<td>default</td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>how many records/rows to be skipped from the source file(s); see <strong>Selecting Input Records</strong> (p. 294).</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>how many records to be read from the source file(s) in turn; all records are read by default; See <strong>Selecting Input Records</strong> (p. 294).</td>
<td>1 - N</td>
</tr>
<tr>
<td>Number of skipped records per source</td>
<td></td>
<td>how many records/rows to be skipped from each source file. By default, the value of <strong>Skip source rows</strong> record property in output port 0 metadata is used. In case the value in metadata differs from the value of this attribute, the <strong>Number of skipped records per source</strong> value is applied, having a higher priority. See <strong>Selecting Input Records</strong> (p. 294).</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Max number of records per source</td>
<td></td>
<td>how many records/rows to be read from each source file; all records from each file are read by default; See <strong>Selecting Input Records</strong> (p. 294).</td>
<td>1 - N</td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>maximum number of tolerated error records in input file(s); applicable only if Controlled <strong>Data Policy</strong> is set</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Treat multiple delimiters as one</td>
<td></td>
<td>If a field is delimited by a multiplied delimiter char, it will be interpreted as a single delimiter when setting to <strong>true</strong>.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Incremental file</td>
<td>1)</td>
<td>Name of the file storing the incremental key, including path. See <strong>Incremental Reading</strong> (p. 293).</td>
<td></td>
</tr>
<tr>
<td>Incremental key</td>
<td>1)</td>
<td>Variable storing the position of the last read record. See <strong>Incremental Reading</strong> (p. 293).</td>
<td></td>
</tr>
<tr>
<td>Verbose</td>
<td></td>
<td>By default, less comprehensive error notification is provided and the performance is slightly higher. However, if switched to <strong>true</strong>, more detailed information with less performance is provided.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Parser</td>
<td></td>
<td>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 <strong>Data Parsers</strong> (p. 400)</td>
<td>auto (default)</td>
</tr>
</tbody>
</table>

**Deprecated**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip first line</td>
<td></td>
<td>By default, the first line is not skipped, if switched to <strong>true</strong> (if it contains a header), the first line is skipped.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

**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:
Chapter 51. Readers

- 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 trailing 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. 156))
- 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. 120).

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. 85) 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.
We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

**Short Summary**

**XLSDataReader** reads data from XLS or XLSX files.

---

### Important

Since Clover 3.3.0, there are new powerful components available for spreadsheet reading/writing - SpreadsheetDataReader (p. 387) and SpreadsheetDataWriter (p. 506) The preceding XLS components (XLSDataReader (p. 402) and XLSDataWriter (p. 529) have remained compatible, though.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLSDataReader</td>
<td>XLS(X) file</td>
<td>0-1</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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. 272) 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.
Port type | Number | Required | Description | Metadata
---|---|---|---|---
Input | 0 | no | For port reading. See Reading from Input Port (p. 289). | One field (byte, cbyte, string).
Output | 0 | yes | For correct data records | Any
| 1-n | no | For correct data records | Output 0

Legend:
1) Metadata can use Autofilling Functions (p. 128). Note: source_timestamp and source_size functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

**XLSDataReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of parser</td>
<td></td>
<td>Specifies the parser to be used. By default, component guesses according the extension (XLS or XLSX).</td>
<td>Auto (default)</td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying what data source(s) will be read (input file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 286).</td>
<td></td>
</tr>
<tr>
<td>Sheet name</td>
<td>1)</td>
<td>Name of the sheet to be read. Wild cards ? and * can be used in the name.</td>
<td></td>
</tr>
<tr>
<td>Sheet number</td>
<td>1)</td>
<td>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, <em>-maxNumber, minNumber</em>- Example: <em>-5, 9-11, 17-</em>.</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See Data Policy (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Metadata row</td>
<td></td>
<td>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. 404) for more information.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Field mapping</td>
<td></td>
<td>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. 404) for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped continuously throughout all source files. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
</tbody>
</table>
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be read continuously throughout all source files. See [Selecting Input Records](p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Number of skipped records per source</td>
<td></td>
<td>Number of records to be skipped from each source file. See [Selecting Input Records](p. 294).</td>
<td>Same as in Metadata (default)</td>
</tr>
<tr>
<td>Max number of records per source</td>
<td></td>
<td>Maximum number of records to be read from each source file. See [Selecting Input Records](p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>Maximum number of allowed errors for the Controlled value of Data Policy before the graph fails.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Incremental file</td>
<td>2)</td>
<td>Name of the file storing the incremental key, including path. See [Incremental Reading](p. 293).</td>
<td></td>
</tr>
<tr>
<td>Incremental key</td>
<td>2)</td>
<td>Variable storing the position of the last read record. See [Incremental Reading](p. 293).</td>
<td></td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start row</td>
<td></td>
<td>Has inclusive meaning: First row that is read. Has lower priority than Number of skipped records.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Final row</td>
<td></td>
<td>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.</td>
<td>all (default)</td>
</tr>
</tbody>
</table>

**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 51.18. XLS Mapping Dialog](Image)

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 51.19. 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 51.5. Field Mapping in XLSData Reader

• 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
Chapter 51. Readers

XMLExtract

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 41, Common Properties of Readers (p. 285)

If you want to find the right Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

XMLExtract reads data from XML files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLExtract</td>
<td>XML file</td>
<td>0-1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See Reading from Input Port (p. 289).</td>
<td>One field (byte, cbyte, string) for specify input of component. Input fields can be mapped to output. See XMLExtract Mapping Definition (p. 409) for more information.</td>
</tr>
</tbody>
</table>
### Chapter 51. Readers

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any(^1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td></td>
<td>For correct data records</td>
<td>Any(^1) (each port can have different metadata)</td>
</tr>
</tbody>
</table>

**Legend:**

1): Metadata on each output port does not need to be the same. Each metadata can use [Autofilling Functions](p. 128).

2): Other output ports are required if mapping requires that.

## XMLExtract Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying what data source(s) will be read (XML file, console, input port, dictionary). See [Supported File URL Formats for Readers](p. 286).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records which are read.</td>
<td>any encoding, default system one by default</td>
</tr>
<tr>
<td>Mapping</td>
<td>1(^1)</td>
<td>Mapping of the input XML structure to output ports. See [XMLExtract Mapping Definition](p. 409) for more information.</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1(^1)</td>
<td>Name of an external file, including its path which defines mapping of the input XML structure to output ports. See [XMLExtract Mapping Definition](p. 409) for more information.</td>
<td></td>
</tr>
<tr>
<td>Namespace Bindings</td>
<td></td>
<td>Allows using arbitrary namespace prefixes in Mapping. See [Namespaces](p. 420).</td>
<td></td>
</tr>
<tr>
<td>XML Schema</td>
<td></td>
<td>URL of the file that should be used for creating the Mapping definition. See [XMLExtract Mapping Editor and XSD Schema](p. 414) for more information.</td>
<td></td>
</tr>
<tr>
<td>Use nested nodes</td>
<td></td>
<td>By default, nested elements are also mapped to output ports automatically. If set to false, an explicit &lt;Mapping&gt; tag must be created for each such nested element.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Trim strings</td>
<td></td>
<td>By default, white spaces from the beginning and the end of the elements values are removed. If set to false, they are not removed.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate</td>
<td></td>
<td>Enables/disables validation of the XML (against a schema)</td>
<td>true</td>
</tr>
<tr>
<td>XML features</td>
<td></td>
<td>Sequence of individual validation 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. 296) for more information.</td>
<td></td>
</tr>
<tr>
<td>Number of skipped mappings</td>
<td></td>
<td>Number of mappings to be skipped continuously throughout all source files. See [Selecting Input Records](p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Max number of mappings</td>
<td></td>
<td>Maximum number of records to be read continuously throughout all source files. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these must be specified. If both are specified, **Mapping URL** has higher priority.
Chapter 51. Readers

Advanced Description

Example 51.6. Mapping in XMLExtract

```xml
<Mappings>
  <TypeOverride elementPath="/employee/child" overridingType="boy" />
  <Mapping element="employee" outPort="0" implicit="false" 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">
      <Mapping element="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey" sequenceField="seqKey" sequenceId="Sequence0"/>
    </Mapping>
    <Mapping element="financial1" 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>
</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> and <TypeOverride> tags. Each of these <Mapping> tags contains some XMLExtract Mapping Tag Attributes (p. 411) See also XMLExtract Type Override Tags (p. 409) or XMLExtract Mapping Tags (p. 409) for more information.

3. XMLExtract Type Override Tags

The Type Override tag can be used to tell the mapping editor, that element on given path should be treated as if it's type was actually the overridingType. This tag has no impact on actual processing of XML file at runtime.

Example:

```xml
<TypeOverride elementPath="/employee/child" overridingType="boy" />  
```

- **elementPath**
  
  Required
  
  Each type override tag must contain one elementPath attribute. The value of this element must be a path from the root of an input XML structure to a node.

  ```xml
  elementPath="/prefix:parent/.../prefix:nodeName"
  ```

- **overridingType**
  
  Required
  
  Each type override tag must contain one overridingType attribute. The value of this element must be a type in the referenced XML schema.

  ```xml
  overridingType="/prefix:typeName"
  ```

4. XMLExtract Mapping Tags

- **Empty Mapping Tag (Without a Child)**
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" Attributes (p. 411) />
```

(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>
```

In addition to nested `<Mapping>` elements, the Mapping can contain `<FieldMapping>` elements to map fields from input record to output record. See XMLExtract Field Mapping Tags (p. 410) for more information.

5. **XMLExtract Field Mapping Tags**

Field Mapping tags allows to map fields from an input record to an output record of parent Mapping element.

Example:

```
<FieldMapping inputField="sessionID" outputField="sessionID" />
```

• **inputField**

  Required

  Specifies a field from an input record, that should be mapped to an output record.

  `inputField="fieldName"`

• **outputField**

  Required

  Specifies a field to which a value from the input field should be stored.

  `outputField="fieldName"`

6. Nested structure of `<Mapping>` tags copies the nested structure of XML elements in input XML files. See example below.
Example 51.7. 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>
  </[[prefixL:]]nameOfElementN>
</[[prefix:]]nameOfElement>
```

• Mapping Can Look Like This:

```
<Mappings>
  <Mapping element="[[prefix:]]nameOfElement" attributes/>
  ...
  <Mapping element="[[prefixK:]]nameOfElementM" attributesKM/>
  <Mapping element="[[prefixL:]]nameOfElementN" attributesLN/>
    ...
    <Mapping element="[[prefixR:]]nameOfElementG" attributesRG/>
  </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.

7. 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).

• **useParentRecord**

  Optional

  If true the mapping will assign mapped values to the record generated by the nearest parent mapping element with outPort specified. Default value of this attribute is false.

  useParentRecord="false|true"

• **implicit**

  Optional

  If false the mapping will not automatically map XML fields to record fields with the same name. Default value of this attribute is true.

  implicit="false|true"

• **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 \texttt{parentKey} attribute.

- \texttt{sequenceField}

Sometimes a pair of \texttt{parentKey} and \texttt{generatedKey} does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has multiple 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 \texttt{parentKey} for the next nested level.

Example: \texttt{sequenceField="sequenceKey"}

- \texttt{sequenceId}

Optional

Sometimes a pair of \texttt{parentKey} and \texttt{generatedKey} does not ensure unique identification of records (the parent-child relation) - this is the case when one parent has multiple 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: \texttt{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 identified using the parent information copied from \texttt{parentKey} to \texttt{generatedKey}. Such information is not sufficient. For this reason, a sequence may be defined to give distinguishing numbers to the multiple child elements.

- \texttt{xmlFields}

If the names of XML nodes or attributes should be changed, it has to be done using a pair of \texttt{xmlFields} and \texttt{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 \texttt{cloverFields} attribute.

Do not forget the values have to correspond to the specified data type.

Example: \texttt{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 \texttt{Source Tab} (p. 416) 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.

Note

If you do not possess a valid XSD schema for your source XML, you can switch to the Mapping tab and click Generate XML Schema which attempts to "guess" the XSD structure from the XML.
Figure 51.20. 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 XML. Every element shows how many occurrences it has in the source file (e.g. [0:n]). In this pane, you need to check the elements that should be mapped to the output ports.

At the top, you specify Output for each selected element by choosing from a drop-down list. Possible values are:

- **Not mapped** - the mapping will not produce a record. By using such mapping elements, you can enforce that any child mapping will be processed only if the parser encounters this element first.

- **Parent record** - the mapping will not produce a record, but it will fill the mapped values to a parent record.

- **portNumber(metadata)** - the mapping will generate a record and write it to a selected output port.

You can then choose from the list of metadata labeled *portNumber(metadata)*, e.g. "3(customer)".

On the right hand side, you can see mapping Input and Output fields. You either map them to each other according to their names (by checking the Map XML by name checkbox) or you map them yourself - explicitly. Please note that in Input - 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 above, the "pref:records" element is selected but we are allowed to leap over its parent element "pref:result" whose field "size" 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.

You can also map the input fields (Input fields section), fields from record produced by parent mapping (Parent fields section) or generate a unique id for record by mapping a sequence from Sequences section to one of the output fields.

**Note**

*sequenceId* and *sequenceField* is set if some sequence is mapped to output metadata field. However it’s possible to set just *sequenceField*. In this case new sequence is created and mapped to the metadata field. The mapping is valid but Mapping Dialog shows warning that metadata field is mapped to non existing sequence.
Once you define all elements, specify output ports, mapping and other properties, you can switch to the Source tab. The mapping code is displayed there. Its structure is the same as described in the preceding sections.

Note

If you do not possess a valid XSD schema for your source XML, you will not be able to map elements visually and you have to do it here in Source.

Note

It's possible to map attribute or element missing at the schema. No validation warning is raised and mapping is visualized at Mapping tab. Italic font is used when displaying mapped elements and attributes missing at the schema.

If you want to map an element to XML fields of its parents, use the ".//" string (like in the file system) before 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. The ".//.//emplID" is a field of "employee" as made available to the currently selected element "customer".

```
<Mapping element="employee">
  <Mapping element="project">
    <Mapping element="customer" outPort="0"
      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 rely on the Use nested nodes property set to true: To reference one parent level using ".//" actually means to reference that ancestor element (over more parents) in the XML which is defined in the direct parent <Mapping> of <Mapping> with the ".//" parent reference.

An example is always a good thing so here it goes. Let us recall the 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" outPort="0"
    xmlFields="name;../empID"
    cloverFields="name;empId"/>
</Mapping>

**Usage of Dot In Mapping**

It is possible to map the value of an element using the `.` dot syntax. The **dot means 'the element itself' (its name).** Every other occurrence of the element's name in mapping (as text, e.g. "customer") represents the element's subelement or attribute. (Note: Available since Clover v. 3.1.0)

The dot can be used in the `xmlFields` attribute just like 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 following chunk of code maps the value of element `customer` on metadata field `customerValue`. Next, `project` (i.e. `customer`'s parent element, that is why `../.`) is mapped on the `projectValue` field.

```
<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:

```
<Mappping ... xmlFields="customer" cloverFields="newFieldName" />
```

Moreover, when you have an XML file containing 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:
Remember the explicit mapping (renaming fields) shown in the examples has a higher priority than the implicit mapping. The implicit mapping can be turned off by setting `implicit` attribute of the corresponding `Mapping` element to `false`.

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:

```xml
<Mapping element="customer" outPort="2"
xmlFields=".;customer"
cloverFields="customerElement;customerAttribute"/>
</Mapping>

<Mapping element="customer" outPort="4" />
// customer's subelement called 'customer' as well
</Mapping>
```

**Usage of useParentRecord attribute**

If you want to map a value from nested element, but you do not want to create a separate record for the parent and nested elements, you may consider using the `useParentRecord` attribute of the `Mapping` element. By setting the attribute to `true`, the values mapped by the `Mapping` element will not be assigned to a new record, but will be set to a parent record. (Note: Available since Clover v.3.3.0-M3)

The following chunk of code maps the value of element `project` on metadata field `projectValue` and value of `customer` element on metadata field `customerValue`. The `customerValue` field is set in the same record as the `projectValue`.

```xml
<Mapping element="project" outPort="0" xmlFields="." cloverFields="projectValue">
  <Mapping element="customer" useParentRecord="true" xmlFields="." cloverFields="customerValue" />
</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:

```xml
<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:

```xml
<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:

```xml
<Mapping element="myElement" templateId="nestedTempl">
   <!-- ... some mapping ... -->
   <Mapping templateRef="nestedTempl" nestedDepth="3"/>
</Mapping> <!-- template declaration ends here -->
```

Note

The following chunk of code:

```xml
<Mapping templateRef="unnestedTempl" nestedDepth="3"/>
```

can be imagined as

```xml
<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.

```xml
<Mapping element="wrap">
   <Mapping element="realElement" templateId="unnestedTempl"/>
</Mapping>
```
In summary, working with `nestedDepth` instead of nested template references always grants transparent results. Its use is recommended.

### Namespaces

If you supply an XML Schema which has a namespace, the namespace is automatically extracted to Namespace Bindings and given a Name. The Name does not have to exactly match the namespace prefix in the input schema, though, as it is only a denotation. You can edit it anytime in the Namespace Bindings attribute as shown below:

![Figure 51.22. Editing Namespace Bindings in XMLExtract](image)

After you open Mapping, namespace prefixes will appear before element and attribute names. If Name was left blank, you would see the namespace URI instead.

**Note**

If your XSD contains two or more namespaces, mapping elements to the output in the visual editor is not supported. You have to switch to the Source tab and handle namespaces yourself. Use the 'Add' button in Namespace Bindings to pre-prepare a namespace. You will then use it in the source code like this:

```
Name = myNs
Value = http://www.w3c.org/foo
```

lets you write

```
myNs:element1
```

instead of

```
{http://www.w3c.org/foo}element1
```

### Selecting subtypes

Sometimes the schema defines an element to be of some generic type, but you know, what the actual specific type of the element will be in the processed XML. If the subtypes of the generic type are also defined in the schema, you may use the Select subtype action. This will open a dialog as shown below. When you choose a subtype, the element in the schema tree will be treated as if it was of the selected type. This way, you will be able to define
the mapping of this element by using Mapping editor. The information will also be stored in the Mapping source - see Type Override Tags (p. 409).

Figure 51.23. Selecting subtype in XMLEXtract

Notes

Consider following XML file

```xml
<customer name="attribute_value">
  <name>element_value</name>
</customer>
```

In this case element `customer` has attribute `name` and child element of same name. If both attribute `name` and element `name` shall be mapped to output metadata, following mapping is incorrect.

```xml
<Mappings>
  <Mapping element="customer" outPort="0"
    xmlFields="{}name"
    cloverFields="field1">
  </Mapping>
</Mappings>
```

Result of this mapping is that both `field1` and `field2` contains value of element `name`. Following mapping shall be used if we need to read value of attribute `name` to some output metadata field.
<Mappings>
    <Mapping element="customer" outPort="0"
        xmlFields="{}name"
        cloverFields="field2">
        <Mapping element="name" useParentRecord="true"
            xmlFields="../{}name"
            cloverFields="field1">
        </Mapping>
    </Mapping>
</Mappings>
XMLReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purposes, see Readers Comparison (p. 286).

**Short Summary**

XMLReader reads data from XML files. It is a powerful new component which supersedes the original XMLXPathReader and XMLExtract.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLReader</td>
<td>XML file</td>
<td>0-1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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. 272) for more information. XMLReader, XMLExtract and XMLXPathReader send data to ports as defined in their Mapping or Mapping URL attribute.

**Abstract**

XMLReader reads data from XML files. 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**

XMLReader

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See Reading from Input Port (p. 289).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
</tbody>
</table>
Port type | Number | Required | Description | Metadata
---|---|---|---|---
Output | 0 ... n-1 | yes | For correct data records. Connect more than one output ports if your mapping requires that. | Any¹)
Output | n | no | Error port | Restricted format²)

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. 128).

2) If you intend to use the last output port for error logging, metadata has to have a fixed format. Field names are arbitrary, field types are these:

- integer - number of the output port where errors occurred
- integer - record number (per source and port)
- integer - field number
- string - field name
- string - value which caused the error
- string - error message
- optional field: string - source name

**XMLReader Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Specifies which data source(s) will be read (XML file, console, input port, dictionary). See Supported File URL Formats for Readers (p. 286).</td>
<td>ISO-8859-1 (default)</td>
</tr>
</tbody>
</table>
| Charset | | Encoding of records that are read. When reading from files, the charset is detected automatically (unless you specify it yourself). **Important**

If you are reading from a port or dictionary, always set **Charset** explicitly (otherwise you will get errors). There is no autodetection as in reading from files. | |
| Data policy | | Determines what should be done when an error occurs. See Data Policy (p. 295) for more information. | Strict (default) | Controlled | Lenient |
| Mapping URL | 1) | External text file containing the mapping definition. See XMLReader Mapping Definition (p. 426) for more information. | |
| Mapping | 1) | Mapping the input XML structure to output ports. See XMLReader Mapping Definition (p. 426) for more information. | |
### Attribute: XML features

Sequence of individual `true/false` expressions related to XML features which should be validated. The expressions are separated from each other by semicolon. See [XML Features](p. 296) for more information.

#### Legend:

1) One of these has to be specified. If both are specified, **Mapping URL** has higher priority.
Advanced Description

Example 51.8. Mapping in XMLReader

```xml
<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">
    <Mapping xpath="sequence" sequenceField="seqKey" sequenceId="Sequence0"/>
  </Context>
  <Context xpath="financial" outPort="3" parentKey="seqKey" generatedKey="seqKey"/>
</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.

**XMLReader 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 **XMLReader Context Tag Attributes** (p. 427) and **XMLReader Mapping Tag Attributes** (p. 428), respectively.

**Important**

By default, mapping definition is **implicit**. Therefore elements (e.g. `salary`) are automatically mapped onto fields of the same name (`salary`) and you do **not** have to write:

```xml
<Mapping xpath="salary" cloverField="salary"/>
```

Thus, use explicit mapping only to populate fields with data from distinct elements.

4. **XMLReader Context Tags and Mapping Tags**

   - **Empty Context Tag (Without a Child)**
     ```xml
     <Context xpath="xpathexpression" **XMLReader Context Tag Attributes** (p. 427) />
     ```

   - **Non-Empty Context Tag (Parent with a Child)**
     ```xml
     <Context xpath="xpathexpression" **XMLReader Context Tag Attributes** (p. 427) >
     ```
(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" XMLReader Mapping Tag Attributes (p. 428) />
  - nodeName is used:
    
    <Mapping nodeName="elementname" XMLReader Mapping Tag Attributes (p. 428) />

5. XMLReader Context Tag and Mapping Tag Attributes

1) XMLReader 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) XMLReader 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.

**Reading Mutlilvalue Fields**

As of Clover 3.3, reading multivalue fields is supported - you can read only lists, however (see Multivalue Fields (p. 163)).

**Note**

Reading maps is handled as reading pure string (for all data types as map's values).
Example 51.9. Reading lists with XMLReader

An example input file containing these elements (just a code snippet):

```xml
...  
<attendees>John</attendees>  
<attendees>Vicky</attendees>  
<attendees>Brian</attendees>  
...  
```

can be read back by the component with this mapping:

```plaintext
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

where `attendanceList` is a field of your metadata. The metadata has to be assigned to the component’s output edge. After you run the graph, the field will get populated by XML data like this (that what you will see in View data):

`[John, Vicky, Brian]`
XMLXPathReader

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 41, Common Properties of Readers (p. 285)

If you want to find the appropriate Reader for your purposes, see Readers Comparison (p. 286).

Short Summary

XMLXPathReader reads data from XML files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data source</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLXPathReader</td>
<td>XML file</td>
<td>0-1</td>
<td>1-n</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 272) 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 the DOM parser). 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For port reading. See Reading from Input Port (p. 289).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any 1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>2)</td>
<td>For correct data records</td>
<td>Any 1) (each port can have different metadata)</td>
</tr>
</tbody>
</table>
**Legend:**

1) Metadata on each output port does not need to be the same. Metadata can use [Autofilling Functions](#) (p. 128). Note: `source_timestamp` and `source_size` functions work only when reading from a file directly (if the file is an archive or it is stored in a remote location, timestamp will be empty and size will be 0).

2) Other output ports are required if mapping requires that.

### XMLXPathReader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Specifies which data source(s) will be read (XML file, console, input port, dictionary). See <a href="#">Supported File URL Formats for Readers</a> (p. 286).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records that are read.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See <a href="#">Data Policy</a> (p. 295) for more information.</td>
<td>Strict (default)</td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition. See <a href="#">XMLXPathReader Mapping Definition</a> (p. 433) for more information.</td>
<td></td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Mapping the input XML structure to output ports. See <a href="#">XMLXPathReader Mapping Definition</a> (p. 433) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML features</td>
<td>Sequence of individual true/false expressions related to XML features which should be validated. The expressions are separated from each other by semicolon. See <a href="#">XML Features</a> (p. 296) for more information.</td>
<td></td>
</tr>
<tr>
<td>Number of skipped mappings</td>
<td>Number of mappings to be skipped continuously throughout all source files. See <a href="#">Selecting Input Records</a> (p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of mappings</td>
<td>Maximum number of records to be read continuously throughout all source files. See <a href="#">Selecting Input Records</a> (p. 294).</td>
<td>0-N</td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these has to be specified. If both are specified, Mapping URL has higher priority.
Advanced Description

Example 51.10. Mapping in XMLXPathReader

```xml
<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. 434) and XMLXPathReader Mapping Tag Attributes (p. 435), respectively.

Important

By default, mapping definition is implicit. Therefore elements (e.g. `salary`) are automatically mapped onto fields of the same name (`salary`) and you do not have to write:

```xml
<Mapping xpath="salary" cloverField="salary"/>
```

Thus, use explicit mapping only to populate fields with data from distinct elements.

4. XMLXPathReader Context Tags and Mapping Tags

   - Empty Context Tag (Without a Child)
     ```xml
     <Context xpath="xpathexpression" XMLXPathReader Context Tag Attributes (p. 434) />
     ```

   - Non-Empty Context Tag (Parent with a Child)
     ```xml
     <Context xpath="xpathexpression" XMLXPathReader Context Tag Attributes (p. 434) >
     ```
(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. 435) />

  • nodeName is used:

    <Mapping nodeName="elementname" XMLXPathReader_Mapping_Tag_Attributes (p. 435) />

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.
Chapter 51. Readers

• 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.

**Reading Multivalue Fields**

As of Clover 3.3, reading multivalue fields is supported - you can read only lists, however (see Multivalue Fields (p. 163)).

**Note**

Reading maps is handled as reading pure `string` (for all data types as map's values).
Example 51.11. Reading lists with XMLXPathReader

An example input file containing these elements (just a code snippet):

```xml
...<attendees>John</attendees>
<attendees>Vicky</attendees>
<attendees>Brian</attendees>
...
```

can be read back by the component with this mapping:

```xml
<Mapping xpath="attendees" cloverField="attendanceList"/>
```

where `attendanceList` is a field of your metadata. The metadata has to be assigned to the component’s output edge. After you run the graph, the field will get populated by XML data like this (that what you will see in View data):

```
[John, Vicky, Brian]
```
Chapter 52. Writers

We assume that you already know what components are. See Chapter 18, Components (p. 94) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

We can distinguish **Writers** according to what they can write:

- One component discards data:
  - **Trash** (p. 524) discards data.

Other **Writers** write data to files.

- Flat files:
  - **UniversalDataWriter** (p. 526) writes data to flat files (delimited or fixed length).

- Other files:
  - **CloverDataWriter** (p. 440) writes data to files in Clover binary format.
  - **XLSDataWriter** (p. 529) writes data to XLS or XLSX files.
  - **StructuredDataWriter** (p. 520) writes data to files with user-defined structure.
  - **XMLWriter** (p. 532) creates XML files from input data records.
  - **DBFDataWriter** (p. 448) writes data to dbase file(s).

Other **Writers** loads data into database.

- Databases Writers:
  - **DBOutputTable** (p. 451) loads data into database using JDBC driver.
  - **QuickBaseRecordWriter** (p. 504) writes data into the QuickBase online database.
  - **QuickBaseImportCSV** (p. 502) writes data into the QuickBase online database.
  - **LotusWriter** (p. 487) writes data into Lotus Notes and Lotus Domino databases.

- High-Speed Database Specific Writers (Bulk Loaders):
  - **DB2DataWriter** (p. 442) loads data into DB2 database using DB2 client.
  - **InfobrightDataWriter** (p. 463) loads data into Infobright database using Infobright client.
  - **InformixDataWriter** (p. 465) loads data into Informix database using Informix client.
Chapter 52. Writers

- **MSSQLDataWriter** (p. 489) loads data into MSSQL database using MSSQL client.
- **MySQLDataWriter** (p. 492) loads data into MYSQL database using MYSQL client.
- **OracleDataWriter** (p. 495) loads data into Oracle database using Oracle client.
- **PostgreSQLDataWriter** (p. 499) loads data into PostgreSQL database using PostgreSQL client.

Other **Writers** send e-mails, JMS messages or write directory structure.

- E-mails:
  - **EmailSender** (p. 459) converts data records into e-mails.

- JMS messages:
  - **JMSWriter** (p. 477) converts data records into JMS messages.

- Directory structure:
  - **LDAPWriter** (p. 485) converts data records into a directory structure.
CloverDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

**Short Summary**

CloverDataWriter writes data to files in our internal binary Clover data format.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloverDataWriter</td>
<td>Clover binary file</td>
<td>1</td>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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**

CloverDataWriter

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For received data records</td>
<td>Any</td>
</tr>
</tbody>
</table>
### CloverDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>File URL</strong></td>
<td>yes</td>
<td>Attribute specifying where received data will be written (Clover data file, console, dictionary). See Supported File URL Formats for Writers (p. 299) See also Output File Structure (p. 441) for more information.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>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).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Save metadata</td>
<td></td>
<td>By default, no file with metadata definition is saved. If set to true, metadata is saved to metadata file. See Output File Structure (p. 441) for more information.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Save index ¹</td>
<td></td>
<td>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. 441) for more information.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>By default, non-existing directories are not created. If set to true, they are created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Compress level</td>
<td></td>
<td>Sets the compression level. By default, zip compression level is used. Level 0 means archiving without compression.</td>
<td>-1 (default)</td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be written to the output file. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
</tbody>
</table>

**Legend:**

¹ Please note this is a deprecated attribute.

### 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 52.1. 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

DB2DataWriter loads data into DB2 database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2DataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For information about incorrect records</td>
<td>Error Metadata for DB2DataWriter (p. 443)</td>
</tr>
</tbody>
</table>

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. 128).

### Table 52.1. Error Metadata for DB2DataWriter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;any_name1&gt;</td>
<td>integer</td>
<td>Number of incorrect record (records are numbered starting from 1)</td>
</tr>
<tr>
<td>1</td>
<td>&lt;any_name2&gt;</td>
<td>integer</td>
<td>number of incorrect field (for delimited records), fields are numbered</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>starting from 1</td>
</tr>
<tr>
<td>2</td>
<td>&lt;any_name3&gt;</td>
<td>string</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### DB2DataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File metadata</td>
<td></td>
<td>Metadata of external file. Must only be delimited. Each column except the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>last one is followed by an identical, one char delimiter. The last</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>delimiter following the last column is \n. Delimiter must not be a part</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>of any field value.</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>yes</td>
<td>Name of the database into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td>Database table</td>
<td>yes</td>
<td>Name of the database table into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td>User name</td>
<td>yes</td>
<td>Database user.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td>yes</td>
<td>Password for database user.</td>
<td></td>
</tr>
<tr>
<td>Load mode</td>
<td></td>
<td>Mode of the action performed when loading data. See [Load mode](p. 446)</td>
<td>insert (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for more information.</td>
<td>replace</td>
</tr>
<tr>
<td>Field mapping</td>
<td>1)</td>
<td>Sequence of individual mappings ($CloverField:=DBField) separated by</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>semicolon, colon, or pipe. See [Mapping of Clover Fields to DB Fields](p.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>446) for more information.</td>
<td></td>
</tr>
<tr>
<td>Clover fields</td>
<td>1)</td>
<td>Sequence of Clover fields separated by semicolon, colon, or pipe. See</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Mapping of Clover Fields to DB Field](p. 446) for more information.</td>
<td></td>
</tr>
<tr>
<td>DB fields</td>
<td>1)</td>
<td>Sequence of DB fields separated by semicolon, colon, or pipe. See</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Mapping of Clover Fields to DB Field](p. 446) for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loader input file</td>
<td>2)</td>
<td>Name of input file to be loaded, including path. See [Loader input file](p.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>447) for more information.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td>All parameters that can be used as parameters by load method. These</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>values are contained in a sequence of pairs of the following form: key=value,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or key only (if the key value is the boolean true) separated from each</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other by semicolon, colon, or pipe. If the value of any parameter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>contains the delimiter as its part, such value must be double quoted.</td>
<td></td>
</tr>
<tr>
<td>Rejected records URL</td>
<td></td>
<td>Name of the file, including path, on DB2 server where rejected records</td>
<td></td>
</tr>
<tr>
<td>(on server)</td>
<td></td>
<td>will be saved. Must be located in the directory owned by database user.</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Batch file URL</td>
<td></td>
<td>URL of the file where the <code>connect</code>, <code>load</code> and <code>disconnect</code> commands for <code>db2</code> load utility are stored. Normally the batch file is automatically generated, stored in current directory and deleted after the load finishes. If the <strong>Batch file URL</strong> 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 <strong>Loader input file</strong> 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.</td>
<td></td>
</tr>
<tr>
<td>DB2 command interpreter</td>
<td></td>
<td>Interpreter that should execute script with <code>DB2</code> commands (<code>connect</code>, <code>load</code>, <code>disconnect</code>). Its form must be the following: <code>interpreterName [parameters] ${}</code> [parameters]. This <code>${}</code> expression must be replaced by the name of this script file.</td>
<td></td>
</tr>
<tr>
<td>Use pipe transfer</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Column delimiter</td>
<td></td>
<td>The first one char field delimiter from <strong>File metadata</strong> or the metadata on the input edge (if <strong>File metadata</strong> 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 <code>coldel</code> parameter in the <strong>Parameters</strong> attribute. If <strong>Column delimiter</strong> is set, <code>coldel</code> in <strong>Parameters</strong> is ignored.</td>
<td></td>
</tr>
<tr>
<td>Number of skipped</td>
<td></td>
<td>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.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>records</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of</td>
<td></td>
<td>Maximum number of records to be loaded into database. The same can be set by specifying the value of the <code>rowcount</code> parameter in the <strong>Parameters</strong> attribute. If <code>rowcount</code> is set in <strong>Parameters</strong>, the <strong>Max number of records</strong> attribute is ignored.</td>
<td>all (default)</td>
</tr>
<tr>
<td>records</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>Maximum number of records after which the load stops. If some number is set explicitly and when it is reached, process can continue in <strong>RESTART</strong> mode. In <strong>REPLACE</strong> mode, process continues from the beginning. The same number can be specified with the help of <code>warningcount</code> in the <strong>Parameters</strong> attribute. If <code>warningcount</code> is specified, <strong>Max error count</strong> is ignored.</td>
<td>all (default)</td>
</tr>
<tr>
<td>Max warning count</td>
<td></td>
<td>Maximum number of printed error messages and/or warnings.</td>
<td>999 (default)</td>
</tr>
<tr>
<td>Fail on warnings</td>
<td></td>
<td>By default, the component fails on errors. Switching the attribute to true, you can make the component fail on warnings. Background: when an underlying bulk-loader utility finishes with a warning, it is just logged to the console. This behavior is sometimes undesirable as warnings from underlying bulk-loaders may seriously impact further processing. For example, ‘Unable to extend table space’ may result in not loading all data records to a database; hence not completing the expected task successfully.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>
Legend:

1) See Mapping of Clover Fields to DB Fields (p. 446) 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. 447) 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.
DBFDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

DBFDataWriter writes data to dbase file(s). Handles Character/Number/Logical/Date dBase data types. Input metadata has to be fixed-length as you are writing binary data.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBFDataWriter</td>
<td>.dbf file</td>
<td>1</td>
<td>0</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Abstract

DBFDataWriter writes data to dbase file(s).

The component can write a single file or a partitioned collection of files.

Important

Remember the output data can be stored only locally. Uploading via a remote transfer protocol and writing ZIP and TAR archives is not supported.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✔️</td>
<td>Incoming data records</td>
<td>Fixed length</td>
</tr>
</tbody>
</table>
# DBFDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>✓</td>
<td>Specifies where data will be written to (path to a .dbf file), see Supported File URL Formats for Writers (p. 299).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Character encoding of records written to the output.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>If records are printed into a non-empty file, they replace the previous content by default (false). If set to true, new records are appended at the end of the existing output file(s).</td>
<td>false (default)</td>
</tr>
<tr>
<td>DBF type</td>
<td></td>
<td>Type of the created DBF file (determined by the first byte of file header). If you are unsure which type to choose, leave the attribute to default.</td>
<td>0x03 FoxBASE+ (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>When true, non-existing directories contained in the File URL path are automatically created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>Maximum number of records to be written to each output file. If specified, the dollar sign(s) $ ('number of digits' placeholder) must be a part of the file name mask, see Supported File URL Formats for Writers (p. 299).</td>
<td>1 - N</td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records/rows to be skipped before writing the first record to the output file, see Selecting Output Records (p. 306).</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Aggregate number of records/rows to be written to all output files, see Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Exclude fields</td>
<td></td>
<td>Sequence of field names that will not be written to the output (separated by semicolon). Can be used when the same fields serve as a part of Partition key.</td>
<td></td>
</tr>
<tr>
<td>Partition key 2)</td>
<td></td>
<td>Sequence of field names defining record distribution among multiple output files - records with the same Partition key are written to the same output file. Use semicolon ':' as field names separator. Depending on selected Partition file tag use appropriate placeholder ($ or #) in the file name mask, see Partitioning Output into Different Output Files (p. 306)</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table 1)</td>
<td></td>
<td>ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag 2) 2)</td>
<td></td>
<td>By default, partitioned 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. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Partition output fields</td>
<td>1/1</td>
<td>Fields of <strong>Partition lookup table</strong> whose values are used as output file(s) names. See [Partitioning Output into Different Output Files](p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition unassigned file name</td>
<td></td>
<td>Name of the file which the unassigned records should be written into (if there are any). Unless specified, data records whose key values are not contained in <strong>Partition lookup table</strong> are discarded. See [Partitioning Output into Different Output Files](p. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

2) Either both or neither of these two attributes must be specified.
1) Either both or neither of these two attributes must be specified.
We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

**Short Summary**

**DBOutputTable** loads data into database using JDBC driver.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBOutputTable</td>
<td>database</td>
<td>1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>For rejected records</td>
<td>Based on Input 0(^1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>For returned values</td>
<td>Any(^2)</td>
</tr>
</tbody>
</table>

**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. 128): 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 $outField1:=sinFieldA,$outField2:=update_count, $outField3:=sinFieldB). 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 number of incoming records.

**DBOutputTable Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB connection</td>
<td>✓</td>
<td>ID of the DB connection to be used.</td>
<td></td>
</tr>
<tr>
<td>Query URL</td>
<td>1)</td>
<td>Name of external file, including path, defining SQL query. See Query or DB Table is Defined (p. 453) for more information.</td>
<td></td>
</tr>
<tr>
<td>SQL query</td>
<td>1)</td>
<td>SQL query defined in the graph. See Query or DB Table is Defined (p. 453) for more information. See also SQL Query Editor (p. 456).</td>
<td></td>
</tr>
<tr>
<td>DB table</td>
<td>1)</td>
<td>Name of DB table. See Query or DB Table is Defined (p. 453) for more information.</td>
<td></td>
</tr>
<tr>
<td>Field mapping</td>
<td>2)</td>
<td>Sequence of individual mappings ($CloverField:=DBField) separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 454) for more information.</td>
<td></td>
</tr>
<tr>
<td>Clover fields</td>
<td>2)</td>
<td>Sequence of Clover fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 454) for more information.</td>
<td></td>
</tr>
<tr>
<td>DB fields</td>
<td>2)</td>
<td>Sequence of DB fields separated by semicolon, colon, or pipe. See Mapping of Clover Fields to DB Fields (p. 454) for more information.</td>
<td></td>
</tr>
<tr>
<td>Query source charset</td>
<td></td>
<td>Encoding of external file defining SQL query.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Batch mode</td>
<td></td>
<td>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. 455) for more information.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Batch size</td>
<td></td>
<td>Number of records that can be sent to database in one batch update. See Batch Mode and Batch Size (p. 455) for more information.</td>
<td>25 (default)</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>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 until the connection is closed during graph freeing. This attribute is ignored if Atomic SQL query is defined.</td>
<td>100 (default)</td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>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. 455) for more information.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Action on error</td>
<td>By default, when the number of errors exceeds <strong>Max error count</strong>, correct records are committed into database. If set to <strong>ROLLBACK</strong>, no commit of the current batch is performed. See <strong>Errors</strong> (p. 455) for more information.</td>
<td>COMMIT (default)</td>
<td>ROLLBACK</td>
</tr>
<tr>
<td>Atomic SQL query</td>
<td>Sets atomicity of executing SQL queries. If set to <strong>true</strong>, all SQL queries for one record are executed as atomic operation, but the value of the <strong>Commit</strong> attribute is ignored and commit is performed after each record. See <strong>Atomic SQL Query</strong> (p. 456) for more information.</td>
<td>false (default)</td>
<td>true</td>
</tr>
</tbody>
</table>

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. 453) for more information.

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. 456).

  - **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. 454) for more information.
    
    See also **SQL Query Editor** (p. 456).

**Example 52.2. Examples of Queries**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert (with clover fields)</td>
<td>insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]</td>
</tr>
<tr>
<td>insert (with question marks)</td>
<td>insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?) [returning $out1field1 := $in0field3[, $out1field2 := auto_generated][, $out1field3 := $in0field7]]</td>
</tr>
</tbody>
</table>
| **DB2, Oracle**
<p>| insert (with clover fields) | insert into mytable [(dbf1,dbf2,...,dbfn)] values ($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[, $out1field3 := dbf7]] |</p>
<table>
<thead>
<tr>
<th>Statement</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert (with clover fields)</td>
<td>insert into mytable [(dbf1,dbf2,...,dbfn)] values ($in0field1, constant1, id_seq.nextvalue, $in0field2, ..., constantk, $in0fieldm)</td>
</tr>
<tr>
<td>insert (with question marks)</td>
<td>insert into mytable [(dbf1,dbf2,...,dbfn)] values (?, ?, id_seq.nextval, ?, constant1, ?, ?, ?, ?, constant2, ?, ?, ?, ?, ?)</td>
</tr>
<tr>
<td>update</td>
<td>update mytable set dbf1 = $in0field1, ..., dbfn = $in0fieldn [returning $out1field1 := $in0field3[, $out1field2 := update_count][, $out1field3 := $in0field7]]</td>
</tr>
<tr>
<td>delete</td>
<td>delete from mytable where dbf1 = $in0field1 and ... and dbfj = ? and dbfn = $in0fieldn</td>
</tr>
</tbody>
</table>

**Legend:**

1) These databases 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. 454) 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**:

```bash
$CloverFieldA:=DBFieldA;...;$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 committed, cannot be rolled back anymore.

**Atomic SQL Query**

- **Atomic SQL query** specifies the way how queries consisting of multiple subqueries concerning a single records will be processed.

By default, each individual subquery is considered separately and in some of these fails, the previous are committed or rolled back according to database.

If the **Atomic SQL query** attribute is set to true, either all subqueries or none of them are committed 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](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 52.1. 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 52.2. 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 52.3. 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

EmailSender sends e-mails.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmailSender</td>
<td>flat file</td>
<td>1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For data for e-mails</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For successfully sent e-mails.</td>
<td>Input 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>For rejected e-mails.</td>
<td>Input 0 plus field named errorMessage1)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMTP server</td>
<td>yes</td>
<td>Name of SMTP server for outgoing e-mails.</td>
<td></td>
</tr>
<tr>
<td>SMTP user</td>
<td></td>
<td>Name of the user for an authenticated SMTP server.</td>
<td></td>
</tr>
<tr>
<td>SMTP password</td>
<td></td>
<td>Password of the user for an authenticated SMTP server.</td>
<td></td>
</tr>
<tr>
<td>Use TLS</td>
<td></td>
<td>By default, TLS is not used. If set to true, TLS is turned on.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Use SSL</td>
<td></td>
<td>By default, SSL is not used. If set to true, SSL is turned on.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Message</strong></td>
<td>yes</td>
<td>Set of properties defining the message headers and body. See E-Mail Message (p. 460) for more information.</td>
<td></td>
</tr>
<tr>
<td>Attachments</td>
<td></td>
<td>Set of properties defining the message attachments. See E-Mail Attachments (p. 461) for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMTP port</td>
<td></td>
<td>Number of the port used for connection to SMTP server.</td>
<td>25 (default)</td>
</tr>
<tr>
<td>Trust invalid SMTP server certificate</td>
<td></td>
<td>By default, invalid SMTP server certificates are not accepted. If set to true, invalid SMTP server certificate (with different name, expired, etc) is accepted.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Ignore send fail</td>
<td></td>
<td>By default, when an e-mail is not successfully sent, graph fails. If set to true, graph execution stops even if no mail can be sent successfully.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

### Advanced Description

- **E-Mail Message**

To define the **Message** attribute, you can use the following wizard:

![EmailSender Message Wizard](image)

*Figure 52.4. 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
```

**Tip**

To send email to multiple recipients, separate their addresses by comma ‘,’. If needed, use the same delimiter in the Cc and Bcc fields.

- **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:

![Edit Attachments Wizard](image)

**Figure 52.5. 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:
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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

**Short Summary**

InfobrightDataWriter loads data into Infobright database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfobrightDataWriter</td>
<td>database</td>
<td>1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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**
Chapter 52. Writers

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For records as they were loaded into database</td>
<td>Corresponding part of metadata on Input 0¹</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB connection</td>
<td>yes</td>
<td>ID of the DB connection object to access the database.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Database table</td>
<td>yes</td>
<td>Name of DB table where data will be loaded.</td>
<td>Text (default)</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Charset used for encoding string values to VAR, VARCHAR column types.</td>
<td></td>
</tr>
<tr>
<td>Data format</td>
<td></td>
<td><code>bh_dataformat</code> supported by Infobright. Options are: <code>txt_variable</code> or <code>binary</code> (this is faster, but works with IEE only).</td>
<td></td>
</tr>
<tr>
<td>Advanced</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clover fields</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Log file</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Append data to log file</td>
<td></td>
<td>By default, new record overwrite the older ones. If set to <code>true</code>, new records are appended to the older ones.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Execution timeout</td>
<td></td>
<td>Timeout for load command (in seconds). Has effect only on Windows platform.</td>
<td>15 (default)</td>
</tr>
<tr>
<td>Check string's and binary's sizes</td>
<td></td>
<td>By default, sizes are not checked before data is passed to database. If set to <code>true</code>, sizes are check - should be set to <code>true</code> if debugging is supported.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Remote agent port</td>
<td></td>
<td>Port to be used when connecting to the server.</td>
<td>55555 (default)</td>
</tr>
</tbody>
</table>
InformixDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

InformixDataWriter loads data into an Informix database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>InformixDataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For information about incorrect records</td>
<td>Input 0 (plus two Error Fields for InformixDataWriter (p. 466))</td>
</tr>
</tbody>
</table>

**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 52.2. Error Fields for InformixDataWriter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastInputField + 1</td>
<td>&lt;anyname1&gt;</td>
<td>integer</td>
<td>Number of row</td>
</tr>
<tr>
<td>LastInputField + 2</td>
<td>&lt;anyname2&gt;</td>
<td>string</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### InformixDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path to dbload utility</td>
<td>yes</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>Host where database server is located.</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>yes</td>
<td>Name of the database into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td>Database table</td>
<td>yes</td>
<td>Name of the database table into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control script</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Error log URL</td>
<td></td>
<td>Name of the error log file, including path. If not set, default error log file is used instead.</td>
<td>/error.log</td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>Maximum number of allowed records. When this number is exceeded, graph fails.</td>
<td>10 (default)</td>
</tr>
<tr>
<td>Ignore rows</td>
<td></td>
<td>Number of rows to be skipped. Is used only if the Use load utility attribute is set to true.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Commit interval</td>
<td></td>
<td>Commit interval in number of rows.</td>
<td>100 (default)</td>
</tr>
<tr>
<td>Column delimiter</td>
<td></td>
<td>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.</td>
<td>&quot;&quot; (default)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Loader input file</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Use load utility</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>User name</td>
<td></td>
<td>Username to be used when connecting to the database. Is used only if the Use load utility attribute is set to true.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password to be used when connecting to the database. Is used only if the Use load utility attribute is set to true.</td>
<td></td>
</tr>
<tr>
<td>Ignore unique key violation</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Use insert cursor</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

**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.
Chapter 52. Writers

JavaBeanWriter

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the appropriate Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

JavaBeanWriter writes a hierarchical structure as JavaBeans into a dictionary. A number of classes is supported for writing. That allows dynamic data interchange between Clover graphs and external environment, such as cloud. Which JavaBean you choose defines the output to a certain extent - that is why you map inputs to a preset but customizable tree structure. Also, you can write data to Java collections (Lists, Maps). When writing, JavaBeanWriter consults your bean's classpath to decide which data types to write. That means it performs type conversions between your metadata field types and JavaBeans types. If a conversion fails, you will experience errors on writing.

If you are looking for a more flexible component which is less restrictive in terms of data types and requires no external classpath, choose JavaMapWriter.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaBeanWriter</td>
<td>dictionary</td>
<td>1-n</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

JavaBeanWriter receives data through all connected input ports and converts Clover records to JavaBean properties based on the mapping you define. At last, the resulting tree structure is written to a dictionary (which is the only possible output). Remember the component cannot write to a file.

The logic of mapping is similar to XMLWriter (p. 534) - if you are familiar with its mapping editor, you will have no problems designing the output tree in this component. The differences are:

- you cannot map input to output freely - the design of the tree structure you can see in the mapping editor is determined by the JavaBean you are using
- JavaBeanWriter allows you to map to Beans, their properties or collections - Lists, Maps
- there are no attributes, wildcard attributes and wildcard elements as in XML

Icon
Chapter 52. Writers

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-N</td>
<td>At least one</td>
<td>Input records to be joined and mapped to JavaBeans.</td>
<td>Any (each port can have different metadata)</td>
</tr>
</tbody>
</table>

**JavaBeanWriter Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary target</td>
<td>yes</td>
<td>The dictionary you want to write JavaBeans to.</td>
<td>Name of a dictionary you have previously defined.</td>
</tr>
<tr>
<td>Bean structure</td>
<td></td>
<td>Click the ‘...’ button to design the structure of your output JavaBean consisting of custom classes, objects, collections or maps.</td>
<td>See Defining Bean Structure (p. 469).</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Defines how input data is mapped to output JavaBeans.</td>
<td>See Mapping Editor (p. 470).</td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition.</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.

**Advanced Description**

**Defining Bean Structure**

Before you can start mapping, you need to define contents of the output JavaBean. Start by editing the **Bean structure** attribute which opens this dialog:

![Image of the Object Inspector dialog](image)

**Figure 52.7. Defining bean structure - click the Select combo box to start.**

- **Java object** - clicking it opens a dialog in which you can choose from Java classes. **Important:** if you intend to use a custom JavaBeans class, place it into the `trans` folder. The class will then be available in this dialog.
• **Collection** - adds a list consisting of other objects, maps or other collections.

• **Map** - adds a key-value map.

### Mapping Editor

Having defined the bean structure, proceed to mapping input records to output JavaBeans. You perform that in a manner which is very close to what you already know from XMLWriter (p. 534). Mapping editors in both components have similar logic.

The very basics of mapping are:

• Edit the component’s **Mapping** attribute. This will open the visual mapping editor:

![Mapping editor in JavaBeanWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree - it is pre-defined by your bean's structure (note: in the example, the bean contains employees and projects they are working on). Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).](image)

- In the right hand pane, you can map input metadata to:
  - Beans
  - Bean properties
  - Lists
  - Maps

Click the green '+' sign to **Add entry**. This adds a new item into the tree - its type depends on context (the node you have selected). Remember the button is not available every time as the output structure is determined by bean structure (p. 469).
• Connect input records to output nodes to create Binding (p. 542).

**Example 52.3. Creating Binding**

![Example mapping in JavaBeanWriter](image)

*Figure 52.9. Example mapping in JavaBeanWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary, i.e. they are used in the mapping.*

• At any time, you can switch to the Source tab (p. 546) and write/check the mapping yourself in code.

• If the basic instructions found here were not satisfying, please consult XMLWriter's Advanced Description (p. 534) where the whole mapping process is described profusely.
JavaMapWriter

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the appropriate Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

**JavaMapWriter** writes JavaBeans (represented as HashMaps) into a dictionary. That allows dynamic data interchange between Clover graphs and external environment, such as cloud. The component is a specific implementation of **JavaBeanWriter** which allows easier mapping. Maps are less restrictive than Beans: there are no data types and type conversion is missing. This gives **JavaMapWriter** a greater flexibility - it always writes data types into Maps just as they were defined in metadata. However, its lower overheads come at the cost of accidental reading a different data type than desired (e.g. if you write string into a Map and then read it back as integer with **JavaBeanReader**, the graph fails).

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaMapWriter</td>
<td>dictionary</td>
<td>1-n</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

**JavaMapWriter** component receives data through all connected input ports and converts data records to Java HashMaps based on the mapping you define. At last, the component writes the resulting tree structure of elements to Maps.

The logic of mapping is similar to XMLWriter (p. 534) - if you are familiar with its mapping editor, you will have no problems designing the output tree in this component. The differences are:

- **JavaMapWriter** allows you to map arrays
- there are no attributes as in XML

Remember the component cannot write to a file - the only possible output is dictionary (p. 218).

Icon
Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-N</td>
<td>At least one</td>
<td>Input records to be joined and mapped to Java Maps.</td>
<td>Any (each port can have different metadata)</td>
</tr>
</tbody>
</table>

JavaMapWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary target</td>
<td>yes</td>
<td>The dictionary you want to write Java Maps to.</td>
<td></td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Defines how input data is mapped onto output Java Maps. See Advanced Description (p. 474).</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition.</td>
<td></td>
</tr>
</tbody>
</table>

Legend:

1) One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.
Advanced Description

You map the input records to the output dictionary in a manner which is very close to what you already know from XMLWriter (p. 534). Mapping editors in both components have similar logic. The very basics of mapping are:

- Connect input edges to **JavaMapWriter** and edit the component's **Mapping** attribute. This will open the visual mapping editor:

![Mapping editor in JavaMapWriter](image)

*Figure 52.10. Mapping editor in JavaMapWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired output tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).*

- In the right hand pane, design your output tree structure consisting of
  - Elements (p. 538)

  **Important**

  Unlike **XMLWriter**, you do not map metadata to any 'attributes'.

  - **Arrays** - arrays are ordered sets of values. To learn how to map them in, see Example 52.5, “Writing arrays” (p. 475).

  - Wildcard elements (p. 540) - another option to mapping elements explicitly. You use the **Include** and **Exclude** patterns to generate element names from respective metadata.

  - Connect input records to output (wildcard) elements to create Binding (p. 542).

**Example 52.4. Creating Binding**
Figure 52.11. Example mapping in JavaMapWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output dictionary.

Note

If you extended your graph and had the output dictionary written to the console, you would get a structure like this. This excerpt is just to demonstrate how Java Maps, mapped in the figure above (p. 475), are stored internally:

```java
[{employees=[{projects=[{manager=John Major, start=01062005, name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006}, {manager=Raymond Brown, start=11052004, name=OLAP, customers=[Sunny], end=31012006}, {manager=Brandon Morrison, start=01032006, name=Javascripting, customers=[Nestele, Traincorp, AnotherBank, Intershop], end=in progress}], lastName=Morrison, firstName=Brandon}], {manager=John Smith, start=06062006, name=JSP, customers=[Sunny, Weblea], end=in progress}], {manager=Simson, firstName=Jane}, {projects=[{manager=John Major, start=01062005, name=Data warehouse, customers=[Hanuman, Weblea, SomeBank], end=31052006}, {manager=Raymond Brown, start=11052004, name=OLAP, customers=[Sunny], end=31012006}, {manager=Brandon Morrison, start=01032006, name=Javascripting, customers=[Nestele, Traincorp, AnotherBank, Intershop], end=in progress}], {manager=John Smith, start=06062006, name=JSP, customers=[Sunny, Weblea], end=in progress}]
```

Example 52.5. Writing arrays

Let us have the following mapping of the input file which contains information about actors. For explanatory reasons, we will part actors' personal data from their countries of origin. The summary of all countries will then be written into an array:
Figure 52.12. Mapping arrays in JavaMapWriter - notice the array contains a dummy element ‘State’ which you bind the input field to.

The array will be written to Maps as e.g.:

```plaintext
[ ...
  Summary={Countries=[USA, ESP, ENG]}]
```

- At any time, you can switch to the Source tab (p. 546) and write/check the mapping yourself in code.
- If the basic instructions found here were not satisfying, please consult XMLWriter’s Advanced Description (p. 534) where the whole mapping process is described profusely.
JMSWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

**JMSWriter** converts Clover data records into JMS messages.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSWriter</td>
<td>jms messages</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For data records</td>
<td>Any&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata on input port may contain a field specified in the **Message body field** attribute.
## JMSWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS connection</td>
<td>yes</td>
<td>ID of the JMS connection to be used.</td>
<td></td>
</tr>
<tr>
<td>Processor code</td>
<td></td>
<td>Transformation of records to JMS messages written in the graph in Java.</td>
<td></td>
</tr>
<tr>
<td>Processor URL</td>
<td></td>
<td>Name of external file, including path, containing the transformation of records to JMS messages written in Java.</td>
<td></td>
</tr>
<tr>
<td>Processor class</td>
<td></td>
<td>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.</td>
<td>DataRecord2JmsMsgProperties (default)</td>
</tr>
<tr>
<td>Processor source charset</td>
<td></td>
<td>Encoding of external file containing the transformation in Java.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Message charset</td>
<td></td>
<td>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.</td>
<td>ISO-8859-1 (default)</td>
</tr>
</tbody>
</table>

### 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. 479) for more information.

See also [Defining Transformations](p. 268) 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. 284).

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.
Chapter 52. Writers

**JSONWriter**

**Commercial Component**

We assume that you have already learned what is described in:

- Chapter 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)
- Chapter 42, *Common Properties of Writers* (p. 298)

If you want to find the appropriate Writer for your purposes, see *Writers Comparison* (p. 299).

**Short Summary**

**JSONWriter** writes data in the JSON format.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Each to all outputs</th>
<th>Different to different outputs</th>
<th>Transformation</th>
<th>Transf. req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSONWriter</td>
<td>JSON file</td>
<td>1-n</td>
<td>0-1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**Abstract**

**JSONWriter** receives data from all connected input ports and converts records to JSON objects based on the mapping you define. Finally, the component writes the resulting tree structure of elements to the output: a JSON file, port or dictionary.

**Icon**

![Icon](image)

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-N</td>
<td>At least one</td>
<td>Input records to be joined and mapped into the JSON structure.</td>
<td>Any (each port can have different metadata)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Optional. For port writing.</td>
<td>Only one field (byte or cbyte or string) is used. The field name is used in File URL to govern how the output records are processed - see Writing to Output Port (p. 301)</td>
</tr>
</tbody>
</table>

**JSONWriter Attributes**
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>yes</td>
<td>The target file for the output JSON. See Supported File URL Formats for Writers (p. 299).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>The encoding of an output file generated by JSONWriter.</td>
<td>ISO-8859-1 (default) &lt;other encodings&gt;</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Defines how input data is mapped onto an output JSON.</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition.</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

1) One of these has to be specified. If both are specified, **Mapping URL** has a higher priority.
Advanced Description

Every JSON object can contain other nested JSON objects. Thus, the JSON format resembles XML and similar tree formats.

As a consequence, you map the input records to the output file in a manner which is very close to what you already know from XMLWriter (p. 534). Mapping editors in both components have similar logic. The very basics of mapping are:

- Connect input edges to JSONWriter and edit the component’s Mapping attribute. This will open the visual mapping editor:

![Mapping editor in JSONWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired JSON tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).](image)

Figure 52.13. Mapping editor in JSONWriter after first open. Metadata on the input edge(s) are displayed on the left hand side. The right hand pane is where you design the desired JSON tree. Mapping is then performed by dragging metadata from left to right (and performing additional tasks described below).

- In the right hand pane, design your JSON tree consisting of
  - Elements (p. 538)

  **Important**

  Unlike XMLWriter, you do not map metadata to any 'attributes'.

  - Arrays - arrays are ordered sets of values in JSON enclosed between the [ and ] brackets. To learn how to map them in JSONWriter, see Example 52.7, “Writing arrays” (p. 484).

  - Wildcard elements (p. 540) - another option to mapping elements explicitly. You use the Include and Exclude patterns to generate element names from respective metadata.

  - Connect input records to output (wildcard) elements to create Binding (p. 542).

Example 52.6. Creating Binding
Figure 52.14. Example mapping in JSONWriter - employees are joined with projects they work on. Fields in bold (their content) will be printed to the output file - see below.

Excerpt from the output file related to the figure above (p. 483) (example of one employee written as JSON):

```
"employee": {
  "firstName": "Jane",
  "lastName": "Simson",
  "projects": [
    "project": {
      "name": "JSP",
      "manager": "John Smith",
      "start": "06062006",
      "end": "in progress",
      "customers": [
        "customer": {
          "name": "Sunny"
        },
        "customer": {
          "name": "Weblea"
        }
      ]
    },
    "project": {
      "name": "OLAP",
      "manager": "Raymond Brown",
      "start": "11052004",
      "end": "31012006",
      "customers": [
        "customer": {
          "name": "Sunny"
        }
      ]
    }
  ]
},
```

Example 52.7. Writing arrays

Let us have the following mapping of the input file which contains information about actors. For explanatory reasons, we will part actors' personal data from their countries of origin. The summary of all countries will then be written into an array

![JSON Writer screenshot](image)

**Figure 52.15. Mapping arrays in JSONWriter - notice the array contains a dummy element 'State' which you bind the input field to.**

The output JSON:

```json
{
  "Actor" : {
    "Name" : "John Malkovich",
    "Sex" : true,
    "Age" : 50
  },
  "Actor" : {
    "Name" : "Liz Hurley",
    "Sex" : false,
    "Age" : 42
  },
  "Actor" : {
    "Name" : "Antonio Banderas",
    "Sex" : true,
    "Age" : 33
  },
  "Summary" : {
    "Countries" : [ "USA", "ENG", "ESP" ]
  }
}
```

- At any time, you can switch to the Source tab (p. 546) and write/check the mapping yourself in code.
- If the basic instructions found here were not satisfying, please consult XMLWriter's Advanced Description (p. 534) where the whole mapping process is described profusely.
Chapter 52. Writers

LDAPWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

LDAPWriter writes information to an LDAP directory.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPWriter</td>
<td>LDAP directory tree</td>
<td>1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
<td>yes</td>
<td>For correct data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0-1</td>
<td>no</td>
<td>For rejected records</td>
<td>Input 0</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDAP URL</td>
<td>yes</td>
<td>LDAP URL of the directory. Can be a list of URLs separated by pipe.</td>
<td>pattern: ldap://host:port/</td>
</tr>
<tr>
<td>Action</td>
<td></td>
<td>Defines the action to be performed with the entry.</td>
<td>replace_attributes (default)</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td>User DN to be used when connecting to the LDAP directory. Similar to the following: cn=john.smith,dc=example,dc=com.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password to be used when connecting to the LDAP directory.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-value separator</td>
<td></td>
<td>LDAPWriter can handle keys with multiple values. These are delimited by this string or character. &lt;none&gt; 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.</td>
<td>&quot;&quot; (default)</td>
</tr>
</tbody>
</table>
LotusWriter

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Chapter 52, Writers (p. 438).

Short Summary

LotusWriter writes data into Lotus Domino databases. Data records are stored as Lotus documents in the database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LotusWriter</td>
<td>Lotus Notes</td>
<td>1</td>
<td>0-1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Abstract

LotusWriter is a component which can write data records to Lotus databases. The writing is 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 library 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 24, Lotus Connections (p. 185)).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✓</td>
<td>for input data records</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for invalid data records</td>
<td></td>
</tr>
</tbody>
</table>
## LotusWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domino connection</td>
<td>✗</td>
<td>ID of the connection to the Lotus Domino database.</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>✗</td>
<td>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.</td>
<td>&quot;insert&quot; (default)</td>
</tr>
<tr>
<td>View</td>
<td>✗</td>
<td>The name of the View in Lotus database within which the data records will be updated.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping</td>
<td>✗</td>
<td>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.</td>
<td>docFieldX := inFieldY; ...</td>
</tr>
<tr>
<td>Compute with form</td>
<td>✗</td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Skip invalid documents</td>
<td>✗</td>
<td>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.</td>
<td>true</td>
</tr>
<tr>
<td>Update mode</td>
<td>✗</td>
<td>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.</td>
<td>&quot;Lazy, first match&quot;(default)</td>
</tr>
<tr>
<td>Multi-value fields</td>
<td>✗</td>
<td>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.</td>
<td>semi-colon separated list of input fields</td>
</tr>
<tr>
<td>Multi-value separator</td>
<td>✗</td>
<td>A string that will be used to separate values from multi-value Lotus fields.</td>
<td>&quot;;;&quot; (default)</td>
</tr>
</tbody>
</table>
MSSQLDataWriter

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

MSSQLDataWriter loads data into MSSQL database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSSQLDataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For information about incorrect records</td>
<td>Input 0 (plus three Error Fields for MSSQLDataWriter (p. 490)²)</td>
</tr>
</tbody>
</table>
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 52.3. Error Fields for MSSQLDataWriter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastInputField + 1</td>
<td>&lt;anyname1&gt;</td>
<td>integer</td>
<td>Number of incorrect row</td>
</tr>
<tr>
<td>LastInputField + 2</td>
<td>&lt;anyname2&gt;</td>
<td>integer</td>
<td>Number of incorrect column</td>
</tr>
<tr>
<td>LastInputField + 3</td>
<td>&lt;anyname3&gt;</td>
<td>string</td>
<td>Error message</td>
</tr>
</tbody>
</table>

MSSQLDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path to bcp utility</td>
<td>yes</td>
<td>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.</td>
</tr>
<tr>
<td>Database</td>
<td>yes</td>
<td>Name of the database where the destination table or view resides.</td>
</tr>
<tr>
<td>Server name</td>
<td></td>
<td>Name of the server to which bcp utility should connect. If bcp utility connects to local named or remote named instance of server, <strong>Server name</strong> should be set to <strong>serverName \instanceName</strong>. If bcp utility connects to local default or remote default instance of server, <strong>Server name</strong> should be set to <strong>serverName</strong>. If it is not set, bcp connects to the local default instance of server on localhost. The same meaning is true for the <strong>serverName</strong> which can be set in Parameters. However, if both <strong>Server name</strong> attribute and the <strong>serverName</strong> parameter are set, <strong>serverName</strong> in Parameters is ignored.</td>
</tr>
<tr>
<td>Database table</td>
<td>1)</td>
<td>Name of the destination table.</td>
</tr>
<tr>
<td>Database view</td>
<td>1)</td>
<td>Name of the destination view. All columns of the view must refer to the same table.</td>
</tr>
<tr>
<td>Database owner</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>User name</td>
<td>yes</td>
<td>Login ID to be used when connecting to the server. The same can be set by specifying the value of the <strong>userName</strong> parameter in the Parameters attribute. If set, <strong>userName</strong> in Parameters is ignored.</td>
</tr>
<tr>
<td>Password</td>
<td>yes</td>
<td>Password for the login ID to be used when connecting to the server. The same can be set by specifying the value of the <strong>password</strong> parameter in the Parameters attribute. If set, <strong>password</strong> in Parameters is ignored.</td>
</tr>
</tbody>
</table>

Advanced
Chapter 52. Writers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column delimiter</td>
<td></td>
<td>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.</td>
<td>All (default)</td>
</tr>
<tr>
<td>Loader input file</td>
<td></td>
<td>Name of input file to be loaded, including path. See Loader input file (p. 491) for more information.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td>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. 491) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

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. 491) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

**Short Summary**

MySQLDataWriter is a high-speed MySQL table loader. Uses MySQL native client.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQLDataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For information about incorrect records</td>
<td>Error Metadata for MySQLDataWriter (p. 493)²</td>
</tr>
</tbody>
</table>

### 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. 128).

### Table 52.4. Error Metadata for MySQLDataWriter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;any_name1&gt;</td>
<td>integer</td>
<td>Number of incorrect record (records are numbered starting from 1)</td>
</tr>
<tr>
<td>1</td>
<td>&lt;any_name2&gt;</td>
<td>integer</td>
<td>number of incorrect column</td>
</tr>
<tr>
<td>2</td>
<td>&lt;any_name3&gt;</td>
<td>string</td>
<td>Error message</td>
</tr>
</tbody>
</table>

### MySQLDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path to mysql utility</td>
<td>yes</td>
<td>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.</td>
<td>localhost (default)</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>Host where database server is located.</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>yes</td>
<td>Name of the database into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td>Database table</td>
<td>yes</td>
<td>Name of the database table into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td>User name</td>
<td>yes</td>
<td>Database user.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td>yes</td>
<td>Password for database user.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path to control script</td>
<td></td>
<td>Name of command file containing the LOAD DATA INFILE statement, including path. See Path to Control Script (p. 494) for more information.</td>
<td></td>
</tr>
<tr>
<td>Lock database table</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Ignore rows</td>
<td></td>
<td>Number of rows of data file that should be skipped. By default, no records are skipped. Valid only for input file with data.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Column delimiter</td>
<td></td>
<td>Delimiter used for each column in data. Field values must not include this delimiter as their part. By default, tabulator is used.</td>
<td>(\backslash ) (default)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Loader input file</td>
<td>1)</td>
<td>Name of input file to be loaded, including path. See Loader input file (p. 494) for more information.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

1) If input port is not connected, **Loader input file** must be specified and contain data. See Loader input file (p. 494) 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.
We assume that you have already learned what is described in:

- Chapter 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)
- Chapter 42, *Common Properties of Writers* (p. 298)

If you want to find the right **Writer** for your purposes, see *Writers Comparison* (p. 299).

**Short Summary**

**OracleDataWriter** loads data into Oracle database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>OracleDataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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**

![Oracle Data Writer](image)

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Rejected records</td>
<td>Input 0</td>
</tr>
</tbody>
</table>
Legend:

1): If no file containing data for loading (Loader input file) is specified, input port must be connected.

### OracleDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path to sqlldr utility</td>
<td>yes</td>
<td>Name of sqlldr utility, including path.</td>
<td></td>
</tr>
<tr>
<td>TNS name</td>
<td>yes</td>
<td>TNS name identifier.</td>
<td></td>
</tr>
<tr>
<td>User name</td>
<td>yes</td>
<td>Username to be used when connecting to the Oracle database.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td>yes</td>
<td>Password to be used when connecting to the Oracle database.</td>
<td></td>
</tr>
<tr>
<td>Oracle table</td>
<td>yes</td>
<td>Name of the database table into which the records should be loaded.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control script</td>
<td></td>
<td>Control script for the sqlldr utility. See Control Script (p. 497) for more information.</td>
<td>append (default)</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>Specifies what should be done with database table. See Append Attribute (p. 497) for more information.</td>
<td>append (default)</td>
</tr>
<tr>
<td>Log file name</td>
<td></td>
<td>Name of the file where the process is logged.</td>
<td>${PROJECT}/loaderinputfile.log</td>
</tr>
<tr>
<td>Bad file name</td>
<td></td>
<td>Name of the file where the records causing errors is written.</td>
<td>${PROJECT}/loaderinputfile.bad</td>
</tr>
<tr>
<td>Discard file name</td>
<td></td>
<td>Name of the file where the records not meeting selection criteria is written.</td>
<td>${PROJECT}/loaderinputfile.dis</td>
</tr>
<tr>
<td>DB column names</td>
<td></td>
<td>Names of all columns in the database table.</td>
<td></td>
</tr>
<tr>
<td>Loader input file</td>
<td></td>
<td>Name of input file to be loaded, including path. See Loader Input File (p. 498) for more information.</td>
<td></td>
</tr>
<tr>
<td>Max error count</td>
<td></td>
<td>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.</td>
<td>50 (default)</td>
</tr>
<tr>
<td>Max discard count</td>
<td></td>
<td>Number of records that can be discarded before the graph stops. If set to 1, even single discarded record stops the graph run.</td>
<td>all (default)</td>
</tr>
<tr>
<td>Ignore rows</td>
<td></td>
<td>Number of rows of the data file that should be skipped when loading data to database.</td>
<td>0 (default)</td>
</tr>
<tr>
<td>Commit interval</td>
<td></td>
<td>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.</td>
<td>64 (default for conventional path)</td>
</tr>
</tbody>
</table>
Chapter 52. Writers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use file for exchange</td>
<td>By default, on Unix pipe transfer is used. If it is set to true and <strong>Loader input file</strong> 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.</td>
<td>false (default on Unix)</td>
<td>true (default on Windows)</td>
</tr>
<tr>
<td>Parameters</td>
<td>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.</td>
<td>false (default)</td>
<td>true</td>
</tr>
<tr>
<td>Fail on warnings</td>
<td>By default, the component fails on errors. Switching the attribute to true, you can make the component fail on warnings. Background: when an underlying bulk-loader utility finishes with a warning, it is just logged to the console. This behavior is sometimes undesirable as warnings from underlying bulk-loaders may seriously impact further processing. For example, 'Unable to extend table space' may result in not loading all data records to a database; hence not completing the expected task successfully.</td>
<td>false (default)</td>
<td>true</td>
</tr>
</tbody>
</table>

**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 52.8. Example of a Control script**

```sql
LOAD DATA
INFILE *
INTO TABLE test
append
{
    name TERMINATED BY ';',
    value TERMINATED BY '
'
}
```

**Append Attribute**

- **Append (default)**

  Specifies that data is simply appended to a table. Existing free space is not used.

- **Insert**
Add 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

PostgreSQLDataWriter loads data into PostgreSQL database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQLDataWriter</td>
<td>database</td>
<td>0-1</td>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Records to be loaded into the database</td>
<td>Any</td>
</tr>
</tbody>
</table>

Legend:

1): If no file containing data for loading (Loader input file) is specified, input port must be connected.
### PostgreSQLDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path to psql utility</td>
<td>yes</td>
<td>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.</td>
<td>localhost</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>Host where database server is located.</td>
<td>localhost</td>
</tr>
<tr>
<td>Database</td>
<td>yes</td>
<td>Name of the database into which the records should be loaded.</td>
<td>Host</td>
</tr>
<tr>
<td>Database table</td>
<td></td>
<td>Name of the database table into which the records should be loaded.</td>
<td>host</td>
</tr>
<tr>
<td>User name</td>
<td></td>
<td>PostgreSQL username to be used when connecting to the server.</td>
<td>User name</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fail on error</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Path to control script</td>
<td></td>
<td>Name of command file containing the \copy statement, including path. See Path to Control Script (p. 500) for more information.</td>
<td>Path to Control Script</td>
</tr>
<tr>
<td>Column delimiter</td>
<td></td>
<td>Delimiter used for each column in data. Field values must not include this delimiter as their part.</td>
<td>tabulator character</td>
</tr>
<tr>
<td>Loader input file</td>
<td></td>
<td>Name of input file to be loaded, including path. See Loader Input File (p. 501) for more information.</td>
<td>Loader Input File</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td>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.</td>
<td>Parameters</td>
</tr>
</tbody>
</table>

### 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

QuickBaseImportCSV adds and updates QuickBase online database table records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBaseImportCSV</td>
<td>QuickBase</td>
<td>1</td>
<td>0-2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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 wraps the API_ImportFromCSV HTTP interaction (http://www.quickbase.com/api-guide/importfromcsv.html).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>for input data records</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for accepted data records</td>
<td>string field for the table Record ID# field values of the imported records</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
<td>✗</td>
<td>for rejected data records</td>
<td>input metadata enriched by up to three Error Fields for QuickBaseImportCSV (p. 503)</td>
</tr>
</tbody>
</table>
Table 52.5. Error Fields for QuickBaseImportCSV

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>specified in the <strong>Error code output field</strong></td>
<td>integer</td>
<td>error code</td>
</tr>
<tr>
<td>1</td>
<td>specified in the <strong>Error message output field</strong></td>
<td>string</td>
<td>error message</td>
</tr>
<tr>
<td>1</td>
<td>specified in the <strong>Batch number output field</strong></td>
<td>integer</td>
<td>index (starting from 1) of the failed batch</td>
</tr>
</tbody>
</table>

1 The error fields must be placed behind the input fields.

**QuickBaseImportCSV Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickBase connection</td>
<td>✔</td>
<td>ID of the connection to the QuickBase online database, see Chapter 23, QuickBase Connections (p. 184)</td>
<td></td>
</tr>
<tr>
<td>Table ID</td>
<td>✔</td>
<td>ID of the table in the QuickBase application data records are to be written into (see the application_stats for getting the table ID)</td>
<td></td>
</tr>
<tr>
<td>Batch size</td>
<td></td>
<td>The maximum number of records in one batch</td>
<td>100 (default)</td>
</tr>
<tr>
<td>Clist</td>
<td></td>
<td>A period-delimited list of table field_ids 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.</td>
<td></td>
</tr>
<tr>
<td>Error code output field</td>
<td></td>
<td>name of the error metadata field for storing the error code, see Error Fields for QuickBaseImportCSV (p. 503)</td>
<td></td>
</tr>
<tr>
<td>Error message output field</td>
<td></td>
<td>name of the error metadata field for storing the error message, see Error Fields for QuickBaseImportCSV (p. 503)</td>
<td></td>
</tr>
<tr>
<td>Batch number output field</td>
<td></td>
<td>name of the error metadata field for storing the index of the corrupted batch, see Error Fields for QuickBaseImportCSV (p. 503)</td>
<td></td>
</tr>
</tbody>
</table>
QuickBaseRecordWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

QuickBaseRecordWriter writes data into QuickBase online database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBaseRecordWriter</td>
<td>QuickBase</td>
<td>1</td>
<td>0-1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>for input data records</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for rejected data records</td>
<td>input metadata enriched by up to two Error Fields for QuickBaseRecordWriter (p. 505)</td>
</tr>
</tbody>
</table>
Table 52.6. Error Fields for QuickBaseRecordWriter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional^1</td>
<td>specified in the error code output field</td>
<td>integer</td>
<td>long</td>
</tr>
<tr>
<td>optional^1</td>
<td>specified in the error message output field</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

^1 The error fields must be placed behind the input fields.

QuickBaseRecordWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickBase connection</td>
<td></td>
<td>ID of the connection to the QuickBase online database, see Chapter 23, QuickBase Connections (p. 184)</td>
<td></td>
</tr>
<tr>
<td>Table ID</td>
<td></td>
<td>ID of the table in the QuickBase application data records are to be written into (see the application_stats for getting the table ID)</td>
<td></td>
</tr>
<tr>
<td>Mapping</td>
<td></td>
<td>List of database table field_ids separated by a semicolon the metadata field values are to be written to.</td>
<td></td>
</tr>
<tr>
<td>Error code output field</td>
<td></td>
<td>Name of the field the error code will be stored in, see Error Fields for QuickBaseRecordWriter (p. 505)</td>
<td></td>
</tr>
<tr>
<td>Error message output field</td>
<td></td>
<td>Name of the field the error message will be stored in, see Error Fields for QuickBaseRecordWriter (p. 505)</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 52. Writers

SpreadsheetDataWriter

Commercial Component

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 42, Common Properties of Writers (p. 298)

If you want to find the appropriate Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

SpreadsheetDataWriter writes data to spreadsheets - XLS or XLSX files. SpreadsheetDataWriter supersedes the original XLSDataWriter with a lot more new features, write modes and improved performance. (XLSDataWriter is still available for backwards compatibility and in the Community edition)

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpreadsheetDataWriter</td>
<td>XLS(X) file</td>
<td>1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

SpreadsheetDataWriter writes data to XLS or XLSX files. It offers advanced features for creating spreadsheets:

• insert/overwrite/append modes
• powerful visual mapping for complex spreadsheets
  • explicitly defined mapping or dynamic auto-mapping
• form writing
• multiline records
• vertical/horizontal writing
• cell formatting support
• streaming mode for performance and huge data loads
• dynamic file/sheet partitioning
• template support

Supported file formats:

• XLS: only Excel 97/2003 XLS files (BIFF8)
• XLSX: Open Document Format, Microsoft Excel 2007 and newer

Supported outputs:

• local or remote (FTP, HTTP, CloverETL Server sandbox, etc. - see File URL in SpreadsheetDataWriter Attributes (p. 507))
• output port
• console
• dictionary

**Icon**

![SpreadsheetDataWriter Icon]

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Incoming records to be written out to a spreadsheet.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Metadata: Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For port writing. See Writing to Output Port (p. 301).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Metadata: One field (byte, cbyte, string).</td>
</tr>
</tbody>
</table>

**SpreadsheetDataWriter Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Specifies where data will be written to: an XLS or XLSX file, the console, an output port or a dictionary. See Supported File URL Formats for Writers (p. 299).</td>
</tr>
<tr>
<td>Sheet</td>
<td></td>
<td>A name or number (zero-based) of the sheet to write into. Unless set, a sheet with a default name is created and inserted after all existing sheets. You can specify multiple sheets separated by a semicolon &quot;,&quot;. For details on partitioning, see Writing Techniques &amp; Tips for Specific Use Cases (p. 516).</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>A visual editor in which you define how input data is mapped to the output spreadsheet(s). See Advanced Description (p. 509) for more information.</td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External file containing the mapping definition.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Write mode</td>
<td></td>
<td>Determines how data is written to the output spreadsheet. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Overwrite in sheet (in-memory)</strong> - overwrites existing cells if present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Insert into sheet (in-memory)</strong> - inserts new data to the mapped area,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shifting existing cells down if present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Append to sheet (in-memory)</strong> - appends data at the end of an existing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data column/row</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Create new file (streaming - XLSX only)</strong> - replaces existing file with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a newly created one making streaming mode possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Create new file (in-memory)</strong> - replaces existing file and work in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-memory mode</td>
</tr>
<tr>
<td>Actions on existing sheets</td>
<td></td>
<td>Defines what action is performed if the specified Sheet already exists in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the target spreadsheet. The attribute works in accordance with <strong>Write mode</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Available options:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Do nothing, keep existing sheets and data</strong> - default option; no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operation is performed prior to writing; insert/overwrite/append modes apply</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Clear target sheet(s)</strong> - specified Sheet(s) are cleared prior to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>writing; <strong>Write mode</strong> setting is ignored</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Replace all existing sheets</strong> - all sheets are removed prior to writing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to the file; equivalent to Create new file option in <strong>Write mode</strong></td>
</tr>
<tr>
<td>Advanced Tabel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Template File URL</td>
<td></td>
<td>A template spreadsheet file which is duplicated into the output file and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>populated with data according to the defined mapping. The template can be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>any spreadsheet, typically containing the header, footer and data sections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(one empty line to be replicated during writing). If looking for more tips,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see <em>Writing Techniques &amp; Tips for Specific Use Cases</em> (p. 516). It is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>required that formats of the output file and the template file match. Usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of XLTX files is limited (see <em>Notes and Limitations</em> (p. 519), rather than</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XLTX use XLSX files as templates.</td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>If set to <code>true</code>, non-existing directories included in the <strong>File URL</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>path will be automatically created.</td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>Maximum number of records that are written to a single file. See</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Partitioning Output into Different Output Files</em> (p. 306)</td>
</tr>
</tbody>
</table>
Chapter 52. Writers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Total number of records throughout all output files that will be skipped. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Total number of records throughout all output files that will be written out. See Selecting Input Records (p. 294).</td>
<td>0-N</td>
</tr>
<tr>
<td>Partition key</td>
<td></td>
<td>A key whose values control the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table</td>
<td>2)</td>
<td>The ID of a lookup table. The table serves for selecting records which should be written to the output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag</td>
<td></td>
<td>By default, output files are numbered. If this attribute is set to Key file tag, output files are named according to values of Partition key or Partition output fields. See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Partition output fields</td>
<td>2)</td>
<td>Fields of Partition lookup table whose values serve for naming output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition unassigned file name</td>
<td></td>
<td>The name of a file unassigned records should be written into (if there are any). Unless specified, data records whose key values are not contained in Partition lookup table are discarded. See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Type of formatter</td>
<td></td>
<td>Specifies the formatter to be used. By default, the component guesses according to the output file extension - XLS or XLSX.</td>
<td>Auto (default)</td>
</tr>
</tbody>
</table>

Legend:

1) The two mapping attributes are mutually exclusive. You either specify the mapping yourself in Mapping, OR supply it in an external file via Mapping URL. The third option is to leave all mapping blank.

2) Either both or neither of these attributes has to be specified.

Advanced Description

Introduction to Spreadsheet Mapping

A mapping tells the component how to write Clover records to a spreadsheet. Mapping defines where to put metadata information, data, format, writing orientation etc.

In the mapping you define a binding between a Clover field and so called leading cell. Data for that field is written into the spreadsheet beginning at the leading cell position either downwards (vertical orientation; default) or to the right (horizontal).

Each leading cell-field binding is independent of each other. That can be used to create complex mappings (e.g. one record can be mapped to multiple rows; see Rows per record global mapping property)

Each Clover field can be mapped to a spreadsheet cell in one of the following Mapping modes:

- Explicit - statically maps a field to a fixed leading cell of your preference. Typically the most used mapping mode for the writer (see Basic Mapping Example (p. 511)). Explicit mode can be combined with other mapping modes.
Tip

To map a field (or a whole record) explicitly, simply drag the field (record) to the spreadsheet preview area and drop it onto desired location. You can select multiple fields.

- **Map by order** - dynamic mapping mode; cells in 'by order' mode are filled in left-right-top-down direction with input record fields by the order in which the fields appear in the input metadata. Only fields which are not mapped explicitly and not mapped by name are taken into account.

- **Map by name** - this mode applies only to cases when you are writing to an already existing sheet(s). Cells mapped "by name" are bound to input fields using 'late binding' on runtime according to their actual content, which presumably is a 'header'. The component tries to match the cell's content with a field name or label (see Field Name vs. Label vs. Description (p. 156)) from input metadata. If such a match could be found then the mapped cell is bound to the corresponding input field. If there is no match for the cell (i.e. cell's content is not a field name/label) then the cell is unresolved - no input field could be assigned. Note that unresolved cells are not a bad thing - you might be writing into say a group of similar templates, each containing just a subset of fields in the input metadata. Mappings with unresolved cells do not result in the graph failing on execution.

  This mode comes in handy when you are writing using pre-defined templates (the Template file URL attribute). See Writing Techniques & Tips for Specific Use Cases, (p. 516).

Note

Both **Map by order** and **Map by name** modes try to automatically map the contents of the output file to the input metadata. Thus these modes are useful in cases when you write into multiple files and you want to design a single 'one-fits-all' generic mapping, typically for multiple templates. Replacing input metadata with another does not require any change in the mapping - it is recomputed accordingly to the mapping logic.

- **Implicit** - default case when the mapping is blank. The component will assume Write header to true and map all input fields by order, starting in top left hand corner.

Spreadsheet Mapping Editor

Spreadsheet mapping editor is the place where you define your mapping and its properties. The mapping editor previews sheets of the output file (if any; otherwise shows a blank spreadsheet). However, the same mapping is applied to a whole group of output files/sheets (e.g. when partitioning into multiple sheets or files).

To start mapping, fill in the File URL and (optionally) Sheet attributes with the file (and sheet name) to write into, respectively. After that, edit Mapping to open the spreadsheet mapping editor. When you write into a new (empty) spreadsheet, the mapping editor will appear blank like this.
In the editor, you map the input fields on the left hand to the spreadsheet on the right hand. Either use mouse drag’n’drop or the **Map by name, Map by order** buttons to create leading cells in the spreadsheet.

You can see the following parts of the editor:

- **Toolbar** - buttons controlling how you Map Clover fields to spreadsheet data (either **by order**, or **by name**) and global **Data offsets** control (see Advanced Mapping Options (p. 512) for an explanation of data offsets).

- **Sheet preview area** - this is where you create and modify all the mapping of the output file.

- **Input metadata** - Clover fields you can map to spreadsheet cells. This is the metadata assigned to the input edge. (You cannot edit it.)

- **Properties** - controls properties of mapped cells and **Global** mapping attributes; can be applied to a single or a group of cells at a time

- **Summary** tab - a place where you can neatly review the Clover-to-spreadsheet mapping you have made.

### Colours in spreadsheet mapping editor

Cells in the preview area highlighted in various colours to identify whether and how they are mapped.

- Orange are the **leading cells** and they form the header. Properties can be adjusted on each orange cell to create complex mappings; see Advanced Mapping Options (p. 512).

- Cells in dashed border, which appear only when a leading cell is selected, indicate the data area.

- Yellow cells demonstrate the first record which will be written.

### Basic Mapping Example

A typical example of what you will want to do in **SpreadsheetDataWriter** is writing into an empty spreadsheet. This section describes how to do that in a few easy steps.

- Open **Spreadsheet Mapping Editor** by editing the **Mapping** attribute.

- Click the whole record in **Input metadata** (*excel_types_nocurrency* in the example below) and drag it to the spreadsheet preview area to cell A1 and drop. You will see that for each field of the input record...
a leading cell is created, producing a default explicit mapping (explained in Introduction to Spreadsheet Mapping (p. 509)). See Figure 52.17, Explicit mapping of the whole record (p. 512)

- In Properties (bottom left hand corner), make sure Write header is set to true. This writes field names (labels actually) to leading cells first, followed by actual data; use this whenever you want to output a header.

- Furthermore in Properties, notice that Orientation is Vertical. This makes the component produce output by rows (opposite to Horizontal orientation, where writing advances by columns).

- Notice that Data offsets (global) is set to 1. That means that data will be written 1 row below the leading cell, making room for the header cell.

  **Note**

  Actually, you will achieve the same result if you leave the mapping blank (implicit mapping). In that case the first row is mapped by order.

![Spreadsheet Mapping Editor](image)

**Figure 52.17. Explicit mapping of the whole record**

**Advanced Mapping Options**

This section provides an explanation of some more advanced concepts building on top of the Basic Mapping Example (p. 511)

- **Data offsets** - determines the position where data is written relative to the leading cell position.

  Basically, its value represents 'a number of rows (in vertical mode) or columns (in horizontal mode) to be skipped before the first record is written (relative to the leading cell).'

  Data offset 0 does not skip anything and data is written right at the leading cell position (Write header option does not work for this setting).

  Data offset 1 is typically used when header is to be written at the leading cell position - so you need to shift the actual data by one row down (or column to the right).
Click arrow buttons in the Data offsets (global) control to adjust data offsets for the whole spreadsheet.

Additionally, you can use the spinner in Properties → Selected cells → Data offset of each leading cell (orange) to adjust data offset locally, i.e. for a particular column only. Notice how modifying data offset is visualised in the sheet preview - the 'omitted' rows change colour. By following dashed cells, which appear when you click a leading cell, you can quickly check where your record will be written.

**Tip**

The arrow buttons in Data offsets (global) only shift the data offset property of each cell either up or down. So mixed offsets are retained, just shifted as desired. To set all data offsets to a single value, enter the value into the number field of Data offsets (global). Note that if there are some mixed offsets, the value is displayed in gray.

![Figure 52.18. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, writing would start at row 4 with no data written to rows 2 and 3.](image)

![Figure 52.19. Global data offsets is set to 1. In the last column, it is locally changed to 4. In the output file, the initial rows of this column would be blank, data would start at D5.](image)

• **Rows per record** - a Global property specifying a gap between rows. Default value is 1 (i.e. there is no gap). Useful when mapping multiple cells above each other (for a single record) or when you need to print blank rows in between your data. Best imagined if you look at the figure below:
With Rows per record set to 2 in leading cells Name and Address, the component always writes one data row, skips one and then writes again. This way various data does not get mixed (overwritten by the other one). For a successful output, make sure Data offsets is set to 2.

- Combination of Data offsets (global and local) and Rows per record - you can put the settings described in preceding bullet points together. See example:

Figure 52.21. Rows per record is set to 3. Data in the first and third column will start in their first row (because of their data offsets being 1). The second and fourth columns have data offsets 2 and 4, respectively. The output will, thus, be formed by ‘zig-zagged’ cells (the dashed ones - follow them to make sure you understand this concept clearly).

- Max number of records - a Global property which you can specify via component attributes, too (see SpreadsheetDataWriter Attributes (p. 507)). If you reduce it, you will notice the number of dashed cells in the spreadsheet preview reduces as well (highlighting only the cells which will be written out in fact).

- Formatting cells (Format Field) - in a spreadsheet, every single cell can have its own format (in Excel, right-click on a cell -> Format cells; Number tab). This format is represented by a format string (not Clover format string, but Excel-specific format string). Since format in Clover is defined globally for a field in metadata, not per record, writing formats to Excel can be tricky. SpreadsheetDataWriter offers two ways of writing Excel-specific format to cells:

  Case 1:

  You can specify the format for a metadata field (its Format property in metadata). That means all values of the field written to the sheet will have the specified format. You need to prefix the Format in metadata with excel: (e.g. excel:0.000% for percents with three decimals) because the component ignores standard format strings (as the Clover-to-Excel format conversion is not possible).

  Case 2:

  You provide two input fields for a single cell: one specifying the cell value and the other defining its format.
Chapter 52. Writers

Note

- This unleashes the full power of Excel where formats are set per-cell rather than per-column.
- You pass the format in the data as an extra 'string' value.
- Remember, the format is specified in Excel terms, not Clover.

* Use **Format field** in **Properties → Selected cells** of the leading (orange) cell to specify the input field containing the format (string).

Which format is used if both are set?

- Do you have the format mapped by the **Format field** property? Yes - the component uses it.
- Is **Format field** not specified OR a value of that particular field is empty (null or empty string)? Yes - use **Format** from the metadata field (if set with *excel:* prefix). See also **Field Details** (p. 158).

You can use **excel:General** format - either in **Format field** or in metadata **Format** - the output will be set to general format (Excel terms).

**Example 52.9. Writing Excel format**

Let us have two fields: **fieldValue** (integer) and **fieldFormat** (string) mapped onto cell A1 (one as value, the other as **Format field**). Imagine these incoming records:

- (100, "#00,0")
  - writes value 100 and format "#00,0" into cell A1
- (100, "General")
  - writes value 100 into cell A1 and sets its format to General
- (100, "") or (100, null)
  - writes value 100 into cell A1 and since **fieldFormat** is empty it looks into the **Format** metadata attribute of **fieldValue** (NOT **fieldFormat**):
    1. if there is no format, uses General
    2. if there is the “excel:XYZ” format string, applies format XYZ to the cell
    3. if there is another format (anything not prefixed by *excel:*), uses General (Clover-to-Excel format conversion is not performed)

Note

When Excel format is specified in **Metadata → Format** it MUST be prefixed by *excel:* so that Clover can know that the format string is specific to Excel-only use. Example: "excel:0.000%"

When Excel format is passed in **data**, as the aforementioned **fieldFormat**, it MUST NOT be prefixed in any way. Example: "0.000%"

Note that the *excel:* format string matters when reading the output back with spreadsheet readers - **SpreadsheetDataReader** or **XLSDataReader**. Common readers (such as **UniversalDataReader**) completely ignore *excel:* . They consider it an empty format string.
Writing Techniques & Tips for Specific Use Cases

- **Writing using template**

Sometimes you may want to prepare in advance a nicely formatted template in Excel, maybe including some static headers, footer, etc. and use Clover to just fill in the data for you. And it might be that you will want to reuse the template without overwriting it.

This is where `SpreadsheetDataWriter` template feature comes in handy. The component can take a previously designed template Excel file (see Template File URL in `SpreadsheetDataWriter Attributes` (p. 507)), make a copy of it into the designated output file (see File URL) and write data to it, retaining the rest of the template.

A template can be any Excel file, usually containing three sections: the header, one template row for data and the rest as the footer.

![Figure 52.22. Writing into a template. Its original content will not be affected, your data will be written into Name, Surname and Age fields.](image)

Notice the template row. It is a row like any other but in the mapping editor, it is designated as the first row of mapped data. The component duplicates that row each time it writes a new data. This way you can assign arbitrary formatting, colors etc. on this data row and it is applied to all written rows.

The template file is not changed or affected in any other way.

**Important**

There is only one reasonable setting when using templates, although all other modes work as expected (they do not, however, produce results that you would want). The settings are:

- **Sheet** - select the sheet from the template (by number or name, do not create new sheet)
- **Mapping** - this is one of the cases where Map by name makes sense. Use the header of the template where applicable. Of course, you can map fields as usual.
- **Write mode** - Insert
- **Actions on existing sheets** - Do nothing, keep existing sheets and data
- **Filling forms** - you can use the component to write into forms without affecting its original boxes. Use these settings:

  Send just one input record to the component's input containing all the form values. Set File URL to the form file to be filled. Then map the input fields explicitly one by one into corresponding form cells using the preview sheet.

Next, use these settings:
• **Write header** - false

• **Data offsets (global)** - 0 (this ensures data will be written right into the leading cells you have mapped - the orange ones)

• **Charts and formulas** - if you use Insert, Append or Overwrite modes, formulas and charts that work with the data areas written in Clover will be properly updated when viewed in Excel.

**Note**
Generating formulas, charts or other Excel objects is not currently supported.

• **Multiple write passes into one sheet**

You can use multiple sequential writes into a single sheet to produce complex patterns. To do so, set up multiple SpreadsheetDataWriter components writing the the same file/sheet and feed them various inputs.

**Important**

Do not forget to put multiple components writing to the same file into different phases. Otherwise the graph will fail on write conflict.

Typically, you will use the [Overwrite in sheet (in-memory) write mode](#) for all components in the sequence.

• **Partitioning** - a neat technique is partitioning into individual sheets according to values of a specified key field (or more fields). Thus you can e.g. write data for different countries into different sheets. Simply choose Country as the partitioning key. This is done by editing the Sheet attribute; switch to [Partition data into sheets by data fields](#) and select a field (or more fields using **Ctrl+click** or **Shift+click**).

![Figure 52.23. Partitioning by one data field](#)

You can partition according to more than one field. In that case, output sheet names will be a compound of field names you have selected. **Example:** You have customer orders stored in one CSV file. You would like to separate them into sheets according to e.g. a name of the shop and a city. Use SpreadsheetDataWriter in create new file mode while partitioning according to the two fields. It will produce sheets like:

- Pete's Grocery, New York
- Hoboken Deli, New Jersey
- Al's Hardware, New York

etc., each of them containing data just for one shop.

See also [Partitioning Output into Different Output Files](#) (p. 306).

• **Writing huge files**
Although Excel format is not primarily designed for big data loads, its processing can easily grow to enormous memory requirements.

The format itself has some limitations:

- Excel 97/2003 - XLS
  - Maximum of 65,535 rows and 256 columns per sheet
  - Maximum number of sheets - 255
- Excel 2007 and newer - XLSX
  - Maximum number of rows: unlimited (but be aware that Excel itself works only with first 1,048,576 rows the file contains). All the data and be read back by SpreadsheetDataReader or other tools that support large files.
  - Maximum number of columns: 16,384
  - Maximum number of sheets: unlimited (as long as you have memory)

Tip

Working with larger spreadsheets is memory consuming and although the component does its best to optimize its memory footprint, bear these few tips in mind:

- When mapping in the Spreadsheet mapping editor, memory consumption for the Designer might temporarily ramp up over a gigabyte of memory - so be sure to set enough heap space for the Designer itself (see Program and VM Arguments (p. 82)).
- Memory consumption is affected by how Excel organizes the file internally so two files with the same amount of data in it can have significantly different memory requirements.
- Use streaming mode whenever possible. Switch to DEBUG mode in graph's Run Configurations to detect whether streaming mode is on or off. To learn how to do that, see Program and VM Arguments (p. 82).

Usually you would use the Create new file (streaming - XLSX only) write mode. Other write modes do not support streaming.

- Reviewing your mapping

In complex mappings with many metadata fields, you might want to check if everything has been mapped properly. Whenever during your work in Spreadsheet Mapping Editor, switch to the Summary tab and observe an overview of leading cells and mappings like this one:
Chapter 52. Writers

Figure 52.24. Mapping summary

Notes and Limitations

- **Encryption** - writing of encrypted XLS or XLSX files is not supported (unlike SpreadsheetDataReader (p. 387) which can read encrypted files)

- **XLTX vs. XLSX templates** - for technical reasons it is currently not possible to use an XLTX template for XLSX output. Nevertheless, the difference between XLTX and XLSX files is minimal. Therefore, we recommend you use XLSX as the format for both the template and output files. For XLS and XLT files, there is no such limitation.

- **Mapping editor on server files** - a spreadsheet mapping editor on server files can operate as usual, except for a case when File URL contains wildcard characters. In that case CloverETL Designer is not able to find matching server files and the mapping editor shows no data in the spreadsheet preview. This is going to be fixed in next releases.

- **Error reporting** - there is no error port on the component. By design, either the component configuration is valid and will then succeed in writing records to a file, or it will fail with a fatal error (invalid configuration, no space left on device, etc.). No errors per input record are generated.
StructuredDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

StructuredDataWriter writes data to files with user-defined structure.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>StructuredDataWriter</td>
<td>structured flat file</td>
<td>1-3</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Records for body</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Records for header</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>no</td>
<td>Records for footer</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For port writing. See Writing to Output Port (p. 301).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
</tbody>
</table>
## StructuredDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying where received data will be written (flat file, console, output port, dictionary). See Supported File URL Formats for Writers (p. 299).</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Char set</td>
<td></td>
<td>Encoding of records written to the output.</td>
<td></td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>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).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Body mask</td>
<td></td>
<td>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. 522) for more information about definition of Body mask and resulting output structure.</td>
<td>Default Body Structure (p. 523) (default)</td>
</tr>
<tr>
<td>Header mask</td>
<td>1)</td>
<td>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. 522) for more information about definition of Header mask and resulting output structure.</td>
<td>empty (default)</td>
</tr>
<tr>
<td>Footer mask</td>
<td>2)</td>
<td>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. 522) for more information about definition of Footer mask and resulting output structure.</td>
<td>empty (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>By default, non-existing directories are not created. If set to true, they are created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>Maximum number of records to be written to one output file.</td>
<td>1-N</td>
</tr>
<tr>
<td>Bytes per file</td>
<td></td>
<td>Maximum size of one output file in bytes.</td>
<td>1-N</td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be written to all output files. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Partition key</td>
<td></td>
<td>Key whose values define the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table</td>
<td>1)</td>
<td>ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag</td>
<td></td>
<td>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. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Partition output fields</td>
<td>1)</td>
<td>Fields of Partition lookup table whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 52. Writers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition unassigned file name</td>
<td>Req</td>
<td>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. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

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 corresponding 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.

![Create Mask Dialog](image)

**Figure 52.25. 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=...`
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:

   ```xml
   < 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right **Writer** for your purposes, see Writers Comparison (p. 299).

**Short Summary**

**Trash** discards data.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trash</td>
<td>none</td>
<td>1–n</td>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**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**

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.

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1-n</td>
<td>yes</td>
<td>For received data records</td>
<td>Any</td>
</tr>
</tbody>
</table>

**Trash Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug print</td>
<td></td>
<td>By default, all records are discarded. If set to <code>true</code>, all records</td>
<td>false (default)</td>
</tr>
</tbody>
</table>
## Chapter 52. Writers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug file URL</td>
<td></td>
<td>Attribute specifying debug output file. See <a href="#">Supported File URL Formats for Writers</a> (p. 299). If path is not specified, the file is saved to the <code>{PROJECT}</code> directory. You do not need to switch Log level from its default value (INFO).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Debug append</td>
<td></td>
<td>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).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of debug output.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Advanced</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print trash ID</td>
<td></td>
<td>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).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>By default, non-existing directories are not created. If set to true, they are created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td>Trash can run in either Performance or Validate records modes. In Performance mode the raw data is discarded, in Validate records Trash simulates a writer - attempting to deserialize the inputs.</td>
<td>Performance (default)</td>
</tr>
</tbody>
</table>
UniversalDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

UniversalDataWriter is a terminative component that writes data to flat files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniversalDataWriter</td>
<td>flat file</td>
<td>1</td>
<td>0-1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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 locally 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✔️</td>
<td>for received data records</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for port writing. See Writing to Output Port (p. 301).</td>
<td>include specific byte/ cbyte/ string field</td>
</tr>
</tbody>
</table>
## UniversalDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>✓</td>
<td>where the received data to be written (flat file, console, output port, dictionary) specified, see [Supported File URL Formats for Writers](p. 299).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>character encoding of records written to the output</td>
<td>ISO-8859-1 (default) &lt;other encodings&gt;</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Quoted strings</td>
<td></td>
<td>When switched to true, all field values (except from byte and cbyte) will be quoted. If you do not set this attribute, its value is inherited from metadata on the input port (and displayed in faded grey text, see also [Record Details](p. 157)).</td>
<td>false</td>
</tr>
<tr>
<td>Quote character</td>
<td></td>
<td>Specifies which kind of quotes will enclose output fields. Applies if Quoted strings is true only. By default, the value of this attribute is inherited from metadata on input port. See also [Record Details](p. 157).</td>
<td>&quot;</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>if set to true, non-existing directories in the File URL attribute path are created</td>
<td>false (default)</td>
</tr>
<tr>
<td>Write field names</td>
<td></td>
<td>Field labels are not written to the output file(s) by default. If set to true, labels of individual fields are printed to the output. Please note field labels differ from field names: labels can be duplicate and you can use any character in them (e.g. accents, diacritics). See [Record Pane](p. 155).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>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. 299)</td>
<td>1 - N</td>
</tr>
<tr>
<td>Bytes per file</td>
<td></td>
<td>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. 299) To avoid splitting a record into two files, max size can be slightly overreached.</td>
<td>1 - N</td>
</tr>
<tr>
<td>Number of skipped</td>
<td></td>
<td>how many records/rows to be skipped before writting the first record to the output file, see [Selecting Output Records](p. 306).</td>
<td>0 (default) - N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>how many records/rows to be written to all output files, see [Selecting Output Records](p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Exclude fields</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Partition key</td>
<td>2)</td>
<td>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. 306)</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table</td>
<td>1)</td>
<td>ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag</td>
<td>2)</td>
<td>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. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Partition output fields</td>
<td>1)</td>
<td>Fields of Partition lookup table whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition unassigned file name</td>
<td></td>
<td>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. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

| Notes:                          |     | Either both or neither of these attributes must be specified.                                                                                |                 |
| Notes 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. 85). 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. 585) component that can split large fields into several records.
Chapter 52. Writers

XLSDataWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 42, Common Properties of Writers (p. 298)

If you want to find the right Writer for your purposes, see Writers Comparison (p. 299).

Short Summary

XLSDataWriter writes data to XLS or XLSX files.

Important

Since Clover 3.3., there are new powerful components available for spreadsheet reading/writing - SpreadsheetDataReader (p. 387) and SpreadsheetDataWriter (p. 506). The preceding XLS components (XLSDataReader (p. 402) and XLSDataWriter (p. 529) have remained compatible, though.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLSDataWriter</td>
<td>XLS(X) file</td>
<td>1</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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 has high memory requirements and may store data in the memory (see the Disable temporary files (inMemory mode) attribute). When working with XLSX files, all data are stored in the memory.

Icon

Ports
Chapter 52. Writers

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For received data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For port writing. See Writing to Output Port (p. 301).</td>
<td>One field (byte, cbyte, string).</td>
</tr>
</tbody>
</table>

XLSDataWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of parser</td>
<td></td>
<td>Specifies the formatter to be used. By default, component guesses according the extension (XLS or XLSX).</td>
<td>Auto (default)</td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Attribute specifying where received data will be written (XLS or XLSX file, console, output port, dictionary). See Supported File URL Formats for Writers (p. 299).</td>
<td></td>
</tr>
<tr>
<td>Sheet name</td>
<td>1)</td>
<td>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.</td>
<td>Sheet[0-9]+ (The number in the name is the number of the last sheet with the same name structure: Sheet[0-9]+.)</td>
</tr>
<tr>
<td>Sheet number</td>
<td>1)</td>
<td>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.</td>
<td>0-N</td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of records written to the output.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Append to the sheet</td>
<td></td>
<td>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).</td>
<td>false (default)</td>
</tr>
<tr>
<td>Metadata row</td>
<td></td>
<td>Number of the row to which the field names should be written. By default, field names are written to the header of the sheet.</td>
<td>0 (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>By default, non-existing directories are not created. If set to true, they are created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Disable temporary files (inMemory mode)</td>
<td></td>
<td>Temporary files, which are created during the writing, are stored on the disk by default. If you set this attribute to true, you will force storing those files in the memory. Note: it can be applied to xls files only.</td>
<td>false</td>
</tr>
<tr>
<td>Start row</td>
<td></td>
<td>Row of the sheet starting from which the records are written. By default, records are written to the sheet starting from the first row.</td>
<td>1 (default)</td>
</tr>
<tr>
<td>Start column</td>
<td></td>
<td>Column of the sheet starting from which the records are written. By default, records are written to the sheet starting from the first column.</td>
<td>A (default)</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>Maximum number of records to be written to one output file.</td>
<td>1-N</td>
</tr>
<tr>
<td>Number of skipped records</td>
<td></td>
<td>Number of records to be skipped. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records to be written to all output files. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
<tr>
<td>Exclude fields</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Partition key</td>
<td></td>
<td>Key whose values define the distribution of records among multiple output files. See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table</td>
<td>2)</td>
<td>ID of lookup table serving for selecting records that should be written to output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag</td>
<td></td>
<td>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. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Partition output fields</td>
<td>2)</td>
<td>Fields of Partition lookup table whose values serve to name output file(s). See Partitioning Output into Different Output Files (p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition unassigned file name</td>
<td></td>
<td>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. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

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!
We assume you have already learned what is described in:

- Chapter 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)
- Chapter 42, *Common Properties of Writers* (p. 298)

If you want to find the appropriate Writer for your purpose, see *Writers Comparison* (p. 299).

### Short Summary

XMLWriter formats records into XML files.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLWriter</td>
<td>XML file</td>
<td>1-n</td>
<td>0-1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

### 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

![XMLWriter Icon](image)

### Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Port number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-N</td>
<td>At least one</td>
<td>Input records to be joined and mapped into an XML file</td>
<td>Any (each port can have different metadata)</td>
</tr>
</tbody>
</table>
### XMLWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>The target file for the output XML. See Supported File URL Formats for Writers (p. 299).</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>The encoding of an output file generated by XMLWriter.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Mapping</td>
<td>1)</td>
<td>Defines how input data is mapped onto an output XML. See Advanced Description (p. 534) for more information.</td>
<td></td>
</tr>
<tr>
<td>Mapping URL</td>
<td>1)</td>
<td>External text file containing the mapping definition. See Creating the Mapping - Mapping Ports and Fields (p. 542) and Creating the Mapping - Source Tab (p. 546) for the mapping file format. Put your mapping to an external file if you want to share a single mapping among multiple graphs.</td>
<td></td>
</tr>
<tr>
<td>XML Schema</td>
<td></td>
<td>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. 545). The schema has to be placed in the meta folder.</td>
<td>none (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create directories</td>
<td></td>
<td>If true, non-existing directories included in the File URL path will be automatically created.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Omit new lines wherever possible</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Cache size</td>
<td></td>
<td>A size 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 the cache is needed to maintain fast processing.</td>
<td>default: auto</td>
</tr>
<tr>
<td>Sorted input</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Sort keys</td>
<td></td>
<td>Tells XMLWriter how the input data is sorted, thus enabling streaming (see Creating the Mapping - Mapping Ports and Fields (p. 542)). 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. 265).</td>
<td></td>
</tr>
<tr>
<td>Records per file</td>
<td></td>
<td>Maximum number of records that are written to a single file. See Partitioning Output into Different Output Files (p. 306).</td>
<td>1-N</td>
</tr>
<tr>
<td>Max number of records</td>
<td></td>
<td>Maximum number of records written to all output files. See Selecting Output Records (p. 306).</td>
<td>0-N</td>
</tr>
</tbody>
</table>
### Attribute Table

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition key</td>
<td></td>
<td>A key whose values control the distribution of records among multiple output files. See [Partitioning Output into Different Output Files](p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition lookup table</td>
<td></td>
<td>The ID of a lookup table. The table serves for selecting records which should be written to the output file(s). See [Partitioning Output into Different Output Files](p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition file tag</td>
<td></td>
<td>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. 306) for more information.</td>
<td>Number file tag (default)</td>
</tr>
<tr>
<td>Partition output fields</td>
<td></td>
<td>Fields of Partition lookup table whose values serve for naming output file(s). See [Partitioning Output into Different Output Files](p. 306) for more information.</td>
<td></td>
</tr>
<tr>
<td>Partition unassigned file name</td>
<td></td>
<td>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. 306) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

**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 52.26, “Mapping Editor” (p. 535)). 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 52.34, *Source tab in Mapping editor*, (p. 546)).

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.
Chapter 52. Writers

Figure 52.26. 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. 545).
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. 542)).

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 52.27. Adding Child to Root Element.](image)

**Important**

For a closer look on adding nodes, manipulating them and using smart drag and drop mouse techniques, see Working with Nodes (p. 541).

**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 52.10. 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.
Chapter 52. Writers

- **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 52.11. 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.

![XML Writer](image)

*Figure 52.28. 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 52.12. 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 52.29. 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 52.30. 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. 542)). 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 52.13. 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 52.14. 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 52.15. Hide Element

Imagine an example XML:

```xml
<address>
  <city>Atlanta</city>
  <state>Georgia</state>
</address>
<address>
  <city>Los Angeles</city>
  <state>California</state>
</address>
```

Then hiding the `address` element produces:

```xml
<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. 306).

Example 52.16. 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

```xml
<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 (p. 536). 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.
Chapter 52. Writers

- **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 52.17. 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 52.31. Mapping editor toolbar.*

The toolbar icons are active depending on the selected node in the tree. Actions you can do comprise:
Chapter 52. Writers

- **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. 535)) 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. 542)

- 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 $\text{portNumber.fieldName}$.

- 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 $\text{portNumber.*}$

- 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=$\text{portNumber.*}$), while the field becomes a subelement whose content is $\text{portNumber.fieldName}$

- 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).

- drag a port/field and drop it onto a property such as Include or Exclude (or any other excluding Input in **Binding**). That can be done either in the **Content** or **Property** panes - as a result, the property receives the value of the port/field. Multiselecting fields and dragging them works, too. Moreover, if you hold down Ctrl while dragging, the port/field value will be added at the end of the property (not replacing it). Say your Include property currently contains e.g. $0.*$. Dragging field1 of port $1$ and dropping it onto Include while holding Ctrl will produce this content: $0.*;\text{field1}$.

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. 535), 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 attribute 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 consequence, a new **Binding** pseudo-element is created, see picture below.

**Note**

Binding an input port to the root element has some limitations. The root can only be bound in this way:

- You have to make sure there will only be one record coming to the input port. Then there is no need to specify partitioning (a warning message will be displayed, though).
- If more than one record is coming to the input port, partitioning has to be specified. Otherwise XMLWriter will generate an invalid XML file (with multiple root elements).

![Figure 52.32. Binding of Port and Element.](image)
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 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. 573) 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 52.18. 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:

```xml
<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>
```

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.

Multivalue Fields

As of Clover v. 3.3, XMLWriter supports mutlivalue fields in metadata. That includes mapping lists and maps to the output XML. For more information, see Multivalue Fields (p. 163) and Data Types in CTL2 (p. 852).

The only thing to mind in XMLWriter is how lists vs. maps look in the output file. A map is written to a single tag (in between the curly { } brackets) while a list is separated to n tags where n is the list's element count. Example:

```xml
<canadianMap>{ot=Ontario, bc=British_Colubmia, at=Alberta, nt=Northern_Territory}</canadianMap> <!--  map with four key-value pairs -->
<valueList>-65.25</valueList> <!-- a three-element list -->
<valueList>-35.02</valueList>
```

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. 542).
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.

![Generating XML from XSD root element](image)

**Figure 52.33. 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**

![Source tab in Mapping editor](image)

**Figure 52.34. Source tab in Mapping editor.**

Here you are the same code as in the figure above for your own experiments:
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.

```
<myNode>
  <movie>
    <name>La La Land</name>
  </movie>
</myNode>
```

**Figure 52.35. Content Assist inside element.**

The inserted code corresponds to nodes and their properties as described in Creating the Mapping - Designing New XML Structure (p. 535)

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. 535)), Insert template or Declare template - see below.

Example 52.19. 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 52.36. 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):

```xml
<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:

```xml
<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:

```xml
<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:

```cloverxml
<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:

```cloverxml
<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 53. Transformers

We assume that you already know what components are. See Chapter 18, Components (p. 94) 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 54, Joiners (p. 624). 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

We can distinguish Transformers according to what they can do.

- One Transformer only copies each input data to all connected outputs.
  - SimpleCopy (p. 618) copies each input data record to all connected output ports.

- One Transformer passes only some input records to the output.
  - DataSampler (p. 559) passes some input records to the output based on one of the selected filtering strategies.

- One Transformer removes duplicate data records.
  - Dedup (p. 561) 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. 572) 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.

- Other Transformer sort data each in different way:
  - ExtSort (p. 574) sorts input data.
  - FastSort (p. 576) sorts input data faster than ExtSort.
  - SortWithinGroups (p. 620) sorts input data within groups of sorted data.

- One Transformer is able to aggregate information about data:
  - Aggregate (p. 552) aggregates information about input data records.

- One Transformer distributes input records among connected output ports:
  - Partition (p. 592) 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. 556) 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. 555) 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. 619) 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. 580) 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. 563) creates single output data record from a group of input data records.

• **Pivot** (p. 599) is a simple form of Denormalizer which creates a pivot table, summarizing input records.

• **Normalizer** (p. 585) creates one or more output data record(s) from a single input data record.

• **MetaPivot** (p. 582) works similarly to Normalizer, but it always performs the same transformation and the output metadata is fixed to data types.

• **Reformat** (p. 603) 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. 606) 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. 559) 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. 622) transforms input data using stylesheets.
Aggregate

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Aggregate computes statistical information about input data records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>n</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For statistical information</td>
<td>Any2</td>
</tr>
</tbody>
</table>
## Aggregate Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregation mapping</td>
<td>Req</td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td>Aggregate key</td>
<td>Req</td>
<td>Key according to which the records are grouped. See Group Key (p. 264) for more information.</td>
<td></td>
</tr>
<tr>
<td>Charset</td>
<td></td>
<td>Encoding of incoming data records.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Sorted input</td>
<td></td>
<td>By default, input data records are supposed to be sorted according to Aggregate key. If they are not sorted as specified, switch this value to false.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values are considered to be different. If set to true, records with null values are considered to be equal.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Old aggregation mapping</td>
<td></td>
<td>Mapping that was used in older versions of CloverETL, its use is deprecated now.</td>
<td></td>
</tr>
</tbody>
</table>

## 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: $outputField=$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 53.1. 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.
Chapter 53. Transformers

Concatenate

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Concatenate gathers data records from multiple inputs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate</td>
<td>yes</td>
<td>no</td>
<td>1-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For input data records</td>
<td>Input 0(1)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For gathered data records</td>
<td>Input 0(1)</td>
</tr>
</tbody>
</table>

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

DataIntersection intrersects data from two inputs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataIntersection</td>
<td>no</td>
<td>yes</td>
<td>2</td>
<td>3</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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

![DataIntersection Icon]
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records (data flow A)</td>
<td>Any(In0)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>For input data records (data flow B)</td>
<td>Any(In1)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For not-changed output data records (contained in flow A only)</td>
<td>Input 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>For changed output data records (contained in both input flows)</td>
<td>Any(Out1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>yes</td>
<td>For not-changed output data records (contained in flow B only)</td>
<td>Input 1</td>
</tr>
</tbody>
</table>

### Legend:

1): Part of them must be equivalent and comparable (Join key).

2): Metadata cannot be propagated through this component.

## DataIntersection Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>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. 558) for more information. Records should be sorted in ascending order to get reasonable results.</td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Definition of the way how records should be intersected written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the way how records should be intersected written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be intersected.</td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allow key duplicates</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
</tbody>
</table>
### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave override key</td>
<td></td>
<td>Older form of <strong>Join key</strong>. Contains fields from the second input port only. This attribute is deprecated now and we suggest you use the current form of the <strong>Join key</strong> attribute.</td>
<td></td>
</tr>
</tbody>
</table>

### 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. 558) or **Java Interfaces for DataIntersection** (p. 558) for more information.

See also **Defining Transformations** (p. 268) 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 53.2. 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 IX, **CTL - CloverETL Transformation Language** (p. 771). (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 template as **Reformat** and **Joiners**. See **CTL Templates for Joiners** (p. 314) for more information.

### Java Interfaces for DataIntersection

**DataIntersection** implements the same interface as **Reformat** and **Joiners**. See **Java Interfaces for Joiners** (p. 317) for more information.
DataSampler

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the appropriate Transformer for your purpose, see Transformers Comparison (p. 309).

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSampler</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-N</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>Yes</td>
<td>For sampled data records</td>
<td>Input0</td>
</tr>
</tbody>
</table>
DataSampler Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampling method</td>
<td>yes</td>
<td>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. 560)</td>
<td>Simple</td>
</tr>
<tr>
<td>Required sample size</td>
<td>yes</td>
<td>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.</td>
<td>(0; 1)</td>
</tr>
<tr>
<td>Sampling key</td>
<td>1)</td>
<td>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).</td>
<td>e.g. Surname(a); FirstName(i); Salary(d)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random seed</td>
<td></td>
<td>A long number that is used in the random generator. It assures that results are random but remain identical on every graph run.</td>
<td>&lt;0; N&gt;</td>
</tr>
</tbody>
</table>

**Legend:**
1) The attribute is required in all sampling methods except for Simple.

**Advanced Description**

A typical use case for DataSampler 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Dedup removes duplicate records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs 1)</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedup</td>
<td>-</td>
<td>✔</td>
<td>1</td>
<td>0-1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✔</td>
<td>for input data records</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✔</td>
<td>for deduplicated data records</td>
<td>equal input</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>for duplicate data records</td>
<td>metadata 1)</td>
</tr>
</tbody>
</table>

1) Metadata can be propagated through this component.
## Dedup Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dedup key</td>
<td></td>
<td>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. 562).</td>
<td></td>
</tr>
<tr>
<td>Keep</td>
<td></td>
<td>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.</td>
<td>First (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of key fields are considered to be equal. If false, they are considered to be different.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Number of duplicates</td>
<td></td>
<td>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.</td>
<td>1 (default)</td>
</tr>
</tbody>
</table>

### 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

**Short Summary**

Denormalizer creates single output record from one or more input records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denormalizer</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✓</td>
<td>for input data records</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✓</td>
<td>for denormalized data records</td>
<td>any</td>
</tr>
</tbody>
</table>

**Denormalizer Attributes**
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>1)</td>
<td>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. 564) for more information.</td>
<td></td>
</tr>
<tr>
<td>Group size</td>
<td>1)</td>
<td>Group may be defined by exact number of its members. E.g. each five records form a single group. The input record count MUST be a multiple of group size. This is mutually exclusive with key attribute.</td>
<td>a number</td>
</tr>
<tr>
<td>Denormalize</td>
<td>2)</td>
<td>Definition of how to denormalize records, written in the graph in CTL.</td>
<td></td>
</tr>
<tr>
<td>Denormalize URL</td>
<td>2)</td>
<td>Name of external file, including path, containing the definition of how to denormalize records, written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Denormalize class</td>
<td>2)</td>
<td>Definition of how to denormalize records, written in the graph in Java.</td>
<td></td>
</tr>
<tr>
<td>Sort order</td>
<td></td>
<td>Order in which groups of input records are expected to be sorted. See Sort order (p. 564)</td>
<td>Auto (default) Ascending Descending Ignore</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of key fields are considered to be equal. If false, they are considered to be different.</td>
<td>true (default) false</td>
</tr>
<tr>
<td>Denormalize source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
</tbody>
</table>

1) Either key or group size must be specified, group size having higher priority.
2) One of them must be specified. Any of these transformation attributes uses the CTL template for Denormalizer or implements a RecordDenormalize interface.

See CTL Scripting Specifics (p. 565) or Java Interfaces for Denormalizer (p. 571) for more information.

See also Defining Transformations (p. 268) 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 53.3. 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 IX, [CTL - CloverETL Transformation Language](p. 771). (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 53.1. Source Tab of the Transform Editor in the Denormalizer Component (I)

Figure 53.2. Source Tab of the Transform Editor in the Denormalizer Component (II)

Table 53.1. Functions in Denormalizer

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean init()</td>
<td>No</td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
</tbody>
</table>
## CTL Template Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invocation</strong></td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>`true</td>
</tr>
</tbody>
</table>

### integer append()

*Required: yes*

*Input Parameters: none*

*Returns: Integer numbers. See [Return Values of Transformations](p. 272) 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:*

```plaintext
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. 272) 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()`:
```java
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()`:
```java
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 numbers are ignored, meaning of 0 and negative values is described in Return Values of Transformations (p. 272)

*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`
```java
function integer appendOnError(
    string errorMessage,
    string stackTrace) {
    printErr(errorMessage);
    return CustomersInGroup;
}
```

**integer transformOnError(Exception exception, stackTrace)**

*Required:* no
### CTL Template Functions

<table>
<thead>
<tr>
<th>Input Parameters</th>
<th>string errorMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>string stackTrace</td>
</tr>
</tbody>
</table>

| Returns                | Integer numbers. See Return Values of Transformations (p. 272) 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

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Called during each graph run after the entire transform was executed.</th>
</tr>
</thead>
</table>
### 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. 284).

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. 272) 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.
Chapter 53. Transformers

ExtFilter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

ExtFilter filters input records according to the specified condition.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtFilter</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For allowed data records</td>
<td>Input 0&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>For rejected data records</td>
<td>Input 0&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.
ExtFilter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter expression</td>
<td>yes</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

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.employeoid.

Note

You can also use the Partition (p. 592) component as a filter instead of ExtFilter. With the Partition (p. 592) component you can define much more sophisticated filter expressions and distribute data records among more output ports.

Important

Or you can use the Reformat (p. 603) component as a filter.

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

ExtSort sorts input records according to a sort key.

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✔</td>
<td>for input data records</td>
<td>the same</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✔</td>
<td>for sorted data records</td>
<td>input and output metadata 1)</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>✗</td>
<td>for sorted data records</td>
<td></td>
</tr>
</tbody>
</table>

1) As all output metadata must be same as the input metadata, they can be propagated through this component.
## ExtSort Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort key</td>
<td>✅</td>
<td>Key according to which the records are sorted. See Sort Key (p. 265) for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer capacity</td>
<td></td>
<td>Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.</td>
<td>8000 (default)</td>
</tr>
<tr>
<td>Number of tapes</td>
<td></td>
<td>Number of temporary files used to perform external sorting. Even number higher than 2.</td>
<td>6 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort order</td>
<td></td>
<td>Order of sorting (Ascending or Descending). Can be denoted by the first letter (A or D) only. The same for all key fields.</td>
<td>Ascending (default)</td>
</tr>
<tr>
<td>Sorting locale</td>
<td></td>
<td>Locale that should be used for sorting.</td>
<td>none (default)</td>
</tr>
<tr>
<td>Case sensitive</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Sorter initial capacity</td>
<td></td>
<td>does the same as Buffer capacity</td>
<td>8000 (default)</td>
</tr>
</tbody>
</table>

### 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

FastSort sorts input records using a sort key. FastSort is faster than ExtSort but requires more system resources.

<table>
<thead>
<tr>
<th>Component</th>
<th>Sort input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastSort</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-N</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✗</td>
<td>for input data records</td>
<td>the same input and output metadata</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✗</td>
<td>for sorted data records</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>✗</td>
<td>for sorted data records</td>
<td></td>
</tr>
</tbody>
</table>

1) As all output metadata must be same as the input metadata, they can be propagated through this component.
# FastSort Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort key</td>
<td>✔</td>
<td>List of fields (separated by semicolon) the data records are to be sorted by, including the</td>
<td>auto (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sorting order for each data field separately, see <code>Sort Key</code> (p. 265)</td>
<td></td>
</tr>
<tr>
<td>Estimated record</td>
<td>1)</td>
<td>Estimated number of input records to be sorted. A rough guess of the order of magnitude is</td>
<td>auto from (if set)</td>
</tr>
<tr>
<td>count</td>
<td></td>
<td>sufficient, see <code>Estimated Record Count</code> (p. 578).</td>
<td></td>
</tr>
<tr>
<td>In memory only</td>
<td></td>
<td>If <code>true</code>, internal sorting is forced and all attributes except <code>Sort key</code> and <code>Run size</code> are</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ignored.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run size (records)</td>
<td>1)</td>
<td>Number of records sorted at once in memory. Largely affects speed and memory requirements,</td>
<td>auto (default)</td>
</tr>
<tr>
<td></td>
<td>2)</td>
<td>see <code>Run Size</code> (p. 578)</td>
<td></td>
</tr>
<tr>
<td>Max open files</td>
<td></td>
<td>Limits the number of temp files that can be created during the sorting. Too low number (500 or</td>
<td>unlimited (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>less) significantly reduces the performance, see <code>Max Open Files</code> (p. 578).</td>
<td></td>
</tr>
<tr>
<td>Concurrency (threads)</td>
<td></td>
<td>Number of worker threads to do the job. The default value ensures the optimal results while</td>
<td>auto (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>overriding the default may even slow the graph run down, see <code>Concurrency</code> (p. 578).</td>
<td></td>
</tr>
<tr>
<td>Number of read buffers</td>
<td>2)</td>
<td>How many chunks of data will be held in memory at a time, see <code>Concurrency</code> (p. 578).</td>
<td>auto (default)</td>
</tr>
<tr>
<td>Average record size</td>
<td></td>
<td>Guess on average byte size of records, see <code>Average Record Size</code> (p. 579).</td>
<td>auto (default)</td>
</tr>
<tr>
<td>(bytes)</td>
<td>2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum memory (MB, GB)</td>
<td>2)</td>
<td>Rough estimate of maximum memory that can be used, see <code>Maximum Memory</code> (p. 578).</td>
<td>auto (default)</td>
</tr>
<tr>
<td>Tape buffer (bytes)</td>
<td></td>
<td>Buffer used by a worker for filling the output. Affects the performance slightly, see `Tape</td>
<td>8192 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buffer` (p. 579).</td>
<td></td>
</tr>
<tr>
<td>Compress temporary</td>
<td></td>
<td>If <code>true</code>, temporary files are compressed. For more information see <code>Compress Temporary Files</code></td>
<td>false (default)</td>
</tr>
<tr>
<td>files</td>
<td></td>
<td>(p. 579).</td>
<td></td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting locale</td>
<td></td>
<td>Locale used for correct sorting order</td>
<td>none (default)</td>
</tr>
<tr>
<td>Case sensitive</td>
<td></td>
<td>By default (<code>Sorting locale</code> is <code>none</code>), upper-case characters are sorted separately and precede lower-case characters that are sorted separately too. If <code>Sorting locale</code> is set, upper- and lower-case characters are sorted together - if <code>Case sensitive</code> is <code>true</code>, a lower-case precedes corresponding upper-case while <code>false</code> preserves the order, data strings appears in the input in.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

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 sizes 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 a great 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 sizes 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:

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Number of temp. files</th>
<th>Default Run size</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>~100</td>
<td>~10,000</td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td>~250</td>
<td>~45,000</td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>20,000 to 2,000</td>
<td>50,000 to 500,000</td>
<td>Depends on available memory</td>
</tr>
</tbody>
</table>

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. 579) 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. 82) 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 necessarily 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Merge merges and sorts data records from two or more inputs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>yes</td>
<td>yes</td>
<td>2-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-1</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>2-n</td>
<td>no</td>
<td>For input data records</td>
<td>Input 0(^1)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For merged data records</td>
<td>Input 0(^1)</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.
## Merge Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge key</td>
<td>yes</td>
<td>Key according to which the sorted records are merged. (Remember that all key fields must be sorted in ascending order!) See Group Key (p. 264) for more information.⁴</td>
<td></td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of key fields are considered to be different. If set to <code>true</code>, they are considered to be equal.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

### Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.
Chapter 53. Transformers

MetaPivot

We suppose that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the appropriate Transformer for your purpose, see Transformers Comparison (p. 309).

Short Summary

MetaPivot converts every incoming record into several output records, each one representing a single field from the input.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaPivot</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 583) All in all, MetaPivot can be used to effectively transform your records to a neat data-dependent structure.

Unlike Normalizer (p. 585), 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

![MetaPivot Icon](image)

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For transformed data records</td>
<td>Any2</td>
</tr>
</tbody>
</table>
MetaPivot Attributes

*MetaPivot* has no component-specific attributes.

**Advanced Description**

**Important**

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) - fields of the same record share the same number (notice in Figure 53.4, *Example MetaPivot Output* (p. 584))
- `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 53.4. Example MetaPivot Transformation

Let us now look at what **MetaPivot** makes to your data. Say you have a delimited file containing data of various data types. You have only two records:

![Figure 53.3. Example MetaPivot Input](image)

Sending these data to **MetaPivot** "classifies" the data to output fields corresponding to their data types:

![Figure 53.4. Example MetaPivot Output](image)

Thus e.g. "hello" is placed in the **valueString** field or "76.70" in **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 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)
- Chapter 43, *Common Properties of Transformers* (p. 309)

If you want to find the right Transformer for your purposes, see *Transformers Comparison* (p. 309).

**Short Summary**

Normalizer creates one or more output records from each single input record.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalizer</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For normalized data records</td>
<td>Any2</td>
</tr>
</tbody>
</table>
## Normalizer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalize</td>
<td>1)</td>
<td>Definition of the way how records should be normalized written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Normalize URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the way how records should be normalized written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Normalize class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be normalized.</td>
<td></td>
</tr>
<tr>
<td>Normalize source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
</tbody>
</table>

### Deprecated

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error actions</td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
</tbody>
</table>

**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. 586) or Java Interfaces for Normalizer (p. 591) for more information.

See also Defining Transformations (p. 268) 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 IX, CTL - CloverETL Transformation Language (p. 771). (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 53.5. Source Tab of the Transform Editor in the Normalizer Component (I)

Figure 53.6. Source Tab of the Transform Editor in the Normalizer Component (II)
Table 53.2. Functions in Normalizer

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th>boolean init()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>No</td>
</tr>
<tr>
<td>Description</td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td>Returns</td>
<td>true</td>
</tr>
</tbody>
</table>

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. 272) 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

```c
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

#### 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

```c
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
- **Returns**: 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 `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 input records. Also error message and stack trace are passed to `transformOnError()`.

#### Example

```c
function integer transformOnError(
    string errorMessage,
    string stackTrace, 
    integer idx) {
    printErr(errorMessage);
    return 1;
}
```
### CTL Template Functions

<table>
<thead>
<tr>
<th>Returns</th>
<th>Integer numbers. See Return Values of Transformations (p. 272) for detailed information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation</td>
<td>Called if <code>transform()</code> throws an exception.</td>
</tr>
<tr>
<td>Description</td>
<td>It creates output records. If any part of the <code>transform()</code> function for some output record causes fail of the <code>transform()</code> function, and if user has defined another function (<code>transformOnError()</code>), processing continues in this <code>transformOnError()</code> at the place where <code>transform()</code> failed. If <code>transform()</code> fails and user has not defined any <code>transformOnError()</code>, the whole graph will fail. The <code>transformOnError()</code> function gets the information gathered by <code>transform()</code> that was get from previously successfully processed code. Also error message and stack trace are passed to <code>transformOnError()</code>.</td>
</tr>
</tbody>
</table>
| Example                                                                | function integer `transformOnError`(`errorMessage`, `stackTrace`, `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. |
Chapter 53. Transformers

### CTL Template Functions

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Called during each graph run after the entire transform was executed.</th>
</tr>
</thead>
</table>

#### 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. 284).

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. 272) 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.
Chapter 53. Transformers

Partition

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Partition distributes individual input data records among different output ports.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n</td>
<td>yes/no&lt;sup&gt;1&lt;/sup&gt;</td>
<td>yes/no&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

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

![Partition](image)

### Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For output data records</td>
<td>Input 0&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>no</td>
<td>For output data records</td>
<td>Input 0&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

**Legend:**

1): Metadata can be propagated through this component.

### Partition Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td>1)</td>
<td>Definition of the way how records should be distributed among output ports</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Partition URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>way how records should be distributed among output ports written in CTL or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java.</td>
<td></td>
</tr>
<tr>
<td>Partition class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be distributed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>among output ports.</td>
<td></td>
</tr>
<tr>
<td>Ranges</td>
<td>1),2</td>
<td>Ranges expressed as a sequence of individual ranges separated from each</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>other by semicolon. Each individual range is a sequence of intervals for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>some set of fields that are adjacent to each other without any delimiter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is expressed also whether the minimum and maximum margin is included</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>to the interval or not by bracket and parenthesis, respectively. Example of</td>
<td></td>
</tr>
<tr>
<td>Partition key</td>
<td>1),2</td>
<td>Key according to which input records are distributed among different output</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ports. Expressed as the sequence of individual input field names separated</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>from each other by semicolon. Example of <strong>Partition key</strong>: first_name;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>last_name.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(default)</td>
<td>other encoding</td>
</tr>
</tbody>
</table>

**Deprecated**
### Chapter 53. Transformers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locale</td>
<td></td>
<td>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 file, see Changing Default CloverETL Settings. (p. 85).</td>
<td>system value or specified default value (default)</td>
</tr>
<tr>
<td>Use internationalization</td>
<td></td>
<td>By default, no internationalization is used. If set to true, sorting according national properties is performed.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

**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. 594) or [Java Interfaces for Partition (and ClusterPartitioner)](p. 598) for more information.

See also [Defining Transformations](p. 268) 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 IX, [CTL - CloverETL Transformation Language](p. 771). (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 template:

### 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 the corresponding button at the upper right corner of the tab.
Figure 53.7. 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 53.3. Functions in Partition (or ClusterPartitioner)

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>void init()</td>
<td></td>
</tr>
<tr>
<td>Required</td>
<td>No</td>
</tr>
<tr>
<td>Description</td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
<tr>
<td>integer getOutputPort()</td>
<td></td>
</tr>
<tr>
<td>Required</td>
<td>yes</td>
</tr>
<tr>
<td>Input Parameters</td>
<td>none</td>
</tr>
<tr>
<td>Returns</td>
<td>Integer numbers. See Return Values of Transformations (p. 272) for detailed information.</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called repeatedly for each input record</td>
</tr>
</tbody>
</table>
### 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 | ```
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. 272) 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);  
|         | }  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string getMessage()</td>
<td></td>
</tr>
</tbody>
</table>
| Required | No  
| Description | Prints error message specified and invoked by user  
| Invocation | Called in any time specified by user (called only when either getOutputPort() or getOutputPortOnError() 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. 284).

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. 272) 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. 272) 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.
Chapter 53. Transformers

**Pivot**

We suppose that you have already learned what is described in:

- Chapter 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)
- Chapter 43, *Common Properties of Transformers* (p. 309)

If you want to find the appropriate Transformer for your purpose, see *Transformers Comparison* (p. 309).

**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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivot</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Note: When using the key attribute, input records should be sorted, though. See *Advanced Description* (p. 600).

**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. 563).

**Icon**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For summarization data records</td>
<td>Any2</td>
</tr>
</tbody>
</table>
# Pivot Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>1)</td>
<td>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.</td>
<td>any input field</td>
</tr>
<tr>
<td>Group size</td>
<td>1)</td>
<td>The number of input records forming one group. When using Group size, the input data do not have to be sorted. <strong>Pivot</strong> then reads a number of records and transforms them to one group. The number is just the value of Group size.</td>
<td>&lt;1; n&gt;</td>
</tr>
<tr>
<td>Field defining output</td>
<td>2)</td>
<td>The input field whose value &quot;maps&quot; to a field name on the output.</td>
<td></td>
</tr>
<tr>
<td>Field defining output</td>
<td>2)</td>
<td>The input field whose value &quot;maps&quot; to a field value on the output.</td>
<td></td>
</tr>
<tr>
<td>Sort order</td>
<td></td>
<td>Groups of input records are expected to be sorted in the order defined here. The meaning is the same as in Denormalizer, see <a href="#">Sort order</a>. Beware that in Pivot, setting this to <strong>Ignore</strong> can produce unexpected results if input is not sorted.</td>
<td>Auto (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>Determines whether two fields containing null values are considered equal.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pivot transformation</td>
<td>3)</td>
<td>Using CTL or Java, you can write your own records transformation here.</td>
<td></td>
</tr>
<tr>
<td>Pivot transformation URL</td>
<td>3)</td>
<td>The path to an external file which defines how to transform records. The transformation can be written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Pivot transformation class</td>
<td>3)</td>
<td>The name of a class that is used for data transformation. It can be written in Java.</td>
<td></td>
</tr>
<tr>
<td>Pivot transformation source charset</td>
<td></td>
<td>The encoding of an external file defining the data transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Defines actions that should be performed when the specified transformation returns an <strong>Error code</strong>. See <a href="#">Return Values of Transformations</a>.</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file which error messages should be written to. These messages are generated during <strong>Error actions</strong>, see above. If the attribute is not set, messages are written to <strong>Console</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

**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. 601).

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. 602).

### 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 53.5. Data Transformation with Pivot - Using Key

Let us have the following input txt file with comma-separated values:

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. 600)

Before writing the transformation, you might want to refer to some of the sections touching the subject:

• Defining Transformations (p. 268)

• writing transformations in Denormalizer, the component Pivot is derived from: CTL Scripting Specifics (p. 565) and Java Interfaces for Denormalizer (p. 571)

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Reformat manipulates record’s structure or content.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reformat</td>
<td>-</td>
<td>✗</td>
<td>1</td>
<td>1-N</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Abstract

Reformat receives potentially unsorted data through single input port, transforms each of them in a user-specified way and sends 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>✔</td>
<td>for input data records</td>
<td>Any(In0)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>✔</td>
<td>for transformed data records</td>
<td>Any(Out0)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>✗</td>
<td>for transformed data records</td>
<td>Any(OutPortNo)</td>
</tr>
</tbody>
</table>
Reformat Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Definition of how records should be reformated. Written in the graph source</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>either in CTL or in Java.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the way</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>how records should be reformated; written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be reformated.</td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
</tbody>
</table>

Deprecated

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error actions</td>
<td>Definition of the action that should be performed when the specified</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transformation returns some <strong>Error code</strong>. See **Return Values of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transformations** (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td>URL of the file to which error messages for specified <strong>Error actions</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>should be written. If not set, they are written to <strong>Console</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

Legend:

1): One of these must be specified. Any of these transformation attributes uses a CTL template for **Reformat** or implements a **RecordTransform** interface.

See **CTL Scripting Specifics** (p. 604) or **Java Interfaces for Reformat** (p. 605) for more information.

See also **Defining Transformations** (p. 268) for detailed information about transformations.

Use Reformat To

- Drop unwanted fields
- Validate fields using functions or regular expressions (p. 921)
- 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 IX, **CTL - CloverETL Transformation Language** (p. 771). (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. 314) for more information.
Java Interfaces for Reformat

Reformat implements the same interface as DataIntersection and Joiners. See Java Interfaces for Joiners (p. 317) for more information.
Rollup

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

Rollup creates one or more output records from one or more input records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rollup</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-N</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any(In0)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For output data records</td>
<td>Any(Out0)</td>
</tr>
<tr>
<td></td>
<td>1-N</td>
<td>no</td>
<td>For output data records</td>
<td>Any(Out1-N)</td>
</tr>
</tbody>
</table>
## Rollup Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group key</td>
<td></td>
<td>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. 264) for more information. If not specified, all records are considered to be members of a single group.</td>
<td></td>
</tr>
<tr>
<td>Group accumulator</td>
<td></td>
<td>ID of metadata that serve to create group accumulators. Metadata serve to store values used for transformation of individual groups of data records.</td>
<td>no metadata (default)</td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Definition of the transformation written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the transformation written in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>Name of external class defining the transformation.</td>
<td></td>
</tr>
<tr>
<td>Transform source</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>charset</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Sorted input</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. 607) or [Java Interfaces for Rollup](p. 616) for more information.

See also [Defining Transformations](p. 268) 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. 616) for detailed information this interface.

For detailed information about Clover Transformation Language see Part IX, [CTL - CloverETL Transformation Language](p. 771). (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 53.8. Source Tab of the Transform Editor in the Rollup Component (I)](image1)

![Figure 53.9. Source Tab of the Transform Editor in the Rollup Component (II)](image2)
Chapter 53. Transformers

Figure 53.10. Source Tab of the Transform Editor in the Rollup Component (III)

Table 53.4. Functions in Rollup

<table>
<thead>
<tr>
<th>CTL Template Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void init()</strong></td>
<td><strong>Required</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Initialize the component, setup the environment, global variables</td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
<td>Called before processing the first record</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>void initGroup(&lt;metadata name&gt; groupAccumulator)</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
<td>yes</td>
</tr>
<tr>
<td><strong>Input Parameters</strong></td>
<td>&lt;metadata name&gt; groupAccumulator (metadata specified by user) If groupAccumulator is not defined, VoidMetadata Accumulator is used in the function signature.</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
<td>Called repeatedly, once for the first input record of each group. Called before updateGroup(groupAccumulator).</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Initializes information for specific group.</td>
</tr>
</tbody>
</table>
| **Example** | function void initGroup(  
  companyCustomers groupAccumulator) {  
  groupAccumulator.count = 0;  
  groupAccumulator.totalFreight = 0;  
} |
### CTL Template Functions

<table>
<thead>
<tr>
<th>boolean updateGroup(&lt;metadata name&gt; groupAccumulator)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
</tr>
<tr>
<td><strong>Input Parameters</strong></td>
</tr>
<tr>
<td><strong>Returns</strong></td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>
| **Example** | function boolean updateGroup(  
| | \quad companyCustomers groupAccumulator) {  
| | \quad groupAccumulator.count++;  
| | \quad groupAccumulator.totalFreight =  
| | \quad groupAccumulator.totalFreight + $0.Freight;  
| | \quad 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 \((\text{transform(counter,groupAccumulator)} \text{ is called})\) \| \(false \text{ (transform(counter,groupAccumulator)} \text{ is not called)}\) |
| **Invocation** | Called repeatedly, once for the last input record of each group. Called after \(\text{updateGroup(groupAccumulator)}\) 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 \(\text{finishGroup()}\) function, and if user has defined another function \(\text{finishGroupOnError()}\), processing continues in this \(\text{finishGroupOnError()}\) at the place where \(\text{finishGroup()}\) failed. If \(\text{finishGroup()}\) fails and user has not defined any \(\text{finishGroupOnError()}\), the whole graph will fail. The \(\text{finishGroup()}\) passes to \(\text{finishGroupOnError()}\) error message and stack trace as arguments. |
### CTL Template Functions

| Example | function boolean finishGroup(   
|         |   companyCustomers groupAccumulator) {   
|         |     groupAccumulator.avgFreight =   
|         |     groupAccumulator.totalFreight   
|         |     / groupAccumulator.count;   
|         |     return true;   
|         | } |

<table>
<thead>
<tr>
<th>integer updateTransform(integer counter, &lt;metadata name&gt; groupAccumulator)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
</tr>
<tr>
<td><strong>Input Parameters</strong></td>
</tr>
<tr>
<td><strong>Returns</strong></td>
</tr>
<tr>
<td><strong>Invocation</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>

| 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;   
|         | } |

<table>
<thead>
<tr>
<th>integer transform(integer counter, &lt;metadata name&gt; groupAccumulator)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
</tr>
</tbody>
</table>
### CTL Template Functions

<table>
<thead>
<tr>
<th>Input Parameters</th>
<th>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.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupAccumulator (metadata specified by user) If groupAccumulator is not defined, VoidMetadata Accumulator is used in the function signature.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>Integer numbers. See Return Values of Transformations (p. 272) for detailed information.</th>
</tr>
</thead>
</table>

| 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;
|         | }

```java
void initGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)
```

<table>
<thead>
<tr>
<th>Required</th>
<th>no</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Input Parameters</th>
<th>string errorMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>stackTrace</td>
<td>string stackTrace</td>
</tr>
<tr>
<td>groupAccumulator (metadata specified by user) If groupAccumulator is not defined, VoidMetadata Accumulator is used in the function signature.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>void</th>
</tr>
</thead>
</table>

| Invocation | Called if `initGroup()` throws an exception. |

| Description | Initializes information for specific group. |

<table>
<thead>
<tr>
<th>Example</th>
<th>function void initGroupOnError( string errorMessage, string stackTrace, companyCustomers groupAccumulator) printErr(errorMessage);</th>
</tr>
</thead>
</table>
### CTL Template Functions

#### boolean updateGroupOnError(string errorMessage, string stackTrace, <metadata name> groupAccumulator)

<table>
<thead>
<tr>
<th>Required</th>
<th>no</th>
</tr>
</thead>
</table>
| 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)

<table>
<thead>
<tr>
<th>Required</th>
<th>no</th>
</tr>
</thead>
</table>
| 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

<table>
<thead>
<tr>
<th>Input Parameters</th>
<th>string errorMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>string stackTrace</td>
</tr>
<tr>
<td>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.)</td>
<td></td>
</tr>
<tr>
<td>&lt;metadata name&gt; groupAccumulator (metadata specified by user) If groupAccumulator is not defined, VoidMetadata Accumulator is used in the function signature.</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td>Integer numbers. See [Return Values of Transformations](p. 272) for detailed information.</td>
</tr>
<tr>
<td>Invocation</td>
<td>Called if updateTransform() throws an exception.</td>
</tr>
<tr>
<td>Description</td>
<td>It creates output records based on individual record information</td>
</tr>
<tr>
<td>Example</td>
<td>function integer updateTransformOnError(</td>
</tr>
<tr>
<td></td>
<td>string errorMessage,</td>
</tr>
<tr>
<td></td>
<td>string stackTrace,</td>
</tr>
<tr>
<td></td>
<td>integer counter,</td>
</tr>
<tr>
<td></td>
<td>companyCustomers groupAccumulator) {</td>
</tr>
<tr>
<td></td>
<td>if (counter &gt;= 0) {</td>
</tr>
<tr>
<td></td>
<td>return SKIP;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>printErr(errorMessage);</td>
</tr>
<tr>
<td></td>
<td>return ALL;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

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 SKIF 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. 272) 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() | String error message specified and invoked 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(), updateTransformOnError(), and transformOnError() functions.

- **Output records or fields**

---
Output records or fields are accessible within the `updateTransform()`, `transform()`, `updateTransformOnError()`, 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()`, `updateTransformOnError()`, 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. 284).

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. 272) 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. 272) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

SimpleCopy copies data to all connected output ports.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleCopy</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Abstract

SimpleCopy receives data records through single input port and copies each of them to all connected output ports.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For copied data records</td>
<td>Input 0(^1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For copied data records</td>
<td>Output 0(^1)</td>
</tr>
</tbody>
</table>

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

SimpleGather gathers data records from multiple inputs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleGather</td>
<td>yes</td>
<td>no</td>
<td>1-n</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For input data records</td>
<td>Input 0&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For gathered data records</td>
<td>Input 0&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

SortWithinGroups sorts input records within groups of records according to a sort key.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortWithinGroups</td>
<td></td>
<td>yes</td>
<td>1</td>
<td>1-n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For sorted data records</td>
<td>Input 0(^1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For sorted data records</td>
<td>Input 0(^1)</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can be propagated through this component. All output metadata must be the same.
## SortWithinGroups Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group key</td>
<td>yes</td>
<td>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. 264) for more information.</td>
<td></td>
</tr>
<tr>
<td>Sort key</td>
<td>yes</td>
<td>Key according to which the records are sorted within each group of adjacent records. See [Sort Key](p. 265) for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer capacity</td>
<td></td>
<td>Maximum number of records parsed in memory. If there are more input records than this number, external sorting is performed.</td>
<td>10485760 (default)</td>
</tr>
<tr>
<td>Number of tapes</td>
<td></td>
<td>Number of temporary files used to perform external sorting. Even number higher than 2.</td>
<td>8 (default)</td>
</tr>
</tbody>
</table>

### 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 43, Common Properties of Transformers (p. 309)

If you want to find the right Transformer for your purposes, see Transformers Comparison (p. 309).

Short Summary

XSLTransformer transforms input data records using an XSL transformation.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLTransformer</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For transformed data records</td>
<td>Any2</td>
</tr>
</tbody>
</table>

XSLTransformer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT file</td>
<td>1)</td>
<td>External file defining the XSL transformation.</td>
<td></td>
</tr>
<tr>
<td>XSLT</td>
<td>1)</td>
<td>XSL transformation defined in the graph.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 53. Transformers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
</table>
| Mapping         | 2)  | Sequence of individual mappings for output fields separated from each other by semicolon. Each individual mapping has the following form: \$outputField:=\text{transform}($inputField) \text{(if inputField should be transformed according to the XSL transformation)} or \$outputField:=$inputField \text{(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**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT file charset</td>
<td>Encoding of external file defining the XSL transformation for all data types except byte or cbyte. Default encoding is UTF-8.</td>
<td>UTF-8 (default)</td>
</tr>
</tbody>
</table>

**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 53.11. 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:

```
$1.first_name=$1.frame
$1.last_name=$1. ln
$1.name=$1.street
$1.cbyte=$1.address
```

*Figure 53.12. 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 54. Joiners

We assume that you already know what components are. See Chapter 18, Components (p. 94) for brief information.

Some components are intermediate nodes of the graph. These are called Joiners or Transformers.

For information about Transformers see Chapter 53, Transformers (p. 550). 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 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 44, Common Properties of Joiners (p. 312)

We can distinguish Joiners according to how they process data. Most 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. 638) joins two or more data inputs according to the equality of key values.
  • ExtMergeJoin (p. 644) 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. 635) joins one input data source and a database according to the equality of key values.
  • LookupJoin (p. 649) 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. 625) 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. 652) joins two or more sorted data inputs according to the user-defined relation of key values (!=, >, >=, <, <=).

• Combine (p. 633) joins data flows by tuples.
ApproximativeJoin

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApproximativeJoin</td>
<td>no</td>
<td>yes</td>
<td>1</td>
<td>2-4</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output port for the joined data with greater conformity</td>
<td>Any, optionally including additional fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><em>total_conformity</em> and <em>keyName_conformity</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Additional fields (p. 631).</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>Output port for the joined data with smaller conformity</td>
<td>Any, optionally including additional fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><em>total_conformity</em> and <em>keyName_conformity</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See Additional fields (p. 631).</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>no</td>
<td>Optional output port for master data records without slave matches</td>
<td>Input 0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>no</td>
<td>Optional output port for slave data records without master matches</td>
<td>Input 1</td>
</tr>
</tbody>
</table>

ApproximativeJoin Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>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. 629).</td>
<td></td>
</tr>
<tr>
<td>Matching key</td>
<td>yes</td>
<td>This key serves to match master and slave records.</td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Transformation in CTL or Java defined in the graph for records with greater conformity.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>External file defining the transformation in CTL or Java for records with greater conformity.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>External transformation class for records with greater conformity.</td>
<td></td>
</tr>
<tr>
<td>Transform for suspicious</td>
<td>2)</td>
<td>Transformation in CTL or Java defined in the graph for records with smaller conformity.</td>
<td></td>
</tr>
<tr>
<td>Transform URL for suspicious</td>
<td>2)</td>
<td>External file defining the transformation in CTL or Java for records with smaller conformity.</td>
<td></td>
</tr>
<tr>
<td>Transform class for suspicious</td>
<td>2)</td>
<td>External transformation class for records with smaller conformity.</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 54. Joiners

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conformity limit (0,1)</td>
<td></td>
<td>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.</td>
<td>0.75 (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locale</td>
<td></td>
<td>Locale to be used when internationalization is used.</td>
<td></td>
</tr>
<tr>
<td>Case sensitive</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
<tr>
<td>Slave override key</td>
<td></td>
<td>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).</td>
<td></td>
</tr>
<tr>
<td>Slave override matching key</td>
<td></td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

**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. 632) or [Java Interfaces](#) (p. 632) for more information.

See also [Defining Transformations](#) (p. 268) 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.)

![Matching Key Wizard (Master Key Tab)](image1)

Figure 54.1. 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.

![Matching Key Wizard (Slave Key Tab)](image2)

Figure 54.2. Matching Key Wizard (Slave Key Tab)

Example 54.1. 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. 629)):

• **Identical** - is a character-by-character comparison. The penalty is given for each different character (similar to `String.equals()`).

• **Tertiary** - ignores 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. 'ć') 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', 'ä', '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.
Chapter 54. Joiners

Figure 54.3. 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 54.4. 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 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:

```plaintext
$driver_field1=$slave_field1(parameters);...;$driver_fieldN=$slave_fieldN(parameters)
```

**Example 54.2. Join Key for ApproximativeJoin**

```plaintext
$first_name=$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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL - CloverETL Transformation Language (p. 771). (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. 314).

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. 317)
Combine

Jobflow Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

The component is located in Palette → Joiners.

Short Summary

Combine takes one record from each input port, combines them according to a specified transformation and sends the resulting records to one or more ports.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Transformation req.</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine</td>
<td>no</td>
<td>no</td>
<td>1–n</td>
<td>1-n</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

In each step, the Combine component takes one record from all input ports, creates single output record, and fills fields of this output record with data from input record (or other data) according to specified transformation.

The simplest way how to define the combination transformation is using the Transform Editor (p. 275) available at the Transform component attribute. There you will see metadata for each input port on the left side and metadata for single output port on the right side. Simply drag and drop fields from the left to the fields on the right to create desired combination transformation.

In default setting, the component assumes that the same number of records will arrive on each input port, and in case that some input edge becomes empty while others still contain some records, the component fails. You can avoid this failures by setting the Allow incomplete tuples attribute to true.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1–n</td>
<td>yes</td>
<td>Input records to be combined.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output record which is the result of combination.</td>
<td>Any</td>
</tr>
</tbody>
</table>
## Combine Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Definition of how input records should be combined into output record. Written in the graph source either in CTL or in Java.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>Name of external file, including path, containing the definition of the way how records should be combined. Written in CTL or in Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be combined.</td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Allow incomplete tuples</td>
<td></td>
<td>Whether each input port has to contribute a record for each output record.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

### Legend:

1): One of these must be specified. Any of these transformation attributes uses a CTL template for **Reformat** or implements a **RecordTransform** interface.

See also [Defining Transformations](p. 268) for detailed information about transformations.
Chapter 54. Joiners

DBJoin

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBJoin</td>
<td>no</td>
<td>no</td>
<td>1 (virtual)</td>
<td>1-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1 (virtual)</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
</tbody>
</table>
### Joiners

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output port for the joined data</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Optional output port for master data records without slave matches. (Only if the <code>Join type</code> attribute is set to <code>inner join</code>.) This applies only to <code>LookupJoin</code> and <code>DBJoin</code>.</td>
<td>Input 0</td>
</tr>
</tbody>
</table>

#### DBJoin Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>Key according to which the incoming data flows are joined. See <code>Join key</code> (p. 637).</td>
<td></td>
</tr>
<tr>
<td>Left outer join</td>
<td></td>
<td>If set to <code>true</code>, also driver records without corresponding slave are parsed. Otherwise, <code>inner join</code> is performed.</td>
<td><code>false</code> (default)</td>
</tr>
<tr>
<td>DB connection</td>
<td>yes</td>
<td>ID of the DB connection to be used as the resource of slave records.</td>
<td></td>
</tr>
<tr>
<td>DB metadata</td>
<td></td>
<td>ID of DB metadata to be used. If not set, metadata is extracted from database using SQL query.</td>
<td></td>
</tr>
<tr>
<td>Query URL</td>
<td>3)</td>
<td>Name of external file, including path, defining SQL query.</td>
<td></td>
</tr>
<tr>
<td>SQL query</td>
<td>3)</td>
<td>SQL query defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>1), 2)</td>
<td>Transformation in CTL or Java defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1), 2)</td>
<td>External file defining the transformation in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1), 2)</td>
<td>External transformation class.</td>
<td></td>
</tr>
<tr>
<td>Cache size</td>
<td></td>
<td>Maximum number of records with different key values that can be stored in memory.</td>
<td>100 (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform charset</td>
<td>source</td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
</tbody>
</table>

**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. 637) or [Java Interfaces](p. 637) for more information.

See also [Defining Transformations](p. 268) 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 54.3. 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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL - CloverETL Transformation Language (p. 771). (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. 314).

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. 317)
ExtHashJoin

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

**Short Summary**

General purpose joiner, merges potentionally unsorted data from two or more data sources on a common key.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtHashJoin</td>
<td>no</td>
<td>no</td>
<td>1-n</td>
<td>1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**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. 643).

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).
Chapter 54. Joiners

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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>2-n</td>
<td>no</td>
<td>Optional slave input ports</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output port for the joined data</td>
<td>Any</td>
</tr>
</tbody>
</table>

ExtHashJoin Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>Key according to which the incoming data flows are joined. See Join key (p. 641).</td>
<td></td>
</tr>
<tr>
<td>Join type</td>
<td></td>
<td>Type of the join. See Join Types (p. 313).</td>
<td>Inner (default)</td>
</tr>
<tr>
<td>Transform</td>
<td>1</td>
<td>Transformation in CTL or Java defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1</td>
<td>External file defining the transformation in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1</td>
<td>External transformation class.</td>
<td></td>
</tr>
<tr>
<td>Allow slave duplicates</td>
<td></td>
<td>If set to true, records with duplicate key values are allowed. If it is false, only the first record is used for join.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Hash table size</td>
<td></td>
<td>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. 643) for more information:</td>
<td>512 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Left outer</td>
<td></td>
<td>If set to <code>true</code>, left outer join is performed. By default it is <code>false</code>. However, this attribute has lower priority than <code>Join type</code>. If you set both, only <code>Join type</code> will be applied.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Full outer</td>
<td></td>
<td>If set to <code>true</code>, full outer join is performed. By default it is <code>false</code>. However, this attribute has lower priority than <code>Join type</code>. If you set both, only <code>Join type</code> will be applied.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>
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. 643) or [Java Interfaces](#) (p. 643) for more information.

See also [Defining Transformations](#) (p. 268) 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.

  $$
  \textit{CUSTOMERID=\textit{CUSTOMERID}#\textit{ORDERID=\textit{ORDERID};\textit{PRODUCTID=\textit{PRODUCTID}}}
  $$

  **Figure 54.6. 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 54.4. Slave Part of Join Key for ExtHashJoin**

  ```
  \textit{\$master\_field1=\$slave\_field1;\$master\_field2=\$slave\_field2;...;\$master\_fieldN=\$slave\_fieldN}
  ```

  - If some `$\textit{slave\_fieldJ}$` is missing (in other words, if the subexpression looks like this: `$\textit{\$master\_fieldJ=}$`), it is supposed to be the same as the `$\textit{\$master\_fieldJ}$`.

  - If some `$\textit{\$master\_fieldK}$` is missing, `$\textit{\$master\_fieldK}$` from the first port is used.
Example 54.5. Join Key for ExtHashJoin

$first_name=$fname;$last_name=$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.

![Hash Join Key Wizard](figure54.7.jpg)

**Figure 54.7. 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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, **CTL - CloverETL Transformation Language** (p. 771). (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. 314).

**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. 317)
ExtMergeJoin

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

**Short Summary**

General purpose joiner, merges sorted data from two or more data sources on a common key.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtMergeJoin</td>
<td>no</td>
<td>yes</td>
<td>1-n</td>
<td>1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**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. 647).

**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.
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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>2-n</td>
<td>no</td>
<td>Optional slave input ports</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output port for the joined data</td>
<td>Any</td>
</tr>
</tbody>
</table>

*ExtMergeJoin Attributes*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>Key according to which the incoming data flows are joined. See <em>Join key</em> (p. 646).</td>
<td></td>
</tr>
<tr>
<td>Join type</td>
<td></td>
<td>Type of the join. See <em>Join Types</em> (p. 313).</td>
<td>Inner (default)</td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Transformation in CTL or Java defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>External file defining the transformation in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>External transformation class.</td>
<td></td>
</tr>
<tr>
<td>Allow slave duplicates</td>
<td></td>
<td>If set to <em>true</em>, records with duplicate key values are allowed. If it is <em>false</em>, only the first record is used for join.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Ascending ordering of inputs</td>
<td></td>
<td>If set to <em>true</em>, incoming records are supposed to be sorted in ascending order. If it is set to <em>false</em>, they are descending.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locale</td>
<td></td>
<td>Locale to be used when internationalization is used.</td>
<td></td>
</tr>
<tr>
<td>Case sensitive</td>
<td></td>
<td>If set to <em>true</em>, upper and lower cases of characters are considered different. By default, they are processed as if they were equal to each other.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some <em>Error code</em>. See <em>Return Values of Transformations</em> (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified <em>Error actions</em> should be written. If not set, they are written to <em>Console</em>.</td>
<td></td>
</tr>
<tr>
<td>Left outer</td>
<td></td>
<td>If set to <em>true</em>, left outer join is performed. By default it is <em>false</em>. However, this attribute has lower priority than <em>Join type</em>. If you set both, only <em>Join type</em> will be applied.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>


### Attribute Request Description Possible values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full outer</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

**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. 648) or [Java Interfaces](p. 648) for more information.

See also [Defining Transformations](p. 268) 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.

  ![Example of the Join Key Attribute in ExtMergeJoin Component](Figure 54.8. 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).

  ![Join Key Wizard (Master Key Tab)](Figure 54.9. 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.

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 54.6. Join Key for ExtMergeJoin**

```plaintext
$first_name;$last_name#$fname;$lname#$f_name;$l_name
```

Following is the part of **Join key** for the master data source (input port 0):

```plaintext
$first_name;$last_name
```

• Thus, these fields are joined with the two fields from the first slave data source (input port 1):

```plaintext
$fname and $lname, respectively.
```

• And, these fields are also joined with the two fields from the second slave data source (input port 2):

```plaintext
$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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL - CloverETL Transformation Language (p. 771). (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. 314).

**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. 317)
LookupJoin

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

For information about lookup tables see Chapter 25, Lookup Tables (p. 186).

Short Summary

General purpose joiner, merges potentially unsorted data from one data source incoming through the single input port with another data source from lookup table based on a common key.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>LookupJoin</td>
<td>no</td>
<td>no</td>
<td>1 (virtual)</td>
<td>1-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

This is a general purpose joiner used in most common situations. It does not require that 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, the second data source is called the slave. Its data is considered as if it were coming through the second (virtual) input port. Each master record is matched to the slave record on one or more fields known as the join key. The output is produced by applying a transformation which maps joined inputs to the output.

Slave data is pulled out from a lookup table, so depending on the lookup table the data can be stored in the memory. That also depends on the lookup table type - e.g. Database lookup stores only the values which have already been queried. Master data is not stored in the memory.

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.
Chapter 54. Joiners

The joined data is then sent to the first output port. The second output port can optionally be used to capture unmatched master records.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1 (virtual)</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>Output port for the joined data</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>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.</td>
<td>Input 0</td>
</tr>
</tbody>
</table>

### LookupJoin Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join key</td>
<td>yes</td>
<td>Key according to which the incoming data flows are joined. See Join key (p. 650).</td>
<td></td>
</tr>
<tr>
<td>Left outer join</td>
<td></td>
<td>If set to true, also driver records without corresponding slave are parsed. Otherwise, inner join is performed.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Lookup table</td>
<td>yes</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Transformation in CTL or Java defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>External file defining the transformation in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>External transformation class.</td>
<td></td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free lookup table after finishing</td>
<td></td>
<td>If set to true, lookup table is emptied after the parsing finishes.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified transformation returns some Error code. See Return Values of Transformations (p. 272).</td>
<td></td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
<td></td>
</tr>
</tbody>
</table>

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. 651) or Java Interfaces (p. 651) for more information.

See also Defining Transformations (p. 268) 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.

![Edit Key Wizard](image)

**Figure 54.11. 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 54.7. 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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, [CTL - CloverETL Transformation Language](p. 771). (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. 314).

**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. 317)
RelationalJoin

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 44, Common Properties of Joiners (p. 312)

If you want to find the right Joiner for your purposes, see Joiners Comparison (p. 312).

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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Slave inputs</th>
<th>Outputs</th>
<th>Output for drivers without slave</th>
<th>Output for slaves without driver</th>
<th>Joining based on equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelationalJoin</td>
<td>no</td>
<td>yes</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Master input port</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>Slave input port</td>
<td>Any</td>
</tr>
</tbody>
</table>
### RelationalJoin Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join key</td>
<td>yes</td>
<td>Key according to which the incoming data flows are joined. See Join key (p. 653).</td>
</tr>
<tr>
<td>Join relation</td>
<td>yes</td>
<td>Defines the way of joining driver (master) and slave records. See Join relation (p. 655).</td>
</tr>
<tr>
<td>Join type</td>
<td></td>
<td>Type of the join. See Join Types (p. 313).</td>
</tr>
<tr>
<td>Transform</td>
<td>1)</td>
<td>Transformation in CTL or Java defined in the graph.</td>
</tr>
<tr>
<td>Transform URL</td>
<td>1)</td>
<td>External file defining the transformation in CTL or Java.</td>
</tr>
<tr>
<td>Transform class</td>
<td>1)</td>
<td>External transformation class.</td>
</tr>
<tr>
<td>Transform source charset</td>
<td></td>
<td>Encoding of external file defining the transformation.</td>
</tr>
</tbody>
</table>

**Possible values**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
</tr>
</tbody>
</table>

**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. 655) or Java Interfaces (p. 655) for more information.

See also Defining Transformations (p. 268) 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 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.

**Figure 54.12. 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).

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.

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 54.8. Join Key for RelationalJoin

\[
\text{$first\_name$;$last\_name$}$\text{fname$;$lname$}$\text{f\_name$;$l\_name$}
\]

Following is the part of Join key for the master data source (input port 0):

\[
\text{$first\_name$} = \text{fname$;$last\_name$} = \text{lname$}
\]

• Thus, these fields are joined with the two fields from the first slave data source (input port 1):

\[
\text{fname$;$lname$}
\]

• And, these fields are also joined with the two fields from the second slave data source (input port 2):

\[
\text{f\_name$;$l\_name$}
\]

• 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 easiest approach. This is when you need to use CTL scripting.

For detailed information about Clover Transformation Language see Part IX, CTL - CloverETL Transformation Language (p. 771). (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. 314).

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. 317)
Chapter 55. Job Control

We assume that you already know what components are. See Chapter 18, Components (p. 94) for brief information.

Some components are focused on execution and monitoring of various job types. We call this group of components: Job control.

Job control components are usually tightly bound with jobflow (p. 240). However, a few of them can be used even in regular ETL graphs.

These components allow running ETL graphs, jobflows and any interpreted scripts. Graphs and jobflows can be monitored and optionally aborted.

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

We can distinguish each component of the Job control group according to the task it performs.

- **Barrier** (p. 657) waits for results of jobs running in parallel and sends an aggregated result an to output port.
- **Condition** (p. 660) routes incoming tokens to one of its output ports based on the result of a specified condition.
- **ExecuteGraph** (p. 663) runs subgraphs with user-specified settings.
- **ExecuteJobflow** (p. 669) runs jobflows with user-specified settings.
- **ExecuteScript** (p. 671) runs either shell scripts or scripts interpreted by a selected interpreter.
- **Fail** (p. 677) aborts parent job.
- **GetJobInput** (p. 680) produces a single record populated by dictionary content.
- **KillGraph** (p. 682) aborts specified graphs.
- **KillJobflow** (p. 686) aborts specified jobflows.
- **MonitorGraph** (p. 688) watches running graphs.
- **MonitorJobflow** (p. 692) watches running jobflows.
- **SetJobOutput** (p. 694) sets incoming values to dictionary content.
- **Success** (p. 696) consumes all incoming tokens or records which are considered successful.
- **TokenGather** (p. 698) copies incoming tokens from any input port to all output ports.
Barrier

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

Barrier allows to wait for results of parallel running jobs and react to the success or failure of groups of jobs in a simple way.

Barrier waits for all input tokens belonging to a group, evaluates this group and based on the results sends output token(s) to first or second output port.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>no</td>
<td>no</td>
<td>1-n</td>
<td>1-2</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

Barrier is mainly used for management of parallel running jobs. Barrier receives all incoming tokens, which carry information about job results, and splits them to logical groups of jobs. Each job group is evaluated independently. Results of groups evaluated as successful are sent to the first output port. Results of the unsuccessful groups are sent to the second output port.

Logical grouping of incoming records

Component attribute 'Input grouping' provides two options how the incoming tokens are split to logical groups of jobs.

- All - all incoming tokens are considered as a single group, exactly one group is processed by the component. This covers the most common scenarios, e.g. checking that all previous jobs were successful.
• **Tuple** - a group consists from a single token from all input ports, i.e. the groups are created by "waves" of tokens coming from input ports. Tokens which arrive first from each input port form the first logical group, the second tokens from each input port form the second logical group etc (i.e. n-th group consists from n-th input token from all input ports). This setting covers checking result of waves of parallel job.

**Group evaluation**

Each token in a group is evaluated by CTL boolean expression from component attribute 'Token evaluation expression' - let's call it job status. The group is considered successful if and only if the job statuses joined by logical operation AND or OR (component attribute 'Group evaluation criteria') is true. So in case of AND operation, all incoming token needs to be successful for success of whole group. On the other hand in case of OR operation, at least one token from the group needs to be successful for group success.

**Generating output tokens**

Successful groups send their results to the first output port and the unsuccessful groups send their results to the second output port. Number and content of output tokens is specified by component attribute 'Output':

- **Single token** - only one token is sent to an output port per group, the token is populated by all group tokens - fields are copied based on fields names (input tokens order to be copied is not guaranteed).

- **All tokens** - each incoming token from the group is sent to the dedicated output port, in case of incompatible metadata fields are copied based on field names.

**Note**

Output ports are not required. Tokens routed to a missing edge are quietly discarded.

**Example usage**

Let's look at simple example of usage.

*Figure 55.1. Example of typical usage of Barrier component*

In this example, three different graphs are synchronously executed by three ExecuteGraph components. All three graphs are running in parallel. Barrier is a collection point for graph execution outcomes; it waits for all graphs to finish prior to moving on to the next step. If all graphs finished successfully, an output token is sent to the first output port. On the other hand, if one or more graphs failed, an output token is sent to the second output port. This component allows simple evaluation for status of the whole job group.

**Icon**

![Barrier icon](image)

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Input tokens with job results.</td>
<td>Any 1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>Input tokens with job results.</td>
<td>Any 1)</td>
</tr>
</tbody>
</table>
### Chapter 55. Job Control

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Tokens for successful groups of jobs.</td>
<td>Any 2)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Tokens for unsuccessful groups of jobs.</td>
<td>Any 2)</td>
</tr>
</tbody>
</table>

**Legend:**

1): Any metadata are possible, only field called 'status' is expected by 'Token evaluation expression' attribute by default ($status == "FINISHED_OK").

2): Any metadata are possible, but tokens sent to output ports are copied based on field names from input ports, so only fields with equal names are populated.
### Barrier Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input grouping</td>
<td>no</td>
<td>Type of algorithm how the incoming tokens are split into groups of jobs, which are evaluated independently. See Logical grouping of incoming records (p. 657)</td>
<td>Tuple (default)</td>
</tr>
<tr>
<td>Token evaluation expression</td>
<td>no</td>
<td>Boolean CTL expression which is applied to each incoming token to decide, whether the token represents successful or unsuccessful job run - final job status. See Group evaluation (p. 658)</td>
<td>default CTL expression</td>
</tr>
<tr>
<td>Group evaluation criteria</td>
<td>no</td>
<td>Logical operation which is applied to the job status (see attribute 'Token evaluation expression') to decide, whether the group of jobs is successful or unsuccessful. See Group evaluation (p. 658)</td>
<td>AND (default)</td>
</tr>
<tr>
<td>Output</td>
<td>no</td>
<td>Defines number of output tokens per group of jobs. See Generating output tokens (p. 658)</td>
<td>Single token (default)</td>
</tr>
</tbody>
</table>

---

### Condition

**Jobflow Component**

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

The component is located in Palette → Job Control.

**Short Summary**

The **Condition** component routes incoming tokens to one of its output ports based on result of specified condition. It is similar to `if` statement in programming languages.

**Note**

To be able to use this component, your license needs to support the Jobflow. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>–</td>
<td>no</td>
<td>1</td>
<td>1–2</td>
<td>no</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Abstract

For each incoming token, the **Condition** component evaluates specified Boolean condition and if the result is true, the token is sent to the first output port, otherwise to the second (optional) output port.

**Condition** works the same way as the **ExtFilter** (p. 572) component.

## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input tokens</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For tokens compliant with the condition</td>
<td>Input 0(^1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>For tokens not satisfying the condition</td>
<td>Input 0(^1)</td>
</tr>
</tbody>
</table>

**Legend:**

1) Metadata can be propagated through this component. All output metadata must be the same.

This component has **Metadata Templates** (p. 264) available.
### Condition Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>yes</td>
<td>Boolean expression according to which the tokens are filtered. Expressed as the sequence of individual expressions for individual input fields separated from each other by semicolon.</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced Description

For more details about the Condition attribute see Filter Expression (p. 573) of the ExtFilter component.
Chapter 55. Job Control

ExecuteGraph

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Tip

If you drag a *.grf file and drop it into the jobflow pane, you will add the ExecuteGraph component.

Short Summary

ExecuteGraph allows running of subgraphs with user-specified settings and provides execution results and tracking details to output ports.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted Inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecuteGraph</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

The ExecuteGraph component runs a subgraph with specific settings, waits for the graph to finish and monitors the graph results and tracking information.

The component can have single input port and two output ports attached. The component reads an input token, executes a subgraph based on incoming data values, waits for the subgraph to finish and sends results of successful subgraph to the first output port and results of failed subgraph to second output port (error port). If the graph run was successful the component continues with processing of next input tokens. Otherwise, component stops executing other graphs and from now all incoming tokens are ignored and information about ignored tokens are sent to error output port. This behaviour can be changed via attribute Stop processing on fail.
In case no input port is attached, only one graph is executed with default settings specified in the component's attributes. In case the first output port is not connected, the component just prints out the subgraph results to the log. In case the second output port (error port) is not attached, first failed subgraph causes interruption of the parent job.

For graph execution, it is necessary to specify subgraph location, execution type and optionally initial values of graph parameters and dictionary content, timeout, execution group and several other attributes. Most of these execution settings can be specified on the component via various component attributes described below. These settings could be considered as a default execution settings. However, these default execution settings can be dynamically changed individually for each graph execution based on data from incoming token. Input mapping attribute is the place where this override is defined.

After the subgraph is finished, results can be mapped to output ports. Output mapping and error mapping attributes define how the output tokens are populated. Information available in graph results comprise mainly from general runtime information, final dictionary content and tracking information.

Tip

Right-click the component and click Open Graph to access the graph that is executed. Similarly, in the ExecuteJobflow component, there is the Open Jobflow option.

## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input tokens with graph execution settings.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Execution information for successful subgraphs.</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Execution information for unsuccessful subgraphs.</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
### ExecuteGraph Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph URL</td>
<td>yes</td>
<td>Path to executed subgraph. Only single graph can be specified in this attribute. Can be overridden by a value from input token, see Input mapping attribute. The graph referenced by this attribute is also used for all mapping dialogs - they display dictionary entries and tracking information based on this graph.</td>
<td></td>
</tr>
<tr>
<td>Execution type</td>
<td>no</td>
<td>Specifies type of execution - synchronous (sequential) or asynchronous (parallel) execution model. Can be overridden by a value from input token, see Input mapping attribute. See [Execution type](p. 665).</td>
<td>synchronous (default)</td>
</tr>
<tr>
<td>Timeout</td>
<td>no</td>
<td>Maximal number of milliseconds dedicated for subgraph run. If the timeout interval elapses the subgraph is aborted. Can be overridden by a value from input token, see Input mapping attribute.</td>
<td>0 (unlimited)</td>
</tr>
<tr>
<td>Input mapping</td>
<td>no</td>
<td>Input mapping defines how data from incoming token overrides default execution settings. See [Input mapping](p. 666).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Output mapping</td>
<td>no</td>
<td>Output mapping maps results of successful graphs to the first output port. See [Output mapping](p. 666).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Error mapping</td>
<td>no</td>
<td>Error mapping maps results of unsuccessful graphs to the second output port. See [Error mapping](p. 668).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>By default, results of failed graphs are sent to the second output port (error port). If this switch is true, results of unsuccessful graphs are sent to the first output port in the same way as successful graphs.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution group</td>
<td>no</td>
<td>Name of execution group to which the executed subgraph belongs. Execution group can be used by KillGraph component as a named handler for a set of graphs to be interrupted.</td>
<td>string</td>
</tr>
<tr>
<td>Preferred cluster node ID</td>
<td>no</td>
<td>Cluster node ID which is preferred for execution of subgraph. The cluster node is not guaranteed, it's a preference.</td>
<td>string</td>
</tr>
<tr>
<td>Execute graph as daemon</td>
<td>no</td>
<td>By default, all subgraphs are executed in non-daemon mode, so none of them can live longer than parent graph. Clover server automatically ensures for finished jobs that all non-daemon subgraphs are interrupted if they are not finished yet. If you want to start a subgraph which can live longer than parent graph, set this switch to true.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Skip checkConfig</td>
<td>no</td>
<td>By default, the pre-execution configuration check of the subgraph is performed if and only if the check was performed on the parent job. This attribute can explicitly enable or disable the check.</td>
<td>boolean</td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, any failed subgraph causes the component stops executing other subgraphs and information about skipped tokens are sent to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

### Execution type
The component supports synchronous (sequential) and asynchronous (parallel) graph execution.

- synchronous execution mode (default) - the component blocks until the graph has finished processing, so graphs are monitored by this component until the end of run.

- asynchronous execution mode - the component starts the graph and immediately sends the status information to the output. The graphs are just started by this component, the component MonitorGraph should be used for monitoring asynchronously executed graphs.

### Input mapping

Input mapping attribute allows to override the settings of the component based on data from incoming token. Moreover, initial dictionary content and graph parameters of executed graph can be changed in input mapping.

Input mapping is regular CTL transformation which is executed before each subgraph execution. Input token if any is only input for this mapping and outputs are up to three records, RunConfig, GraphParameters and Dictionary.

- **RunConfig** record represents execution settings. If a field of the record is not populated by this mapping, default value from respective attribute of the component is used instead.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobURL</td>
<td>string</td>
<td>Overrides component attribute Graph URL.</td>
</tr>
<tr>
<td>executionType</td>
<td>string</td>
<td>Overrides component attribute Execution type.</td>
</tr>
<tr>
<td>timeout</td>
<td>long</td>
<td>Overrides component attribute Timeout.</td>
</tr>
<tr>
<td>executionGroup</td>
<td>string</td>
<td>Overrides component attribute Execution group.</td>
</tr>
<tr>
<td>clusterNodeId</td>
<td>string</td>
<td>Overrides component attribute Preferred cluster node ID.</td>
</tr>
<tr>
<td>daemon</td>
<td>boolean</td>
<td>Overrides component attribute Execute graph as daemon.</td>
</tr>
<tr>
<td>skipCheckConfig</td>
<td>boolean</td>
<td>Overrides component attribute Skip checkConfig.</td>
</tr>
</tbody>
</table>

- **GraphParameters** record represents all internal and external graph parameters of triggered graph.

- **Dictionary** record represents input dictionary entries of triggered graph.

**Note**

GraphParameters and Dictionary records are available in transform dialog only if component attribute Graph URL links to an existing graph which is used as a template for extraction of graph parameters and dictionary structure. Only graph parameters and input dictionary entries from this graph can be populated by input mapping, no matter which graph will be actually executed in runtime.

### Output mapping

Output mapping is regular CTL transformation which is used to populate token passed to the first output port. The mapping is executed for successful graphs. Up to four input data records are available for this mapping.

- The input token based on which the graph was executed (is not available for component usage without input connector), this is very helpful for passing through some fields from input token to output token. This record has Port 0 displayed in the Type column.

- **RunStatus** record provides information about graph execution.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| runId      | long    | Unique identifier of subgraph run. In case of asynchronous execution type the value can be used for graph monitoring or interruption.
### Field Name | Type | Description
---|---|---
originalJobURL | string | Path to executed subgraph.
startTime | date | Time of graph execution.
endTime | date | Time of graph finish or null for asynchronous execution.
duration | long | Graph execution time in milliseconds.
status | string | Final graph execution status (FINISHED_OK | ERROR | ABORTED | TIMEOUT | RUNNING for asynchronous execution).
errException | string | Cause exception for failed graphs only.
errMessage | string | Error message for failed graphs only.
errComponent | string | Component ID which caused graph fail.
errComponentType | string | Type of component which caused graph fail.

• Dictionary record provides content of output dictionary entries of the subgraph. This record is available for the mapping only if the attribute ‘Graph URL’ refers to a subgraph instance, which has an output dictionary entry.

• Tracking record provides tracking information usually available only in JMX interface of the running graph. This record is available for the mapping only if the attribute ‘Graph URL’ refers to a subgraph instance.

• Tracking fields available for whole graph:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startTime</td>
<td>date</td>
<td>Time of graph execution.</td>
</tr>
<tr>
<td>endTime</td>
<td>date</td>
<td>Time of graph finish or null for running graph.</td>
</tr>
<tr>
<td>executionTime</td>
<td>long</td>
<td>Graph execution time in milliseconds.</td>
</tr>
<tr>
<td>graphName</td>
<td>string</td>
<td>Name of executed graph.</td>
</tr>
<tr>
<td>result</td>
<td>string</td>
<td>Graph execution status (FINISHED_OK</td>
</tr>
<tr>
<td>runningPhase</td>
<td>integer</td>
<td>Index of running phase or null if graph is already finished.</td>
</tr>
<tr>
<td>usedMemory</td>
<td>integer</td>
<td>Memory footprint (in bytes) of executed graph.</td>
</tr>
</tbody>
</table>

• Tracking fields available for a graph phase:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startTime</td>
<td>date</td>
<td>Time of phase execution.</td>
</tr>
<tr>
<td>endTime</td>
<td>date</td>
<td>Time of phase finish or null for running phase.</td>
</tr>
<tr>
<td>executionTime</td>
<td>long</td>
<td>Phase execution time in milliseconds.</td>
</tr>
<tr>
<td>memoryUtilization</td>
<td>string</td>
<td>Graph memory footprint (in bytes).</td>
</tr>
<tr>
<td>result</td>
<td>string</td>
<td>Phase execution status (FINISHED_OK</td>
</tr>
</tbody>
</table>

• Tracking fields available for a component:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>Name of the component.</td>
</tr>
</tbody>
</table>
### Chapter 55. Job Control

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>usageCPU</td>
<td>number</td>
<td>Actual CPU time used by the component expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).</td>
</tr>
<tr>
<td>usageUser</td>
<td>number</td>
<td>Actual CPU time used by the component in user mode expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).</td>
</tr>
<tr>
<td>peakUsageCPU</td>
<td>number</td>
<td>Maximal CPU time used by the component expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).</td>
</tr>
<tr>
<td>peakUsageUser</td>
<td>number</td>
<td>Maximal CPU time used by the component in user mode expressed by number from interval (0, 1) (0 means 0% of CPU is used by the component, 1 means 100% of CPU is used by the component).</td>
</tr>
<tr>
<td>totalCPUTime</td>
<td>long</td>
<td>Number of milliseconds of CPU time used by this component.</td>
</tr>
<tr>
<td>totalUsetTime</td>
<td>number</td>
<td>Number of milliseconds of CPU time in user mode used by this component.</td>
</tr>
<tr>
<td>memoryUtilization</td>
<td>long</td>
<td>Graph memory footprint (in bytes).</td>
</tr>
<tr>
<td>result</td>
<td>string</td>
<td>Component execution status (FINISHED_OK</td>
</tr>
<tr>
<td>usedMemory</td>
<td>integer</td>
<td>Memory footprint (in bytes) of this component. Only experimental implementation.</td>
</tr>
</tbody>
</table>

- Tracking fields available for an input or output port:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteFlow</td>
<td>integer</td>
<td>Number of bytes passed through this port per seconds.</td>
</tr>
<tr>
<td>bytePeak</td>
<td>integer</td>
<td>Maximal byteFlow registered on this port.</td>
</tr>
<tr>
<td>totalBytes</td>
<td>long</td>
<td>Number of bytes passed through this port.</td>
</tr>
<tr>
<td>recordFlow</td>
<td>integer</td>
<td>Number of records passed through this port per seconds.</td>
</tr>
<tr>
<td>recordPeak</td>
<td>integer</td>
<td>Maximal recordFlow registered on this port.</td>
</tr>
<tr>
<td>totalRecords</td>
<td>integer</td>
<td>Number of records passed through this port.</td>
</tr>
<tr>
<td>waitingRecords</td>
<td>integer</td>
<td>Number of records cached on the edge connected to the port.</td>
</tr>
<tr>
<td>averageWaitingRecords</td>
<td>integer</td>
<td>Average number of records cached on the edge connected to the port.</td>
</tr>
<tr>
<td>usedMemory</td>
<td>integer</td>
<td>Memory footprint (in bytes) of the attached edge.</td>
</tr>
</tbody>
</table>

### Error mapping

Attitude of error mapping is almost identical to output mapping. Unlike the output mapping this error mapping is used if and only if the graph finished unsuccessfully and the second output port is populated instead of the first one.
Chapter 55. Job Control

ExecuteJobflow

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Tip

If you drag a *.jbf file and drop it into the jobflow pane, you will add the ExecuteJobflow component.

Short Summary

ExecuteJobflow allows running of jobflows with user-specified settings and provides execution results and tracking details to output ports.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecuteJobflow</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

This component works similarly to ExecuteGraph. See ExecuteGraph (p. 663) component documentation.

Icon

Ports

Please refer to ExecuteGraph Ports (p. 664).
ExecuteJobflow Attributes

Please refer to ExecuteGraph Attributes (p. 665).
ExecuteScript

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

Short Summary

ExecuteScript is a component that runs either shell scripts or scripts interpreted by a selected interpreter. It either runs a script only once or the script is executed repeatedly for each incoming record. Each incoming record can redefine almost all parameters of run including the script itself.

Abstract

ExecuteScript runs a script with a given interpreter (default system shell by default).

When there is no edge connected to an input port, the component runs a script only once. One output record is produced in this case.

When there are records coming to an input port, one script execution per record is performed and one output record per script execution is produced.

If the script was successful the component continues with processing of next input tokens. Otherwise, component stops executing other scripts and from now all incoming tokens are ignored and information about ignored tokens are sent to error output port. This behaviour can be changed in parameter Stop processing on fail.

Output record contains all important information about a script run (times, exit value, error reports and standard output). Mapping of these values to user-defined output metadata can be defined in Output Mapping and Error Mapping attributes.

All script execution parameters can be set via input records with use of Input Mapping attribute. The mapping sets which values for the input are used as script execution parameters. Input and output mapping are common to job control (p. 656) components.

A single run of a script is performed as follows:

- The script code is copied to a temporary batch file.
- An interpreter is run with a ${} string substituted with the name of the temporary file
- After a successful run or interruption with timeout the output record is produced depending on connection to output ports and Redirect error output setting.

For more detailed information see attribute description (p. 673).
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Parameters of script run (including a script itself if needed).</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Component input and results of a script run.</td>
<td>Any</td>
</tr>
<tr>
<td>Error</td>
<td>1</td>
<td>no</td>
<td>Component input and results of a script run. Records for scripts that cannot run or return 1.</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has [Metadata Templates](p. 264) available.
## ExecuteScript Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Script</td>
<td>no</td>
<td>Code of a script to be executed. If an interpreter attribute value is kept default, script must be a code of a shell script. Thus, it can easily be used for running one or more system commands. Otherwise, the code format depends on a chosen interpreter.</td>
<td></td>
</tr>
<tr>
<td>Script URL</td>
<td>no</td>
<td>URL of a script to be executed. If an interpreter attribute value is kept default, script must be a code of a shell script. Thus, it can easily be used for running one or more system commands. Otherwise, the code format depends on a chosen interpreter. If both, script and script URL attributes are specified, only script URL is used.</td>
<td></td>
</tr>
<tr>
<td>Script charset</td>
<td>no</td>
<td>This character encoding is used for an executed script file. A script file reference is specified either in 'Script URL' attribute or temporary batch file is created automatically from 'Script' attribute.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Working directory</td>
<td>no</td>
<td>Working directory of the executed script. All relative paths used inside the script will be interpreted with respect the this directory. By default it is set to the root of CloverETL project containing the graph.</td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td>no</td>
<td>A limit for script execution (in milliseconds). When a script runs longer than the limit the components kills it. In this case in the output record fields are set as follows: exitValue is set to 1, reachedTimeout to true and duration is greater or equal to timeout.</td>
<td>0 (unlimited)</td>
</tr>
<tr>
<td>Input Mapping</td>
<td>no</td>
<td>Input mapping defines how data from incoming token overrides default component settings. See input mapping fields description (p. 674)</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Output Mapping</td>
<td>no</td>
<td>Output mapping maps results of successful script executions to the first output port. See output mapping fields description (p. 675)</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Error Mapping</td>
<td>no</td>
<td>Error mapping maps results of unsuccessful scripts to the second output port. See output mapping fields description (p. 675)</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Redirect Error Output</td>
<td>no</td>
<td>By default, results of failed scripts are sent to the second output port (error port). If this switch is true, results of unsuccessful scripts are sent to the first output port in the same way as successful scripts.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Interpreter</td>
<td>no</td>
<td>Set an interpreter to be used for running a script. When an interpreter is executed $() is substituted with a name of a temporary batch file that contains a copy of the script. If an interpreter is sensitive to an extension of a script file, it is necessary to set Batch file extension property so that a temporary file will have the right extension.</td>
<td>A path to an interpreter followed by $(). By default a script is interpreted by a system shell (e.g. cmd in Windows and sh in Linux).</td>
</tr>
</tbody>
</table>
### Chapter 55. Job Control

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment variables</td>
<td>no</td>
<td>Sets environment variables values in the script. It allows either to setting or appending to them. Appending to a non-existing variable leads to defining it and setting its value. Note that variable values are only visible inside of a script, i.e. setting PATH cannot be used for setting path to an interpreter. Variable values set in this property can be overridden by mapping of input to EnvironmentVariables metadata in an input mapping dialog.</td>
<td></td>
</tr>
<tr>
<td>Standard input</td>
<td>no</td>
<td>Contents of standard input that will be sent to the script. Be aware that if the script expects more input lines than available, it may hang.</td>
<td>string</td>
</tr>
<tr>
<td>Standard input file URL</td>
<td>no</td>
<td>File URL to contents of standard input that will be sent to the script. Be aware that if the script expects more input lines than available, it may hang.</td>
<td></td>
</tr>
<tr>
<td>Standard output file URL</td>
<td>no</td>
<td>File URL of a file to store standard output of the script. The file content is either rewritten or appended depending on the append flag.</td>
<td></td>
</tr>
<tr>
<td>Standard error file URL</td>
<td>no</td>
<td>File URL of a file to store error output of the script. The file content is either rewritten or appended depending on the append flag.</td>
<td></td>
</tr>
<tr>
<td>Append</td>
<td>no</td>
<td>Sets whether standard output and error output written into files (attributes Standard output file URL and Error output file URL) should rewrite existing content or it should be appended.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Data charset</td>
<td>no</td>
<td>Character encoding used to encode standard input passed from input port and to decode standard and error output to be passed to output ports.</td>
<td>ISO-8859-1</td>
</tr>
<tr>
<td>Batch file extension</td>
<td>no</td>
<td>Sets an extension of a batch file that is given to thr interpreter (its name is substituted for ${} in the interpreter setting).</td>
<td>bat</td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, any failed script causes the component stops executing other scripts and information about skipped tokens are sent to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

**Note**

The contents of script attribute are copied to a temporary batch file. On Microsoft Windows, it is often useful to start the script with `@echo off` to disable echoing the executed commands.

### Input Mapping Fields Description

Input records can be mapped to two different metadata: RunConfig and EnvironmentVariables.

Fields of RunConfig have the following functionality:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>script</td>
<td>Overrides component attribute Script</td>
<td></td>
</tr>
<tr>
<td>scriptURL</td>
<td>Overrides component attribute Script URL</td>
<td></td>
</tr>
<tr>
<td>scriptCharset</td>
<td>Overrides component attribute Script charset</td>
<td></td>
</tr>
<tr>
<td>interpreter</td>
<td>Overrides component attribute Interpreter</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 55. Job Control

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>workingDirectory</td>
<td>Overrides component attribute Working Directory</td>
</tr>
<tr>
<td>timeout</td>
<td>Overrides component attribute Timeout</td>
</tr>
<tr>
<td>environmentVariables</td>
<td>Overrides component attribute Environment Variables. It is expected that the value contains a list of variable assignments delimited with &quot;;&quot;. An assignment with simple &quot;=&quot; symbol assigns a value to an assigned environment variable. An assignment with &quot;+=&quot; symbol appends a value to an assigned environment variable.</td>
</tr>
<tr>
<td>stdIn</td>
<td>Overrides component attribute Standard Input</td>
</tr>
<tr>
<td>stdInFileURL</td>
<td>Overrides component attribute Standard Input File URL</td>
</tr>
<tr>
<td>stdOutFileURL</td>
<td>Overrides component attribute Standard Output File URL</td>
</tr>
<tr>
<td>errOutFileURL</td>
<td>Overrides component attribute Error Output File URL</td>
</tr>
<tr>
<td>append</td>
<td>Overrides component attribute Append</td>
</tr>
<tr>
<td>dataCharset</td>
<td>Overrides component attribute Data charset</td>
</tr>
<tr>
<td>redirectErrorOutput</td>
<td>Overrides component attribute Redirect Error Output</td>
</tr>
<tr>
<td>batchFileExtension</td>
<td>Overrides component attribute Batch File Extension</td>
</tr>
</tbody>
</table>

### Note

In **Input mapping**, you can use the `$out.0.script` field to create a dynamic command line. Just map a script and its parameters onto the field. Example:

```
$out.0.script = "md5.exe " + $in.0.filePath;
```

### Note

**Environment variables** provided to the executed script can be defined in three different ways.

1. Use component's attribute **Environmental variables** for statical definition of environment variables. Variable names and values are defined once for all script executions.

2. The output record **EnvironmentVariables** populated in **Input mapping** is the second way how the environment variables can be defined. Set of variable names is still statically defined by the record structure, but values of variables can be derived from input tokens.

3. The most complex way how the environment variables can be defined is to populate **environmentVariables** field in the output record **RunConfig** in **Input mapping**. Value of this field has same syntax and meaning like component's attribute **Environment variables**. Both, set of variables and their values can be defined fully dynamically in this case.

### Note

If you want to append a string to an environment variable in **Input mapping**, use **getEnvironmentVariables()** CTL function. Example:

```
$out.1.PATH = getEnvironmentVariables()['PATH'] + ":" + $in.0.additionalPath;
```

### Output Mapping Fields Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdOut</td>
<td>Standard output of a script.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>errOut</td>
<td>Error output of a script.</td>
</tr>
<tr>
<td>startTime</td>
<td>Start time of a script.</td>
</tr>
<tr>
<td>stopTime</td>
<td>Stop time of a script.</td>
</tr>
<tr>
<td>duration</td>
<td>Duration of a script. (duration = stopTime - startTime)</td>
</tr>
<tr>
<td>exitValue</td>
<td>Value returned by the script. Typically 0 means no error, non-zero values stand for errors.</td>
</tr>
<tr>
<td>reachedTimeout</td>
<td>Boolean determining whether the script reached timeout.</td>
</tr>
<tr>
<td>errException</td>
<td>If the script call finished with error, it may contain an exception that caused the error. This happens only in situation when the script has not started (e.g. path to the script is not valid) or its run has been interrupted (e.g. when a timeout has been reached).</td>
</tr>
<tr>
<td>errMessage</td>
<td>Message reported by the exception in errException.</td>
</tr>
</tbody>
</table>
Fail

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, **Common Properties of All Components** (p. 256)
- Chapter 40, **Common Properties of Most Components** (p. 264)
- Chapter 48, **Common Properties of Job Control** (p. 322)

If you want to find the right Job Control component for your purposes, see **Job control Comparison** (p. 322).

The component is located in Palette → Job Control.

**Short Summary**

The **Fail** component aborts the parent job (jobflow or graph) with user-specified error message.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

**Abstract**

**Fail** interrupts parent job (jobflow or graph). First incoming token to the component throws exception (org.jetel.exception.UserAbortException) with user defined error message. The job finishes immediately with final status ERROR. Moreover, the dictionary content can be changed before the job is interrupted. In general, the component allows to interrupt the job and at the same time return some results through dictionary.

The component Fail works even without input port attached. In this case, the job is interrupted immediately when the phase with the Fail component is started up.

**Icon**

![Fail Icon](image)

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Any input token from this port interrupts the jobflow (or graph).</td>
<td>Any 1</td>
</tr>
</tbody>
</table>

**Legend:**
1): Input field named 'errorMessage' is automatically used for user-specific error message, which interrupts the job, if it is not specified in mapping other way.
Chapter 55. Job Control

Fail Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error message</td>
<td>no</td>
<td>In case the job is interrupted, the exception is thrown with this error message. The error message can be dynamically changed in mapping.</td>
<td>“user abort” (default)</td>
</tr>
<tr>
<td>Mapping</td>
<td>no</td>
<td>Mapping is used for dynamical assembling of error message, which is thrown in case the job is going to be interrupted. Moreover, dictionary content of interrupted job can be changed as well. See Advanced Description (p. 679).</td>
<td></td>
</tr>
</tbody>
</table>

Advanced Description

- Mapping details

  Mapping in Fail component is generally used for two purposes:

  - Assembling of error message from incoming record.
  - Populating dictionary content from incoming record.

  **Note**

  Only output dictionary entries can be changed.

  ![Figure 55.2. Example of mapping for Fail component](image)

  Error message compiled by the mapping has the highest priority. If the mapping does not set ‘errorMessage’, the error message from component attribute is used instead. If even this attribute is not set, predefined text ”user abort” is used instead.
Chapter 55. Job Control

GetJobInput

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

GetJobInput produces single record populated by dictionary content and/or graph parameters.

Abstract

The component GetJobInput retrieves requested job parameters and sends them to output port. The component produces single output record which is populated by mapping. Auto-mapping is applied when mapping is not specified, e.g. input dictionary entries are automatically copied to output fields with identical names.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For record with job input.</td>
</tr>
</tbody>
</table>

Legend:

1): fields with identical names with input dictionary entries are populated automatically
## GetJobInput Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
</table>
| Basic     |     | **Mapping**  
Mapping populates the output record of the component. Input dictionary entries and graph parameters are natural input values for the mapping. In fact, mapping attribute is a regular CTL transformation from record, which represents input dictionary entries, to record with output metadata structure. Mapping is invoked exactly once. |                 |
KillGraph

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

KillGraph aborts specified graphs and passes their final status to output port.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KillGraph</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

The KillGraph component aborts graphs specified by run ID or by execution group (all graphs belonging to the execution group are aborted). Final execution status of interrupted graphs is passed to output port or just printed out to log. Moreover, you can choose if even daemon children of interrupted graphs are aborted (non-daemon children are interrupted in any case) - see Execute graph as daemon attribute of ExecuteGraph (p. 663).

The component reads input token, extracts run ID or execution group from incoming data (see Input mapping attribute), interrupts the requested graphs and writes final status of interrupted graph to the output port (see Output mapping attribute).

In case the input port is not attached, just the graphs specified in Run ID attribute or in Execution group attribute are interrupted.

Icon
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input tokens with identifications of interrupted graphs.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Final graph execution status.</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has [Metadata Templates](p. 264) available.
KillGraph Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run ID</td>
<td>no</td>
<td>Specifies run ID of interrupted graph. Has higher priority than 'Execution group' attribute. This attribute can be overridden in input mapping.</td>
</tr>
<tr>
<td>Execution group</td>
<td>no</td>
<td>All graphs belonging to the specified execution group are interrupted. 'Run ID' attribute has higher priority. This attribute can be overridden in input mapping.</td>
</tr>
<tr>
<td>Kill daemon children</td>
<td>no</td>
<td>Specifies whether even daemon children are interrupted. Non-daemon children are aborted in any case. This attribute can be overridden in input mapping.</td>
</tr>
<tr>
<td>Input mapping</td>
<td>no</td>
<td>Input mapping defines how to extract run ID or execution group to be interrupted from incoming token. See <em>Input mapping</em> (p. 684).</td>
</tr>
<tr>
<td>Output mapping</td>
<td>no</td>
<td>Output mapping defines how to populate the output token by final graph status of interrupted graph. See <em>Output mapping</em> (p. 684).</td>
</tr>
</tbody>
</table>

Input mapping

Input mapping is regular CTL transformation which is executed for each input token to extract run ID or execution group to be interrupted. Output record has following structure:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runId</td>
<td>long</td>
<td>Overrides component attribute Run ID</td>
</tr>
<tr>
<td>executionGroup</td>
<td>string</td>
<td>Overrides component attribute Execution group</td>
</tr>
<tr>
<td>killDaemonChildren</td>
<td>boolean</td>
<td>Overrides component attribute Kill daemon children</td>
</tr>
</tbody>
</table>

Output mapping

Output mapping is regular CTL transformation which is executed for interrupted graph to populate the output token. Available input data has following structure:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runId</td>
<td>long</td>
<td>run ID of interrupted graph</td>
</tr>
<tr>
<td>originalJobURL</td>
<td>string</td>
<td>path to interrupted graph</td>
</tr>
<tr>
<td>version</td>
<td>string</td>
<td>version of interrupted graph</td>
</tr>
<tr>
<td>startTime</td>
<td>date</td>
<td>time of graph execution</td>
</tr>
<tr>
<td>endTime</td>
<td>date</td>
<td>time of graph finish</td>
</tr>
<tr>
<td>duration</td>
<td>long</td>
<td>graph run execution time in milliseconds (endTime - startTime)</td>
</tr>
<tr>
<td>status</td>
<td>string</td>
<td>final graph execution status (FINISHED_OK</td>
</tr>
<tr>
<td>errException</td>
<td>string</td>
<td>cause exception for failed graphs</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>errMessage</td>
<td>string</td>
<td>error message for failed graphs</td>
</tr>
<tr>
<td>errComponent</td>
<td>string</td>
<td>component ID which caused graph fail</td>
</tr>
<tr>
<td>errComponentType</td>
<td>string</td>
<td>type of component which caused graph fail</td>
</tr>
</tbody>
</table>
KillJobflow

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

KillJobflow aborts specified jobflows and passes their final status to output port.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>KillJobflow</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

This component works similarly to KillGraph. See KillGraph (p. 682) component documentation.

Icon

Ports

Please refer to KillGraph Ports (p. 683).
KillJobflow Attributes

Please refer to KillGraph Attributes (p. 684).
MonitorGraph

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette \(\rightarrow\) Job Control.

Short Summary

MonitorGraph allows watching of running graphs. Component can either wait for final execution status or periodically monitor current execution status.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonitorGraph</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

The MonitorGraph component allows watching of running graphs. Each incoming token triggers new monitor of a graph specified by run ID extracted from the token. It is possible to monitor multiple graphs at once.

A single graph monitor watches the graph and waits for it to finish. When the graph is finished running, graph results are sent to an output port in the same manner as in ExecuteGraph component; results of successful graphs are sent to the first output port and unsuccessful graphs are sent to the second output port. Moreover whenever time specified in the monitoring interval attribute elapses, the graph monitors send current graph status information even for still running graphs.

In case no input port is attached, only one graph is monitored with the settings specified in component's attributes. In case the first output port is not connected, the component just prints out the subgraph results to the log. In case the second output port (error port) is not attached, first subgraph that fails would cause interruption of the parent job.

Input mapping defines how to extract run ID and other settings of the graph monitor from incoming token. Whenever graph results or actual graph status need to be mapped to output ports, output mapping and error mapping attributes are used to populate output tokens. Information available in graph results comprise mainly from general runtime information, dictionary content and tracking information.
Note

Only graphs executed by the current jobflow (direct children) can be watched by the MonitorGraph component.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input tokens with identification of monitored graph.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Execution information for successful graphs.</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Execution information for unsuccessful graphs.</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
# MonitorGraph Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph URL</td>
<td>no</td>
<td>Path to a graph which represents typical monitored graph. The graph referenced by this attribute is also used for all mapping dialogs - they display dictionary entries and tracking information based on this graph.</td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td>no</td>
<td>Maximal number of milliseconds dedicated for graph run. If the graph is running longer than the time specified in this attribute, current graph information with TIMEOUT status is send to error output port. This is just default value for all graph monitors. Can be overridden in input mapping individually for each graph monitor.</td>
<td>0 (unlimited)</td>
</tr>
<tr>
<td>Monitoring interval</td>
<td>no</td>
<td>Whenever time specified in this attribute elapses, the graph monitor send actual graph status information to the first output port. By default, only final graph results are sent to output ports. This is just default value for all graph monitors. Can be overridden in input mapping individually for each graph monitor.</td>
<td>none (default)</td>
</tr>
<tr>
<td>Input mapping</td>
<td>no</td>
<td>Input mapping defines how to extract run ID and other graph monitor settings from incoming token. See Input mapping (p. 690).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Output mapping</td>
<td>no</td>
<td>Output mapping maps results of successful graphs to the first output port. Output mapping is used also for sending of current status in case the monitoring interval is specified. See Output mapping (p. 691).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Error mapping</td>
<td>no</td>
<td>Error mapping maps results of unsuccessful graphs to the second output port. See Error mapping (p. 691).</td>
<td>CTL transformation</td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>By default, results of failed graphs are sent to the second output port (error port). If this switch is true, results of unsuccessful graphs are sent to the first output port in the same way as successful graphs.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run ID</td>
<td>no</td>
<td>Statically defined run ID of monitored graph. This attribute is usually overridden in input mapping by data from incoming token.</td>
<td>string</td>
</tr>
</tbody>
</table>

## Input mapping

Input mapping is regular CTL transformation which is executed for each incoming token to specify run ID of monitored graph and settings of respective graph monitor. Available output fields:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runId</td>
<td>long</td>
<td>Run ID of monitored graph. Overrides component attribute Run ID.</td>
</tr>
</tbody>
</table>
### Output mapping

Output mapping is regular CTL transformation which is used to populate token passed to the first output port. The mapping is executed for successful graphs or for current status of still running graphs, which is sent in case monitoring interval is specified. More details about input records for this output mapping is available in documentation for [ExecuteGraph](#) component.

The graph monitor finishes watching the graph after the graph is complete or timeout elapses. Another option how to stop graph monitoring is to return STOP constant in output mapping.

### Error mapping

Error mapping is almost identical to output mapping. This error mapping is used only if the graph finished unsuccessfully or timeout elapsed. The second output port is populated by error mapping.
MonitorJobflow

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

MonitorJobflow allows watching of running jobflows. Component can either wait for final execution status or periodically monitor current execution status.

Note

To be able to use this component, you need a separate jobflow license. Also, the component requires your project is executed on Clover Server.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonitorJobflow</td>
<td>no</td>
<td>no</td>
<td>0-1</td>
<td>0-2</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

This component works similarly to MonitorGraph. See MonitorGraph (p. 688) component documentation.

Icon

Ports

Please refer to MonitorGraph Ports (p. 689).
MonitorJobflow Attributes

Please refer to MonitorGraph Attributes (p. 690).
SetJobOutput

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control component for your purposes, see Job control Comparison (p. 322).

The component is located in Palette → Job Control.

Short Summary

SetJobOutput receives input records and sets the incoming values to dictionary content.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetJobOutput</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

The component SetJobOutput writes incoming records to output dictionary entries. Output dictionary entries are populated according to mapping. If no mapping is specified, field values are set to dictionary entries with identical names.

First input record sets values of dictionary entries, and subsequent input records override the existing values.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For records to be written to dictionary.</td>
<td>Any 1)</td>
</tr>
</tbody>
</table>

Legend:

1): field values with identical names with output dictionary entries are written automatically
### SetJobOutput Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td><strong>Mapping</strong>&lt;br&gt;This attribute specifies mapping from input record metadata to output dictionary entries. Each incoming record is processed by this mapping and its values are mapped to dictionary. In fact, mapping attribute is a regular CTL transformation from input metadata structure to record, which represents output dictionary entries.</td>
<td></td>
</tr>
</tbody>
</table>


Success

Jobflow Component

We assume that you have already learned what is described in:
- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

The component is located in Palette → Job Control.

Short Summary

Success is a successful endpoint in a jobflow.

<table>
<thead>
<tr>
<th>Component</th>
<th>Data output</th>
<th>Input ports</th>
<th>Output ports</th>
<th>Transformation</th>
<th>Transf. required</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>none</td>
<td>0-1</td>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Abstract

Success is a successful endpoint in a jobflow. Tokens that flow into the component are not processed anymore - they are considered to be successfully processed within the jobflow. The component can serve as a visual marker of success in a jobflow.

The component can log a message and set contents of dictionary - it is similar to the Fail (p. 677) component.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0-1</td>
<td>yes</td>
<td>For received tokens</td>
<td>Any</td>
</tr>
</tbody>
</table>

Success Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced Description

- **Mapping details**

  Mapping in Success component is generally used for two purposes:

  - Assembling of log message from incoming record.
  - Populating dictionary content from incoming record.

  **Note**

  Only output dictionary entries can be changed.

  Log message compiled by the mapping has the highest priority. If the mapping does not set 'message', the message from component attribute is used instead. If no message is set via attribute or mapping, nothing is logged.
TokenGather

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 48, Common Properties of Job Control (p. 322)

If you want to find the right Job Control for your purposes, see Job control Comparison (p. 322).

**Short Summary**

TokenGather copies each incoming token from any input port to all connected output ports. If input metadata differs from output metadata copying based on field names is used. This component is typically used to collect all tokens from several parallel execution branches and send them to one unified output.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TokenGather</td>
<td>no</td>
<td>no</td>
<td>1-n</td>
<td>1-n</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Abstract**

The TokenGather component receives incoming tokens from any input port and copies them to all connected output ports - each incoming token is copied to all output ports. Input ports and output ports can have any metadata. Copying from input metadata to output metadata is based on field names - field value is moved to output token if and only if output token has field with identical name.

**Icon**

This component has Metadata Templates (p. 264) available.

**Legend:**

Metadata Templates (p. 264)
1): only fields with identical names with input fields are populated
Chapter 56. File Operations

We assume that you already know what components are. See Chapter 18, Components (p. 94) for brief information.

The group of components designed for file system manipulation is called File Operations.

File Operations components can create, copy, move, delete files and directories, list directories, and read file attributes. The components can work with local files and remote files via FTP. Access to sandboxes is also supported when running on the Server. It also offers limited support on other protocols (e.g. copy files from the web using the HTTP protocol); however, archived content manipulation is not supported (e.g. zip, gzip and tar protocols).

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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 49, Common Properties of File Operations (p. 323)
Chapter 56. File Operations

CopyFiles

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)
• Chapter 49, Common Properties of File Operations (p. 323)

If you want to find the right File Operation component for your purposes, see File Operations Comparison (p. 323).

The component is located in Palette → File Operations.

Short Summary

CopyFiles can be used to copy files and directories.

Note

To be able to use this component, you need a separate jobflow license.

<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyFiles</td>
<td>0-1</td>
<td>0-2</td>
</tr>
</tbody>
</table>

Abstract

CopyFiles can copy multiple sources into one destination directory or a regular source file to a target file. Directories can be copied recursively. Optionally, existing files may be skipped or updated based on the modification date of the files.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input data records to be mapped to component attributes.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Results</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Errors</td>
<td>Any</td>
</tr>
</tbody>
</table>
This component has Metadata Templates (p. 264) available.
## CopyFiles Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Basic</strong></td>
<td></td>
</tr>
<tr>
<td>Source file URL</td>
<td>yes</td>
<td>Path to the source file or directory (see [Supported URL Formats for File Operations](p. 324)).</td>
<td></td>
</tr>
<tr>
<td>Target file URL</td>
<td>yes</td>
<td>Path to the destination file or directory (see [Supported URL Formats for File Operations](p. 324)). When it points to a directory, the source will be copied into the directory. It must be a path to a single file or directory.</td>
<td></td>
</tr>
<tr>
<td>Recursive</td>
<td>no</td>
<td>Copy directories recursively.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Overwrite</td>
<td>no</td>
<td>Specifies whether existing files shall be overwritten. In update mode, the target will be overwritten only when the source file is newer than the destination file.</td>
<td>always (default)</td>
</tr>
<tr>
<td>Create parent directories</td>
<td>no</td>
<td>Attempt to create non-existing parent directories.</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the Create parent directories option is enabled and the Target file URL ends with a slash ('/'), it is treated as the parent directory, i.e. the source directory or file is copied into the target directory, even if it does not exist.</td>
<td></td>
</tr>
<tr>
<td>Input mapping</td>
<td>2)</td>
<td>Defines mapping of input records to component attributes.</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Defines mapping of results to standard output port.</td>
<td></td>
</tr>
<tr>
<td>Error mapping</td>
<td>2)</td>
<td>Defines mapping of errors to error output port.</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>If enabled, errors will be sent to the output port instead of the error port.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Verbose output</td>
<td>no</td>
<td>If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.</td>
<td>false (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, a failure causes the component to skip all subsequent operations and send the information about skipped operations to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

### Legend

1) The attribute is required, unless specified in the Input mapping.

2) Required if the corresponding edge is connected.
Advanced Description

Editing any of the Input, Output or Error mapping opens the Transform Editor (p. 275).

**Input mapping** - the editor allows you to override selected attributes of the component with the values of the input fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Attribute</th>
<th>Type</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceURL</td>
<td>Source file URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>targetURL</td>
<td>Target file URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td>Recursive</td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td>overwrite</td>
<td>Overwrite</td>
<td>string</td>
<td>&quot;always&quot;</td>
</tr>
<tr>
<td>makeParentDirs</td>
<td>Create parent directories</td>
<td>boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

**Output mapping** - the editor allows you to map the results and the input data to the output port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceURL</td>
<td>string</td>
<td>URL of the source file.</td>
</tr>
<tr>
<td>targetURL</td>
<td>string</td>
<td>URL of the destination.</td>
</tr>
<tr>
<td>resultURL</td>
<td>string</td>
<td>New URL of the successfully copied file. Only set in <em>Verbose output</em> mode.</td>
</tr>
<tr>
<td>result</td>
<td>boolean</td>
<td>True if the operation has succeeded (can be false when <em>Redirect error output</em> is enabled).</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>If the operation has failed, the field contains the error message (used when <em>Redirect error output</em> is enabled).</td>
</tr>
</tbody>
</table>

**Error mapping** - the editor allows you to map the errors and the input data to the error port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>boolean</td>
<td>Will always be set to false.</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>The error message.</td>
</tr>
<tr>
<td>sourceURL</td>
<td>string</td>
<td>URL of the source file.</td>
</tr>
<tr>
<td>targetURL</td>
<td>string</td>
<td>URL of the destination.</td>
</tr>
</tbody>
</table>
CreateFiles

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 49, Common Properties of File Operations (p. 323)

If you want to find the right File Operation component for your purposes, see File Operations Comparison (p. 323).

The component is located in Palette → File Operations.

Short Summary

CreateFiles can be used to create files and directories and to set their modification date.

Note

To be able to use this component, you need a separate jobflow license.

<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateFiles</td>
<td>0-1</td>
<td>0-2</td>
</tr>
</tbody>
</table>

Abstract

CreateFiles can create files and directories. It is also capable of setting the modification date of both existing and newly created files and directories. Optionally, non-existing parent directories may also be created.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input data records to be mapped to component attributes.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Results</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Errors</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
## CreateFiles Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Path to the file or directory to be created (see <a href="#">Supported URL Formats for File Operations</a> (p. 324)). If it ends with a slash ('/'), it denotes that a directory should be created, which can also be specified using the Create as directory attribute.</td>
</tr>
<tr>
<td>Create as directory</td>
<td>no</td>
<td>Specifies that directories should be created instead of regular files.</td>
</tr>
<tr>
<td>Create parent directories</td>
<td>no</td>
<td>Attempt to create non-existing parent directories.</td>
</tr>
<tr>
<td>Last modified date</td>
<td>no</td>
<td>Set the last modified date of existing and newly created files to the specified value. Format of the date is defined in the DEFAULT_DATETIME_FORMAT property (<a href="#">Changing Default CloverETL Settings</a> (p. 85)).</td>
</tr>
<tr>
<td>Input mapping</td>
<td>2)</td>
<td>Defines mapping of input records to component attributes.</td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Defines mapping of results to standard output port.</td>
</tr>
<tr>
<td>Error mapping</td>
<td>2)</td>
<td>Defines mapping of errors to error output port.</td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>If enabled, errors will be sent to the standard output port instead of the error port.</td>
</tr>
<tr>
<td>Verbose output</td>
<td>no</td>
<td>If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, a failure causes the component to skip all subsequent operations and send the information about skipped operations to the error output port. This behaviour can be turned off by this attribute.</td>
</tr>
</tbody>
</table>

### Legend

1) The attribute is required, unless specified in the Input mapping.

2) Required if the corresponding edge is connected.

### Advanced Description

Editing any of the Input, Output or Error mapping opens the Transform Editor (p. 275).

**Input mapping** - the editor allows you to override selected attributes of the component with the values of the input fields.
### Field Name | Attribute | Type | Possible values
--- | --- | --- | ---
fileURL | File URL | string | 
directory | Create as directory | boolean | true | false
makeParentDirs | Create parent directories | boolean | true | false
modifiedDate | Last modified date | date | 

**Output mapping** - the editor allows you to map the results and the input data to the output port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileURL</td>
<td>string</td>
<td>URL of the target file or directory.</td>
</tr>
<tr>
<td>result</td>
<td>boolean</td>
<td>True if the operation has succeeded (can be false when <strong>Redirect error output</strong> is enabled).</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>If the operation has failed, the field contains the error message (used when <strong>Redirect error output</strong> is enabled).</td>
</tr>
</tbody>
</table>

**Error mapping** - the editor allows you to map the errors and the input data to the error port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>boolean</td>
<td>Will always be set to false.</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>The error message.</td>
</tr>
<tr>
<td>fileURL</td>
<td>string</td>
<td>URL of the target file or directory.</td>
</tr>
</tbody>
</table>
Chapter 56. File Operations

DeleteFiles

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 49, Common Properties of File Operations (p. 323)

If you want to find the right File Operation component for your purposes, see File Operations Comparison (p. 323).

The component is located in Palette → File Operations.

Short Summary

DeleteFiles can be used to delete files and directories.

Note

To be able to use this component, you need a separate jobflow license.

```
<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteFiles</td>
<td>0-1</td>
<td>0-2</td>
</tr>
</tbody>
</table>
```

Abstract

DeleteFiles can be used to delete files and directories (also recursively).

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input data records to be mapped to component attributes.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Results</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Errors</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
## DeleteFiles Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes</td>
<td>Path to the file or directory to be deleted (see Supported URL Formats for File Operations (p. 324)).</td>
<td></td>
</tr>
<tr>
<td>Recursive</td>
<td>no</td>
<td>Delete directories recursively.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Input mapping</td>
<td>2)</td>
<td>Defines mapping of input records to component attributes.</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Defines mapping of results to standard output port.</td>
<td></td>
</tr>
<tr>
<td>Error mapping</td>
<td>2)</td>
<td>Defines mapping of errors to error output port.</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>If enabled, errors will be sent to the standard output port instead of the error port.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Verbose output</td>
<td>no</td>
<td>If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, a failure causes the component to skip all subsequent operations and send the information about skipped operations to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

**Legend**

1) The attribute is required, unless specified in the **Input mapping**.

2) Required if the corresponding edge is connected.

### Advanced Description

Editing any of the **Input, Output** or **Error mapping** opens the Transform Editor (p. 275).

**Input mapping** - the editor allows you to override selected attributes of the component with the values of the input fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Attribute</th>
<th>Type</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileURL</td>
<td>File URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td>Recursive</td>
<td>boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

**Output mapping** - the editor allows you to map the results and the input data to the output port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileURL</td>
<td>string</td>
<td>Path to the file or directory that was deleted.</td>
</tr>
<tr>
<td>result</td>
<td>boolean</td>
<td>True if the operation has succeeded (can be false when Redirect error output is enabled).</td>
</tr>
</tbody>
</table>
### Field Name | Type | Description
---|---|---
errorMessage | string | If the operation has failed, the field contains the error message (used when **Redirect error output** is enabled).

**Error mapping** - the editor allows you to map the errors and the input data to the error port.

### Field Name | Type | Description
---|---|---
result | boolean | Will always be set to false.
errorMessage | string | The error message.
fileURL | string | URL of the deleted file or directory.
ListFiles

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 49, Common Properties of File Operations (p. 323)

If you want to find the right File Operation component for your purposes, see File Operations Comparison (p. 323).

The component is located in Palette → File Operations.

Short Summary

ListFiles can be used to list directory contents and to retrieve file attributes, such as size or modification date.

Note

To be able to use this component, you need a separate jobflow license.

<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListFiles</td>
<td>0-1</td>
<td>1-2</td>
</tr>
</tbody>
</table>

Abstract

ListFiles lists directory contents including detailed information about individual files. Subdirectories may be listed recursively.

Icon

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input data records to be mapped to component attributes.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>One record per each entry in the target directory</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Errors</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
ListFiles Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File URL</td>
<td>yes 1)</td>
<td>Path to the file or directory to be listed (see Supported URL Formats for File Operations (p. 324)).</td>
<td></td>
</tr>
<tr>
<td>Recursive</td>
<td>no</td>
<td>List subdirectories recursively.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Input mapping</td>
<td>2)</td>
<td>Defines mapping of input records to component attributes.</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Defines mapping of results to standard output port.</td>
<td></td>
</tr>
<tr>
<td>Error mapping</td>
<td>2)</td>
<td>Defines mapping of errors to error output port.</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>If enabled, errors will be sent to the standard output port instead of the error port.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, a failure causes the component to skip all subsequent operations and send the information about skipped operations to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

Legend

1) The attribute is required, unless specified in the Input mapping.

2) Required if the corresponding edge is connected.

Advanced Description

Editing any of the Input, Output or Error mapping opens the Transform Editor (p. 275).

**Input mapping** - the editor allows you to override selected attributes of the component with the values of the input fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Attribute</th>
<th>Type</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileURL</td>
<td>File URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td>Recursive</td>
<td>boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

**Output mapping** - the editor allows you to map the results and the input data to the output port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>string</td>
<td>URL of the file or directory.</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>File name.</td>
</tr>
<tr>
<td>canRead</td>
<td>boolean</td>
<td>True if the file can be read.</td>
</tr>
<tr>
<td>canWrite</td>
<td>boolean</td>
<td>True if the file can be modified.</td>
</tr>
<tr>
<td>canExecute</td>
<td>boolean</td>
<td>True if the file can be executed.</td>
</tr>
<tr>
<td>isDirectory</td>
<td>boolean</td>
<td>True if the file exists and is a directory.</td>
</tr>
<tr>
<td>isFile</td>
<td>boolean</td>
<td>True if the file exists and is a regular file.</td>
</tr>
</tbody>
</table>
### Field Name | Type | Description
--- | --- | ---
isHidden | boolean | True if the file is hidden.
lastModified | date | The time that the file was last modified.
size | long | True size of the file in bytes.
result | boolean | True if the operation has succeeded (can be false when Redirect error output is enabled).
errorMessage | string | If the operation has failed, the field contains the error message (used when Redirect error output is enabled).

**Error mapping** - the editor allows you to map the errors and the input data to the error port.

### Field Name | Type | Description
--- | --- | ---
result | boolean | Will always be set to false.
errorMessage | string | The error message.
MoveFiles

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 49, Common Properties of File Operations (p. 323)

If you want to find the right File Operation component for your purposes, see File Operations Comparison (p. 323).

The component is located in Palette → File Operations.

**Short Summary**

**MoveFiles** can be used to move files and directories.

**Note**

To be able to use this component, you need a separate jobflow license.

<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveFiles</td>
<td>0-1</td>
<td>0-2</td>
</tr>
</tbody>
</table>

**Abstract**

*MoveFiles* can move multiple sources into one destination directory or a regular source file to a target file. Optionally, existing files may be skipped or updated based on the modification date of the files.

**Icon**

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>Input data records to be mapped to component attributes.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>Results</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>Errors</td>
<td>Any</td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.
### MoveFiles Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source file URL</td>
<td>yes</td>
<td>Path to the source file or directory (see Supported URL Formats for File Operations (p. 324)).</td>
<td></td>
</tr>
<tr>
<td>Target file URL</td>
<td>yes</td>
<td>Path to the destination file or directory (see Supported URL Formats for File Operations (p. 324)). When it points to a directory, the source will be moved into the directory. It must be a path to a single file or directory.</td>
<td></td>
</tr>
<tr>
<td>Overwrite</td>
<td>no</td>
<td>Specifies whether existing files shall be overwritten. In update mode, the target will be overwritten only when the source file is newer than the destination file.</td>
<td>always (default)</td>
</tr>
<tr>
<td>Create parent directories</td>
<td>no</td>
<td>Attempt to create non-existing parent directories. When the Create parent directories option is enabled and the Target file URL ends with a slash ('/'), it is treated as the parent directory, i.e. the source directory or file is moved into the target directory, even if it does not exist.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Input mapping</td>
<td>2)</td>
<td>Defines mapping of input records to component attributes.</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Defines mapping of results to standard output port.</td>
<td></td>
</tr>
<tr>
<td>Error mapping</td>
<td>2)</td>
<td>Defines mapping of errors to error output port.</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td>no</td>
<td>If enabled, errors will be sent to the output port instead of the error port.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Verbose output</td>
<td>no</td>
<td>If enabled, one input record may cause multiple records to be sent to the output (e.g. as a result of wildcard expansion). Otherwise, each input record will yield just one cumulative output record.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stop processing on fail</td>
<td>no</td>
<td>By default, a failure causes the component to skip all subsequent operations and send the information about skipped operations to the error output port. This behaviour can be turned off by this attribute.</td>
<td>true (default)</td>
</tr>
</tbody>
</table>

**Legend**

1) The attribute is required, unless specified in the Input mapping.

2) Required if the corresponding edge is connected.

**Advanced Description**
Chapter 56. File Operations

Editing any of the **Input**, **Output** or **Error mapping** opens the Transform Editor (p. 275).

**Input mapping** - the editor allows you to override selected attributes of the component with the values of the input fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Attribute</th>
<th>Type</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceURL</td>
<td>Source file URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>targetURL</td>
<td>Target file URL</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>overwrite</td>
<td>Overwrite</td>
<td>string</td>
<td>&quot;always&quot;</td>
</tr>
<tr>
<td>makeParentDirs</td>
<td>Create parent directories</td>
<td>boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

**Output mapping** - the editor allows you to map the results and the input data to the output port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceURL</td>
<td>string</td>
<td>URL of the source file.</td>
</tr>
<tr>
<td>targetURL</td>
<td>string</td>
<td>URL of the destination.</td>
</tr>
<tr>
<td>resultURL</td>
<td>string</td>
<td>New URL of the successfully moved file. Only set in <strong>Verbose output</strong> mode.</td>
</tr>
<tr>
<td>result</td>
<td>boolean</td>
<td>True if the operation has succeeded (can be false when <strong>Redirect error output</strong> is enabled).</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>If the operation has failed, the field contains the error message (used when <strong>Redirect error output</strong> is enabled).</td>
</tr>
</tbody>
</table>

**Error mapping** - the editor allows you to map the errors and the input data to the error port.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>boolean</td>
<td>Will always be set to false.</td>
</tr>
<tr>
<td>errorMessage</td>
<td>string</td>
<td>The error message.</td>
</tr>
<tr>
<td>sourceURL</td>
<td>string</td>
<td>URL of the source file.</td>
</tr>
<tr>
<td>targetURL</td>
<td>string</td>
<td>URL of the destination.</td>
</tr>
</tbody>
</table>
Chapter 57. Cluster Components

We assume that you already know what components are. See Chapter 18, Components (p. 94) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 45, Common Properties of Cluster Components (p. 319)

We can distinguish either Cluster component by what it does:

- ClusterPartitioner (p. 720) distributes data records among various CloverETL Server nodes of a single Cluster.
- ClusterGather (p. 718) gathers data records from various CloverETL Server nodes of a single Cluster.
Chapter 57. Cluster Components

ClusterGather

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 45, Common Properties of Cluster Components (p. 319)

Short Summary

ClusterGather gathers data records from multiple cluster nodes.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterGather</td>
<td>yes</td>
<td>no</td>
<td>1-n (virtual)</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Abstract

ClusterGather receives potentially unsorted data records through single input port. ClusterGather gathers records as specified in the Merge type attribute.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For gathered data records</td>
<td>Input 0(^1)</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can be propagated through this component.
## ClusterGather Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging type</td>
<td></td>
<td>The way how records should be gathered. By round robin, sorted by key or gathered as soon as possible.</td>
<td>Simple gather (default)</td>
</tr>
<tr>
<td>Merge key</td>
<td>yes</td>
<td>Key according to which the sorted records are merged if <strong>Merge by key</strong> is selected. (Remember that all key fields must be sorted in ascending order!) See <strong>Group Key</strong> (p. 264) for more information.</td>
<td></td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of key fields are considered to be different. If set to <code>true</code>, they are considered to be equal.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Data policy</td>
<td></td>
<td>Determines what should be done when an error occurs. See <strong>Data Policy</strong> (p. 295) for more information.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 57. Cluster Components

ClusterPartitioner

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)
- Chapter 45, Common Properties of Cluster Components (p. 319)

Short Summary

ClusterPartitioner distributes individual input data records among different Cluster nodes.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterPartitioner</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n (virtual)</td>
<td>yes/no</td>
<td>yes/no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For output data records</td>
<td>Input 0¹</td>
</tr>
</tbody>
</table>

Legend:

1): Metadata can be propagated through this component.
# Partition Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td>1)</td>
<td>Definition of the way how records should be distributed among Cluster nodes written in the graph in CTL or Java.</td>
<td></td>
</tr>
<tr>
<td>Partition URL</td>
<td>1)</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Partition class</td>
<td>1)</td>
<td>Name of external class defining the way how records should be distributed among Cluster nodes.</td>
<td></td>
</tr>
<tr>
<td>Ranges</td>
<td>1),2</td>
<td>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: <code>&lt;1,9) (,31.12.2008);&lt;1,9) &lt;31.12.2008,);&lt;9,) (,31.12.2008); &lt;9,) &lt;31.12.2008)</code>.</td>
<td></td>
</tr>
<tr>
<td>Partition key</td>
<td>1),2</td>
<td>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: <code>first_name;last_name</code>.</td>
<td></td>
</tr>
<tr>
<td>Load balancer queue size</td>
<td></td>
<td>Size of the queue for load balancer.</td>
<td></td>
</tr>
</tbody>
</table>

| **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. 722) or [Java Interfaces for ClusterPartitioner](p. 722) for more information.

See also [Defining Transformations](p. 268) 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 IX, [CTL - CloverETL Transformation Language](p. 771). (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 template as *Partition*. See [CTL Templates for Partition (or ClusterPartitioner)](p. 594).

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. 598).
Chapter 58. Data Quality

We assume that you already know what components are. See Chapter 18, Components (p. 94) for brief information.

Some components are focused on determining and assuring the quality of your data. We call this group of components: Data Quality.

Data Quality 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

As Data Quality are heterogeneous group of components, they have no common properties.

We can distinguish each component of the Data Quality group according to the task it performs.

- Address Doctor 5 (p. 724) validates or fixes address format.
- EmailFilter (p. 728) validates e-mail addresses and sends out the valid ones. Data records with invalid e-mail addresses can be sent out through the optional second output port.
- ProfilerProbe (p. 733) performs statistical analyses of data flowing through the component.
Address Doctor 5

Commercial Component

We suppose that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the appropriate Data Quality component for your purpose, see Data Quality Comparison (p. 321).

Note

Despite being a Transformer, the component is located in Palette → Data Quality.

Short Summary

Address Doctor 5 validates, corrects or completes the address format of your input records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Doctor 5</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For transformed data records</td>
<td>Any2</td>
</tr>
</tbody>
</table>
Chapter 58. Data Quality

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>1</td>
<td>no</td>
<td>For records that could not be transformed (error port)</td>
<td>Any2</td>
</tr>
</tbody>
</table>

### Address Doctor 5 Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Config file</td>
<td>1)</td>
<td>External file defining the configuration.</td>
<td></td>
</tr>
<tr>
<td>Parameter file</td>
<td>1)</td>
<td>External file defining parameters.</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>2)</td>
<td>Specifies the address database and its location.</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>2)</td>
<td>Controls how the transformation is performed.</td>
<td></td>
</tr>
<tr>
<td>Input mapping</td>
<td>yes</td>
<td>Determines what will be processed.</td>
<td></td>
</tr>
<tr>
<td>Output mapping</td>
<td>yes</td>
<td>Controls what will be mapped to the output.</td>
<td></td>
</tr>
<tr>
<td>Element item</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delimiter</td>
<td></td>
<td>If the whole address is stored on a single line, this attribute specifies which special character separates the address fields.</td>
<td>delimiter is not used (default)</td>
</tr>
</tbody>
</table>

**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. 725)

### 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).
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**.
• 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 58.3. Output mapping](image)

• 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.
EmailFilter
Commercial Component

We assume that you have already learned what is described in:
• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Data Quality component for your purposes, see Data Quality Comparison (p. 321).

Note
Despite being a Transformer, the component is located in Palette \rightarrow Data Quality.

Short Summary
EmailFilter filters input records according to the specified condition.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmailFilter</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>0-2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For valid data records</td>
<td>Input 0&lt;sup&gt;1)&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>For rejected data records</td>
<td>Any&lt;sup&gt;2)&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

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 58.1. Error Fields for EmailFilter

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FieldA</td>
<td>the Error field</td>
<td>string</td>
<td>Error field</td>
</tr>
<tr>
<td>FieldB</td>
<td>the Status field</td>
<td>integer(^1)</td>
<td>Status field</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field list</td>
<td>yes</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Level of inspection</td>
<td></td>
<td>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. 731) for more information.</td>
<td>SYNTAX</td>
</tr>
<tr>
<td>Accept empty</td>
<td></td>
<td>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. 731) for more information.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Error field</td>
<td></td>
<td>Name of the output field to which error message can be written (for rejected records only).</td>
<td></td>
</tr>
<tr>
<td>Status field</td>
<td></td>
<td>Name of the output field to which error code can be written (for rejected records only).</td>
<td></td>
</tr>
<tr>
<td>Multi delimiter</td>
<td></td>
<td>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.</td>
<td>[;:</td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Accept condition</td>
<td></td>
<td>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.</td>
<td>LENIENT (default)</td>
</tr>
<tr>
<td>Advanced</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E-mail buffer size</td>
<td></td>
<td>Maximum number of records that are read into memory after which they are bulk processed. See Buffer and Cache Size (p. 731) for more information.</td>
<td>2000 (default)</td>
</tr>
<tr>
<td>E-mail cache size</td>
<td></td>
<td>Maximum number of cached e-mail address verification results. See Buffer and Cache Size (p. 731) for more information.</td>
<td>2000 (default)</td>
</tr>
<tr>
<td>Domain cache size</td>
<td></td>
<td>Maximum number of cached DNS query results. Is ignored at SYNTAX level.</td>
<td>3000 (default)</td>
</tr>
<tr>
<td>Domain retry timeout (ms)</td>
<td></td>
<td>Timeout in millisecond for each DNS query attempt. Thus, maximum time in milliseconds spent to resolving equals to Domain retry timeout multipliclated by Domain retry count.</td>
<td>800 (default)</td>
</tr>
<tr>
<td>Domain retry count</td>
<td></td>
<td>Number of retries for failed DNS queries.</td>
<td>2 (default)</td>
</tr>
<tr>
<td>Domain query A records</td>
<td></td>
<td>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.</td>
<td>true (default)</td>
</tr>
<tr>
<td>SMTP connect attempts (ms,...)</td>
<td></td>
<td>Attempts for connection and HELO. Expressed as a sequence of numbers separated by comma. The numbers are delays between individual attempts to connect.</td>
<td>1000,2000 (default)</td>
</tr>
<tr>
<td>SMTP anti-greylisting attempts (s,...)</td>
<td></td>
<td>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 Greylisting Attempts (p. 732) for more information.</td>
<td>30,120,240 (default)</td>
</tr>
<tr>
<td>SMTP retry timeout (s)</td>
<td></td>
<td>TCP timeout in seconds after which a SMTP request fails.</td>
<td>300 (default)</td>
</tr>
<tr>
<td>SMTP concurrent limit</td>
<td></td>
<td>Maximum number of parallel tasks when anti-greylisting is on.</td>
<td>10 (default)</td>
</tr>
<tr>
<td>Mail From</td>
<td></td>
<td>The From field of a dummy message sent at MAIL level.</td>
<td>CloverETL <a href="mailto:clover@cloveretl.org">clover@cloveretl.org</a> (default)</td>
</tr>
<tr>
<td>Mail Subject</td>
<td></td>
<td>The Subject field of a dummy message sent at MAIL level.</td>
<td>Hello, this is a test message (default)</td>
</tr>
</tbody>
</table>
| Mail Body | | The Body of a dummy message sent at MAIL level. | Hello,
This is CloverETL
text message.
Please 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, This is CloverETL test message. Please 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.
Chapter 58. Data Quality

ProfilerProbe

Commercial Component

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Data Quality component for your purposes, see Data Quality Comparison (p. 321).

The component is located in Palette → Data Quality.

Short Summary

ProfilerProbe analyses (profiles) input data. The component is a light-weight version of the Data Profiler application, fully integrated within the CloverETL environment. The big advantage of the component is the combined power of CloverETL solutions with data profiling features. Thus, it makes profiling accessible in very complex workflows, such as data integration, data cleansing, and other ETL tasks.

ProfilerProbe is not limited to only profiling isolated data sources; instead, it can be used for profiling data from various sources (including popular DBs, flat files, spreadsheets etc.). ProfilerProbe is capable of handling all data sources supported by CloverETL’s Readers (p. 328). As a result, this component is considered a major step forward comparing to the stand-alone Clover Profiling Jobs (cpj) in Data Profiler.

Note

To be able to use this component, you need a separate data profiling license.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProfilerProbe</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Abstract

ProfilerProbe calculates metrics of the data that is coming through its first input port. You can choose which metrics you want to apply on each field of the input metadata. You can use this component as a 'probe on an edge' to get a more detailed (statistical) view of data that is flowing in your graph.

The component sends an exact copy of the input data to output port 0 (behaves as SimpleCopy). That means you can use ProfilerProbe in your graphs to examine data flowing in it - without affecting the graph’s business logic itself.

The remaining output ports contain results of profiling, i.e. metric values for individual fields.

Icon
## Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>Input data records to be analysed by metrics.</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>A copy of input data records.</td>
<td>Input port 0</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>Results of data profiling per individual field.</td>
<td>Any</td>
</tr>
</tbody>
</table>
## ProfilerProbe Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metrics</td>
<td>1)</td>
<td>Statistics you want to be calculated on metadata fields. You can apply all metrics as in Data Profiler stand-alone, <strong>except for charts.</strong> Learn more about metrics here.</td>
<td>List of all metrics</td>
</tr>
<tr>
<td>Output mapping</td>
<td>2)</td>
<td>Maps profiling results to output ports, starting from port number 1. See <a href="#">Advanced Description</a> (p. 735).</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metrics URL</td>
<td>1)</td>
<td>Profiler job file containing the <strong>Metrics</strong> settings.</td>
<td>*.cpj</td>
</tr>
<tr>
<td>Output mapping URL</td>
<td>2)</td>
<td>External XML file containing the <strong>Output mapping</strong> definition.</td>
<td></td>
</tr>
</tbody>
</table>
| Processing mode      |     | **Always active** (default) - default mode to execute **ProfilerProbe** component locally and remotely (if executed on the server). **Debug mode only** - select this mode to capture execution data for debugging purpose, similar to debug mode on component edges - please note that when executing a graph with this mode selected for **ProfilerProbe**:  
  - runs as expected when server debug_mode = true (a server graph configuration property - see Clover Server docs).  
  - when server debug_mode = false, the input data would continue through the 1st output port, but it does not send profiling of data to subsequent output ports. | **Always active** (default) | **Debug mode only** |

### Legend

1) Specify only one of these attributes. (If both are set, **Metrics URL** has a higher priority.)

2) Specify only one of these attributes. (If both are set, **Output mapping URL** has a higher priority.)

### Advanced Description

- **Output mapping** - editing the attribute opens the Transform Editor (p. 275) where you can decide which metrics to send to output ports.
The dialog provides you with all the power and features known from Transform Editor and CTL (p. 849). In addition, notice metadata on the left hand side has a special format. It is a tree of input fields AND metrics you assigned to them via the Metrics attribute. Fields and metrics are grouped under the RunResults record. Each field in RunResults record has a special name: fieldName__metric_name (note the underscore is doubled as a separator), e.g. firstName__avg_length. Additionally is another special record containing one field - inputRecordCount. After you run your graph, the field will store the total number of records which were profiled by the component. You can right-click a field/metric and Expand All, or Collapse All metrics.

Note

The ProfilerComponent can report an error similar to:

```
CTL code compilation finished with 1 errors
Error: Line 5 column 23 - Line 5 column 39: Field 'field1__avg_length' does not exist in record 'RunResults'!
```

This means that you're accessing a disabled metric in output mapping - in this example the Average length is not enabled on the field field1.

To do the mapping in a few basic steps:

1. Provided you already have some output metadata, just left-click a metric in the left-hand pane and drag it onto an output field. This will send profiling results of that particular metric to the output.

2. If you do not have any output metadata:
   a. Drag a Field from the left hand side pane and drop it into the right hand pane (an empty space).
   b. This produces a new field in the output metadata. Its format is: fieldName__metric_name (note the underscore is doubled as a separator), e.g. firstName__avg_length.
c. You can map metrics to fields of any output port, except for port 0. That port is reserved for input data (which just 'flows through' the component without being affected in a way).

**Note**

Output mapping uses CTL (you can switch to the **Source** tab). All kinds of functions are available that help you learn even more about your data. Example:

```java
double uniques = $out.0.firstName__uniques; // conversion from integer
double uniqInAll = (uniques / $in.0.recordCount) * 100;
```

calculates the per cent of unique frist names in all records.

**• Importing and Externalizing metrics** - in the **Metrics** dialog, you can have your settings of fields and their metrics externalized to a Profiler job (*.cpj) file, or imported from a Profiler job (*.cpj) file into this attribute. There are two buttons at the bottom of the dialog for this purpose: **Import from .cpj** and **Externalize to .cpj**. The externalized .cpj file can be used in the **Metrics URL** attribute. The **Externalize to .cpj** action fills in this attribute automatically.

![Import/Externalize metrics buttons](image)

**ProfilerProbe Notes & Limitations**

This short section describes the main differences between using the **ProfilerProbe** component and profiling data via *.cpj jobs.

• The component does not support charts at the moment (interval / frequency / time unit).

• It performs analyses just on the data which comes through its input edge. Profiling results are sent to output ports. Please note you do not need any results database.

• It is able to use data profiling jobs (*.cpj) via the **Metrics URL** attribute.

• If you want to use sampling of the input data, connect the **DataSampler** (or other filter) component to your graph. There is no built-in sampling in **ProfilerProbe**.
Chapter 59. Others

We assume that you already know what components are. See Chapter 18, Components (p. 94) 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

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. 764) executes system commands.
- JavaExecute (p. 752) executes Java commands.
- DBExecute (p. 743) executes SQL/DML/DDL statements against database.
- RunGraph (p. 756) runs specified CloverETL graph(s).
- HTTPConnector (p. 747) sends HTTP requests and receives responses from web server.
- WebServiceClient (p. 767) calls a web-service and maps response to output ports.
- CheckForeignKey (p. 739) checks foreign key values and replaces those invalid by default values.
- SequenceChecker (p. 760) checks whether input data records are sorted.
- LookupTableReaderWriter (p. 754) reads data from a lookup table and/or write data to a lookup table.
- SpeedLimiter (p. 762) slows down data flowing throughout the component.
CheckForeignKey

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

CheckForeignKey checks the validity of foreign key values and replaces invalid values by valid ones.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckForeignKey</td>
<td>-</td>
<td>no</td>
<td>2</td>
<td>1-2</td>
<td>-</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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 on 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 port 1), 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For data with foreign key</td>
<td>Any1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>yes</td>
<td>For data with primary key</td>
<td>Any2</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For data with corrected key</td>
<td>Input 0¹</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>no</td>
<td>For data with invalid key</td>
<td>Input 0¹</td>
</tr>
</tbody>
</table>
Legend:

1): Metadata cannot be propagated through this component.

**CheckForeignKey Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foreign key</td>
<td>yes</td>
<td>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. 740) for more information.</td>
<td></td>
</tr>
<tr>
<td>Default foreign key</td>
<td>yes</td>
<td>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. 740) for more information.</td>
<td></td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of fields are considered to be different. If set to true, nulls are considered to be equal.</td>
<td>false (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash table size</td>
<td></td>
<td>Table for storing key values. Should be higher than the number of records with unique key values.</td>
<td>512 (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary key</td>
<td></td>
<td>Sequence of field names from the second input port separated from each other by semicolon. See Deprecated: Primary Key (p. 742) for more information.</td>
<td></td>
</tr>
</tbody>
</table>

**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.
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.

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 59.3. 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.
Chapter 59. Others

DBExecute

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

DBExecute executes SQL/DML/DDL statements against a database.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBExecute</td>
<td>-</td>
<td>✗</td>
<td>0-1</td>
<td>0-2</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>Input records for stored procedure or the whole SQL commands</td>
<td>any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>2)</td>
<td>Output parameters of stored procedure or result set of the query</td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>✗</td>
<td>for error information</td>
<td>based on input metadata 3)</td>
</tr>
</tbody>
</table>

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. 128): ErrCode and ErrText.
### DBExecute Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DB connection</td>
<td>yes</td>
<td>ID of the DB connection to be used.</td>
<td></td>
</tr>
<tr>
<td>Query URL</td>
<td>1)</td>
<td>One of these two options: Either the name of external file, including path, defining SQL query with the same characteristics as described in the <strong>SQL query</strong> attribute, or the <strong>File URL</strong> attribute string that is used for port reading. See [SQL Query Received from Input Port](p. 745) for details.</td>
<td></td>
</tr>
<tr>
<td>SQL query</td>
<td>1)</td>
<td>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 following: `{[? = ]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 (<strong>Query input parameters</strong>, <strong>Query output parameters</strong> and/or <strong>Result set output fields</strong>, respectively). In addition, if the query consists of multiple statements, they must be separated from each other by specified <strong>SQL statement delimiter</strong>. Statements will be executed one by one.</td>
<td></td>
</tr>
<tr>
<td>SQL statement delimiter</td>
<td></td>
<td>Delimiter between individual SQL statements in the <strong>SQL query</strong> or <strong>Query URL</strong> attribute. Default delimiter is semicolon.</td>
<td>; (default)</td>
</tr>
<tr>
<td>Print statements</td>
<td></td>
<td>By default, SQL commands are not printed. If set to true, they are sent to stdout.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Transaction set</td>
<td></td>
<td>Specifies whether the statements should be executed in transaction. See [Transaction Set](p. 745) for more information. Is applied only if database supports transactions.</td>
<td>SET (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query source charset</td>
<td></td>
<td>Encoding of external file specified in the <strong>Query URL</strong> attribute.</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Call as stored procedure</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Query input parameters</td>
<td></td>
<td>Used when stored procedure/function with input parameters is called. It is a sequence of the following type: <code>1:=$inputField1;...;n:=$inputFieldn</code>. Value of each specified input field is mapped to corresponding parameter (whose position in <strong>SQL query</strong> equals to the specified number). This attribute cannot be specified if SQL commands should be received through input port.</td>
<td></td>
</tr>
</tbody>
</table>
## Attribute | Req | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Query output parameters</td>
<td></td>
<td>Used when stored procedure/function with output parameters or return value is called. It is a sequence of the following type: (1:=outputField1;\ldots;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.</td>
</tr>
<tr>
<td>Result set output fields</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Error actions</td>
<td></td>
<td>Definition of the action that should be performed when the specified query throws an SQL Exception. See Return Values of Transformations (p. 272).</td>
</tr>
<tr>
<td>Error log</td>
<td></td>
<td>URL of the file to which error messages for specified Error actions should be written. If not set, they are written to Console.</td>
</tr>
</tbody>
</table>

**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. 292).

#### 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**

• *Transferring 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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

HTTPConnector sends HTTP requests to a web server and receives responses.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs?</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPConnector</td>
<td>-</td>
<td>no</td>
<td>0-1</td>
<td>0-2</td>
<td>-</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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. The response can be sent to an output port, stored to a specified file or stored to a temporary file. Path to the file can then be sent to a specified output port.

Icon

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For setting various attributes of the component</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>no</td>
<td>For response content, response file path, status code, component attributes...</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 no For error details</td>
<td></td>
</tr>
</tbody>
</table>

This component has Metadata Templates (p. 264) available.

HTTPConnector Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>URL</td>
<td>1)</td>
<td>URL of the HTTP server the component connects to. May contain one or more placeholders in the following form: <em>{&lt;field name&gt;}</em>. See the section called “Reading of Remote Files” (p. 288) for the URL format. The HTTP, HTTPS, FTP and SFTP protocols are supported. Connecting via a proxy server is available, too, in a way like: http://(proxy://proxyHost:proxyPort)/www.domain.com.</td>
<td></td>
</tr>
<tr>
<td>Request method</td>
<td></td>
<td>Method of request.</td>
<td>GET (default)</td>
</tr>
<tr>
<td>Add input fields as parameters</td>
<td></td>
<td>Specifies whether additional parameters from the input edge should be added to the URL. Note: When parameters are read from the input edge and put to the query string, they can even contain special characters (?@:, etc.). Do not replace such characters with %-notation. <strong>HTTPConnector</strong> automatically makes them URL-encoded. This feature was introduced in Clover 3.3-M3 and causes backwards incompatibility.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Add input fields as parameters to</td>
<td></td>
<td>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.</td>
<td>QUERY (default)</td>
</tr>
<tr>
<td>Ignored fields</td>
<td></td>
<td>Specifies which input fields are not added as parameters. List of input fields separated by semicolon is expected.</td>
<td></td>
</tr>
<tr>
<td>Additional HTTP header properties</td>
<td></td>
<td>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. The value may refer to a field using ${fieldName} notation.</td>
<td></td>
</tr>
<tr>
<td>Multipart entities</td>
<td></td>
<td>Specifies fields, that should be added as multipart entities to a POST request. Field name is used as an entity name. List of input fields separated by semicolon is expected.</td>
<td></td>
</tr>
<tr>
<td>Request/response charset</td>
<td></td>
<td>Character encoding of the input/output files</td>
<td>ISO-8859-1 (default)</td>
</tr>
<tr>
<td>Request content</td>
<td></td>
<td>Request content defined directly in the graph.</td>
<td></td>
</tr>
<tr>
<td>Input file URL</td>
<td></td>
<td>URL of the file from which single HTTP request is read. See [URL File Dialog](p. 68).</td>
<td></td>
</tr>
<tr>
<td>Output file URL</td>
<td></td>
<td>URL of the file to which HTTP response is written. See [URL File Dialog](p. 68). The output files are not deleted automatically and must be removed by hand or as a part of transformation.</td>
<td></td>
</tr>
<tr>
<td>Append output</td>
<td></td>
<td>By default, any new response overwrites the older one. If you switch this attribute to true, new response is appended to the older ones. Is applied to output files only.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Input Mapping</td>
<td></td>
<td>Allows to set various properties of the component by mapping their values from input record.</td>
<td></td>
</tr>
<tr>
<td>Output Mapping</td>
<td></td>
<td>Allows to map response data (like content, status code, ... ) to the output record. It is also possible to map values from input fields and error details (if Redirect error output is set to true).</td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 59. Others

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Mapping</td>
<td></td>
<td>Allows to map error message to the output record. It is also possible to map values from input fields and attributes.</td>
<td></td>
</tr>
<tr>
<td>Redirect error output</td>
<td></td>
<td>Allows to redirect error details to standard output port.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

#### Advanced Description

| Authentication method      |     | Specifies which authentication method should be used.                                                                                                                                                         | HTTP BASIC (default) | HTTP DIGEST | ANY |
|---------------------------|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|------------|
| Username                  |     | Username required to connect to the server                                                                                                                                                                    |                      |            |     |
| Password                  |     | Password required to connect to the server                                                                                                                                                                    |                      |            |     |
| OAuth Consumer key        |     | Consumer key associated with a service. Defines the access token (2-legged OAuth) for signing requests - together with OAuth Consumer secret.                                                               |                      |            |     |
| OAuth Consumer secret     |     | Consumer secret associated with a service. Defines the access token (2-legged OAuth) for signing requests - together with OAuth Consumer key.                                                                |                      |            |     |
| Store HTTP response to file | 2) | If this attribute is switched to true, response is written to temporary files with the prefix specified in the Prefix for response names attribute. The path to these temporary files can be retrieved using Output Mapping. Storing response to temporary files is necessary in case the response body is too large to be stored in a single string data field. The temporary files are deleted automatically after graph finishes (if not run in Debug mode). | false (default) | true       |     |
| Prefix for response files |     | Prefix that will be used in the name of each output file with HTTP response. To this prefix, distinguishing numbers are appended.                                                                             | "http-response-" (default) | other prefix |     |
| Redirect error output     |     | If true the error details will be sent to a Output port 0 instead of Output port 1                                                                                                                          | false (default) | true       |     |

#### Deprecated

| 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. 288) for the URL format. The HTTP, HTTPS, FTP and SFTP protocols are supported. |                      |            |     |
| 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              |     | 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.                                                                               |                      |            |     |

1) URL must be specified by setting one of URL or URL from field attributes or mapping it in the Input mapping.
2) The response can be stored either to a file specified in Output file URL or to a temporary file (when Store response file URL to output field is set to true) - it is not possible to use both the options.
• **Input mapping** - editing the attribute opens the Transform Editor (p. 275) where you can decide which component attributes should be set using input record.

![Figure 59.4. Transform Editor in HTTPConnector](image)

The dialog provides you with all the power and features known from Transform Editor and CTL (p. 849).

**Note**

All kinds of CTL functions are available to modify the input field value to be used.

• **Output mapping** - editing the attribute opens the Transform Editor (p. 275) where you can decide what should be sent to an output port.

![Figure 59.5. Transform Editor in HTTPConnector](image)

The dialog provides you with all the power and features known from Transform Editor and CTL (p. 849).

To do the mapping in a few basic steps:

1. Provided you already have some output metadata, just left-click an item in the left-hand pane and drag it onto an output field. This will send the result data to the output.

2. If you do not have any output metadata:
   a. Drag a **Field** from the left hand side pane and drop it into the right hand pane (an empty space).
   b. This produces a new field in the output metadata.

You can map various data to output port:
• **Values of fields from input metadata** - you can send values from input fields to the output port. This is mainly useful, when you are using some kind of session identifier for HTTP requests.

• **Result** - provides result data. These includes:
  
  • **content** - the content of the HTTP response. This field will be `null`, if the response is written to a file.
  
  • **outputFilePath** - the path to a file, where the response has been written. Will be `null`, if the response is not written to a file.
  
  • **statusCode** - HTTP status code of the response.
  
  • **header** - map representing HTTP header properties from response.
  
  • **errorMessage** - error message in case, that the error output is redirected to a standard output port.

• **Attributes** - provides values of the component attributes:
  
  • **URL** - the URL where the request has been sent.
  
  • **requestMethod** - method that was used for the request.
  
  • **requestContent** - content of the request, that has been sent.
  
  • **inputFileUrl** - URL of the file containing request content.

**Note**

Output mapping uses CTL (you can switch to the **Source** tab). All kinds of functions are available to modify the value to be stored in the output field.

```java
$out.0.prices = find($in.1.content, "price: .\d+ USD")
```

finds all occurrences of the form `price: [some text] USD` in the response content.

• **Error mapping** - editing the attribute opens the Transform Editor (p. 275) where you can map error details to an output port. The behavior is very similar to the **Output mapping** (p. )

**Notes**

• Since v3.3.0-M3 it is no longer necessary to encode field values used as Query parameters before passing them to **HTTPConnector** - they are encoded automatically. This, however, breaks backward compatibility, so be aware of this fact.

• Since v3.3.0-M3 it is possible to use **Output mapping** to retrieve path to an output file, when the response is stored to a file (whether it is stored to temporary file or user-specified file). The file path is no longer sent to an output port automatically (as was the case for temporary files).
JavaExecute

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

JavaExecute executes Java commands.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs?</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaExecute</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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. 284).

Below is the list of JavaRunnable interface methods. See Java Interfaces for JavaExecute (p. 753) for more information.

Icon

JavaExecute has neither input nor output port.

JavaExecute Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>1)</td>
<td>Runnable transformation in Java defined in the graph.</td>
<td></td>
</tr>
<tr>
<td>Runnable URL</td>
<td>1)</td>
<td>External file defining the runnable transformation in Java.</td>
<td></td>
</tr>
<tr>
<td>Runnable class</td>
<td>1)</td>
<td>External runnable transformation class.</td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>Req</td>
<td>Description</td>
<td>Possible values</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Runnable source</td>
<td>charset</td>
<td>Encoding of external file defining the transformation.</td>
<td>ISO-8859-1 (default)</td>
</tr>
</tbody>
</table>

**Advanced**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties</td>
<td>Properties to be used when executing Java command.</td>
</tr>
</tbody>
</table>

**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. 753) for more information.

See also [Defining Transformations](p. 268) 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. 284).

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()`.
Chapter 59. Others

LookupTableReaderWriter

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

LookupTableReaderWriter reads data from lookup table and/or writes it to lookup table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LookupTableReaderWriter</td>
<td>-</td>
<td>no</td>
<td>0-1</td>
<td>0-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>1)</td>
<td>For data records to be written to lookup table</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0-n</td>
<td>1)</td>
<td>For data records to be read from lookup table</td>
<td>Input 01</td>
</tr>
</tbody>
</table>

Legend:

1): At least one of them has to be connected. If the input port is connected, the component receives data through it and writes it to the lookup table. If an output port is connected, the component reads data from the lookup table and sends it out through this port.
If the input port is connected and the component cannot write into the Look-up table (see LookupTableReaderWriter Attributes (p. 755)) you have specified, an error will be shown.

**Important**

Please note writing into Database lookup table is not supported. You should use DBOutputTable (p. 451) instead.

### LookupTableReaderWriter Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lookup table</td>
<td>yes</td>
<td>ID of the lookup table to be used as</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a source of records when the component is used as a reader, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a deposit when the component is used as a writer, or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• both when it is used both for reading and writing.</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free lookup table after finishing</td>
<td></td>
<td>By default, contents of the lookup table are not deleted after the graph</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>finishes. If set to true, the lookup table is emptied after the processing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ends.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>
RunGraph

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

RunGraph runs CloverETL graphs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunGraph</td>
<td>-</td>
<td>no</td>
<td>0-1</td>
<td>1-2</td>
<td>-</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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.

Ports

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For graph names and Clover command line arguments</td>
<td>Input Metadata for RunGraph (In-Out Mode) (p. 757)</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For information about graph execution</td>
<td>Output Metadata for RunGraph (p. 757)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2)</td>
<td>For information about unsuccessful graph execution</td>
<td>Output Metadata for RunGraph (p. 757)</td>
</tr>
</tbody>
</table>

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 59.1. Input Metadata for RunGraph (In-Out Mode)

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;anyname1&gt;</td>
<td>string</td>
<td>Name of the graph to be executed, including path</td>
</tr>
<tr>
<td>1</td>
<td>&lt;anyname2&gt;</td>
<td>string</td>
<td>Clover command line argument. <strong>Warning:</strong> Arguments sent in this field are ignored when the <strong>Same JVM</strong> attribute is true (see RunGraph Attributes (p. 757)).</td>
</tr>
</tbody>
</table>

### Table 59.2. Output Metadata for RunGraph

<table>
<thead>
<tr>
<th>Field number</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>graph</td>
<td>string</td>
<td>Name of the graph to be executed, including path</td>
</tr>
<tr>
<td>1</td>
<td>result</td>
<td>string</td>
<td>Result of graph execution (Finished OK, Aborted, or Error)</td>
</tr>
<tr>
<td>2</td>
<td>description</td>
<td>string</td>
<td>Detailed description of graph fail</td>
</tr>
<tr>
<td>3</td>
<td>message</td>
<td>string</td>
<td>Value of <code>org.jetel.graph.Result</code></td>
</tr>
<tr>
<td>4</td>
<td>duration</td>
<td>integer, long, or decimal</td>
<td>Duration of graph execution in milliseconds</td>
</tr>
<tr>
<td>5</td>
<td>runID</td>
<td>decimal</td>
<td>Identification of the execution of the graph which runs on CloverETL Server.</td>
</tr>
</tbody>
</table>

### RunGraph Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Graph URL</strong></td>
<td>1)</td>
<td>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)</td>
<td></td>
</tr>
<tr>
<td><strong>The same JVM</strong></td>
<td></td>
<td>By default, the same JVM instance is used to run the specified graphs. If switched to <strong>false</strong>, graph(s) run as external processes. When working in the server environment, this attribute always has to be <strong>true</strong> (thus, you cannot pass graph arguments through field 1 of port 0, see Ports (p. 756)).</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Graph parameters to pass</strong></td>
<td></td>
<td>Parameters that are used by executed graphs. List a sequence separated by semicolon. If <strong>The same JVM</strong> attribute is switched to <strong>false</strong>, this attribute is ignored. See Advanced Description (p. 758) for more information.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 59. Others

### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative JVM command</td>
<td>2)</td>
<td>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.</td>
<td>java -cp (default)</td>
</tr>
<tr>
<td>Log file URL</td>
<td></td>
<td>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. 68).</td>
<td></td>
</tr>
<tr>
<td>Append to log file</td>
<td>2)</td>
<td>By default, data in the specified log file is overwritten on each graph run.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Graph execution class</td>
<td>2)</td>
<td>Full class name to execute graph(s).</td>
<td>org.jetel.main.runGraph (default)</td>
</tr>
<tr>
<td>Command line arguments</td>
<td>2)</td>
<td>Arguments of Java command to be executed when running graph(s).</td>
<td></td>
</tr>
<tr>
<td>Ignore graph fail</td>
<td></td>
<td>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.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

### Advanced Description

**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 59.1. Working with Quoted Command Line Arguments

Let us have the following list of arguments:

```java
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 39, *Common Properties of All Components* (p. 256)
- Chapter 40, *Common Properties of Most Components* (p. 264)

If you want to find the right **Other** component for your purposes, see *Others Comparison* (p. 320).

**Short Summary**

*SequenceChecker* checks the sort order of input data records.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceChecker</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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**

![SequenceChecker](image)

**Ports**

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0-n</td>
<td>no</td>
<td>For checked and copied data records</td>
<td>Input 0(1)</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort key</td>
<td>yes</td>
<td>Key according to which the records should be sorted. If they are sorted in any other way, graph fails. See Sort Key (p. 265) for more information.</td>
<td></td>
</tr>
<tr>
<td>Unique keys</td>
<td></td>
<td>By default, values of Sort key should be unique. If set to false, values of Sort key can be duplicated.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Equal NULL</td>
<td></td>
<td>By default, records with null values of fields are considered to be equal. If set to false, nulls are considered to be different.</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort order</td>
<td></td>
<td>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.</td>
<td>Ascending (default)</td>
</tr>
<tr>
<td>Locale</td>
<td></td>
<td>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. 85).</td>
<td>system value or specified default value (default)</td>
</tr>
<tr>
<td>Use internationalization</td>
<td></td>
<td>By default, no internationalization is used. If set to true, sorting according national properties is performed.</td>
<td>false (default)</td>
</tr>
</tbody>
</table>
Chapter 59. Others

SpeedLimiter

We assume that you have already learned what is described in:

• Chapter 39, Common Properties of All Components (p. 256)
• Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

SpeedLimiter slows down data records going through it.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedLimiter</td>
<td>-</td>
<td>no</td>
<td>1</td>
<td>1-n</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

1) Component sends each data record to all connected output ports.

Abstract

SpeedLimiter receives data records through its single input port, delays each input record by a specified number of milliseconds and copies each input record 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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>yes</td>
<td>For input data records</td>
<td>Any</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>yes</td>
<td>For copied data records</td>
<td>Input 0(1)</td>
</tr>
<tr>
<td></td>
<td>1-n</td>
<td>no</td>
<td>For copied data records</td>
<td>Input 0(1)</td>
</tr>
</tbody>
</table>

Legend:

1): Unlike in SimpleCopy (p. 618), 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>yes</td>
<td>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.</td>
<td>0-N</td>
</tr>
</tbody>
</table>

### Tips & Tricks

When using Speedlimiter, do not forget records are sent out from the component only after its buffer gets full (by default). Sometimes you might need to:

1. send a record to Speedlimiter
2. have it delayed by a specified amount of seconds
3. send this very record to output ports immediately

In such case, you have to change settings of the edge outgoing from Speedlimiter to Direct fast propagate. See Types of Edges (p. 97) for more details.
SystemExecute

We assume that you have already learned what is described in:

- Chapter 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

**Short Summary**

SystemExecute executes system commands.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemExecute</td>
<td>-</td>
<td>no</td>
<td>0-1</td>
<td>0-1</td>
<td>-</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For standard input of the specified system command (input of the process)</td>
<td>Any1</td>
</tr>
<tr>
<td>Output</td>
<td>0</td>
<td>1)</td>
<td>For standard output of the specified system command (output of the process)</td>
<td>Any2</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Basic</strong></td>
<td></td>
</tr>
<tr>
<td>System command</td>
<td>yes</td>
<td>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. 766) for details.</td>
<td></td>
</tr>
<tr>
<td>Process input/output</td>
<td></td>
<td>Encoding used for formatting/parsing data for input/from output of system process.</td>
<td>ISO-8859-1</td>
</tr>
<tr>
<td>charset</td>
<td></td>
<td>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. 58) for more information.</td>
<td></td>
</tr>
<tr>
<td>Output file URL</td>
<td>1)</td>
<td>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.</td>
<td>false (default)</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>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: <code>&lt;interpreter name&gt; [parameters] ${} [parameters]</code>. 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.</td>
<td>current directory (default)</td>
</tr>
<tr>
<td>Working directory</td>
<td></td>
<td>Working directory of the component.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td>Number of error lines</td>
<td></td>
<td>Number of lines that are printed if a command finishes with errors.</td>
<td>2 (default)</td>
</tr>
<tr>
<td>Delete tmp batch file</td>
<td></td>
<td>IMPORTANT: This attribute has been removed from Clover v. 3.3.0.M3 because of a clash with temp space management. By default, the created temporary batch file is deleted after command execution. If set to false, it will not be deleted.</td>
<td>true (default)</td>
</tr>
<tr>
<td>Environment</td>
<td></td>
<td>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 mean that /home/user/mydir!true 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).</td>
<td>For example: PATH=/home/user/mydir!true (default)</td>
</tr>
<tr>
<td>Timeout for producer/</td>
<td></td>
<td>Timeout in milliseconds. See [Timeout](p. 766) for details.</td>
<td>0 (without limitation)</td>
</tr>
<tr>
<td>consumer workers (ms)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 39, Common Properties of All Components (p. 256)
- Chapter 40, Common Properties of Most Components (p. 264)

If you want to find the right Other component for your purposes, see Others Comparison (p. 320).

Short Summary

WebServiceClient calls a web-service.

<table>
<thead>
<tr>
<th>Component</th>
<th>Same input metadata</th>
<th>Sorted inputs</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Each to all outputs</th>
<th>Java</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebServiceClient</td>
<td>-</td>
<td>no</td>
<td>0-1</td>
<td>0-n</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port type</th>
<th>Number</th>
<th>Required</th>
<th>Description</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>no</td>
<td>For request</td>
<td>Any1(In0)</td>
</tr>
<tr>
<td>Output</td>
<td>0-N</td>
<td>no(^1)</td>
<td>For response mapped to these ports</td>
<td>Any2(Out#)</td>
</tr>
</tbody>
</table>

Legend:

1): Response does not need to be sent to output if output ports are not connected.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Req</th>
<th>Description</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSDL URL</td>
<td>yes</td>
<td>URL of the WSD server to which component will connect. Connecting via a proxy server is available, too, in a way like: http://(proxy://proxyHost:proxyPort)//www.domain.com.</td>
<td></td>
</tr>
<tr>
<td>Operation name</td>
<td>yes</td>
<td>Name of the operation to be performed. See Advanced Description (p. 768).</td>
<td></td>
</tr>
<tr>
<td>Request Body structure</td>
<td>yes</td>
<td>Structure of the request that is received from input port or written directly to the graph.</td>
<td></td>
</tr>
<tr>
<td>Request Header structure</td>
<td></td>
<td>Optional attribute to Request Body structure. If not specified, automatic generation is disabled.</td>
<td></td>
</tr>
<tr>
<td>Response mapping</td>
<td></td>
<td>Mapping of successful response to output ports. The same mapping as in XMLExtract. See XMLExtract Mapping Definition (p. 409) for more information.</td>
<td></td>
</tr>
<tr>
<td>Fault mapping</td>
<td></td>
<td>Mapping of fault response to output ports. The same mapping as in XMLExtract. See XMLExtract Mapping Definition (p. 409) for more information.</td>
<td></td>
</tr>
<tr>
<td>Namespace bindings</td>
<td></td>
<td>A set of name-value assignments defining custom namespaces.</td>
<td>e.g. weather = <a href="http://ws.cdyne.com/WeatherWS/">http://ws.cdyne.com/WeatherWS/</a></td>
</tr>
<tr>
<td>Use nested nodes</td>
<td></td>
<td>When true, all elements with the same name are mapped, no matter their depth in the tree. See example in Advanced Description (p. 768).</td>
<td>true (default)</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Username</td>
<td>1)</td>
<td>Username to be used when connecting to the server.</td>
<td></td>
</tr>
<tr>
<td>Password</td>
<td>1)</td>
<td>Password to be used when connecting to the server.</td>
<td></td>
</tr>
<tr>
<td>Auth Domain</td>
<td>1)</td>
<td>Authentication domain. If not set, the NTLM authentication scheme will be disabled. Does not affect Digest and Basic authentication methods.</td>
<td></td>
</tr>
<tr>
<td>Auth Realm</td>
<td>1)</td>
<td>Authentication realm to which specified credentials apply. If left empty, the credentials will be used for any realm. Does not affect NTLM authentication scheme.</td>
<td></td>
</tr>
<tr>
<td>Timeout (ms)</td>
<td></td>
<td>Timeout for the request in milliseconds</td>
<td></td>
</tr>
<tr>
<td>Override Server URL</td>
<td></td>
<td>Specifies a URL that should be used for the requests instead of the one specified in WSDL definition.</td>
<td></td>
</tr>
<tr>
<td>Override Server URL from field</td>
<td></td>
<td>Specifies a field containing a URL that should be used for the requests instead of the one specified in WSDL definition.</td>
<td></td>
</tr>
<tr>
<td>Disable SSL Certificate Validation</td>
<td></td>
<td>If true, component ignores certificate validation problems for SSL connection.</td>
<td>true</td>
</tr>
</tbody>
</table>

1) See Authentication (p. 770) section for more details.

**Advanced Description**
• After sending your request to the server, **WebServiceClient** waits up to 10 mins for a response. If there is none, the component fails on error.

• If you switch log level (p. 82) to `DEBUG`, you can examine the full SOAP request and response in log. This is useful for development and issue investigation purposes.

• **Operation name** opens a dialog, depicted in the figure below, in which you can select a WS operation - just double click one of them. Operations not supporting the document style of the input meassage are displayed with a red error icon.

![Figure 59.6. Choosing WS operation name in WebServiceClient.](image)

• **Example 59.2. Use nested nodes example**

Mapping

```xml
<Mappings>
  <Mapping element="request">
    <Mapping element="message" outPort="0" />
  </Mapping>
</Mappings>
```

applied to

```xml
<request>
  <message>msg1</message>
  <operation>msg2</operation>
</request>
```

produces:

• `msg1` and `msg2` with **Use nested nodes** switched on (default behaviour)

• `msg1` with **Use nested nodes** switched off. In order to extract `msg2`, you would need to create an explicit `<Mapping>` tag (one for every nested element).
Authentication

If authentication is required by the web service server, **Username, Password** and, if NTLM authentication scheme is required, **Auth Domain** component properties need to be set.

There are currently three authentication schemes supported: NTLM, Digest and Basic. NTLM is the most secure, Basic is the least secure of these methods. Server advertizes which authentication methods it supports and **WebServiceClient** automatically selects the most secure one.

**Auth Realm** can be used to restrict specified credential only to desired realm in case Basic or Digest authentication schema is selected.

Note

**Auth Domain** is required by the NTLM authentication. If it is not set, only Digest and Basic authentication schemes will be enabled.

In case server requires NTLM authentication, but **Auth Domain** is left empty, you will get error like these in graph execution log:

- **ERROR [Axis2 Task]** - Credentials cannot be used for NTLM authentication: org.apache.commons.httpclient.UsernamePasswordCredentials

Also note that it is not possible to specify the domain as part of the **Username** in form of domain \username as is sometimes customary. The domain name has to be specified separately in the **Auth Domain** component property.
Part IX. CTL - CloverETL
Transformation Language
Chapter 60. 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 sufficient 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. 271), 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. 789)
- [Functions Reference](p. 819)

---

**CTL2 reference and built-in functions:**

- [Language Reference](p. 850)
- [Functions Reference](p. 878)

---

**Note**

CTL2 version of Clover transformation language is recommended.
Chapter 61. CTL1 vs. CTL2 Comparison

CTL2 is a new version of CloverETL transformation language. It adds many improvements to the CTL concept.

<table>
<thead>
<tr>
<th>Feature</th>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly typed</td>
<td>☒</td>
<td>☑</td>
</tr>
<tr>
<td>Interpreted mode</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Compiled mode</td>
<td>☒</td>
<td>☑</td>
</tr>
<tr>
<td>Speed</td>
<td>slower</td>
<td>faster</td>
</tr>
</tbody>
</table>

**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 you can use your metadata names directly in your code to declare a variable:

```java
Employee tmpEmployee;
recordName1 myRecord;
```
Chapter 61. CTL1 vs. CTL2 Comparison

The following table presents an overview of differences between both versions of CTL.

### Table 61.2. CTL Version Differences

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header (interpreted mode)</strong></td>
<td><strong>Header (compiled mode)</strong></td>
</tr>
<tr>
<td><code>//#TL</code></td>
<td><code>//#CTL2</code></td>
</tr>
<tr>
<td></td>
<td><code>//#CTL2:COMPILED</code></td>
</tr>
<tr>
<td><strong>Declaration of primitive variables</strong></td>
<td></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>integer</code></td>
</tr>
<tr>
<td><code>bytearray</code></td>
<td><code>byte</code></td>
</tr>
<tr>
<td><strong>Declaration of container variables</strong></td>
<td></td>
</tr>
<tr>
<td><code>list myList;</code></td>
<td><code>&lt;element type&gt;[] myList;</code></td>
</tr>
<tr>
<td>Example:</td>
<td>Example:</td>
</tr>
<tr>
<td><code>integer[]&gt; myList;</code></td>
<td><code>map&lt;string,boolean&gt; myMap;</code></td>
</tr>
<tr>
<td></td>
<td><code>map[&lt;type of key&gt;, &lt;type of value&gt;] myMap;</code></td>
</tr>
<tr>
<td><strong>Declaration of records</strong></td>
<td></td>
</tr>
<tr>
<td><code>record (&lt;metadataID&gt;) myRecord;</code></td>
<td><code>&lt;metadataName&gt; myRecord;</code></td>
</tr>
<tr>
<td><strong>Declaration of functions</strong></td>
<td></td>
</tr>
<tr>
<td><code>function fName(arg1,arg2) {</code></td>
<td><code>function &lt;data type&gt; fName(&lt;type1&gt; arg1,&lt;type2&gt; arg2) {</code></td>
</tr>
<tr>
<td><code>&lt;functionBody&gt; }</code></td>
<td><code>&lt;functionBody&gt; }</code></td>
</tr>
<tr>
<td><strong>Mapping operator</strong></td>
<td></td>
</tr>
<tr>
<td><code>$0.field1 := $0.field1;</code> (please note ':'=' vs '=')</td>
<td><code>$0.field1 = $0.field1;</code> (please note ':'=' vs '=')</td>
</tr>
<tr>
<td><strong>Accessing input records</strong></td>
<td></td>
</tr>
<tr>
<td><code>@&lt;port No&gt;</code></td>
<td>unavailable, may be replaced with:</td>
</tr>
<tr>
<td></td>
<td><code>$&lt;port No&gt;.*</code></td>
</tr>
<tr>
<td><code>@&lt;metadata name&gt;</code></td>
<td>unavailable, may be replaced with:</td>
</tr>
<tr>
<td></td>
<td><code>$&lt;metadata name&gt;.*</code></td>
</tr>
<tr>
<td><strong>Accessing field values</strong></td>
<td></td>
</tr>
<tr>
<td><code>@&lt;port No&gt;[&lt;field No&gt;]</code></td>
<td>unavailable, may be replaced with:</td>
</tr>
<tr>
<td></td>
<td><code>$&lt;port No&gt;.&lt;field name&gt;</code></td>
</tr>
<tr>
<td><code>@&lt;metadata name&gt;[&lt;field No&gt;]</code></td>
<td>unavailable, may be replaced with:</td>
</tr>
<tr>
<td></td>
<td><code>$&lt;metadata name&gt;.&lt;field name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;record variable name&gt;&quot;&lt;field name&gt;&quot;</code></td>
<td>unavailable, may be replaced with:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;record variable name&gt;.&lt;field name&gt;</code></td>
</tr>
<tr>
<td><strong>Conditional fail expression (interpreted mode only)</strong></td>
<td>$0.field1 := expr1 : expr2 : ... : exprN;</td>
</tr>
<tr>
<td>unavailable</td>
<td>myVar = expr1 : expr2 : ... : exprN;</td>
</tr>
<tr>
<td>unavailable</td>
<td>myFunction(expr1 : expr2 : ... : exprN)</td>
</tr>
<tr>
<td><strong>Dictionary declaration</strong></td>
<td></td>
</tr>
</tbody>
</table>
## Chapter 61. CTL1 vs. CTL2 Comparison

### CTL1

<table>
<thead>
<tr>
<th>need not be defined</th>
<th>must always be defined</th>
</tr>
</thead>
</table>

### Dictionary entry types

| string, readable.channel, writable.channel | boolean, byte, date, decimal, integer, long, number, string, readable.channel, writable.channel, object |

### Writing to dictionary

<table>
<thead>
<tr>
<th>signature:</th>
<th>syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write_dict(string name, string value)</td>
<td>dictionary.&lt;entry name&gt; = value;</td>
</tr>
</tbody>
</table>

**Example 1:**
```
write_dict("customer", "John Smith");
```

**Example 2:**
```
string customer;
write_dict(customer, "John Smith");
```

<table>
<thead>
<tr>
<th>signature:</th>
<th>syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean dict_put_str(string name, string value);</td>
<td></td>
</tr>
</tbody>
</table>

**Example 3:**
```
dict_put_str("customer", "John Smith");
```

**Example 4:**
```
string customer;
dict_put_str(customer, "John Smith");
```

### Reading from dictionary

<table>
<thead>
<tr>
<th>signature:</th>
<th>syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>string read_dict(string name)</td>
<td>value = dictionary.&lt;entry name&gt;;</td>
</tr>
</tbody>
</table>

**Example 1:**
```
string myString;
myString = read_dict("customer");
```

**Example 2:**
```
string myString;
string customer;
myString = read_dict(customer);
```

<table>
<thead>
<tr>
<th>signature:</th>
<th>syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>string dict_get_str(string name)</td>
<td></td>
</tr>
</tbody>
</table>

**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) | 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) | unavailable |

### Sequence functions

| sequence(<sequence ID>).current | sequence(<sequence name>).current() |
### Chapter 61. CTL1 vs. CTL2 Comparison

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence(&lt;sequence ID&gt;).next</td>
<td>sequence(&lt;sequence name&gt;).next()</td>
</tr>
<tr>
<td>sequence(&lt;sequence ID&gt;).reset</td>
<td>sequence(&lt;sequence name&gt;).reset()</td>
</tr>
</tbody>
</table>

#### 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]
```

#### Following set of optional functions can be used in both CTL1 and CTL2:

<required template function>OnError() (e.g. transformOnError(), etc.)

#### Jump statements

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>break;</td>
</tr>
<tr>
<td>continue</td>
<td>continue;</td>
</tr>
<tr>
<td>return Expression</td>
<td>return Expression;</td>
</tr>
</tbody>
</table>

#### Contained-in operator

```
myVar .in. myContainer
```

```
in(myVar,myContainer)
or
myVar.in(myContainer)
```

#### Eval functions

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval()</td>
<td>unavailable</td>
</tr>
<tr>
<td>eval_exp()</td>
<td>unavailable</td>
</tr>
</tbody>
</table>

#### Ternary operator

```
unavailable
```

```
Condition ? ExprIfTrue : ExprIfFalse
```

```
but
```

```
but
```

```
iif(Condition,ExprIfTrue,ExprIfFalse)
```

```
iif(Condition,ExprIfTrue,ExprIfFalse)
```

```
can be used instead
```

```
also exists
```

#### Legend:

1) These functions do nothing since version 3.0 of CloverETL and can be removed from the code.
Chapter 62. 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 interpreted 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

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted mode</td>
<td></td>
</tr>
<tr>
<td>//#TL</td>
<td>//#CTL2</td>
</tr>
<tr>
<td>//#CTL1</td>
<td></td>
</tr>
</tbody>
</table>

Compiled mode

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unavailable</td>
<td>//#CTL2:COMPILED</td>
</tr>
</tbody>
</table>

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), precision 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.

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer data type</td>
<td></td>
</tr>
<tr>
<td>int myInt;</td>
<td>integer myInt;</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte data type</td>
<td></td>
</tr>
<tr>
<td>bytearray myByte;</td>
<td>byte myByte;</td>
</tr>
</tbody>
</table>
### 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 follows 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 from CTL1 syntax to that of CTL2, replace the older declaration 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 identified with their ID, in CTL2 metadata are identified by their unique name.
Chapter 62. Migrating
CTL1 to CTL2

See the table below on how records may be declared in CTL1 and in CTL2.

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration of a list</strong></td>
<td></td>
</tr>
<tr>
<td>list myList;</td>
<td>&lt;element type&gt;[] myList;</td>
</tr>
<tr>
<td>e.g.: string[] myList;</td>
<td></td>
</tr>
<tr>
<td><strong>Declaration of a map</strong></td>
<td></td>
</tr>
<tr>
<td>map myMap;</td>
<td>map[type of key, type of value] myMap;</td>
</tr>
<tr>
<td>e.g.: map[string, integer] myMap;</td>
<td></td>
</tr>
<tr>
<td><strong>Declaration of a record</strong></td>
<td></td>
</tr>
<tr>
<td>record (&lt;metadata ID&gt;) myRecord;</td>
<td>&lt;metadata name&gt; myRecord;</td>
</tr>
<tr>
<td>record (@&lt;port number&gt;) myRecord;</td>
<td></td>
</tr>
<tr>
<td>record (@&lt;metadata name&gt;) myRecord;</td>
<td></td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>function transform(idx) {</td>
<td>function integer transform(integer idx) {</td>
</tr>
<tr>
<td>&lt;other function body&gt;</td>
<td>&lt;other function body&gt;</td>
</tr>
<tr>
<td>$0.customer := $0.customer;</td>
<td>$0.customer = $0.customer;</td>
</tr>
<tr>
<td>}</td>
<td>return 0;</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;record variable name&gt; = @&lt;port number&gt;;</td>
<td>&lt;record variable name&gt;.* = $&lt;port number&gt;.*;</td>
</tr>
<tr>
<td>&lt;record variable name&gt; = @&lt;metadata name&gt;</td>
<td>&lt;record variable name&gt;.* = $&lt;metadata name&gt;.*;</td>
</tr>
</tbody>
</table>

**Note**

Note that you should also change the syntax of groupAccumulator usage in Rollup.
Chapter 62. Migrating CTL1 to CTL2

### 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.

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupAccumulator[&quot;&lt;field name&gt;&quot;]</td>
<td>groupAccumulator.&lt;field name&gt;</td>
</tr>
</tbody>
</table>

### Example 62.1. Example of dictionary usage

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Writing to dictionary</strong></td>
<td></td>
</tr>
<tr>
<td>write_dict(string &lt;entry name&gt;, string &lt;entry value&gt;);</td>
<td></td>
</tr>
<tr>
<td>dict_put_str(string &lt;entry name&gt;, string &lt;entry value&gt;);</td>
<td></td>
</tr>
<tr>
<td><strong>Reading from dictionary</strong></td>
<td></td>
</tr>
<tr>
<td>string myVariable;</td>
<td></td>
</tr>
<tr>
<td>myVariable = read_dict(&lt;entry name&gt;);</td>
<td></td>
</tr>
<tr>
<td>string myVariable;</td>
<td></td>
</tr>
<tr>
<td>myVariable = dict_get_str(&lt;entry name&gt;);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CTL1</th>
<th>CTL2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Writing to dictionary</strong></td>
<td></td>
</tr>
<tr>
<td>write_dict(&quot;Mount_Everest&quot;, &quot;highest&quot;);</td>
<td></td>
</tr>
<tr>
<td>dict_put_str(&quot;Mount_Everest&quot;, &quot;highest&quot;);</td>
<td></td>
</tr>
<tr>
<td><strong>Reading from dictionary</strong></td>
<td></td>
</tr>
<tr>
<td>string myVariable;</td>
<td></td>
</tr>
<tr>
<td>myVariable = read_dict(&quot;Mount_Everest&quot;);</td>
<td></td>
</tr>
<tr>
<td>string myVariable;</td>
<td></td>
</tr>
<tr>
<td>myVariable = dict_get_str(&quot;Mount_Everest&quot;);</td>
<td></td>
</tr>
</tbody>
</table>

### Step 7: Add semicolons where necessary

In CTL1, 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. 878) for list of CTL2 functions.

<table>
<thead>
<tr>
<th>CTL1 function</th>
<th>CTL2 function</th>
</tr>
</thead>
<tbody>
<tr>
<td>uppercase()</td>
<td>upperCase()</td>
</tr>
<tr>
<td>bit_invert()</td>
<td>bitNegate()</td>
</tr>
</tbody>
</table>
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 curly 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 curly 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.

<table>
<thead>
<tr>
<th>CTL version</th>
<th>Switch statement syntax</th>
</tr>
</thead>
</table>
| CTL1        |```intel 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        |```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 should change the code containing lookup table or sequence syntax. Note that these are identified with either IDs (in CTL1) or with their names.

<table>
<thead>
<tr>
<th>CTL version</th>
<th>Sequence syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL1</td>
<td>// Transforms input record into output record.</td>
</tr>
<tr>
<td></td>
<td>function transform() {</td>
</tr>
<tr>
<td></td>
<td>$0.field1 := $0.field1;</td>
</tr>
<tr>
<td></td>
<td>$0.field2 := $0.field2;</td>
</tr>
<tr>
<td></td>
<td>$0.field3 := sequence(Sequence0).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field4 := sequence(Sequence0).next();</td>
</tr>
<tr>
<td></td>
<td>$0.field5 := sequence(Sequence0, string).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field6 := sequence(Sequence0, string).next();</td>
</tr>
<tr>
<td></td>
<td>$0.field7 := sequence(Sequence0, long).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field8 := sequence(Sequence0, long).next();</td>
</tr>
<tr>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>CTL2</td>
<td>// Transforms input record into output record.</td>
</tr>
<tr>
<td></td>
<td>function integer transform() {</td>
</tr>
<tr>
<td></td>
<td>$0.field1 = $0.field1;</td>
</tr>
<tr>
<td></td>
<td>$0.field2 = $0.field2;</td>
</tr>
<tr>
<td></td>
<td>$0.field3 = sequence(seqCTL2).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field4 = sequence(seqCTL2).next();</td>
</tr>
<tr>
<td></td>
<td>$0.field5 = sequence(seqCTL2, string).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field6 = sequence(seqCTL2, string).next();</td>
</tr>
<tr>
<td></td>
<td>$0.field7 = sequence(seqCTL2, long).current();</td>
</tr>
<tr>
<td></td>
<td>$0.field8 = sequence(seqCTL2, long).next();</td>
</tr>
<tr>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
### CTL1

```java
// 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
}
```
###CTL2

```plaintext
// 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:

```plaintext
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:

```plaintext
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.

<table>
<thead>
<tr>
<th>CTL version</th>
<th>Transformation with mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>//=TL</td>
</tr>
<tr>
<td></td>
<td>int retInt;</td>
</tr>
<tr>
<td></td>
<td>function transform() {</td>
</tr>
<tr>
<td></td>
<td>if ($0.field3 &lt; 5) retInt = 0; else retInt = 1;</td>
</tr>
<tr>
<td></td>
<td>// the following part is the mapping:</td>
</tr>
<tr>
<td></td>
<td>$0.* := $0.*;</td>
</tr>
<tr>
<td></td>
<td>$0.field1 := uppercase($0.field1);</td>
</tr>
<tr>
<td></td>
<td>$1.* := $0.*;</td>
</tr>
<tr>
<td></td>
<td>$1.field1 := uppercase($0.field1);</td>
</tr>
<tr>
<td></td>
<td>return retInt</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>CTL2</td>
<td>//=CTL2</td>
</tr>
<tr>
<td></td>
<td>integer retInt;</td>
</tr>
<tr>
<td></td>
<td>function integer transform() {</td>
</tr>
<tr>
<td></td>
<td>// the following part is the mapping:</td>
</tr>
<tr>
<td></td>
<td>$0.* = $0.*;</td>
</tr>
<tr>
<td></td>
<td>$0.field1 = upperCase($0.field1);</td>
</tr>
<tr>
<td></td>
<td>$1.* = $0.*;</td>
</tr>
<tr>
<td></td>
<td>$1.field1 = upperCase($0.field1);</td>
</tr>
<tr>
<td></td>
<td>if ($0.field3 &lt; 5) return = 0; else return 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Chapter 63. 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. 789)
- Functions Reference (p. 819)

Example 63.1. Example of CTL1 syntax (Rollup)

```plaintext
//#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. 790)
- Comments (p. 790)
- Import (p. 790)
- Data Types in CTL (p. 791)
- Literals (p. 793)
- Variables (p. 795)
- Operators (p. 796)
- Simple Statement and Block of Statements (p. 801)
- Control Statements (p. 801)
- Functions (p. 806)
- Conditional Fail Expression (p. 808)
- Accessing Data Records and Fields (p. 809)
- Mapping (p. 812)
- Parameters (p. 818)
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. 793) 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. 108)

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 12 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. 809).

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 63.1. Literals

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
<th>Declaration syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>digits representing integer number</td>
<td>[0-9]+</td>
<td>95623</td>
</tr>
<tr>
<td>long integer</td>
<td>digits representing integer numbers with absolute value even greater than 2^{31}, but less than 2^{63}</td>
<td>[0-9]+L?</td>
<td>257L, 9562307813123123</td>
</tr>
<tr>
<td>hexadecimal integer</td>
<td>digits and letters representing integer number in hexadecimal form</td>
<td>0x[0-9A-F]+</td>
<td>0xA7B0</td>
</tr>
<tr>
<td>octal integer</td>
<td>digits representing integer number in octal form</td>
<td>0[0-7]*</td>
<td>0644</td>
</tr>
<tr>
<td>number</td>
<td>floating point number represented by 64bits in double precision format</td>
<td>[0-9]+.[0-9]++</td>
<td>456.123</td>
</tr>
<tr>
<td>decimal</td>
<td>digits representing a decimal number</td>
<td>[0-9]+.[0-9]+D</td>
<td>123.456D</td>
</tr>
<tr>
<td>double quoted string</td>
<td>string value/literal enclosed in double quotes; escaped characters [\n,\r,\t, ', &quot;, \b] get translated into corresponding control chars</td>
<td>&quot;...anything except ['&quot;]...&quot;</td>
<td>&quot;hello\tworld\n\r&quot;</td>
</tr>
<tr>
<td>single quoted string</td>
<td>string value/literal enclosed in single quotes; only one escaped character ['] gets translated into corresponding char [']</td>
<td>¦...anything except ['']...¦</td>
<td>¦hello\tworld\n\r¦</td>
</tr>
<tr>
<td>list of literals</td>
<td>list of literals where individual literals can also be other lists/maps/records</td>
<td>[ &lt;any literal&gt; , &lt;any literal&gt; ]*</td>
<td>[10, 'hello', &quot;world&quot;, 0x1A, 2008-01-01, [1, 2], [3, 4]]</td>
</tr>
<tr>
<td>date</td>
<td>date value</td>
<td>this mask is expected: yyyy-MM-dd</td>
<td>2008-01-01</td>
</tr>
<tr>
<td>datetime</td>
<td>datetime value</td>
<td>this mask is expected: yyyy-MM-dd HH:mm:ss</td>
<td>2008-01-01 18:55:00</td>
</tr>
</tbody>
</table>

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 apply it on an argument value.

For information on these conversion functions see Conversion Functions (p. 820)

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. 796)
- Relational Operators (p. 798)
- Logical Operators (p. 800)

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 \texttt{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 multiply 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 a string and a regular expression (p. 921). It returns \texttt{true} if the whole string matches the regular expression, otherwise returns \texttt{false}.

\[ =~ \]
\[ .regex. \]

• 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:

  ```c
  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:

  ```c
  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. 801)
- **Iteration Statements** (p. 802)
- **Jump Statements** (p. 803)

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

```c
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:

```c
if (Condition) Statement1 else Statement2
```

The Statement2 can even be another if statement and also with else branch:

```c
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.

```java
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.

```java
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:

```java
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.

```java
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.

```java
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.

```java
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:

```java
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:

```java
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:

```java
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:

```csharp
string MyException;
```

After that, in the code of your transformation, you can use the following `try-catch` statement:

```csharp
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:

```csharp
appendOnError(), transformOnError(), etc.
```

Each of these required functions may have its (optional) counterpart 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:

```javascript
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. 804). 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.

```javascript
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 identifiers and even terminate expressions with a semicolon. The evaluated expression has a limited scope and cannot see variables declared outside of it.

**Example 63.2. Eval() Function Examples**

**Example 1:**

```c
int value = eval_exp("2+5"); // the result of the expression is stored to 'value'
```

**Example 2:**

```c
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):

```plaintext
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:

```c
int i;
for (i=0; i<length(@0); i++) {
    print_err(@0[i]);
}
```

You can also define records in CTL code. Such definitions 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 63.3. 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 63.4. 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.

```c
//#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 \texttt{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 63.5. Example of Mapping with Wild Cards**

The mapping must also be defined at the end of a function (the \texttt{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 \texttt{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 \texttt{below} the mapping with wild cards.

This is useful when there are many unchanged fields and a few that will be changed.

\[
\text{//#TL}
\]

\[
// declare variable for returned value (number of output port)
int retInt;
\]

\[
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 := strtoupper($0.field1);
    $1.* := $0.*;  
    $1.field1 := strtoupper($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 63.6. 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.

```cpp
// 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 63.7. 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 variabile 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.

```c
//#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. 213).
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. 820)
- **Date Functions** (p. 825)
- **Mathematical Functions** (p. 828)
- **String Functions** (p. 832)
- **Miscellaneous Functions** (p. 842)
- **Dictionary Functions** (p. 844)
- **Lookup Table Functions** (p. 845)
- **Sequence Functions** (p. 847)
- **Custom CTL Functions** (p. 848)

**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. 159) 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 Date and Time Format (p. 110).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 117).
- For detailed information about locale see Locale (p. 123).

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. 85).

Here we provide the list of these functions:

- bytearray 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 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 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> <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 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 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 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` 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` 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` 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` 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` 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` function accepts three arguments: string, the name of any numeric data type and string. 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` 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.

* return datatype try_convert(<any type> from, typename return datatype);

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, nameof datetype, 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, string typename string, string format);

The try_convert(date, nameof string datatype, 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 a 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 Date and Time Format (p. 110).

• For detailed information about locale see Locale (p. 123).

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. 85).

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`. 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.
The trunc(<numeric type>) function takes one argument of any numeric data type and returns its truncated long value.

The trunc(list) function takes one list argument, empties its values and returns null.

The trunc(map) function takes one map argument, empties its values and returns null.

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:

- \texttt{<numeric type> abs(<numeric type> arg);}

  The \texttt{abs(<numeric type>)} function takes one argument of any numeric data type and returns its absolute value.

- \texttt{long bit_and(<numeric type> arg1, <numeric type> arg2);}

  The \texttt{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, \texttt{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, which corresponds to decimal 3. Return data type is \texttt{long}, but if it is sent to other numeric data type, it is expressed in its numeric representation.

- \texttt{long bit_invert(<numeric type> arg);}

  The \texttt{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, \texttt{bit_invert(11)} returns -12.) The function inverts all bits in an argument. Return data type is \texttt{long}, but if it is sent to other numeric data type, it is expressed in its numeric representation.

- \texttt{boolean bit_is_set(<numeric type> arg, <numeric type> Index);}

  The \texttt{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 \texttt{Index} and returns \texttt{true} or \texttt{false}, if the bit is 1 or 0, respectively.
  (For example, \texttt{bit_is_set(11, 3)} returns \texttt{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 \texttt{true}. And \texttt{bit_is_set(11, 2)} would return \texttt{false}.

- \texttt{long bit_lshift(<numeric type> arg, <numeric type> Shift);}

  The \texttt{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 (\texttt{Shift} number of bits on the left side are added and set to 0.)
  (For example, \texttt{bit_lshift(11, 2)} returns 44.) As decimal 11 can be expressed as bitwise 1011, the two bits on the right side (10) are added and the result is 101100, which corresponds to decimal 44. Return data type is \texttt{long}, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- \texttt{long bit_or(<numeric type> arg1, <numeric type> arg2);}

  The \texttt{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, \texttt{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, which corresponds to decimal 15. Return data type is \texttt{long}, but if it is sent to other numeric data type, it is expressed in its numeric data type.

- \texttt{long bit_rshift(<numeric type> arg, <numeric type> Shift);}

  The \texttt{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 (\texttt{Shift} number of bits on the right side are removed.)
  (For example, \texttt{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, which corresponds to decimal 2. Return data type is \texttt{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 arguments 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 Date and Time Format (p. 110).

• For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 117).

• For detailed information about locale see Locale (p. 123).

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. 85).

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://docs.oracle.com/javase/6/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://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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 \( \text{maxDifference} + 1 \) 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 \[ http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html \] for details about \textit{Locale}.

- list \textit{find}(string \textit{arg}, string \textit{regex});

The \textit{find}(string, string) function accepts two string arguments. The second one is regular expression (p. 921). 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 \textit{get_alphanumeric_chars}(string \textit{arg});

The \textit{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 \textit{get_alphanumeric_chars}(string \textit{arg}, boolean \textit{takeAlpha}, boolean \textit{takeNumeric});

The \textit{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 \textit{index_of}(string \textit{arg}, string \textit{substring});

The \textit{index_of}(string, string) function accepts two strings. It takes them and returns the index of the first appearance of \textit{substring} in the string specified as the first argument.

- int \textit{index_of}(string \textit{arg}, string \textit{substring}, int \textit{fromIndex});

The \textit{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 \textit{substring} counted from the character located at the position specified by the third argument.

- boolean \textit{is_ascii}(string \textit{arg});

The \textit{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 \textit{is_blank}(string \textit{arg});

The \textit{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 \textit{is_date}(string \textit{arg}, string \textit{pattern});

The \textit{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 \textit{locale}, according to the specified \textit{pattern}, the function returns true. If it is not possible, it returns false.

(For more details, see another version of the \textit{is_date()} function below - the \textit{is_date}(string, string, string, boolean) function.)

This function is a variant of the mentioned \textit{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://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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, 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://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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](http://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](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 (p. 921), 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 (p. 921). 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:

1. **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.

2. **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.

3. **<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).

4. **<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).

5. **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`.

6. **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.

7. **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 \texttt{reverse} (list \texttt{arg});

The \texttt{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 \texttt{sort} (list \texttt{arg});

The \texttt{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. 159) 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:

```c
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)**

  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)**

  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:

\texttt{\texttt{lookup(<lookup ID>,keyValue).<field name>}}

\texttt{\texttt{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 \texttt{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/inherited 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](http://wiki.cloveretl.org/doku.php?id=function_building).
Chapter 64. 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. 850)
- Functions Reference (p. 878)

Example 64.1. Example of CTL2 syntax (Rollup)

```csharp
//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;
}
```
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. 851)
- Comments (p. 851)
- Import (p. 851)
- Data Types in CTL2 (p. 852)
- Literals (p. 855)
- Variables (p. 857)
- Dictionary in CTL2 (p. 858)
- Operators (p. 859)
- Simple Statement and Block of Statements (p. 864)
- Control Statements (p. 864)
- Functions (p. 869)
- Conditional Fail Expression (p. 870)
- Accessing Data Records and Fields (p. 871)
- Mapping (p. 873)
- Parameters (p. 877)
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. 855) 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. 108)

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 64.2. 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 12 and 2, respectively), it will be converted to 33.33D.

You can cast any float number to the decimal data type by appending the d letter to its end.

**integer**

Its declaration looks like this: `integer identifier;`

If you append 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:

```java
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 filled 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 lists 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. 854).

**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. 871).

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 64.1. Literals

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
<th>Declaration syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>digits representing integer number</td>
<td>[0-9]+</td>
<td>95623</td>
</tr>
<tr>
<td>long integer</td>
<td>digits representing integer number with absolute value even greater than $2^{31}$, but less than $2^{63}$</td>
<td>[0-9]+\L?</td>
<td>257L, or 9562307813123123</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>digits and letters representing integer number in hexadecimal form</td>
<td>0x[0-9A-F]+</td>
<td>0xA7B0</td>
</tr>
<tr>
<td>octal</td>
<td>digits representing integer number in octal form</td>
<td>0[0-7]*</td>
<td>0644</td>
</tr>
<tr>
<td>number</td>
<td>floating point number represented by 64bits in double precision format</td>
<td>[0-9]+.[0-9]+</td>
<td>456.123</td>
</tr>
<tr>
<td>decimal</td>
<td>digits representing a decimal number</td>
<td>[0-9]+.[0-9]+D</td>
<td>123.456D</td>
</tr>
<tr>
<td>double quoted</td>
<td>string value/literal enclosed in double quotes; escaped characters [n\r\n\t,\,&quot;,\b] get translated into corresponding control chars</td>
<td>&quot;...anything except [&quot;]...&quot;</td>
<td>&quot;hello\tworld\n\r&quot;</td>
</tr>
<tr>
<td>single quoted</td>
<td>string value/literal enclosed in single quotes; only one escaped character ['] gets translated into corresponding char [']</td>
<td>'...anything except [']...'</td>
<td>'hello\tworld\n\r'</td>
</tr>
<tr>
<td>list of literals</td>
<td>list of literals where individual literals can also be other lists/maps/records</td>
<td>(&lt;any literal&gt;, &lt;any literal&gt;)* ]</td>
<td>[10, 'hello', &quot;world&quot;, 0xA, 2008-01-01, [ [ 1, 2 ] ], [ 3, 4 ]]</td>
</tr>
<tr>
<td>date</td>
<td>date value</td>
<td>this is expected: yyyy-MM-dd</td>
<td>2008-01-01</td>
</tr>
<tr>
<td>datetime</td>
<td>datetime value</td>
<td>this mask is expected: yyyy-MM-dd HH:mm:ss</td>
<td>2008-01-01 18:55:00</td>
</tr>
</tbody>
</table>

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 apply it on an argument value.

For information on these conversion functions see Conversion Functions (p. 880)

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 access an entry from CTL2, you must define it in the graph as shown in Chapter 29, Dictionary (p. 218).

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. 859)
• Relational Operators (p. 861)
• Logical Operators (p. 863)

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 multiply 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. These signs can be used more times in one expression. In such a case you can express priority of comparisons by parentheses.

Important

If you choose the .operator. syntax, operator must be surrounded by white spaces. Example syntax for the eq operator:

- 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.

```plaintext
>  
.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.

```plaintext
>=  
=>  
.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.

```plaintext
<  
.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.
• 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.

!=
<>.
eq.

• Matches regular expression

The operator serves to compare string and some regular expression (p. 921). It returns true if the whole string matches the regular expression, otherwise returns false.

~=.
.regex.

• Contains regular expression

The operator serves to compare string and some regular expression (p. 921). It returns true if the string contains a substring that matches the regular expression, otherwise returns false.

?=

<=
<=
.le.

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.`

Assignment Operator

As of Clover 3.3, the = operator does not just pass object references, but performs a deep copy of values. That is of course more demanding in terms of performance.

Example 64.3. Clearing a copied list

```java
    integer[] list1 = [1, 2, 3];
    integer[] list2;
    list2 = list1;

    list1.clear(); // only list1 is cleared (older implementation: list2 was cleared,
```


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. 864)
- **Iteration Statements** (p. 865)
- **Jump Statements** (p. 866)

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.

```plaintext
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:

```plaintext
if (Condition) Statement1 else Statement2
```

The Statement2 can even be another if statement and also with else branch:

```plaintext
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 corresponding 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 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.

Note

It is possible to iterate over values of a map (i.e. not whole <entries>). Remember the type of the loop variable has to match the type of map's values:

map[string, integer] myMap;
myMap["first"] = 1;
myMap["second"] = 2;

foreach(integer value: myMap) {
  printErr(value); // prints 1 and 2
}

To obtain map's keys as a list [], use the getKeys() (p. 903) function.

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:

```java
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.

```java
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) counterpart 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:

```javascript
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. 867). 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.

```javascript
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

**Important**

Since v. 3.2, the syntax has changed to:

$\text{in.portID.fieldID}$ and $\text{out.portID.fieldID}$

e.g. $\text{in.0.* = out.0.*}$;

That way, you can clearly distinguish input and output metadata.

Transformations you have written before will be compatible with the old syntax.

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:

- $\langle\text{port number}\rangle.\langle\text{field number}\rangle$
  
  Example: $0.2$
  
  $0.*$ means all fields on the first port (port 0).

- $\langle\text{port number}\rangle.\langle\text{field name}\rangle$
  
  Example: $0.firstname$

- $\langle\text{metadata name}\rangle.\langle\text{field number}\rangle$
  
  Example: $customers.2$
  
  $customers.*$ means all fields on the first port (port 0).

- $\langle\text{metadata name}\rangle.\langle\text{field name}\rangle$
  
  Example: $customers.firstname$

You can also define records in CTL code. Such definitions can look like these:

- $\langle\text{metadata name}\rangle$ MyCTLRecord;
  
  Example: $customers$ myCustomers;

- After that, you can use the following expressions:
  
  $\langle\text{record variable name}\rangle.\langle\text{field name}\rangle$
  
  Example: $myCustomers.firstname$;

Mapping of records to variables looks like this:

- $\text{myVariable = }\langle\text{port number}\rangle.\langle\text{field number}\rangle;$
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:

```plaintext
$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:

```plaintext
void copyByName(record to, record from);
```

**Example 64.4. Mapping of Metadata by Name (using the copyByName() function)**

```plaintext
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 output, 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 use the following function:

```c
void copyByPosition(record to, record from);
```

Example 64.5. Mapping of Metadata by Position

```c
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 64.6. 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.

```javascript
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 64.7. 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.

```javascript
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 64.8. 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.

```c
// 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. 213).
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. 880)
- **Container Functions** (p. 902)
- **Record functions (dynamic field access)** (p. 906)
- **Date Functions** (p. 887)
- **Mathematical Functions** (p. 889)
- **String Functions** (p. 893)
- **Miscellaneous Functions** (p. 910)
- **Lookup Table Functions** (p. 914)
- **Sequence Functions** (p. 917)
- **Custom CTL Functions** (p. 918)

**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.:

```plaintext
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. 910).
**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. 159) 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 Date and Time Format (p. 110).

• For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 117).

• For detailed information about locale see Locale (p. 123).

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. 85).

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.

• string byte2str(byte payload, string charset);

  Returns bytes converted to string using a given charset.

• 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. 123) 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(975922372036854775807.25)` 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.

- **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 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-1 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-1 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`.

• **string str2byte**(string payload, string charset);

Returns a string converted from input bytes using a given charset encoder.

• **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);

• **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 `str2double(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 \texttt{str2long(string arg, integer \, \textit{radix})};

The \texttt{str2long} function accepts two arguments: \texttt{string} and \texttt{integer}. It takes the first argument as if it were expressed in the \texttt{radix} based numeric system representation and returns its corresponding long decimal value.

• long \texttt{str2long(string arg, string \, \textit{format})};

The \texttt{str2long} function takes the first \texttt{string} argument as decimal string representation of a long number corresponding to the \texttt{format} specified as the second argument and the system locale and converts it to the corresponding long value.

• long \texttt{str2long(string arg, string \, \textit{format}, string \, \textit{locale})};

The \texttt{str2long} function takes the first \texttt{string} argument as decimal string representation of a long number corresponding to the \texttt{format} specified as the second argument and the \texttt{locale} specified as the third argument and converts it to the corresponding long value.

• string \texttt{toString(<numeric|list|map type> \, \textit{arg})};

The \texttt{toString} 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 \texttt{xml2json(string arg)};

The \texttt{xml2json} function takes one \texttt{string} argument that is XML formatted and converts it to a JSON formatted string. Its counterpart is the \texttt{json2xml} 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 Date and Time Format (p. 110).

• For detailed information about locale see Locale (p. 123).

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. 85).

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. The function's argument is not modified by the return value.

• date extractTime(date arg);

The extractTime(date) function takes one date argument and returns only the information containing hours, minutes, seconds, and milliseconds. The function's argument is not modified by the return value.

• 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 string arguments. 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 `zeroDate()``

The `zeroDate()` function accepts no argument and returns 1.1.1970.

**Note**

The following two functions are deprecated. Their return value modifies the argument at the same 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.
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, 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, 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 16.

• 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(numerictype)` 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 Date and Time Format (p. 110).
- For detailed information about formatting and/or parsing of any numeric data type see Numeric Format (p. 117).
- For detailed information about locale see Locale (p. 123).

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. 85).

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, \texttt{cut("somestringasanexample", [2,3,1,5])} returns \texttt{["mes","omest"]}.

- \textbf{integer \texttt{editDistance}(string \texttt{arg1}, string \texttt{arg2});}

  The \texttt{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 \texttt{maxDifference + 1} as the return value.

  For more details, see another version of the \texttt{editDistance()} function below - the \texttt{editDistance (string, string, integer, string, integer)} (p. 895) function.

- \textbf{integer \texttt{editDistance}(string \texttt{arg1}, string \texttt{arg2}, string \texttt{locale});}

  The \texttt{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 \texttt{maxDifference + 1} as the return value.

  For more details, see another version of the \texttt{editDistance()} function below - the \texttt{editDistance (string, string, integer, string, integer)} (p. 895) function.

  See \texttt{http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html} for details about \texttt{Locale}.

- \textbf{integer \texttt{editDistance}(string \texttt{arg1}, string \texttt{arg2}, integer \texttt{strength});}

  The \texttt{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 \texttt{maxDifference + 1} as the return value.

  For more details, see another version of the \texttt{editDistance()} function below - the \texttt{editDistance (string, string, integer, string, integer)} (p. 895) function.

- \textbf{integer \texttt{editDistance}(string \texttt{arg1}, string \texttt{arg2}, integer \texttt{strength}, string \texttt{locale});}

  The \texttt{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 \texttt{maxDifference + 1} as the return value.

  For more details, see another version of the \texttt{editDistance()} function below - the \texttt{editDistance (string, string, integer, string, integer)} (p. 895) function.
See [http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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)` function.

  See [http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html](http://docs.oracle.com/javase/6/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)` 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 \texttt{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. 919).

See \url{http://docs.oracle.com/javase/6/docs/api/java/util/Locale.html} for details about \texttt{Locale}.

- \texttt{string \_escapeUrl(string \_arg)};

  The function escapes illegal characters within components of specified URL (see \texttt{isUrl()} CTL2 function (p. 898) 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.

- \texttt{string[] \_find(string \_arg, string \_regex)};

  The \_find(string, string) function accepts two string arguments. The second one is a regular expression (p. 921). 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.

- \texttt{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.

- \texttt{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.

- \texttt{string \_getFieldLabel(reference \_record, string \_arg)};

  The function returns a label of a field whose name is specified in \_arg. The fields are taken from \_record.

- \texttt{string \_getFieldLabel(reference \_record, integer \_arg)};

  The function returns a label of a field whose index is specified in \_arg. The fields are taken from \_record.

- \texttt{string \_getUrlHost(string \_arg)};

  The function parses out host name from specified URL (see \texttt{isUrl()} CTL2 function (p. 898) 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, \texttt{null} is returned.

- \texttt{string \_getUrlPath(string \_arg)};

  The function parses out path from specified URL (see \texttt{isUrl()} CTL2 function (p. 898) 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, \texttt{null} is returned.

- \texttt{integer \_getUrlPort(string \_arg)};

  The function parses out port number from specified URL (see \texttt{isUrl()} CTL2 function (p. 898) 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.

- \texttt{string \_getUrlProtocol(string \_arg)};
The function parses out protocol name from specified URL (see isUrl() CTL2 function (p. 898) 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. 898) 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. 898) 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. 898) 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, substring) function returns the index (zero-based) of the first occurrence of substring in the string. Returns -1 if no occurrence is found.

• integer indexOf(string arg, string substring, integer fromIndex);

The indexOf(string, substring, fromIndex) function returns the index (zero-based) of the first occurrence of substring in the string, starting from fromIndex. Returns -1 if no occurrence is found.

• 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.
Chapter 64. CTL2

• 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://docs.oracle.com/javase/6/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
```

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 being 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 (p. 921). 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 function returns a string consisting of lowercase letters. Its length is between `<minLength; maxLength>`. Characters in the generated string always belong to `[a-z]` (no special symbols).

  Example random string: `qjfxq`

- `string randomUUID();`

  Generates a random but undoubtedly unique string identifier. The generated string has this format:

  ```
  hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhh
  ```

  where `h` belongs to `[0-9a-f]`. In other words, you generate hexadecimal code of a random 128bit number.

  Example generated string: `cee188a3-aa67-4a68-bcd2-52f3ec0329e6`

  For more details on the algorithm used, browse the Java documentation.

- `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 (p. 921), 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 3rd 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".
Chapter 64. CTL2

- `string[] split(string arg, string regex);`

  The `split(string, string)` function accepts two string arguments. The second is some regular expression (p. 921). 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. 898) 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 working with containers (list, map, record), you will typically use the 'contained in' function call - in. This call identifies whether a value is contained in a list or a map of other values. There are two syntactic options:

```java
boolean myBool;
string myString;
string[] myList;
...

myBool = myString.in(myList)

myBool = in(myString, myList);
```

In the table below, examine (im)possible ways of using in:

<table>
<thead>
<tr>
<th>Code</th>
<th>Working?</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot; in [&quot;a&quot;, &quot;b&quot;]</td>
<td>true</td>
</tr>
<tr>
<td>in(10, [10, 20])</td>
<td>true</td>
</tr>
<tr>
<td>10 in [10, 20]</td>
<td>false</td>
</tr>
</tbody>
</table>

As for lists and maps in metadata, use in like this:

```java
"abc" in "$in.0.listField"  // operator syntax for list field
"abc" in "$in.0.mapField"  // operator syntax for map field
```

**Note**

The operator syntax has a con. Searching through lists will always be slow (since it has to be linear).

**Functions you can use with containers**

- `<element type>[] 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.

- **boolean containsAll(<element type>[] list, <element type>[] subList);**
  
The `containsAll(<element type>[][], <element type>[][])` function returns `true` if the list passed as the first argument contains every element of the second passed list, i.e. the second list is a sublist of the first list.

- **boolean containsKey(map[type of key],<type of value>] map, <type of key> key);**
  
The `containsKey(map[type of key],<type of value>][], <type of key>)` function returns `true` if the specified map contains a mapping for the specified key.

- **boolean containsValue(map[type of key],<type of value>] map, <type of value> value);**
  
The `containsValue(map[type of key],<type of value>][], <type of value>)` function returns `true` if the specified map maps one or more keys to the specified value.

- **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.

- **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.

- **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.

- **list[] getKeys(map[type of key], <type of value>] arg);**
  
The function returns a list of your map's `keys`. Remember the list has to be the same type as map's keys, e.g.:
map[string, integer] myMap;

// filling the map with values, e.g. myMap["first"] = 1;

string[] listOfKeys = getKeys(myMap);

• <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.
Record functions (dynamic field access)

These functions are to be found in the Functions tab, section Dynamic field access library inside the Transform Editor (p. 275).

• integer **length();**
  
  Returns the number of fields of a record the function is called on.

• integer **compare(reference record1, string field1, reference record2, string field2);**
  
  Compares two fields of given records. The fields are identified by their name. The function returns an integer value which is either:
  
  1. < 0 ... field2 is greater than field1
  2. > 0 ... field2 is lower than field1
  3. 0 ... fields are equal

• integer **compare(reference record1, integer field1, reference record2, integer field2);**
  
  Compares two fields of given records. The fields are identified by their index (0 is the first field). The function returns an integer value which is either:
  
  1. < 0 ... field2 is greater than field1
  2. > 0 ... field2 is lower than field1
  3. 0 ... fields are equal

• boolean **getBoolValue(reference record, integer field);**
  
  Returns the value of a boolean field. The field is identified by its index.

• boolean **getBoolValue(reference record, string field);**
  
  Returns the value of a boolean field. The field is identified by its name.

• byte **getByteValue(reference record, integer field);**
  
  Returns the value of a byte field. The field is identified by its index.

• byte **getByteValue(reference record, string field);**
  
  Returns the value of a byte field. The field is identified by its name.

• date **getDateValue(reference record, integer field);**
  
  Returns the value of a date field. The field is identified by its index.

• date **getDateValue(reference record, string field);**
  
  Returns the value of a date field. The field is identified by its name.

• decimal **getDecimalValue(reference record, integer field);**
  
  Returns the value of a decimal field. The field is identified by its index.
• **decimal `getDecimalValue` (reference `record`, string `field`);**
  Returns the value of a decimal field. The field is identified by its name.

• **integer `getFieldIndex` (reference `record`, string `field`);**
  Returns the index (zero-based) of a field which is identified by its name. If the field name is not found in the record, the function returns -1.

• **string `getFieldLabel` (reference `record`, integer `field`);**
  Returns the label of a field which is identified by its index. Please note a label is not a field's name, see Field Name vs. Label vs. Description (p. 156).

• **string `getFieldName` (record `argRecord`, integer `index`);**
  The `getFieldName` 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`);**
  Returns the type of a field you specify by its index (i.e. field's number starting from 0). Field type is returned as a simple string. Example code:

  ```java
  string dataType = getFieldType($in.0, 2);
  ```

  will return the data type of the third field for each incoming record (e.g. decimal).

  **Important**
  Records as arguments look like the records for the `getFieldName` function. See above.

• **integer `getIntegerValue` (reference `record`, integer `field`);**
Returns the value of an integer field. The field is identified by its index.

- `integer getValueAsInteger(referencerecord, stringfield);

Returns the value of an integer field. The field is identified by its name.

- `long getValueAsLong(referencerecord, integerfield);

Returns the value of a long field. The field is identified by its index.

- `long getValueAsLong(referencerecord, stringfield);

Returns the value of a long field. The field is identified by its name.

- `number getValueAsNumber(referencerecord, integerfield);

Returns the value of a number field. The field is identified by its index.

- `number getValueAsNumber(referencerecord, stringfield);

Returns the value of a number field. The field is identified by its name.

- `string getValueAsString(referencerecord, integerfield);

Attempts to return the value of a field (no matter its type) as a common string. The field is identified by its index.

- `string getValueAsString(referencerecord, stringfield);

Attempts to return the value of a field (no matter its type) as a common string. The field is identified by its name.

- `boolean isNull(referencerecord, stringfield);

Checks whether a given field is null. The field is identified by its name.

- `boolean isNull(referencerecord, integerfield);

Checks whether a given field is null. The field is identified by its index.

- `void setValueAsBoolean(referencerecord, integerfield, booleanvalue);

Sets a boolean value to a field. The field is identified by its index.

- `void setValueAsBoolean(referencerecord, stringfield, booleanvalue);

Sets a boolean value to a field. The field is identified by its name.

- `void setValueAsByte(referencerecord, integerfield, bytevalue);

Sets a byte value to a field. The field is identified by its index.

- `void setValueAsByte(referencerecord, stringfield, bytevalue);

Sets a byte value to a field. The field is identified by its name.
• void **setDateValue**(reference record, integer field, date value);
  
  Sets a date value to a field. The field is identified by its index.

• void **setDateValue**(reference record, string field, date value);
  
  Sets a date value to a field. The field is identified by its name.

• void **setDecimalValue**(reference record, integer field, decimal value);
  
  Sets a decimal value to a field. The field is identified by its index.

• void **setDecimalValue**(reference record, string field, decimal value);
  
  Sets a decimal value to a field. The field is identified by its name.

• void **setIntValue**(reference record, integer field, integer value);
  
  Sets an integer value to a field. The field is identified by its index.

• void **setIntValue**(reference record, string field, integer value);
  
  Sets an integer value to a field. The field is identified by its name.

• void **setLongValue**(reference record, integer field, long value);
  
  Sets a long value to a field. The field is identified by its index.

• void **setLongValue**(reference record, string field, long value);
  
  Sets a long value to a field. The field is identified by its name.

• void **setNumValue**(reference record, integer field, number value);
  
  Sets a number value to a field. The field is identified by its index.

• void **setNumValue**(reference record, string field, number value);
  
  Sets a number value to a field. The field is identified by its name.

• void **setStringValue**(reference record, integer field, string value);
  
  Sets a string value to a field. The field is identified by its index.

• void **setStringValue**(reference record, string field, string value);
  
  Sets a string value to a field. The field is identified by its name.
Chapter 64. CTL2

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. 878).
• map[string,string] getEnvironmentVariables();
Returns an unmodifiable map of system environment variables. An environment variable is a system-dependent
external named value. Similar to Java function System.getenv(). Note that the keys are case-sensitive.
Example call:
string envPath = getEnvironmentVariables()["PATH"];
• map[string,string] getJavaProperties();
The function returns the map of Java
System.getProperties(). Example call:

VM

system

properties.

Similar

to

Java

function

string operatingSystem = getJavaProperties()["os.name"];
• string getParamValue(string paramName);
Returns the value of the specified graph parameter. The argument is the name of the graph parameter without
the ${ } characters, e.g. PROJECT_DIR. The returned value is resolved, i.e. it doesn't contain any references
to other graph parameters.
The function returns null for non-existent parameters.
Example call:
string datainDir = getParamValue("DATAIN_DIR"); // will contain "./data-in"
• map[string,string] getParamValues();
Returns a map of graph parameters and their values. The keys are the names of the parameters without the ${ }
characters, e.g. PROJECT_DIR. The values are resolved, i.e. they don't contain any references to other graph
parameters. The map is unmodifiable. Example call:
string datainDir = getParamValues()["DATAIN_DIR"]; // will contain "./datain"
• <any type> iif(boolean
iffalsevalue);

con,

<any

type>

iftruevalue,

<any

type>

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.

910


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. 159) 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.

- **string resolveParams(string text);**

  Behaves as string resolveParams(string, false, true). See the related function (p. 912).

- **string resolveParams(string parameter, boolean bool1, boolean bool2);**

  The function takes a string and substitutes all graph parameters in it by their respective values. So each occurrence of pattern \${<PARAMETER_NAME>\} which is referencing an existing graph parameter is replaced by the parameter's value. This is always carried out no matter the values of both boolean arguments. The function can resolve system properties in a similar manner - e.g. PATH or JAVA_HOME.

  Moreover, you can control what else will be resolved:

  - **bool1 - resolve special characters (e.g. \n \u). Example: Let us have a metadata field called asterisk whose content is \u002A. Then transformation**

    ```
    resolveParams(asterisk, true, false);
    ```

    Produces the * character.

  - **bool2 - resolve CTL code. Note that CTL code inside the inverted commas is interpreted as CTL1 (p. 788). Example: Let us have a metadata field called code whose content is 1 plus 2 equals `to_string(1+2)`. Then transformation**

    ```
    resolveParams(code, false, true);
    ```

    Produces "1 plus 2 equals 3".

**Note**

If you are passing the parameter directly, not as a metadata field, e.g. like this:

```
string special = "\u002A"; // Unicode for asterisk - *
resolveParams(special, true, false); // this line is not needed
printErr(special);
```

it is automatically resolved. The code above will print an asterisk, even if you omit the second line. It is because resolving is triggered when processing the quotes which surround the parameter.

- **void sleep(long duration);**
The function pauses the execution for specified milliseconds.

- `string toAbsolutePath(string path);`

The function converts the specified path to an OS-dependent absolute path to the same file. The input may be a path or a URL. If the input path is relative, it is resolved against the context URL of the running graph.

If running on the Server, the function can also handle sandbox URLs (p. 290). However, a sandbox URL can only be converted to an absolute path, if the file is locally available on the current server node.

Returns `null` if the conversion fails.

**Note**

The returned path will always use forward slashes as directory separator, even on Microsoft Windows systems.

If you need the path to contain backslashes, use the `translate()` function:

```java
string absolutePath = toAbsolutePath(path).translate('/', '\');
```
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 64.9. Usage of Lookup Table Functions

```c
// 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:

```plaintext
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:

```plaintext
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/inherited 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. 895) 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 strength you define.

### Table 64.2. National Characters

<table>
<thead>
<tr>
<th>Locale</th>
<th>National Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA - Catalan</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>CZ - Czech</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>DA - Danish and Norwegian</td>
<td>““â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>DE - German</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>ES - Spanish</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>ET - Estonian</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>FI - Finnish</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>FR - French</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>HR - Croatian</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>HU - Hungarian</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>IS - Icelandic</td>
<td>“â€š”, “â€œ”, “â€§”, “â€™”, “â€œ”, “â€”</td>
</tr>
<tr>
<td>Locale</td>
<td>National Characters</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>IT - Italian</td>
<td>&quot;a=â&quot;, &quot;è=ê&quot;, &quot;é=ê&quot;, &quot;ì=î&quot;, &quot;ù=û&quot;</td>
</tr>
<tr>
<td>LV - Latvian</td>
<td>&quot;a=â&quot;, &quot;ē=ê&quot;, &quot;ı=î&quot;, &quot;ū=û&quot;, &quot;ž=ž&quot;, &quot;š=š&quot;</td>
</tr>
<tr>
<td>PL - Polish</td>
<td>&quot;a=â&quot;, &quot;ą=ą&quot;, &quot;ć=ć&quot;, &quot;ę=ę&quot;, &quot;ł=ł&quot;, &quot;ń=ń&quot;, &quot;ó=ó&quot;, &quot;ś=ś&quot;, &quot;ż=ż&quot;, &quot;ź=ź&quot;</td>
</tr>
<tr>
<td>PT - Portuguese</td>
<td>&quot;a=â&quot;, &quot;ã=ã&quot;, &quot;ê=ê&quot;, &quot;ô=ô&quot;, &quot;õ=õ&quot;, &quot;ú=ú&quot;, &quot;ü=ü&quot;, &quot;ç=ç&quot;</td>
</tr>
<tr>
<td>RO - Romanian</td>
<td>&quot;a=â&quot;, &quot;â=â&quot;, &quot;î=î&quot;, &quot;î=î&quot;</td>
</tr>
<tr>
<td>RU - Russian</td>
<td>&quot;а=а&quot;, &quot;ё=ё&quot;, &quot;ё=ё&quot;, &quot;й=й&quot;, &quot;й=й&quot;, &quot;с=с&quot;, &quot;ч=ч&quot;, &quot;р=р&quot;, &quot;т=т&quot;, &quot;у=у&quot;, &quot;ю=ю&quot;, &quot;я=я&quot;</td>
</tr>
<tr>
<td>SL - Slovenian</td>
<td>&quot;a=â&quot;, &quot;š=š&quot;, &quot;ž=ž&quot;, &quot;ž=ž&quot;</td>
</tr>
<tr>
<td>SQ - Albanian</td>
<td>&quot;a=â&quot;, &quot;ç=ç&quot;, &quot;ç=ç&quot;</td>
</tr>
<tr>
<td>SV - Swedish</td>
<td>&quot;a=â&quot;, &quot;ö=ö&quot;, &quot;ö=ö&quot;</td>
</tr>
</tbody>
</table>
Chapter 65. Regular Expressions

A *regular expression* is a formalism used to specify a set of strings with a single expression. Since the implementation of regular expressions comes from the Java standard library, the syntax of expressions is the same as in Java.

**Example 65.1. Regular Expressions Examples**

\[p-s\]{5}

- means the string has to be exactly five characters long and it can only contain the \( p, q, r, \) and \( s \) characters

\[^a-d].*\]

- this example expression matches any string which starts with a character other than \( a, b, c, d \) because
  - the ^ sign means exception
  - a-d means characters from a to d
  - these characters can be followed by zero or more (*) other characters
  - the dot stands for an arbitrary character

For more detailed explanation of how to use regular expressions see the Java documentation for `java.util.regex.Pattern`.

The meaning of regular expressions can be modified using embedded flag expressions. The expressions include the following:

- \(?i\) – Pattern.CASE_INSENSITIVE
  - Enables case-insensitive matching.
- \(?s\) – Pattern.DOTALL
  - In dotall mode, the dot . matches any character, including line terminators.
- \(?m\) – Pattern.MULTILINE
  - In multiline mode you can use ^ and $ to mean the beginning and end of the line, respectively (that includes at the beginning and end of the entire expression).

Further reading and description of other flags can be found at http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html.
List of Figures

1.1. Family of CloverETL Products .............................................. 2
1.2. 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
5.4. CloverETL Help ..................................................................... 16
6.1. Available Software ............................................................... 17
7.1. License Manager showing installed licenses. .............................. 19
7.2. CloverETL License dialog ..................................................... 20
7.3. CloverETL License wizard ..................................................... 21
7.4. Select Activate using license key radio button and click Next. .... 22
7.5. Enter the path to the license file or copy and paste the license text. 22
7.6. Confirm you accept the license agreement and click Finish button. 23
7.7. Select Activate online radio button, enter your license number and password and click Next. 24
7.8. Confirm you accept the license agreement and click Finish button. 24
8.1. Giving a Name to a CloverETL Project .................................... 26
8.2. CloverETL Server Project Wizard .......................................... 27
8.3. Giving a Name to the New CloverETL Server Project .............. 27
8.4. CloverETL Examples Project Wizard ....................................... 29
8.5. Renaming CloverETL Examples Projects .................................. 29
9.1. Project Folder Structure inside Navigator Pane ......................... 31
9.2. Opening the Workspace.prm File .......................................... 32
9.3. Workspace.prm File ............................................................. 32
9.4. Basic Eclipse Perspective ..................................................... 33
9.5. Selecting CloverETL Perspective ............................................ 33
9.6. CloverETL Perspective .......................................................... 34
10.1. CloverETL Perspective ....................................................... 35
10.2. Graph Editor with an Opened Palette of Components .............. 36
10.3. Closing the Graphs ............................................................. 37
10.4. Rulers in the Graph Editor .................................................. 37
10.5. Grid in the Graph Editor ..................................................... 38
10.6. A Graph before Selecting Auto-Layout .................................. 38
10.7. A Graph after Selecting Auto-Layout .................................... 39
10.8. Six New Buttons in the Tool Bar Appear Highlighted (Align Middle is shown) ......................................................... 39
10.9. Alignments from the Context Menu ...................................... 40
10.10. Navigator Pane ............................................................... 40
10.11. Outline Pane ................................................................. 41
10.12. Another Representation of the Outline Pane ......................... 41
10.13. Accessing a locked graph element - you can add any text you like to describe the lock. 42
10.14. Properties Tab ............................................................... 43
10.15. Console Tab .................................................................. 43
10.16. Problems Tab .................................................................. 44
10.17. Clover - RegEx Tester Tab .................................................. 44
10.18. Clover - Graph Tracking Tab .............................................. 44
10.19. Clover - Log Tab ............................................................... 45
11.1. Creating a New Graph ......................................................... 47
11.2. Giving a Name to a New CloverETL Graph ......................... 47
11.3. Selecting the Parent Folder for the Graph .............................. 48
11.4. CloverETL Perspective with Highlighted Graph Editor .......... 48
11.5. Graph Editor with a New Graph and the Palette of Components 49
11.6. Components Selected from the Palette ................................. 50
11.7. Components are Connected by Edges .................................. 51
<table>
<thead>
<tr>
<th>11.8. Creating an Input File</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.9. Creating the Contents of the Input File</td>
<td>52</td>
</tr>
<tr>
<td>11.10. Metadata Editor with Default Names of the Fields</td>
<td>52</td>
</tr>
<tr>
<td>11.11. Metadata Editor with New Names of the Fields</td>
<td>53</td>
</tr>
<tr>
<td>11.12. Edge Has Been Assigned Metadata</td>
<td>53</td>
</tr>
<tr>
<td>11.13. Metadata Have Been Propagated through the Component</td>
<td>54</td>
</tr>
<tr>
<td>11.14. Opening the Attribute Row</td>
<td>54</td>
</tr>
<tr>
<td>11.15. Selecting the Input File</td>
<td>55</td>
</tr>
<tr>
<td>11.16. Input File URL Attribute Has Been Set</td>
<td>55</td>
</tr>
<tr>
<td>11.17. Output File URL without a File</td>
<td>56</td>
</tr>
<tr>
<td>11.18. Output File URL with a File</td>
<td>56</td>
</tr>
<tr>
<td>11.19. Defining a Sort Key</td>
<td>57</td>
</tr>
<tr>
<td>11.20. Sort Key Has Been Defined</td>
<td>57</td>
</tr>
<tr>
<td>11.21. Running the Graph</td>
<td>58</td>
</tr>
<tr>
<td>11.22. Result of Successful Run of the Graph</td>
<td>58</td>
</tr>
<tr>
<td>11.23. Contents of the Output File</td>
<td>59</td>
</tr>
<tr>
<td>12.1. Running a Graph from the Main Menu</td>
<td>60</td>
</tr>
<tr>
<td>12.2. Running a Graph from the Context Menu</td>
<td>61</td>
</tr>
<tr>
<td>12.3. Running a Graph from the Upper Tool Bar</td>
<td>61</td>
</tr>
<tr>
<td>12.4. Successful Graph Execution</td>
<td>62</td>
</tr>
<tr>
<td>12.5. Console Tab with an Overview of the Graph Execution</td>
<td>62</td>
</tr>
<tr>
<td>12.6. Counting Parsed Data</td>
<td>63</td>
</tr>
<tr>
<td>12.7. Run Configurations Dialog</td>
<td>63</td>
</tr>
<tr>
<td>13.1. Selecting Cheat Sheets</td>
<td>65</td>
</tr>
<tr>
<td>13.2. The Cheat Sheet Selection Wizard</td>
<td>65</td>
</tr>
<tr>
<td>13.3. CloverETL and Standard Eclipse Commands (Collapsed)</td>
<td>66</td>
</tr>
<tr>
<td>13.4. CloverETL and Standard Eclipse Commands (Expanded)</td>
<td>66</td>
</tr>
<tr>
<td>13.5. CloverETL Designer Reference Cheat Sheet</td>
<td>67</td>
</tr>
<tr>
<td>13.6. Locating a Custom Cheat Sheet</td>
<td>67</td>
</tr>
<tr>
<td>14.1. URL File Dialog</td>
<td>68</td>
</tr>
<tr>
<td>14.2. Edit Value Dialog</td>
<td>69</td>
</tr>
<tr>
<td>14.3. Find Wizard</td>
<td>69</td>
</tr>
<tr>
<td>14.4. Go to Line Wizard</td>
<td>70</td>
</tr>
<tr>
<td>14.5. Open Type Dialog</td>
<td>70</td>
</tr>
<tr>
<td>15.1. Import (Main Menu)</td>
<td>71</td>
</tr>
<tr>
<td>15.2. Import (Context Menu)</td>
<td>71</td>
</tr>
<tr>
<td>15.3. Import Options</td>
<td>72</td>
</tr>
<tr>
<td>15.4. Import Projects</td>
<td>72</td>
</tr>
<tr>
<td>15.5. Import from CloverETL Server Sandbox Wizard (Connect to CloverETL Server)</td>
<td>73</td>
</tr>
<tr>
<td>15.6. Import from CloverETL Server Sandbox Wizard (List of Files)</td>
<td>73</td>
</tr>
<tr>
<td>15.7. Import Graphs</td>
<td>74</td>
</tr>
<tr>
<td>15.8. Import Metadata from XSD</td>
<td>75</td>
</tr>
<tr>
<td>15.9. Import Metadata from DDL</td>
<td>76</td>
</tr>
<tr>
<td>16.1. Export Options</td>
<td>77</td>
</tr>
<tr>
<td>16.2. Export Graphs</td>
<td>77</td>
</tr>
<tr>
<td>16.3. Export Graphs to HTML</td>
<td>78</td>
</tr>
<tr>
<td>16.4. Export metadata to XSD</td>
<td>79</td>
</tr>
<tr>
<td>16.5. Export to CloverETL Server Sandbox</td>
<td>80</td>
</tr>
<tr>
<td>16.6. Export Image</td>
<td>81</td>
</tr>
<tr>
<td>17.1. Setting Up Memory Size</td>
<td>84</td>
</tr>
<tr>
<td>17.2. Custom Clover Settings</td>
<td>85</td>
</tr>
<tr>
<td>17.3. Enlarging the Font of Numbers</td>
<td>88</td>
</tr>
<tr>
<td>17.4. Setting the Font Size</td>
<td>88</td>
</tr>
<tr>
<td>17.5. Setting The Java Runtime Environment</td>
<td>89</td>
</tr>
<tr>
<td>17.6. Preferences Wizard</td>
<td>90</td>
</tr>
<tr>
<td>17.7. Installed JREs Wizard</td>
<td>90</td>
</tr>
<tr>
<td>17.8. Adding a Java Development Kit</td>
<td>91</td>
</tr>
<tr>
<td>17.9. Searching for JDK Jars</td>
<td>92</td>
</tr>
</tbody>
</table>
17.10. Adding JDK Jars ................................................................. 92
19.1. Selecting the Edge Type .................................................... 97
19.2. Creating Metadata on an empty Edge .................................... 98
19.3. Assigning Metadata to an Edge ........................................... 99
19.4. Metadata in the Tooltip .................................................. 100
19.5. Properties of an Edge ...................................................... 101
19.6. Filter Editor Wizard .......................................................... 101
19.7. Debug Properties Wizard .................................................. 103
19.8. View Data Dialog ............................................................ 103
19.9. Viewing Data on Debugged Edge .......................................... 103
19.10. Hide/Show Columns when Viewing Data .............................. 104
19.11. View Record Dialog ....................................................... 104
19.12. Find Dialog ................................................................. 105
19.13. Copy Dialog ................................................................. 105
20.1. Creating Internal Metadata in the Outline Pane ...................... 130
20.2. Creating Internal Metadata in the Graph Editor ..................... 131
20.3. Externalizing and/or Exporting Internal Metadata .................. 132
20.4. Selecting a Location for a New Externalized and/or Exported Internal Metadata .......................................................... 132
20.5. Creating External (Shared) Metadata in the Main Menu and/or in the Navigator Pane .................................................. 133
20.6. Internalizing External (Shared) Metadata .............................. 134
20.7. Extracting Metadata from Delimited Flat File .......................... 135
20.8. Extracting Metadata from Fixed Length Flat File .................... 136
20.9. Setting Up Delimited Metadata .......................................... 137
20.10. Setting Up Fixed Length Metadata .................................... 139
20.11. Extract Metadata from Excel Spreadsheet Wizard .................. 140
20.12. Format Extracted from Spreadsheet Cell .............................. 141
20.13. Extracting Internal Metadata from a Database ...................... 142
20.14. Database Connection Wizard ............................................. 143
20.15. Selecting Columns for Metadata ........................................ 143
20.16. Generating a Query ........................................................ 144
20.17. DBF Metadata Editor ....................................................... 146
20.18. Specifying Lotus Notes connection for metadata extraction ........ 147
20.19. Lotus Notes metadata extraction wizard, page 2 .................... 148
20.20. Merging two metadata - conflicts can be resolved in one of the three ways (notice radio buttons at the bottom). .......... 149
20.21. Creating Database Table from Metadata and Database Connection 151
20.22. Metadata Editor for a Delimited File ................................ 154
20.23. Metadata Editor for a Fixed Length File .............................. 155
20.24. Trackable Fields Selection in Metadata Editor ....................... 155
21.1. Creating Internal Database Connection .................................. 168
21.2. Externalizing Internal Database Connection .......................... 169
21.3. Internalizing External (Shared) Database Connection .............. 171
21.4. Database Connection Wizard ............................................ 172
21.5. Adding a New JDBC Driver into the List of Available Drivers .... 172
21.6. Running a Graph with the Password Encrypted ...................... 176
21.7. Connecting to MS SQL with Windows authentication. Setting-up a database connection like this is not sufficient. Additional steps explained below this figure need to be performed. 177
21.8. Adding path to the native dll to VM arguments ....................... 178
22.1. Edit JMS Connection Wizard ........................................... 182
23.1. QuickBase Connection Dialog ........................................... 184
24.1. Lotus Notes Connection Dialog ......................................... 185
25.1. Creating Internal Lookup Table ........................................ 187
25.2. Externalizing Wizard ..................................................... 188
25.3. Selecting Lookup Table Item ............................................ 190
25.4. Lookup Table Internalization Wizard .................................. 191
25.5. Lookup Table Wizard ..................................................... 192
25.6. Simple Lookup Table Wizard ............................................ 192
25.7. Edit Key Wizard ........................................................... 193
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.8.</td>
<td>Simple Lookup Table Wizard with File URL</td>
<td>193</td>
</tr>
<tr>
<td>25.9.</td>
<td>Simple Lookup Table Wizard with Data</td>
<td>194</td>
</tr>
<tr>
<td>25.10.</td>
<td>Changing Data</td>
<td>194</td>
</tr>
<tr>
<td>25.11.</td>
<td>Database Lookup Table Wizard</td>
<td>195</td>
</tr>
<tr>
<td>25.12.</td>
<td>Appropriate Data for Range Lookup Table</td>
<td>196</td>
</tr>
<tr>
<td>25.13.</td>
<td>Range Lookup Table Wizard</td>
<td>196</td>
</tr>
<tr>
<td>25.14.</td>
<td>Persistent Lookup Table Wizard</td>
<td>198</td>
</tr>
<tr>
<td>25.15.</td>
<td>Aspell Lookup Table Wizard</td>
<td>200</td>
</tr>
<tr>
<td>26.1.</td>
<td>Creating a Sequence</td>
<td>202</td>
</tr>
<tr>
<td>26.2.</td>
<td>Editing a Sequence</td>
<td>205</td>
</tr>
<tr>
<td>26.3.</td>
<td>A New Run of the Graph with the Previous Start Value of the Sequence</td>
<td>205</td>
</tr>
<tr>
<td>27.1.</td>
<td>Creating Internal Parameters</td>
<td>208</td>
</tr>
<tr>
<td>27.2.</td>
<td>Externalizing Internal Parameters</td>
<td>209</td>
</tr>
<tr>
<td>27.3.</td>
<td>Internalizing External (Shared) Parameter</td>
<td>211</td>
</tr>
<tr>
<td>27.4.</td>
<td>Example of a Parameter-Value Pairs</td>
<td>212</td>
</tr>
<tr>
<td>29.1.</td>
<td>Dictionary Dialog with Defined Entries</td>
<td>219</td>
</tr>
<tr>
<td>30.1.</td>
<td>Pasting a Note to the Graph Editor Pane</td>
<td>222</td>
</tr>
<tr>
<td>30.2.</td>
<td>Enlarging the Note</td>
<td>222</td>
</tr>
<tr>
<td>30.3.</td>
<td>Highlighted Margins of the Note Have Disappeared</td>
<td>223</td>
</tr>
<tr>
<td>30.4.</td>
<td>Changing the Note Label</td>
<td>223</td>
</tr>
<tr>
<td>30.5.</td>
<td>Writing a New Description in the Note</td>
<td>224</td>
</tr>
<tr>
<td>30.6.</td>
<td>A New Note with a New Description</td>
<td>224</td>
</tr>
<tr>
<td>30.7.</td>
<td>Folding the Note</td>
<td>225</td>
</tr>
<tr>
<td>30.8.</td>
<td>Properties of a Note</td>
<td>225</td>
</tr>
<tr>
<td>31.1.</td>
<td>CloverETL Search Tab</td>
<td>226</td>
</tr>
<tr>
<td>31.2.</td>
<td>Search Results</td>
<td>227</td>
</tr>
<tr>
<td>33.1.</td>
<td>Valid selections</td>
<td>230</td>
</tr>
<tr>
<td>33.2.</td>
<td>How to select the FTL wizard</td>
<td>230</td>
</tr>
<tr>
<td>33.3.</td>
<td>Select a wizard (new wizard selection window)</td>
<td>231</td>
</tr>
<tr>
<td>33.4.</td>
<td>Deselection</td>
<td>231</td>
</tr>
<tr>
<td>33.5.</td>
<td>New Graph Name Page</td>
<td>232</td>
</tr>
<tr>
<td>33.6.</td>
<td>Output Page</td>
<td>232</td>
</tr>
<tr>
<td>33.7.</td>
<td>File Selection Page</td>
<td>233</td>
</tr>
<tr>
<td>33.8.</td>
<td>URL Dialog</td>
<td>233</td>
</tr>
<tr>
<td>33.9.</td>
<td>Database Connection Page</td>
<td>234</td>
</tr>
<tr>
<td>33.10.</td>
<td>Fact Table Selection Page</td>
<td>234</td>
</tr>
<tr>
<td>33.11.</td>
<td>Dimension Table Selection Page</td>
<td>235</td>
</tr>
<tr>
<td>33.12.</td>
<td>Order Table Page</td>
<td>235</td>
</tr>
<tr>
<td>33.13.</td>
<td>Mapping Page</td>
<td>236</td>
</tr>
<tr>
<td>33.14.</td>
<td>Fact Mapping Page</td>
<td>236</td>
</tr>
<tr>
<td>33.15.</td>
<td>Summary Page</td>
<td>237</td>
</tr>
<tr>
<td>33.16.</td>
<td>Created Graph</td>
<td>237</td>
</tr>
<tr>
<td>33.17.</td>
<td>Graph Parameters</td>
<td>238</td>
</tr>
<tr>
<td>37.1.</td>
<td>Selecting Components</td>
<td>252</td>
</tr>
<tr>
<td>37.2.</td>
<td>Components in Palette</td>
<td>252</td>
</tr>
<tr>
<td>37.3.</td>
<td>Removing Components from the Palette</td>
<td>253</td>
</tr>
<tr>
<td>38.1.</td>
<td>Find Components dialog - the searched text is highlighted both in component names and description.</td>
<td>254</td>
</tr>
<tr>
<td>38.2.</td>
<td>Add Components dialog - finding a sorter</td>
<td>255</td>
</tr>
<tr>
<td>39.1.</td>
<td>Edit Component Dialog (Properties Tab)</td>
<td>256</td>
</tr>
<tr>
<td>39.2.</td>
<td>Edit Component Dialog (Ports Tab)</td>
<td>256</td>
</tr>
<tr>
<td>39.3.</td>
<td>Simple Renaming Components</td>
<td>260</td>
</tr>
<tr>
<td>39.4.</td>
<td>Running a Graph with Various Phases</td>
<td>261</td>
</tr>
<tr>
<td>39.5.</td>
<td>Setting the Phases for More Components</td>
<td>261</td>
</tr>
<tr>
<td>39.6.</td>
<td>Running a Graph with Disabled Component</td>
<td>262</td>
</tr>
<tr>
<td>39.7.</td>
<td>Running a Graph with Component in PassThrough Mode</td>
<td>263</td>
</tr>
<tr>
<td>40.1.</td>
<td>Creating Metadata from a Template</td>
<td>264</td>
</tr>
<tr>
<td>40.2.</td>
<td>Defining Group Key</td>
<td>265</td>
</tr>
<tr>
<td>40.3.</td>
<td>Defining Sort Key and Sort Order</td>
<td>266</td>
</tr>
</tbody>
</table>
40.4. Define Error Actions Dialog ................................................................. 274
40.5. Transformations Tab of the Transform Editor ....................................... 275
40.6. Copying the Input Field to the Output ............................................... 276
40.7. Transformation Definition in CTL (Transformations Tab) ....................... 277
40.8. Mapping of Inputs to Outputs (Connecting Lines) ................................. 277
40.9. Editor with Fields and Functions ....................................................... 278
40.10. Input Record Mapped to Output Record Using Wildcards ....................... 278
40.11. Transformation Definition in CTL (Source Tab) .................................. 279
40.12. Java Transform Wizard Dialog ......................................................... 279
40.13. Confirmation Message .................................................................. 280
40.14. Transformation Definition in CTL (Transform Tab of the Graph Editor) 280
40.15. Outline Pane Displaying Variables and Functions ............................... 281
40.16. Content Assist (Record and Field Names) ........................................... 281
40.17. Content Assist (List of CTL Functions) .............................................. 282
40.18. Error in Transformation ................................................................ 282
40.19. Converting Transformation to Java .................................................... 282
40.20. Transformation Definition in Java ...................................................... 283
41.1. Viewing Data in Components ............................................................ 290
41.2. Viewing Data as Plain Text ............................................................... 291
41.3. Viewing Data as Grid .................................................................. 291
41.4. Plain Text Data Viewing .................................................................. 291
41.5. Grid Data Viewing ........................................................................ 292
41.6. XML Features Dialog .................................................................... 296
42.1. Viewing Data on Components ............................................................ 303
42.2. Viewing Data as Plain Text ............................................................... 303
42.3. Viewing Data as Grid .................................................................... 304
42.4. Plain Text Data Viewing .................................................................. 304
42.5. Grid Data Viewing ........................................................................ 304
44.1. Source Tab of the Transform Editor in Joiners ................................... 315
51.1. Configuring prefix selector in ComplexDataReader. Rules are defined in the Selector properties pane. Notice the two extra attributes for regular expressions. .................................................. 338
51.2. Sequences Dialog ........................................................................ 342
51.3. A Sequence Assigned .................................................................... 342
51.4. Edit Key Dialog .............................................................................. 343
51.5. Source Tab of the Transform Editor in DataGenerator ....................... 344
51.6. Generated Query with Question Marks ............................................ 351
51.7. Generated Query with Output Fields ............................................... 352
51.8. Mapping to Clover fields in EmailReader ........................................... 355
51.9. Example mapping of nested arrays - the result .................................. 370
51.10. SpreadsheetDataReader Mapping Editor ......................................... 391
51.11. Basic Mapping – notice leading cells and dashed borders marking the area data will be taken from ................................................................. 392
51.12. The difference between global data offsets set to 1 (default) and 3. In the right hand figure, reading would start at row 4 (ignoring data in rows 2 and 3). .................................................. 393
51.13. Global data offset is set to 1 to all columns. In the third column, it is locally changed to 3. ................................. 393
51.14. Rows per record is set to 4. This makes SpreadsheetDataReader take 4 Excel rows and create one record out of their cells. Cells actually becoming fields of a record are marked by a dashed border, therefore the record is not populated by all data. Which cells populate a record is also determined by the data offsets setting, see the following bullet point. ........................................... 393
51.15. Rows per record is set to 3. The first and third columns 'contribute' to the record by their first row (because of the global data offset being 1). The second and fourth columns have (local) data offsets 2 and 4, respectively. The first record will, thus, be formed by 'zig-zagged' cells (the yellow ones – follow them to make sure you understand this concept clearly). ........................................... 394
51.16. Retrieving format from a date field. Format Field was set to the "Special" field as target. ........................................... 394
51.17. Reading mixed data using two leading cells per column. Rows per record is 2, Data offset needed to be raised to 2 – looking at the first leading cell which has to start reading on the third row. ........................................... 395
51.18. XLS Mapping Dialog .................................................................... 404
51.19. XLS Fields Mapped to Clover Fields ................................................. 405
51.20. The Mapping Dialog for XMLEXtract ................................................. 415
52.22. Writing into a template. Its original content will not be affected, your data will be written under
understand this concept clearly). ................................................................. 470
52.13. Mapping editor in JSONWriter after first open. Metadata on the input edge(s) are displayed on
the left hand side. The right hand pane is where you design the desired output tree - it is pre-defined by
your bean's structure (note: in the example, the bean contains employees and projects they are working
on). Mapping is then performed by dragging metadata from left to right (and performing additional tasks
described below). ................................................................. 476
52.14. Example mapping in JSONWriter - employees are joined with projects they work on. Fields in
bold (their content) will be printed to the output file - see below. ................................................................. 483
52.23. Parent Elements .................................................................................................................. 416
52.24. Mapping summary ............................................................................................................. 419
52.25. Create Mask Dialog ........................................................................................................... 421
52.26. Mapping Editor ................................................................................................................... 422
52.27. Adding Child to Root Element. ............................................................................................ 426
52.28. Wildcard attribute and its properties. .................................................................................... 431
52.29. Attribute and its properties. ................................................................................................. 432
52.30. Element and its properties. .................................................................................................. 433
52.31. Mapping editor toolbar. ...................................................................................................... 434
52.32. Binding of Port and Element. ................................................................................................. 435
52.33. Generating XML from XSD root element. .......................................................................... 436
52.34. Source tab in Mapping editor. .............................................................................................. 437
52.8. Mapping editor in JavaBeanWriter after first open. Metadata on the input edge(s) are displayed on
the left hand side. The right hand pane is where you design the desired output tree. Mapping is then
performed by dragging metadata from left to right (and performing additional tasks described below). 
52.9. Example mapping in JavaBeanWriter - employees are joined with projects they work on. Fields in
bold (their content) will be printed to the output dictionary. ................................................................. 475
52.10. Mapping editor in JavaMapWriter after first open. Metadata on the input edge(s) are displayed on
the left hand side. The right hand pane is where you design the desired output tree. Mapping is then
performed by dragging metadata from left to right (and performing additional tasks described below). ........... 474
52.11. Example mapping in JavaMapWriter - employees are joined with projects they work on. Fields in
bold (their content) will be printed to the output dictionary. ................................................................. 475
52.12. Mapping arrays in JavaMapWriter - notice the array contains a dummy element 'State' which you
bind the input field to. .................................................................................................................. 476
52.21. Parent Elements ...................................................................................................................... 416
52.22. Editing Namespace Bindings in XMLExtract ....................................................................... 420
52.23. Selecting subtype in XMLExtract .......................................................................................... 421
52.24. Generated Query with Question Marks ............................................................................ 457
52.25. Generated Query with Input Fields .................................................................................... 457
52.26. Generated Query with Returned Fields ............................................................................. 458
52.27. Adding Child to Root Element. ............................................................................................ 462
52.28. Wildcard attribute and its properties. .................................................................................... 463
52.29. Attribute and its properties. .................................................................................................. 464
52.30. Element and its properties. .................................................................................................. 465
52.31. Mapping editor toolbar. ...................................................................................................... 466
52.32. Binding of Port and Element. ................................................................................................. 467
52.33. Generating XML from XSD root element. .......................................................................... 468
52.34. Source tab in Mapping editor. .............................................................................................. 469
List of Tables

6.1. Sites with CloverETL ................................................................. 17
9.1. Standard Folders and Parameters ............................................. 31
19.1. Memory Demands per Edge Type ............................................. 106
20.1. Data Types in Metadata ......................................................... 108
20.2. Available date engines ......................................................... 110
20.3. Date Format Pattern Syntax (Java) .......................................... 111
20.4. Rules for Date Format Usage (Java) ......................................... 112
20.5. Date and Time Format Patterns and Results (Java) ....................... 113
20.6. Date Format Pattern Syntax (Joda) .......................................... 114
20.7. Rules for Date Format Usage (Joda) ......................................... 114
20.8. Numeric Format Pattern Syntax .............................................. 117
20.9. BNF Diagram ........................................................................ 118
20.10. Used Notation ........................................................................ 118
20.11. Locale-Sensitive Formatting .................................................. 118
20.12. Numeric Format Patterns and Results ..................................... 120
20.13. Available Binary Formats ..................................................... 123
20.14. List of all Locale .................................................................. 123
20.15. CloverETL-to-SQL Data Types Transformation Table (Part I) .... 152
20.16. CloverETL-to-SQL Data Types Transformation Table (Part II) ... 152
40.1. Transformations Overview ..................................................... 271
41.1. Readers Comparison ............................................................... 286
42.1. Writers Comparison ............................................................... 299
43.1. Transformers Comparison ..................................................... 309
44.1. Joiners Comparison ............................................................... 312
44.2. Functions in Joiners, DataIntersection, and Reformat ................. 315
45.1. Cluster Components Comparison ........................................... 319
46.1. Others Comparison ............................................................... 320
47.1. Data Quality Comparison ....................................................... 321
48.1. Job control Comparison .......................................................... 322
49.1. File Operations Comparison .................................................. 323
51.1. Functions in DataGenerator ................................................... 344
51.2. Error Metadata for Parallel Reader .......................................... 381
51.3. Error Metadata for QuickBaseRecordReader ......................... 384
51.4. Error Port Metadata - first ten fields have mandatory types, names can be arbitrary ........................................ 388
51.5. Format strings ...................................................................... 395
51.6. Error Metadata for UniversalDataReader ................................. 398
52.1. Error Metadata for DB2DataWriter .......................................... 443
52.2. Error Fields for InformixDataWriter ....................................... 466
52.3. Error Fields for MSSQLDataWriter ........................................ 490
52.4. Error Metadata for MySQLDataWriter ................................... 493
52.5. Error Fields for QuickBaseImportCSV ................................... 503
52.6. Error Fields for QuickBaseRecordWriter ................................ 505
53.1. Functions in Denormalizer .................................................... 566
53.2. Functions in Normalizer ....................................................... 588
53.3. Functions in Partition (or ClusterPartitioner) ......................... 595
53.4. Functions in Rollup ............................................................... 609
58.1. Error Fields for EmailFilter .................................................... 729
59.1. Input Metadata for RunGraph (In-Out Mode) ............................ 757
59.2. Output Metadata for RunGraph .............................................. 757
61.1. CTL Version Comparison ..................................................... 774
61.2. CTL Version Differences ...................................................... 775
63.1. Literals .............................................................................. 793
64.1. Literals ............................................................................. 855
64.2. National Characters ............................................................. 919
List of Examples

20.1. String Format .................................................................................................................. 122
20.2. Examples of Locale ........................................................................................................ 123
20.3. Example situations when you could take advantage of multivalue fields .................. 163
20.4. Integer lists which are (not) equal - symbolic notation .................................................. 166
27.1. Canonizing File Paths ..................................................................................................... 214
33.1. Example of usage ............................................................................................................ 229
34.1. Example jobflow log - token starting a subgraph ............................................................ 245
38.1. Finding a sort component ................................................................................................ 255
40.1. Sorting ............................................................................................................................ 266
40.2. Example of the Error Actions Attribute ......................................................................... 274
51.1. Example State Function ................................................................................................ 337
51.2. ........................................................................................................................................ 339
51.3. Example Mapping in JavaBeanReader .......................................................................... 358
51.4. Reading lists with JavaBeanReader .............................................................................. 361
51.5. Field Mapping in XLSDataReader .................................................................................. 405
51.6. Mapping in XMLExtract .............................................................................................. 409
51.7. From XML Structure to Mapping Structure ................................................................. 411
51.8. Mapping in XMLReader ............................................................................................... 426
51.9. Reading lists with XMLReader ..................................................................................... 430
51.10. Mapping in XMLXPathReader ................................................................................... 433
51.11. Reading lists with XMLXPathReader .......................................................................... 437
52.1. Internal Structure of Archived Output File(s) ............................................................... 441
52.2. Examples of Queries ...................................................................................................... 453
52.3. Creating Binding ............................................................................................................. 471
52.4. Creating Binding ............................................................................................................. 474
52.5. Writing arrays ................................................................................................................ 475
52.6. Creating Binding ............................................................................................................. 482
52.7. Writing arrays ................................................................................................................ 484
52.8. Example of a Control script .......................................................................................... 497
52.9. Writing Excel format ..................................................................................................... 515
52.10. Using Expressions in Ports and Fields ....................................................................... 536
52.11. Include and Exclude property examples ..................................................................... 537
52.12. Attribute value examples ............................................................................................. 537
52.13. Writing null attribute .................................................................................................... 539
52.14. Omitting Null Attribute ............................................................................................... 539
52.15. Hide Element ................................................................................................................. 540
52.16. Partitioning According to Any Element ....................................................................... 540
52.17. Writing and omitting blank elements .......................................................................... 541
52.18. Binding that serves as JOIN ........................................................................................ 545
52.19. Insert Wildcard attributes in Source tab .................................................................... 548
53.1. Aggregation Mapping ................................................................................................... 554
53.2. Join Key for DataIntersection ..................................................................................... 558
53.3. Key for Denormalizer .................................................................................................... 564
53.4. Example MetaPivot Transformation .............................................................................. 584
53.5. Data Transformation with Pivot - Using Key .............................................................. 601
54.1. Matching Key ................................................................................................................. 628
54.2. Join Key for ApproximativeJoin .................................................................................. 631
54.3. Join Key for DBJoin ....................................................................................................... 637
54.4. Slave Part of Join Key for ExtHashJoin ........................................................................ 641
54.5. Join Key for ExtHashJoin ............................................................................................. 642
54.6. Join Key for ExtMergeJoin ........................................................................................... 647
54.7. Join Key for LookupJoin .............................................................................................. 651
54.8. Join Key for RelationalJoin .......................................................................................... 655
59.1. Working with Quoted Command Line Arguments ........................................................ 759
59.2. Use nested nodes example ............................................................................................ 769