Oracle Retail® Extract Transform and Load™ 11.3

Programmer's Guide

ORACLE

Corporate Headquarters:

Oracle
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

European Headquarters:

Oracle
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

The Apache Software License, Version 1.1

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Oracle customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
E-mail	support@retek.com
Internet (ROCS)	rocs.retek.com Oracle Retail's secure client Web site to update and view issues
Phone	+1 612 587 5800

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
Hong Kong	800 96 4262
Korea	00 308 13 1342
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835
Mail	Oracle Customer Support 950 Nicollet Mall Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step-by-step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

Chapter 1 – Introduction

RETL is a high performance, scalable, platform independent, and parallel processing data movement tool. RETL attempts to address several primary needs:

- Database independent applications.
- Platform independent applications.
- Developing such applications more quickly than possible with conventional coding methods (e.g. custom crafted C and/or C++ code).
- High performance data processing.

In order to provide for these needs, the RETL defines an interface (in XML) that applications can call to define ETL functions. This interface is in a well-defined XML form that allows access to any database that the RETL supports.

RETL is a cross-platform development tool supported on many different platforms (e.g. AIX, SunOS, and HP-UX). The XML definitions do not change on a per platform basis so moving between hardware platforms is a snap if you are using RETL.

Development of the XML instructions for RETL is much simpler, faster and less error prone than writing C and/or C++ code. The result is that applications can be completed much faster than possible with previous methods.

This guide shows you, the application developer, how to rapidly deploy RETL in your application and use it to manage your parallel processing system. Specifically, the guide provides information to:

- Install and set up RETL
- Configure RETL
- Administer RETL
- Develop applications using RETL

Select the RETL operators you need to perform your application's data processing tasks. Manage your system resources to scale your application by partitioning data across multiple processing nodes.

RETL resides as an executable on your application's platform, and on any other system that serves as a source or target of data.

The 11.x releases of RETL introduce a new high performance architecture written in Java.

Enhancements of 11.x versions of RETL

True Platform Independence – The new architecture of the 11.x releases has been written entirely in Java. As a result, RETL may now run on platforms that previously would have required significant efforts to port to. See the FAQ question related to platform independence and the Compatibility Matrix for supported platforms. As a direct result of the platform independence, significant RETL resources in platform maintenance will be freed up to develop new features going forward.

Improved performance – RETL 11.x releases include a new architecture that optimizes pipeline parallelism by consolidating connected operators into the same pipeline where possible. This reduces the number of threads required, minimizing context switching and thrashing among threads, and thereby allowing a framework that supports improved performance. Optimizations have been made in certain operators, namely ORAREAD/ORAWRITE and IMPORT/EXPORT, which have been shown to positively affect performance in comparison to 10.x releases. Internal performance testing between 10.x and 11.x releases has shown 11.x releases to be anywhere from 15%-150% faster. With RETL 11.2, significant performance enhancements have been made in internal data structure representation and algorithms, which give an even larger boost to flows.

Improved partitioning – RETL 11.2 introduces an enhanced partitioning engine that supports true parallelism within the framework and RETL operators. Parallel file and database reads/writes directly tie into data partitions to produce a truly parallel system less inhibited by bottlenecks in a flow.

Error handling and debugging – 11.x releases introduce better flow debugging by giving line and column specific error messages when RETL encounters problems reading an XML flow definition. Additionally, 11.x's exception handling allows for better error handling and debugging.

Online Help – A command-line option has been added to 11.x releases that allow flow developers to view operator syntax and usage via the command-line without having to refer to this document.

Simplified installation and configuration – 11.x releases consolidate code into a single binary rather than the 26 binaries of 10.x release. This considerably eases installation. In addition, there is less environment setup that needs to be done in order to complete an installation.

Backwards compatibility with previous releases – A requirement of 11.x versions is that they be backwardly-compatible with the 10.x versions. 11.x versions are stricter on enforcement of valid XML flow interfaces, input schemas, and data fields. In some instances, the 11.x product will find data and/or flow errors that may have previously been unreported. The result may be more discarded or rejected records and/or error messages than previously identified.

Technical specifications

RETL is certified on the platforms listed in this section. With the new architecture of 11.x releases of RETL, only one binary/runtime environment is necessary for all the supported platforms. For historical reasons the executable for RETL is called "rfx" and when referencing the executable we will use that name.

The release_notes.txt that documents the current configurations is included with the RETL install package. If you have a configuration that is not included in the release_notes.txt, verify with Retek Customer Support to see if your configuration is now available.

Supported operating systems

- Sun Solaris (versions 8, 9)
- IBM AIX (versions 5.1, 5.2 (RETL 11.2+) RETL 11.2 will not run on AIX 4.3.3)
- HP-UX (versions 11 and 11i)

Note: See "What if I want to run RETL 11.x on unsupported platforms?"

Database management systems

- Oracle (8i and 9i)

Note: "What if I want to run RETL 11.x on unsupported databases?"

Data integration

Integrate external data of these forms:

- UNIX flat files
- Oracle database tables

Chapter 2 – Installation and system configuration

Installation

Install RETL on each server system that will be involved in inputting, outputting, or processing data. For example, if one system outputs data files and another system inputs that data and processes it, install RETL on both systems.

1 Log in as the `root` user on the appropriate host.

2 Create UNIX groups for the following environments:

- rfx – group that owns the RETL software

3 Create a UNIX operating system account on the appropriate host, using ksh as the default shell. `rfx - rfx group`

4 Create a directory where you will install this software.

5 Log in to the UNIX server as `rfx`.

6 Download retl_<version>_install.zip (the install package) and retl_<version>_doc.zip (the documentation) from the Retek Fulfillment center and place the RETL install package on your UNIX server.

7 Extract retl_<version>_install.zip:

$ unzip retl_<version>_install.zip

8 Change directories to the location where the package is installed at `<install_path>/<rfx_dir>`.

9 At a UNIX prompt enter: `>./install.sh`

Note: You must be in the `<cdrom_path>/<rfx_dir >` for the installation to successfully complete.

10 Follow the prompts to install RETL for your configuration.

```
$ ./install.sh
Enter directory for RETL software:
---> <enter path to RFX_HOME>
Is this the correct directory for the install? y or n
RFX_HOME: <path to RFX_HOME>
---> y
Creating RFX_HOME directory <path to RFX_HOME> ...
Creating install directory in <path to RFX_HOME> ...
Copying Library Files...
Copying Sample Files...
Copying Executables...
Copying Config File...
Copying JRE Files...
Installing The Java Runtime Environment (JRE) for <platform>...
JRE installed successfully at <path to RFX_HOME>/JRE/HP-UX/jre
Successful completion of RETL Install
To complete the RETL setup and installation:
1) Place the following in a .kshrc/ .profile to retain setup
variables:
RFX_HOME=/release/pkg_mocks/retl/test
PATH=/release/pkg_mocks/retl/test/bin:$PATH
2) Be sure to verify any additional environment setup as per the
"Setup" section
of the Programmers Guide.
3) Verify the installation by running the following command:
$RFX_HOME/bin/retl -version
$
```

11 Review the install.log file in the `<base directory>/install` to verify that RETL was installed successfully.

12 Set up your environment as per the Setup section below.

13 Verify the installation and setup by running the “verify_retl” script (see “Testing the Setup (verify_retl)”).

Upgrading from a 10.x release of RETL

11.x releases are required to be backwardly compatible with the XML flow interface of 10.x releases. As a result, minimal changes to a setup should be necessary to upgrade 10.x releases to 11.x releases. In short, these are the steps that you should follow:

1 Choose a new (and different) location to install the new version of RETL.

2 Install the new version of RETL following the installation instructions.

3 Change the environment that RETL runs within so that it refers to the new RFX_HOME (e.g. change within .kshrc or .cshrc).

4 Double check the environment variables to make sure that you weren't explicitly referring to old RETL directories explicitly.

5 Read the Backwards Compatibility section to determine the need to change flows/scripts.

6 Make any changes necessary to correct your flows to be able to run in 11.x versions.

7 Verify your flows/scripts run as expected. If any scripts/flows fail in running with the 11.x release but pass in running with the 10.x release and changes aren't otherwise noted in the Backwards-compatibility notes, FAQ, or RETL 11 Release Notes, notify support of this condition.

To make future upgrades more seamless, separate the RETL specific changes that you make to your environment so that the environment variables can be easily removed, modified and/or replaced when necessary.

Set up

1 After installation, set up your UNIX environment for RETL. The following example shows the variables you need in your UNIX profile to run RETL properly.

```
export RFX_HOME=<base directory>
export PATH=$RFX_HOME/lib:$RFX_HOME/bin:${PATH}
```

2 If you've selected an installation with database support, set up database and environment variables required for basic database setup. Please refer to the Appendix C – Database Configuration and Troubleshooting Guide for more information on setting up your database.

```
export ORACLE_HOME=/your/Oracle_home/directory
export PATH=$ORACLE_HOME/bin;$PATH
```

3 Log in to the UNIX server as `rfx`. At the UNIX prompt, enter:

```
>rfx
```

4 Make any changes to the operator defaults in rfx.conf. For example, Oracle database operators require hostname/port to be specified. These changes may be made to rfx.conf as DEFAULTS for convenience. See the example rfx.conf for details.

5 Make any changes to the temporary directory settings in rfx.conf. See the documentation for the TEMPDIR element in the "Configuration" section.

6 Make any changes to the default performance log file location configured in logger.conf.

The distributed logger.conf specifies /tmp/rfx.log as the default performance log file. If this file cannot be created, the log4j logging facility will report an error.

To change the performance log file location, modify logger.conf and change the value of the name parameter in the PERFORMANCE-APPENDER element. The line looks like the following:

```
<param name="file" value="/tmp/rfx.log"/>
```

For example, to change the performance log file location to /var/tmp/rfx/rfx-performance.log, change the line to look like this:

```
<param name="file" value=”/var/tmp/rfx/rfx-performance.log"/>
```

7 If RETL is installed correctly and the .profile is correct, the following results:

```
Error : Flow file argument ('-f') required!
```

Note: See the Troubleshooting section at the end of this document if you have problems or issues.

The RETL Package Verification Tool (verify_retl)

When setting up RETL in a new environment (or if you'd just like to verify that the RETL environment is set up properly) run the "verify_retl" script located in the `/bin` directory of the RETL installation (note verify_retl is available as of release 10.2).

Note: RFX_HOME should be set properly prior to running verify_retl.

The RETL package verification tool performs the following checks:

1 Verifies environment variables/etc is set up properly.

2 Ensures that the RETL binary is installed properly and can run.

3 Runs a series of system tests derived from the samples directory.

4 Logs information about environment setup in a log file.

The usage for verify_retl is as follows:

```
verify_retl [-doracle] [-nodb] [-h]
```

Option	Description
-doracle	Checks environment variables etc, for the Oracle installation of RETL.
-nodb	Checks environment variables for the standalone version of RETL.
-h	Displays the help message.

This generates the following output if successful:

```
Checking RETL Environment...found ORACLE environment...passed!
Checking RETL binary...passed!
Running samples...passed!
=================================================================
=====
Congratulations! Your RETL environment and installation passed all
tests. See the programmer's guide for more information about how to
further test your database installation (if
applicable)
=================================================================
=====

Exiting...saving output in /files0/retl/tmp/verifyretl-20384.log
```

Check the "RETL ENVIRONMENT SETUP" section of the log file for important information on environment variables that must be set.

Backwards Compatibility Notes

A major requirement for RETL 11.x releases is that they be backwardly compatible in the XML interface with 10.3.x releases. Much effort has been expended to provide this compatibility; however, there are a few small changes that must be made for certain operators when upgrading to 11.x from 10.x releases of RETL.

XML flow interface/operator differences between 10.x and 11.x releases:

Operator	Property	Backwards Compatibility Notes
Filter	filter	DEPRECATED SYNTAX – Previous (10.x) versions produced warning messages to correct 'filter' syntax. 11.x will not accept the following syntax in the filter property : >, <, >=, <=, =. These operations are replaced by GT, LT, GE, LE, EQ, respectively.

Database read operators(ORAREAD)	query	INVALID SYNTAX – Previous (10.x) versions allowed input of invalid XML in the query property of the dbread operators. Characters such as '>' and '<' in the query property will cause 11.x versions to produce an error message.
		An example follows:
		Previous XML valid in 10.x versions:
		`<PROPERTY name="query" value="SELECT * FROM ANY_TABLE WHERE ANY_COLUMN > 1" />`
		Property should now appear as the following in 11.x versions:
		`<PROPERTY name="query">`
		`<![CDATA[`
		`SELECT * FROM ANY_TABLE WHERE ANY_COLUMN > 1`
		`]]>`
		`</PROPERTY>`
		Note: A script has been provided to modify any flows to conform to the XML standard. This script is located in $RFX_HOME/bin/fix_flow_xml.ksh. View the header of the script for usage information.

Database read/write operators (ORAREAD/ORAWRITE)	dbname	NEW REQUIRED PROPERTY – The database to connect to. This property should be used instead of 'sid' going forward.
	port	NEW REQUIRED PROPERTY – The port on which the database listener resides. **Note:** For Oracle databases, use tnsping to obtain the port number. The default for Oracle is 1521. **Note:** this may need to be specified only once in the rfx.conf configuration file for convenience.

Operator	Property	Backwards Compatibility Notes
	hostname	NEW OPTIONAL PROPERTY – The fully specified hostname or IP address where the database resides. **Note:** This property should only be specified when it is known that connections are being made to a remote database **Note:** This may need to be specified only once in the rfx.conf configuration file for convenience.

Example (in operator):

```
<OPERATOR type="orawrite">
<OUTPUT name="test.v"/>
      <PROPERTY name="dbname"           value="databasename"/>
      <PROPERTY name="connectstring"    value="userid/password"/>
      <PROPERTY name="tablename"        value="mytable"/>
      <PROPERTY name="createtablemode" value="recreate"/>
      <PROPERTY name="hostname"         value="myhostname"/>
      <PROPERTY name="port"             value="1521"/>
</OPERATOR>
```

Example (in rfx.conf):

```
<DEFAULTS operator="orawrite">
      <PROPERTY name="hostname"         value="myhostname"/>
      <PROPERTY name="port"             value="1521"/>
</OPERATOR>
```

Feature Support differences between 10.x and 11.x releases

RETL 11.x does not fully implement certain command-line arguments and will not accept long argument options (e.g. –flow-file instead of –f)

Hardware requirements differences between 10.x and 11.x versions

In general, more physical memory is required in order to run the 11.x versions of RETL. There is no general formula or guideline for the additional memory requirement since it strongly correlates to the flow, data, configuration, etc.

rfx command line options

You can get help on rfx options on the command line by typing “rfx –h” on the command line. You should see something like the following:

```
>rfx -h
rfx [ OPTIONS ]

-h          Print help and exit
-oOPNAME    Print operator help. Valid values:
              operator name or 'ALL' for all ops
-e          Print RETL environment variable usage
-v          Print version and exit
-cFILE      Configuration File
-nNUMPARTS  Number of Partitions (SMP only)
-x          Disable partitioning (default=off)
-sTYPE      Display schema as TYPE. Valid values:
              NONE,SCHEMAFILE (default=NONE)
-lLOGFILE   Log statistics/times to LOGFILE
-fFLOWFILE  XML file containing flow
-d          Produce daVinci files (default=off)
-g          Produce flow graphs (default=off)
```

These options are discussed in more detail in the table below:

Option	Default Value	Description
-h	n/a	Shows the help message shown above.
-oOPNAME	n/a	Displays syntax usage for operator specified in OPNAME, or for all operators if OPNAME is 'ALL'. Valid operator names are the same as those operators used in the XML flow interfaces. The intention with this option is to provide 'online syntax help' for flow developers, reducing the need to refer to this document for syntax usage. See the Online Help section for more information about this option.
-e	n/a	Prints RETL environment variables that can be used for things such as turning on verbose debugging, setting JVM parameters, etc. See RETL Environment Variables for more information
-V	n/a	Displays the version and build number.
-cSTRING	$RFX_HOME/etc/rfx.conf	Overrides the default configuration file.
-nINT	As specified in the rfx.conf – or 1 if no rfx.conf is found.	The number of partitions to use. This feature is intended for RETL experts only.
-x	Partitioning as defined in the rfx.conf.	Disables partitioning.
-sTYPE	NONE	Prints the input and output schemas for each operator. Valid values and descriptions: NONE – rfx will not print any schema information. SCHEMAFILE – If specified, this option prints the input and output for each operator in schema file format so that developers can quickly and easily cut and paste rfx output to a file and break up flows. Developers could then modify these files for the purposes of specifying IMPORT and EXPORT schema files.  **Note:** rfx should be run with the –sNONE option in production systems where unnecessary output is not needed. The –sSCHEMAFILE option is often useful in development environments where it is desirable to debug RETL flows by breaking them up into smaller portions.

Option	Default Value	Description
-lLOGFILE	n/a	Specifies the log file in which to log RETL statistics/times. If the log file path is relative, the log file will be placed in the directory as defined in the TEMPDIR element of the RETL configuration file (rfx.conf). This changes the default log file as specified in rfx.conf and will turn on logging only if the log level in rfx.conf is set to "1" or more. For more information about the LOGGER feature, see the next section, entitled "Configuration"

-fSTRING	n/a	Specifies the file to use as input to rfx. This is where the XML flow is located. If no file is specified, rfx will read from standard input. The following is the syntax to use when reading from stdin via the korn shell (ksh):
		`rfx –f - <<EOF`
		`<FLOW>`
		…
		`</FLOW>`
		`EOF`

--davinci-files	Off	This option is not currently supported.
-g	Off	Produce visual graphs of a flow. **Note:** The flow is not run. On Solaris, this will produce an HTML web page that displays the graph. On AIX and HP-UX, this will produce a DOT file, which can be loaded with a tool called DOTTY. See http://www.research.att.com/sw/tools/graphviz/download.html for more information on DOTTY. See "Producing graphical output of flows with RETL" for more information on how to use this option

RETL Environment variables

You can retrieve a list of environment variables that can be set via the following ksh syntax:

```
export VARIABLE=VALUE
```

Option	Valid Values	Description
RFX_DEBUG	0,1	Option to turn on verbose RETL debugging to standard output. Set to '1' to turn on. Default is '0'.
RFX_SHOW_SQL	0,1	Option to turn on verbose RETL database debugging to standard output. Set to '1' to turn on. Default is '0'.
RETL_ENABLE_ASSERTIONS	0,1	Option to enable assertion checking in RETL (use only when there appears to be a bug in RETL itself). Set to '1' to turn on. Default is '0'.
RETL_VM_MODE	'highvol', 'lowvol'	Volume option. Set to 'highvol' to turn on JVM options for RETL in high volume environments with longer-running processes. Default is 'highvol' and this should not be changed unless the flow has been shown to run faster in 'lowvol' mode.
RETL_INIT_HEAP_SIZE	xxxM, where xxx is a number in Megabytes	Setting for the initial heap size for the Java Virtual Machine (JVM). Default is 50M.
RETL_MAX_HEAP_SIZE	xxxM, where xxx is a number in Megabytes	Setting for the maximum heap size for the Java Virtual Machine (JVM). Default is 300M.
RETL_JAVA_HOME	Any path to a valid Java Runtime Environment (JRE)	Option to reset the location of the Java Runtime Environment (JRE).
JAVA_ARGS	Valid JVM arguments	Option to set any JVM parameters. These will be placed on the command-line as arguments to the 'java' command. This option should not be used unless instructed to do so by Retek support, or if the user is aware of the implications of setting JVM parameters and has tested the results of making any changes.

Configuration

A configuration file may be specified by a user to control how RETL uses system resources, where to store temporary files, and to set default values for operators.

Configuration field descriptions

These explanations should assist you in modifying your configuration file. The RETL configuration file, rfx.conf, is located in the <`base_directory>/etc` directory.

Element Type	Attribute Name	Attribute Value	Description
CONFIGURATION			The root element of the RETL Configuration file. This element can have either NODE or DEFAULTS elements.
NODE	hostname	Host name	The name of the UNIX server where RETL is installed.
	bufsize	8..n	The number of records allowed between operators at any given time. The default bufsize is 2048 records. The bufsize can have a significant impact on performance. Setting this value too low causes a significant amount of contention between processes – slowing down RETL. A value that is too high can also slow RETL down because it will consume more memory than it needs to. Finding the right value for your hardware configuration and flow is part of tuning with RETL.
	numpartitions	1..n	Optional value, defaults to 1.

Element Type	Attribute Name	Attribute Value	Description
TEMPDIR	Path	Valid path	TEMPDIR is a child element of the NODE Element. It is the path to the directory where the RETL writes temporary files. We recommend that the number of temporary directories equals the number of partitions in order to maximize performance. Ideally each temp directory should be on a separate disk controller.  **Note:** These directories must always be local to the host where rfx is running. Use of network drives can have a drastic impact on performance. TEMPDIR by default points at /tmp. This should be changed to be a local disk after installation, as /tmp on many platforms is a memory device used for O/S swap space. If this is not changed, the box could be exhausted of physical memory. Care should be taken to protect the files within this directory since they can sometimes contain userid and password information. Please talk to your system administrator about setting the permissions so that only the appropriate personnel can access these files (e.g. by setting: umask 077). Temporary files should be removed from temporary directories on a daily or weekly basis. This very important server maintenance task aids in RETL debugging, reviewing database-loading utility log files, and so on. If a RETL module fails, the user should not re-run the module before removing the temporary files that were generated by the failing module.
GLOBAL			This element specifies global options for global settings within RETL.
	bytes_per_character	1,2, or 3	This setting specifies how RETL treats character data. In short, it affects how many bytes are in a character and can be used to allow RETL to work seamlessly with UNICODE data.

Element Type	Attribute Name	Attribute Value	Description
LOGGER			The LOGGER element specifies a facility for RETL performance logging. This gives a dynamic view of RETL to allow developers to get some information about the data that flows through each operator. This also enables developers to determine if/when deadlock conditions occur, debug problems and tune performance. See the properties that follow on how to configure the LOGGER for RETL.
	type	File	The type of logging facility to use. Currently the only value RETL allows is to log to a 'file'.
	dest	<output filename>	An optional output destination to log to. Currently, this value must be an absolute or relative filename. If the filename is relative, the log file will be placed in the directory as defined in the TEMPDIR element of the RETL configuration file (rfx.conf). The log file can be overridden by specifying –lLOGFILE as a command-line parameter to rfx. The default value is "rfx.log".  **Note:** The dest filename overrides the first performance log file location specified in logger.conf. See the "Performance Logger" section for more information.

Element Type	Attribute Name	Attribute Value	Description
	level	0,1,2	Specifies the level of detailed information recorded in the log file. Higher levels mean more information is logged. log level values have the following meaning: "0" = no logging. "1" = logging of operator start times "2" = logging of the above plus: • Operator end times • Record count per operator • Records per second per operator • Start and stop times for major events that are performed by RETL (e.g. query execution, database utility execution, external sorts, etc.) If the flow name is provided in the FLOW element, it is logged with each logger entry. When the log level is set to "2", a performance report in HTML format will also be written at the end of each RETL run. This report will show hotspots in the RETL flow, in actual time spent per operator.  **Note:** Leave logging turned off in production systems where performance is a critical factor. The logging feature is not designed to log errors, but is intended to give rough measures of flow performance characteristics and aid in debugging flows. It is recommended to periodically remove the log file to free unneeded disk space.
DEFAULTS			This element is a child of CONFIGURATION element. This section is used to define one or more default PROPERTY values for operators that are reused frequently. Care should be taken when using and changing these defaults since they can change the results of a RETL flow without changing individual flows.
	operator		Name of the operator to assign default values. Refer to the following chapters for the operators that are available.
PROPERTY			This element is a child of the DEFAULTS element.

Element Type	Attribute Name	Attribute Value	Description
	name		The name of the operator property to assign a default value. Refer to Chapter 5 for the property names for each operator.
	value		The value assigned as the default for the specified operator property. Refer to Chapter 5 for valid property values for each operator.

The following is a sample resource configuration for RETL:

```
<CONFIG>
   <NODE    hostname="localhost" numpartitions="1" bufsize="2048" >
      <TEMPDIR path=”/u00/rfx/tmp”/>
      <TEMPDIR path=”/u01/rfx/tmp”/>
   </NODE>
   <GLOBAL  bytes_per_character="1" >
      <LOGGER  type="file" dest="rfx.log" level="0" />
   </GLOBAL>
   <DEFAULTS   operator="oraread">
      <PROPERTY name="maxdescriptors" value="100"/>
      <PROPERTY name=”hostname” value=”mspdev25”/>
      <PROPERTY name=”port” value=”1521”/>
   </DEFAULTS>
   <DEFAULTS   operator="orawrite">
      <PROPERTY name=”hostname” value=”mspdev25”/>
      <PROPERTY name=”port” value=”1521”/>
   </DEFAULTS>
</CONFIG>
```

Logger Configuration

Logging in RETL is performed using the log4j logging facility. log4j is an extremely flexible open-source package maintained by the Apache Software Foundation.

log4j can be configured to send logged output to the console, a file, or even a file that automatically rolls over at a given frequency. log4j can also send logged information to more than one destination. Additionally, log4j loggers have a tunable logging level that can control how much information is logged. All of the capabilities of log4j cannot be documented here. Refer to http://logging.apache.org/log4j/docs/documentation.html for more documentation on log4j.

RETL uses two log4j loggers. The first logger is the performance logger. The second logger, the output logger, handles RETL output that is normally sent to the terminal.

Performance Logger

The performance logger logs operator statistics such as start time, stop time, and records processed per second. It also records the start and stop time of various system events, such as sorts and database queries. See the "Configuration" section for more information.

The performance logger is configured in the logger.conf file located in the `<base_directory>/etc` directory. The performance logger's log4j name is name is "retek.retl.performance".

To turn on performance logging, edit logger.conf and find the "logger" XML element where the name attribute is "retek.retl.performance". Change the level to "DEBUG". (By default the level is "WARN".)

Performance information will be logged to `/tmp/rfx.log`. To change the location of this file, change the file specified in the PERFORMANCE-APPENDER appender.

Note: If a file is specified in the LOGGER element in the rfx.conf file, it will override the first file specified in the logger.conf file.

Output Logger

The output logger logs informational, warning, and error messages. By default, all of these messages are written to the terminal. By configuring the output logger, the destination for these messages can be changed.

If you want to use any of the advanced features of log4j, change the output logger's settings in logger.conf. The log4j name of the output logger is "retek.retl".

For example, if you want to log all errors into a file and also want them displayed on the terminal, make the following changes to the logger.conf file:

1 Add the following before the logger element for the "retek.retl" logger, replacing *file-name-for-RETL-errors* with the name of the file you want to use:

```
<appender name="ERRORSTOFILE" class="org.apache.log4j.FileAppender">
 <param name="file" value="file-name-for-RETL-errors"/>
 <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="[%d] %-5p - %m%n"/>
 </layout>
 <filter class="org.apache.log4j.varia.LevelRangeFilter">
    <param name="acceptOnMatch" value="true"/>
    <param name="levelMin" value="ERROR"/>
    <param name="levelMax" value="FATAL"/>
 </filter>
</appender>
```

2 Add the following to the "retek.retl" logger element:

```
<appender-ref ref="ERRORSTOFILE"/>
```

The first step creates an "appender." Appenders tell the log4j logging system where and how to log messages. The example above tells log4j that the ERRORSTOFILE appender is a file appender that writes to the file specified in the file parameter. The pattern layout shows how to format the message, and the filter dictates that only messages logged at level ERROR or above are logged.

The second step associates the ERRORSTOFILE appender with the "retek.retl." logger.

Refer to http://logging.apache.org/log4j/docs/documentation.html for more documentation on log4j.

Multi-byte character support

The optional variable “bytes_per_character” allows RETL to provide multi-byte character support. The number of bytes read or written is dependent upon the size of a character. RETL determines the size of a string by multiplying the number of characters by bytes_per_character. If you are using databases, bytes_per_character is set up by consulting your database configuration.

For Imports and Exports, RETL uses the schema file to determine the number of bytes a particular string will have. For a fixed length field the number of bytes is simply <number of chars> * <bytes_per_character>. For delimited fields, the maximum number of bytes is <maxlength>*<bytes_per_character>.

Chapter 3 – RETL Interface

There are two interfaces to RETL. The traditional interface is through a flow file, which is a set of processing instructions in XML format. An additional interface allowing direct access to RETL functionality through a Java class library was introduced in RETL 11.3.

Note: Neither interface is preferred over the other. All functionality in one interface is available in the other. Developers should choose the interface to RETL depending on the specific integration requirements.

Terms

Term	Definition
Flow	The instructions that RETL executes. A flow is a collection of operators.
Record	A RETL record is like a database record. It is a collection of fields of different datatypes.
Dataset	A dataset is a collection of records. It is similar to a database table. A dataset that is input to an operator is called an *input dataset*. A dataset that is output from a record is called an *output dataset*.
Operator	A RETL operator creates, transforms, or writes records in a dataset. When speaking of an operator independent of an interface type in this document, all uppercase letters are used.
Property	Operators have properties that tell the operator more information on how it should perform its job. For example, the IMPORT operator has an inputfile property that tells the IMPORT operator which file to import.

RETL XML Interface

When using the RETL XML interface, the flow is specified in a file in XML format. (Consult a reference manual or website if you are unfamiliar with XML.) The XML format follows these rules:

- The root node is named FLOW. That is, the entire XML file contents are contained within <FLOW> and </FLOW> tags.
- The FLOW element has an optional attribute named “name”. The value of this attribute is used when logging information about the flow.
- The FLOW element requires two or more OPERATOR children elements that specify the operators.
- The OPERATOR element has a mandatory attribute named “type”. The value of this attribute specifies what type of operator is being specified. The operator type is not case sensitive, although all lowercase is recommended. The different types of operators are detailed later in this document.

- The properties of an operator are specified using a PROPERTY element of the OPERATOR element. The PROPERTY element has two attributes named “name” and “value”. The “name” specifies the name of the property and the “value” specifies the value of the property. The valid properties for each operator are detailed later in this document.
- The input dataset(s) to an operator are specified using the INPUT child element of the OPERATOR element. The INPUT element requires a single attribute named “name”. Its value is the name of the dataset. Each input dataset must be an output dataset of another operator.
- The output dataset(s) of an operator are specified using the OUTPUT child element of the OPERATOR element. The OUTPUT element requires a single attribute named “name”. Its value is the name of the dataset. Each output dataset must be an input dataset to some other operator.
- XML comments are supported and encouraged.

Here is an example of a flow that reads in data from a file using the IMPORT operator and writes the records to an Oracle database using the ORAWRITE operator:

```
<FLOW name="dataload_flow">
   <!-- Import the data.txt file. -->
   <OPERATOR type="import">
      <PROPERTY name="inputfile" value="data.txt"/>
      <PROPERTY name="schemafile" value="data.schema.xml"/>
      <OUTPUT name="data.v"/>
   </OPERATOR>

   <!-- Write to the MYDATA table on MYDATABASE -->
   <OPERATOR type="orawrite">
      <INPUT name="data.v"/>
      <PROPERTY name="dbuserid" value="username/password"/>
      <PROPERTY name="dbname" value="MYDATABASE"/>
      <PROPERTY name="tablename" value="MYDATA"/>
   </OPERATOR>
</FLOW>
```

Operator Nesting

"Operator Nesting" is a way of tying logically related operators together and reduces the number of lines in a flow. An OUTPUT for an operator can be replaced with the operator that would receive the corresponding INPUT.

Note: Using operator nesting will not change the performance of a flow.

For example, the following two flows are equivalent. The second flow replaces OUTPUT of "sales.v" in the IMPORT with the ORAWRITE, and the ORAWRITE does not have an INPUT element:

```
<FLOW>
   <OPERATOR type="import">
      <PROPERTY name="inputfile" value="sales.txt"/>
      <PROPERTY name="schemafile" value="sales-schema.xml"/>
      <OUTPUT name=”sales.v”>
   </OPERATOR>
   <OPERATOR type=”orawrite”>
      <INPUT name=”sales.v”>
      <PROPERTY name=”tablename” value=”SALES”/>
      ...
   </OPERATOR>
</FLOW>

<FLOW>
   <OPERATOR type="import">
      <PROPERTY name="inputfile" value="sales.txt"/>
      <PROPERTY name="schemafile" value="sales-schema.xml"/>
      <OPERATOR type=”orawrite”>
         <PROPERTY name=”tablename” value=”SALES”/>
         ...
      </OPERATOR>
   </OPERATOR>
</FLOW>
```

RETL Java Interface

The RETL Java Interface exposes the RETL functionality through a Java class library, i.e. an API. This section documents how the Java interface compares with the XML interface. Documentation of each class can be found in the Javadocs distributed with the other RETL documentation.

Unless otherwise noted, all classes reside in the `com.retek.retl` package.

Note: When the RETL Java interface is used, RETL runs within the same JVM as the initiating application. This could lead to resource contention issues. The initiating application must be started with appropriate settings for initial heap size, maximum heap size, thread stack size, as well as any garbage collection parameters.

For more information on parameters to set, consult the "Performance Tuning Guide" and the retl shell script.

Initialization

When the XML interface is used, the RETL runtime environment is set through environment variables and command line options to the retl shell script. In the Java interface, the RETL runtime environment must be initialized using the `RETL` class. This class has methods that allow you to set the various runtime parameters. See the Javadoc for more information.

Flows

Just like the XML interface, the Java interface runs flows, only in the Java interface a flow is an instance of the `Flow` class. Operators are added to the flow using the `add()` method. After all of the operators have been added to the flow, the flow is executed using the `run()` method.

Operators

For each operator type in the XML interface there is a corresponding class in the Java interface. For example, the IMPORT operator's functionality is contained in a class named `Import`. The operators are documented in chapters 7 through 13.

The class name is always an upper case first letter followed by lower case letters, even for operator names that consist of more than one word. For example, the class name for the CHANGECAPTURE operator is `Changecapture`, not `ChangeCapture`.

Properties

Each operator property is set through a method with the same name as the property. The valid properties for each operator are documented in chapters 7 through 13.

The method name for setting a property is all lowercase, even when the property name consists of more than one word. For example, the schemafile property of the IMPORT operator is set through the `Import.schemafile()` method, not `Import.schemaFile()`.

Currently, there are only setter methods; there are no getter methods implemented.

Datasets

Operators have `input()` and `output()` methods that set the name of the input and output datasets of an operator. These methods correspond to the INPUT and OUTPUT elements of the XML interface.

The following Java class runs a flow that is identical to the “dataload_flow” example flow from the XML Interface:

```
import com.retek.retl.*;
import com.retek.retl.base.RETLException;

...

   try
   {
      RETLProperties retlProperties = new RETLProperties();
         retlProperties.setRETLHomeDirectory(...);
      RETL.initialize(retlProperties);

      Flow flow = new Flow();

      Import opImport = new Import();
      opImport.inputfile("data.txt");
      opImport.schemafile("data.schema.xml");
      opImport.output("data.v");
      flow.add(opImport);

      Orawrite opOrawrite = new Orawrite();
      opOrawrite.input("data.v");
      opOrawrite.dbuserid("username/password");
      opOrawrite.dbname("MYDATABASE");
      opOrawrite.tablename("MYDATA");
      flow.add(opOrawrite);

      flow.run();
   }
   catch (RETLException e)
   {
      RETL.handleException(e);
      System.exit(1);
   }
```

Chapter 4 – RETL program flow

Program flow overview

The following text and diagram provides an overview of the RETL input-process-output model. The data input operators to RETL processing include the following:

- DBREAD, where data is read directly from a database
- IMPORT, where RETL accepts a data file
- GENERATOR, where RETL itself creates data input

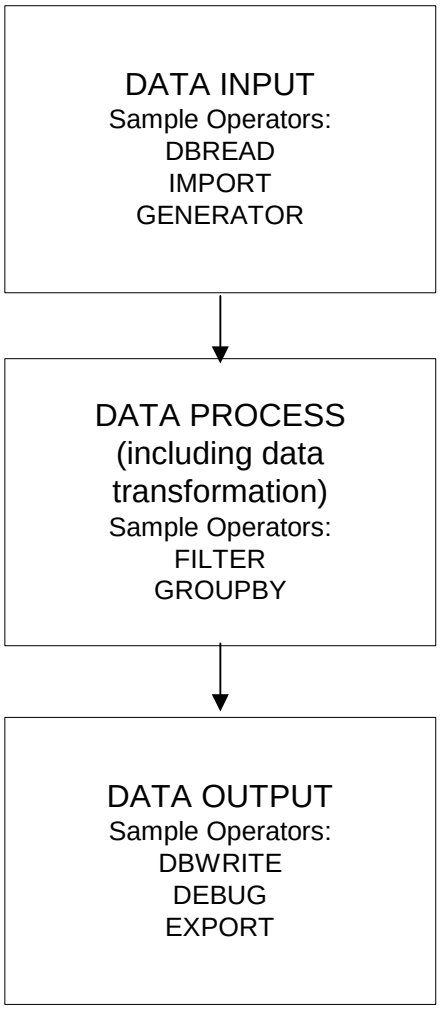
Data process operators commonly include those that transform data in the dataset being processed, including:

- GROUPBY, which can be used to sum values in a table column

The data output process can include the use of the following operators:

- DBWRITE, where RETL writes data directly to the database
- DEBUG, which can be used to print records directly to the screen (stdout).
- EXPORT, which can export data to a flat file

The general flow



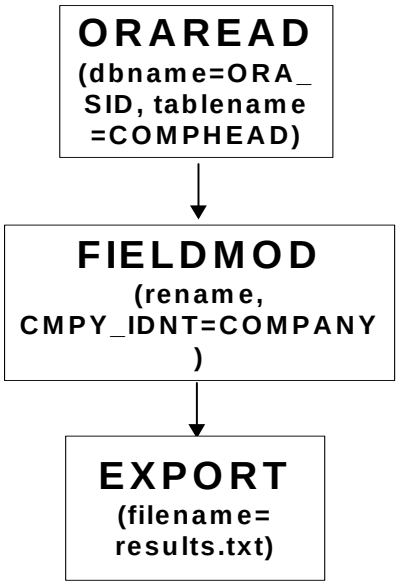
The diagram above is a general representation of the form that all flows take. Every flow must have a Data Input (or Source) and a Data Output (or Sink). These flows may optionally perform transformations on the input data before sending it to the Data Output.

Some things to note about all flows:

- When an operator generates an output dataset, that output dataset must be input into another operator. The data "flows" from one operator to another along the datasets (the arrows between the operators above).
- DATA INPUT operators have no dataset inputs – they get data from outside sources (e.g. Oracle database or flat file).
- Similarly, DATA OUTPUT operators are terminal points within the flow. They do not generate output datasets – rather they export records to an external repository (for example, Oracle or flat files).

In the next several sections we look at specific flows.

A simple flow



The diagram above is one way to represent a simple flow. This flow contains three operators: ORAREAD, FIELDMOD and EXPORT. This flow also contains two datasets – these are the simple arrows that connect the operator boxes above. The arrow from ORAREAD to FIELDMOD means that records flow from the ORAREAD operator to the FIELDMOD. Likewise the arrow from FIELDMOD to EXPORT indicates that the records from the FIELDMOD are passed onto the EXPORT operator.

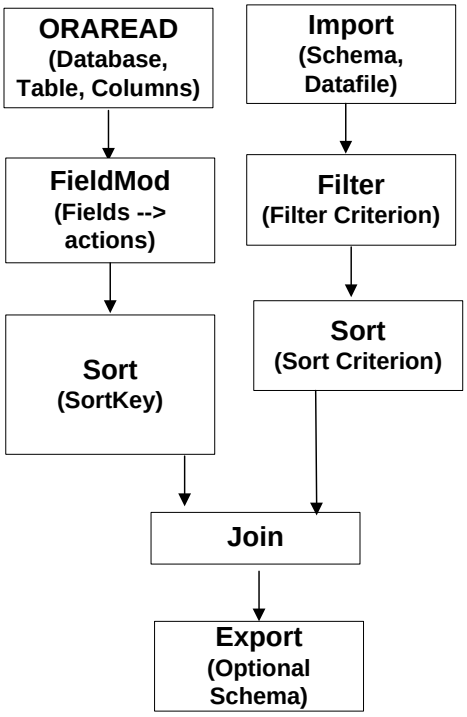
In this flow diagram, several attributes are also shown to give us more information about what exactly this flow does. This flow pulls the “COMPHEAD” table out of the “ORA_SID” database (via the ORAREAD operator), renames the column “CMPY_IDNT” to “COMPANY” (via the FIELDMOD operator), and exports the resulting records to the file “results.txt” (via the EXPORT operator).

Here is XML for the above flow:

```
<FLOW name="cmpy.flw">
   <OPERATOR type="oraread">
   <PROPERTY name="dbname"         value="ORA_SID"/>
   <PROPERTY name="connectstring" value="userid/passwd"/>
   <PROPERTY name="query">
      <![CDATA[
         select * from COMPHEAD
      ]]>
   </PROPERTY>
   <OUTPUT name = “data.v"/>
</OPERATOR>
 <!-- This is the format for a comment.
   --  They can of course be
```

```
    --  multi-line, but cannot be nested.
  -->
    <OPERATOR type="fieldmod">
       <PROPERTY name="rename" value="CMPY_IDNT=COMPANY"/>
       <INPUT name = “data.v"/>
       <OUTPUT name = "comphead.v"/>
    </OPERATOR>
    <OPERATOR type="export">
       <INPUT name="comphead.v"/>
       <PROPERTY name="outputfile" value="results.txt"/>
    </OPERATOR>
</FLOW>
```

A more complex flow



Flows can get much more complex than the simple flow shown above.

The above flow pulls in data simultaneously from a flat file and Oracle Database. The dataset from the file is filtered based upon the specified filter criterion then sorted. In parallel, the dataset from the Oracle database is modified via the FIELDMOD operator and then sorted. The resulting datasets are joined and the results are output to a flat file via the Export operator.

Several things to note:

- RETL is capable of dealing with multiple data sources within a single flow.
- RETL is capable of outputting results to multiple files and/or database tables.

Online Help

RETL 11.x introduces an online help system that enables flow developers to obtain operator syntax and usage information directly from the rfx binary. The following help can be obtained:

- specific property names for each operator
- confirmation of whether a property is required or optional
- valid values for each property
- a brief description of each property
- default values for each property (if property is optional)

For more information about this feature, see the –o option in the section entitled RFX Command line options

Debugging with RETL

It is often useful for RETL flow programmers or system administrators to diagnose what is happening inside of RETL. This can assist in tracking down errors or performance bottlenecks. There are several mechanisms that RETL offers to aid in debugging:

- Verbose messages

  Verbose can be turned on via the following variables:

  - export RFX_DEBUG=1
  - export RFX_SHOW_SQL=1

RFX_DEBUG will turn on additional informational messages about RETL operations to standard output.

RFX_SHOW_SQL will turn on additional logging of database operations to standard output.

Printing in schema file format – See "Configure RETL to print schema output in Schema File format"

Printing graphical output of flows – See "Producing graphical output of flows with RETL"

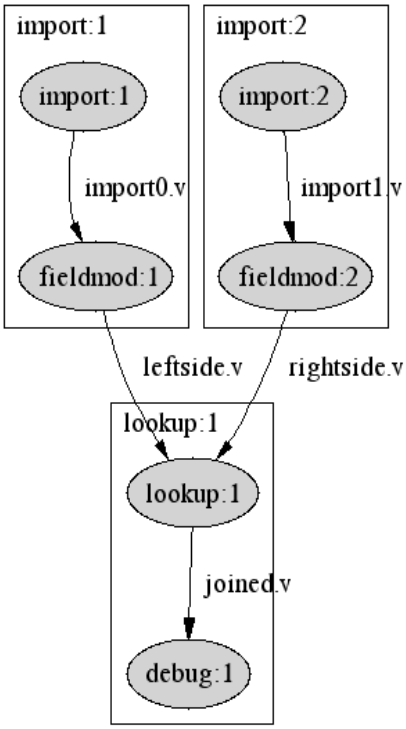
Performance logging – See "Configuration Options – LOGGER"

Producing graphical output of flows with RETL

Beginning with version 11.2, RETL offers an option to print out graphs of flows. This allows a flow developer or user to get an actual 'picture' of a RETL flow, of RETL operators and link names that connect each operator. In addition to being a visual aid, this graph is invaluable for tracing down problems with a flow. RETL error messages will often denote which operator a particular error was thrown from. By noting the link names that connect operators, one can relatively easily trace problems directly back to the source operator, fix the problem, and re-run the flow. An example of running a flow with the '-g' command-line option is as follows:

1 rfx –f <flow name.xml> -g

2 load rfx<random#>_serial_flow.html in any web browser to view the output

XML Flow	RETL visual graph
<FLOW name="example.flw"> <OPERATOR type="import"> <PROPERTY name="inputfile" value="input0.dat"/> <PROPERTY name="schemafile" value="schema0.xml"/> <OUTPUT name="import0.v" /> </OPERATOR> <OPERATOR type="fieldmod"> <INPUT name="import0.v"/> <PROPERTY name="keep" value="A D"/> <OUTPUT name="leftside.v"/> </OPERATOR> <OPERATOR type="import"> <PROPERTY name="inputfile" value="input1.dat"/> <PROPERTY name="schemafile" value="schema1.xml"/> </OPERATOR> <OPERATOR type="fieldmod"> <INPUT name="import1.v"/> <PROPERTY name="keep" value="B D"/> <OUTPUT name="rightside.v"/> </OPERATOR> <OPERATOR type="lookup"> <INPUT name="leftside.v" /> <INPUT name="rightside.v" /> <PROPERTY name="tablekeys" value="D"/> <OUTPUT name="joined.v" />	

XML Flow	RETL visual graph
</OPERATOR> <OPERATOR type="debug"> <INPUT name="joined.v"/> </OPERATOR> </FLOW>	

This is an invaluable tool when debugging a RETL flow. For example, if RETL came back with an error and a user wanted to figure out which lookup operator the error came from:

```
Exception in operator [lookup:1]
Record (A|B) can't be found in lookup relation!
```

1 Print RETL flow graph (see RETL syntax above)

2 Locate operator with name ‘lookup:1’ in flow graph (also can be located directly in the XML by counting down to the first lookup operator in the flow)

3 Locate link names that connect to ‘lookup:1’ and search the XML flow file for the lookup operator that contains these link names. For example, in the above flow, one can see that ‘lookup:1’ has links named ‘leftside.v’, ‘rightside.v’ as INPUTs, and ‘joined.v’ as OUTPUTs. Any of these can be used to search for the appropriate lookup in the flow since link names must be unique.

Performance Logging with RETL

RETL records the following diagnostic information when the retek.retl.performance log4j logger is set to level DEBUG:

1 Processing time for each operator

2 Count of processed records for each operator

3 Throughput in records per second for each operator

4 Major events taking place within each operator (for example, start/stop times for queries in DBREAD operators, calls to native database load utilities in DBWRITE operators, or calls to external gsort in SORT).

This information is logged in the performance log file. Additionally, the first three items are used to create an Operator Time Comparison Chart and a Performance Graph.

The file locations of the Operator Time Comparison Chart and the Performance Graph are output when RETL completes processing:

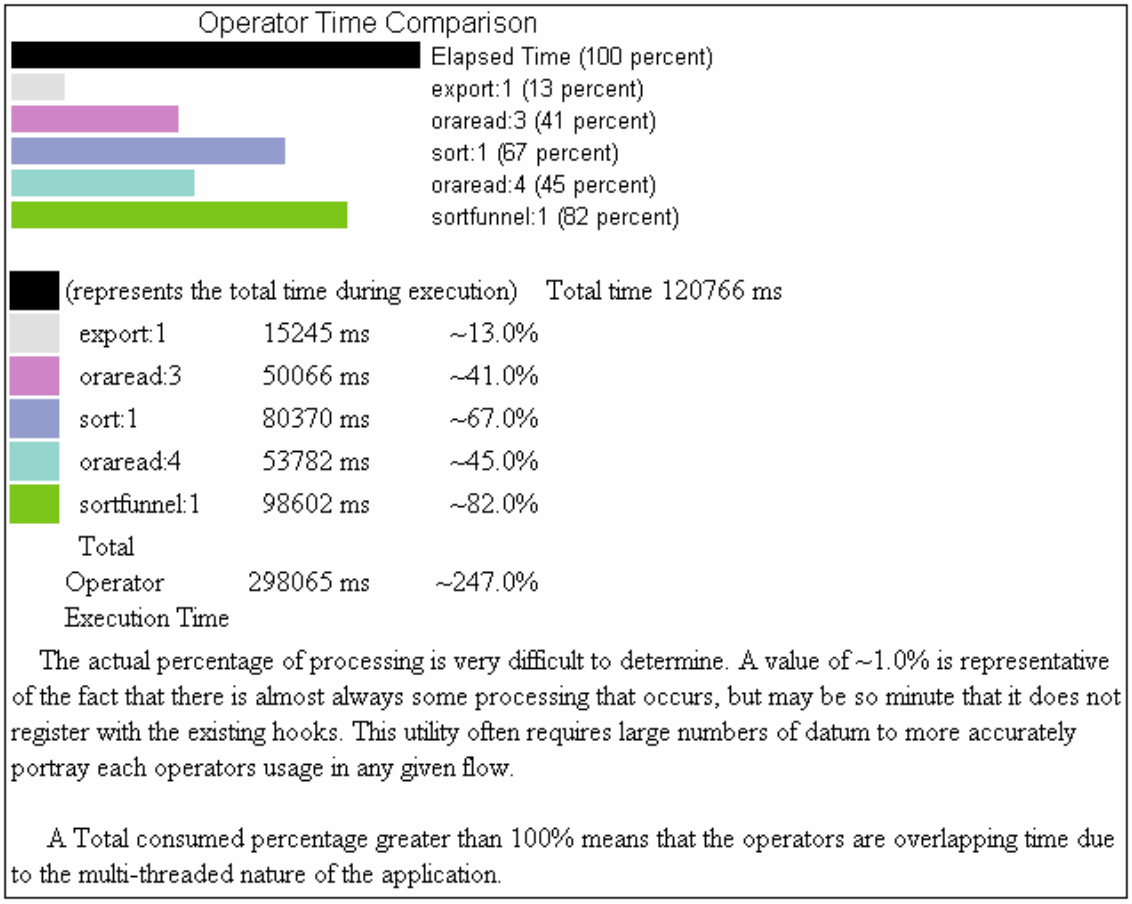
```
All threads complete

Flow ran successfully
Open operator-time-comparison-chart-file in your browser to see the
performance page.
1 File written: performance-graph-file
To view the output, use this command to load the file into
dotty:<br>
dotty performance-graph-file
```

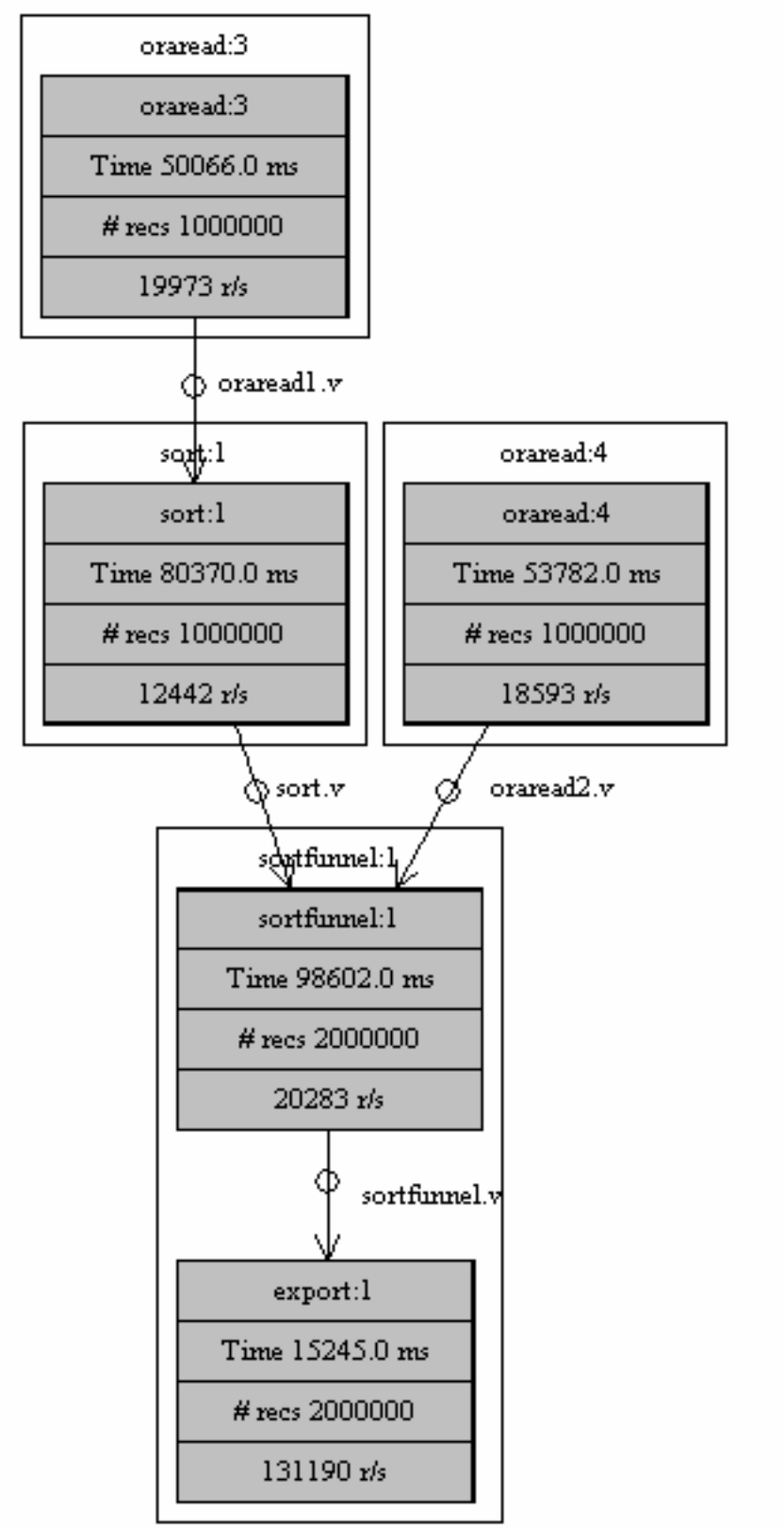
Sample log file

```
[2004-12-08 16:04:15,193]PERF: oraread:3 Starting query 1 of 1:
select * from stresstests_1000000
[2004-12-08 16:04:15,256]PERF: oraread:3 Ending query 1
[2004-12-08 16:04:15,384]PERF: oraread:4 Starting query 1 of 1:
select * from stresstests_1000000 ORDER BY SEQFIELD
[2004-12-08 16:04:15,420]PERF: oraread:4 Ending query 1
[2004-12-08 16:04:15,479]User time: sortfunnel:1 started!
[2004-12-08 16:04:15,480]User time: sort:1 started!
[2004-12-08 16:04:15,480]User time: oraread:3 started!
[2004-12-08 16:04:15,481]User time: oraread:4 started!
[2004-12-08 16:05:06,460]User time: Operator "pipeline:2" finished -
processed 1,000,000 records in 50.977 seconds ( 19,616.69
records/sec)
[2004-12-08 16:05:06,464]User time: Operator "oraread:3" finished -
processed 1,000,000 records in 50.968 seconds ( 19,620.154
records/sec)
[2004-12-08 16:05:06,475]PERF: sort:1 Starting gsort
[2004-12-08 16:05:20,426]PERF: sort:1 Ending gsort
[2004-12-08 16:05:20,529]User time: export:1 started!
[2004-12-08 16:06:14,122]User time: Operator "import:1" finished -
processed 1,000,000 records in 53.685 seconds ( 18,627.177
records/sec)
[2004-12-08 16:06:14,125]User time: Operator "pipeline:1" finished -
processed 1,000,000 records in 118.643 seconds ( 8,428.647
records/sec)
[2004-12-08 16:06:14,128]User time: Operator "sort:1" finished -
processed 1,000,000 records in 118.643 seconds ( 8,428.647
records/sec)
[2004-12-08 16:06:14,191]User time: Operator "pipeline:3" finished -
processed 1,000,000 records in 118.708 seconds ( 8,424.032
records/sec)
[2004-12-08 16:06:14,193]User time: Operator "oraread:4" finished -
processed 1,000,000 records in 118.708 seconds ( 8,424.032
records/sec)
[2004-12-08 16:06:14,201]User time: Operator "pipeline:0" finished -
processed 2,000,000 records in 118.72 seconds ( 16,846.361
records/sec)
[2004-12-08 16:06:14,204]User time: Operator "sortfunnel:1" finished
- processed 2,000,000 records in 118.72 seconds ( 16,846.361
records/sec)
[2004-12-08 16:06:14,206]User time: Operator "export:1" finished -
processed 2,000,000 records in 53.675 seconds ( 37,261.295
records/sec)
```

Sample Operator Time Comparison Chart



Sample Performance Graph



Chapter 5 – RETL Schema Files

RETL can process datasets either directly from a database or from file-based systems. Depending upon the data source, one of two operators can perform this function. For reading data directly from database tables, use the ORAREAD operator. See Chapter 7 - Database operators, for more information about how to use ORAREAD. For situations where you expect RETL to process data in a file format, use the IMPORT operator. This operator imports a disk file into RETL, translating the data file into a RETL dataset.

Schema file requirements

RETL stores information about each dataset that describes the data structures (the metadata). The means by which you supply metadata is dependent on whether you interface data using the DBREAD or IMPORT operator. When datasets are created using DBREAD, the metadata is read along with the data directly from the database table. In this case, RETL requires no further metadata. On the other hand, if the IMPORT operator is used for dataset input, RETL requires the use of a schema.

The schema ensures that datasets are consistent from the source data to the target data. RETL reads in a schema file that is specific to the dataset that is being imported. RETL uses the schema file to validate the input dataset structure. For example, if the dataset ultimately populates a target table in a database, the schema file dictates the type and order of the data, as well as the data characteristics, such as the character type, length, and nullability.

Two types of schema files are defined for use with an incoming text data file:

- Delimited file type
- Fixed length file type

Schema file XML specification

Element Type	Attribute Name	Attribute Value	Description
RECORD			The root element.
	type	“delimited” or “fixed”	Describes whether the schema is fixed records or delimited records.
	final_delimiter	character	Required field. End of record delimiter that is used within the input file to distinguish the end of a record. This is extremely important to help distinguish between records in the event that the input data file is corrupt. Generally this is a newline character for readability.
	len	1..n	Required only for fixed length fields. This is the total length of the fixed length record. Note that this must be equivalent to the sum of all aggregate field lengths.
	sortkeys	<space separated list of keys >	Optional property to specify the space-separated keys that the records are presorted by. This should be used only when it is known that the records will always be sorted by specified keys and it is necessary to remove the sort for performance purposes. This property must be specified in conjunction with ‘sortorder’.
	sortorder	asc or desc	Optional property to specify the sort order of the record. Specify ‘asc’ for ascending sorted order and ‘desc’ for descending sorted order. This should be used only when it is known that the records will always be sorted by specified keys and it is necessary to remove the sort for performance purposes. This must be specified in conjunction with ‘sortkeys’.
FIELD			Child element of the RECORD.
	name		The column name of the given field.
	delimiter		This optional attribute is for delimited fields only. It specifies the end of field delimiter, which can be any character, but the default value is the pipe character (‘|’).

Element Type	Attribute Name	Attribute Value	Description
	datatype	Any one of: “int8”, “int16”, “int32”, “int64”, “uint8”, “uint16”, “uint32”, “uint64”, “dfloat”, “sfloat”, “string”, “date”, “time”, or “timestamp”	The datatype of the field. If the date data type is declared, the incoming data must be formatted as ‘YYYYMMDD’. If the time data type is declared, the incoming data must be formatted as ‘HH24MISS’. If the timestamp data type is declared, the incoming data must be formatted as ‘YYYYMMDDHH24MISS’
	nullable	“true” or “false”	Optional field describing whether null values are allowed in the field. The default value is false. If true then a nullvalue MUST be specified.
	nullvalue	Quoted string	Required if nullable=”true”. This is the text string that an IMPORT operator looks at to determine whether or not a field is null when reading flat files. This should be a unique value that will not otherwise be found within the dataset. For fixed length fields, this value must be equal in length to the field length so that the record length remains fixed. **Note:** The null string may specified (“”) for a fixed length field rather than specifying a number of blanks equivalent to the length of the field. Null values must be a valid value for the type specified. For example, ‘00000000’ as a nullvalue for a date is invalid.
	maxlength	Number	Required only for delimited “string” fields. Specifies the maximum allowable length of the string field.
	len	Number	Required only for all fixed length fields. Specifies the length of the field.
	strip	“leading”, “trailing”, “both”, or “none”	Optional attribute used for delimited “string” fields to determine whether and how white space (tabs and spaces) should be stripped from input field. The default value is “none”.

Delimited record schema

When you create a schema for use with a delimited file type, follow these guidelines:

- The delimiter can be any symbol.

  Note: Make sure the delimiter is never part of your data.

- To define the field's nullability, set nullable to 'true' or 'false'.
- If the field is nullable, set the default nullvalue. This is often set to "" – the null string.
- For the string data type, specify the maxlength.

An example delimited schema file

```
<RECORD type="delimited" final_delimiter="0x0A">

     <FIELD name="colname1" delimiter="|" datatype="int8"
nullable="false"/>

     <FIELD name="colname2" delimiter="|" datatype="int16"
nullable="true"
nullvalue=""/>

     <FIELD name="colname3" delimiter="|" datatype="int32"
nullable="true"  nullvalue=""/>

     <FIELD name="colname4" delimiter="|" datatype="int64"
nullable="true"  nullvalue=""/>

     <FIELD name="colname5" delimiter="|" datatype="dfloat"
nullable="true"  nullvalue=""/>

     <FIELD name="colname6" delimiter="|" datatype="string"
maxlength="5" nullable="false“                    />

     <FIELD name="colname7" delimiter="|" datatype="date"
nullable="false“                    />

     <FIELD name="colname7" delimiter="|" datatype="timestamp"
nullable="false“                    />

     <FIELD name="colname8" delimiter="|"   datatype="uint8"
nullable="false"                    />

     <FIELD name="colname9" delimiter="|"   datatype="uint16"
nullable="true"  nullvalue=""/>

     <FIELD name="colname10" delimiter="|" datatype="uint32"
nullable="true"  nullvalue=""/>

     <FIELD name="colname11" delimiter="|" datatype="uint64"
nullable="true"  nullvalue=""/>

     <FIELD name="colname12" delimiter="|" datatype="sfloat"
nullable="true"  nullvalue=""/>

</RECORD>
```

Fixed length record schema

When you create a schema for use with fixed length records, follow these guidelines:

- Specify the length (“len”) of the record.
- Specify a “len” value for every field. The total of all fields must be equivalent to the record length.
- To define the field’s nullability, set nullable to “true” or “false”.
- If the field is nullable, set the default nullvalue. Remember that since this is a fixed length field the null value must be the correct length for the field.
- If possible, specify the “final_delimiter” to ensure that input files are at least record-delimited so that RETL can recover in the event that data is corrupted or invalid.

 **Note:** Make sure the final_delimiter is never part of your data.

An example fixed length schema file

```
<RECORD type="fixed"     len=”99”        final_delimiter="0x0A">
     <FIELD name="colname1"    len=”2”    datatype="int8"
nullable="false"                    />
     <FIELD name="colname2"    len=”5”    datatype="int16"
nullable="false"/>
     <FIELD name="colname3"    len=”10”  datatype="int32"
nullable="false"/>
     <FIELD name="colname4"    len=”18”  datatype="int64"
nullable="false"/>
     <FIELD name="colname5"    len=”6”  datatype="dfloat"
nullable="true"  nullvalue="NULVAL"/>
     <FIELD name="colname6"    len=”5”   datatype="string"
nullable="false“                    />
     <FIELD name="colname7"    len=”8”   datatype="date"
nullable="false“                    />
     <FIELD name="colname7"    len="14" datatype="timestamp"
nullable="false“                    />
     <FIELD name="colname8"    len=”2”   datatype="uint8"
nullable="false"                    />
     <FIELD name="colname9"    len=”4”   datatype="uint16"
nullable="true"  nullvalue="XXXX"/>
     <FIELD name="colname10"  len=”5”   datatype="uint32"
nullable="true"  nullvalue="     "/>
     <FIELD name="colname11"  len=”18” datatype="uint64"
nullable="false"/>
     <FIELD name="colname12"  len="18" datatype="sfloat"
nullable="false"/>
</RECORD>
```

nullvalue Considerations

Because nullvalues can be changed between import and export schemas, it is possible to lose data if you are not careful. However, this property can also be used to assign default values to null fields on export. No matter how you use them, care should be taken when selecting values for a field's nullvalue. Unless you are assigning a default value to a null field on export we strongly recommend against using a value that can actually appear as a valid value within a given dataset.

To illustrate consider the following schema (named "1.schema"):

```
<RECORD type="delimited" final_delimiter="0x0A">
  <FIELD name="colname1" delimiter="|" datatype="int8"
nullable="false" />
  <FIELD name="colname2" delimiter="|" datatype="dfloat"
nullable="true" nullvalue=""/>
</RECORD>
```

Also consider this schema (named "2.schema") where the nullvalue has been modified for the second field:

```
<RECORD type="delimited" final_delimiter="0x0A">
  <FIELD name="colname1" delimiter="|" datatype="int8"
nullable="false" />
  <FIELD name="colname2" delimiter="|" datatype="dfloat"
nullable="true" nullvalue="0.000000"/>
</RECORD>
```

When running through the following flow:

```
<FLOW>
  <OPERATOR type="import">
    <PROPERTY name="inputfile" value="1.dat" />
    <PROPERTY name="schemafile" value="1.schema" />
    <OUTPUT name="import.v" />
  </OPERATOR>
  <OPERATOR type="export">
    <INPUT name="import.v" />
    <PROPERTY name="outputfile" value="2.dat" />
    <PROPERTY name="schemafile" value="2.schema" />
  </OPERATOR>
</FLOW>
```

Where 1.dat contains the following data:

```
2|
3|4.000000
4|5.000000
5|0.000000
```

The following is the resulting output to 2.dat:

```
2|0.000000
3|4.000000
4|5.000000
5|0.000000
```

By using an export schema that has valid values to represent the nullvalue for the second field, this flow has assigned a default value to all of the second field's null values.

Then, if you change the input and schema files as in the following flow:

```
<FLOW>
  <OPERATOR type="import">
    <PROPERTY name="inputfile" value="2.dat" />
    <PROPERTY name="schemafile" value="2.schema" />
    <OUTPUT name="import.v" />
  </OPERATOR>
  <OPERATOR type="export">
    <INPUT name="import.v" />
    <PROPERTY name="outputfile" value="3.dat" />
    <PROPERTY name="schemafile" value="1.schema" />
  </OPERATOR>
</FLOW>
```

Output file 3.dat will look like this:

```
2|
3|4.000000
4|5.000000
5|
```

Note: Null values are always considered the smallest value in a field. That is, for any value n, null < n. In sorted order terms, null values come first (precede other values) in ascending and come last (follow other values) in descending order.

Configure RETL to print schema output in Schema File format

One error prone area in the development of RETL flows is when developers break up and put together flows. The interface between flows is the schema files that tell RETL how to read and write files. Writing schema files manually is tedious and error prone. We highly recommend that developers use the '-sSCHEMAFILE' command line option to speed up development. For more information about this feature, see the section RFX Command line options.

Chapter 6 – RETL Parallel Processing

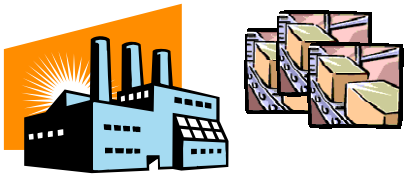
RETL Parallelism Overview

Parallel processing is the ability to break down divisible tasks into granular units of work, and to execute those units of work concurrently and independently of one another. The end goal of parallelism is to achieve speedup by executing tasks concurrently. RETL makes uses of parallelism in several ways:

- Pipeline parallelism
- Framework parallelism / Data Partitioning

Pipeline Parallelism

Pipeline parallelism is readily demonstrated in manufacturing by an assembly line, where each worker can operate on his/her tasks independently of other workers. At each stage in the assembly line, the worker is assigned a task and completes this task on the evolving product. Many tasks can be completed simultaneously (albeit after each worker has product to work on). This is akin to how RETL is pipeline parallel. RETL operators effectively act as workers on an assembly line, whose goal is to produce records of a specific format. Each operator has its own assignment – for example, to sort, join, merge – and it executes these tasks independently of other operator's tasks.

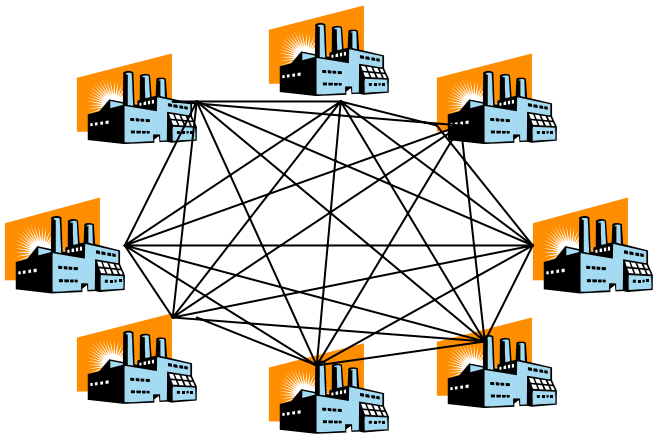


Framework Parallelism

While pipeline parallelism is a great means of increasing throughput, it is only good for doing so until a certain point, where each worker reaches capacity. Worse yet, some workers may be slower at their tasks than others, resulting in bottlenecks in the process. In the manufacturing world, an obvious way of solving this dilemma would be to increase the number of assembly lines. Now each assembly line can work independently of other assembly lines, hopefully uninhibited by bottlenecks. This is akin to RETL's framework-based parallelism. RETL implements framework-based parallelism in a concept called data partitioning. The idea is to split data from the source and pass each 'chunk' of data to separate and concurrent pipelines.

Where can bottlenecks arise? Back to the manufacturing example, there may be only one inlet for transports to bring in raw materials, or one outlet for transports to ship out final product. These both limit the productivity of all assembly lines. The same can be said for RETL's processing. Input data being read from databases or files and output data being written to the same need to be parallelized, in order to fully maximize data partitioning and to be uninhibited by task bottlenecks. Starting with RETL 11.2 and beyond, RETL supports this additional enhancement to parallelize input and output tasks, by multi-threading file and database reads/writes.

RETL uses both framework data partitioning and pipeline parallelism to achieve overall speedups in throughput.



Future versions of RETL will include support for Massively Parallel Processing (MPP), which maximizes processing on multiple interconnected machines. In the manufacturing example, this would be represented by a series of interconnected factories, each working simultaneously to produce a finished product.

RETL Data Partitioning

(This section assumes you have familiarized yourself with the graphical representation of a flow. Graphical representation of flows is detailed in the "**Error! Reference source not found.**" section. If you have not done so, please read this section.)

RETL Data Partitioning, or simply partitioning, is the process RETL uses to increase the number of operators processing records, in an attempt to increase overall throughput. Partitioning consists of splitting up the datasets into subsets and processing each subset with additional pipelines constructed from the originals defined in the flow.

Partitioning does not happen automatically in RETL; the flow must be configured to tell RETL how to split up the data into subsets and partitioning must be enabled.

Enabling RETL Data Partitioning

Partitioning is enabled in one of two ways:

- Specifying a value greater than 1 for the numpartitions attribute of the NODE element in the RETL configuration file. This method enables partitioning for all invocations of RETL that use the configuration file.
- Specifying a value greater than 1 for the –n command line parameter. This method enables partitioning for just the single invocation of RETL and overrides the setting in the configuration file.

Partition Construction

The partitioned data and the duplicated pipelines that process the partitioned data together make up a partition.

Partitions are created by RETL when one or more partitioning operators are included in the flow and are configured for partitioning. A partitioning operator, also known as a partitioner, is an operator that splits the data into subsets. The operators that partition data are IMPORT, DBREAD, HASH, SPLITTER, and ORAREAD.

The partition pipelines are duplicated automatically by RETL from the operators specified in the flow. For example, if the original flow has a BINOP operator, multiple copies of the BINOP operator are created, one for each partition.

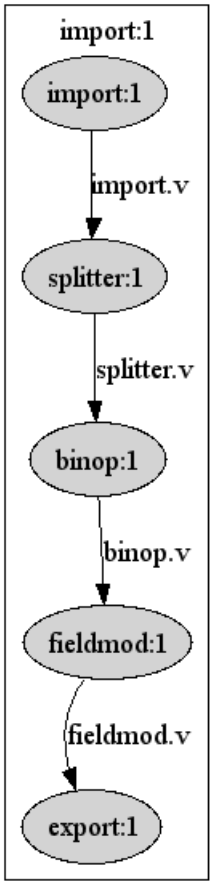
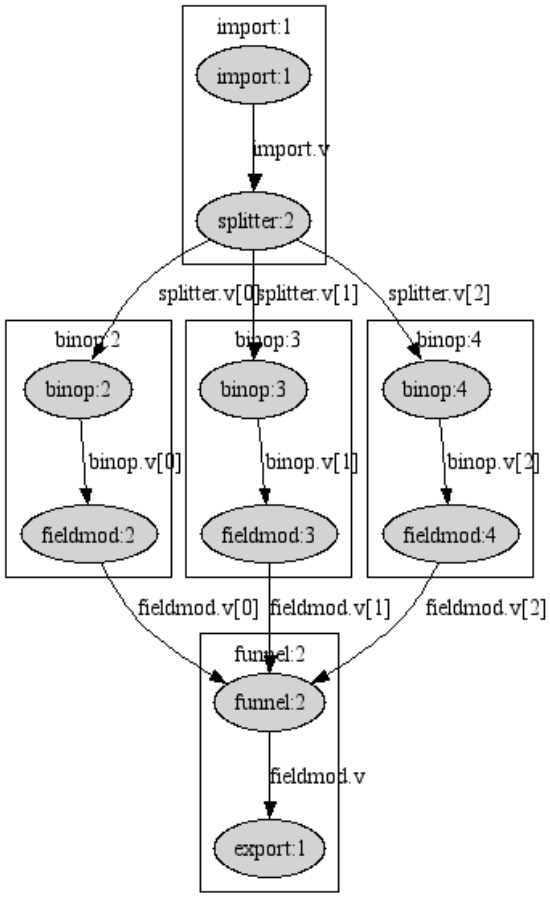
A partition pipeline ends when an operator is encountered that does not support partitioning or does not support the type of partitioning performed by the partitioner. If necessary, RETL inserts a funnel in order to “unpartition” the data subsets back into a single dataset.

For a simple example, suppose you have a text file that contains records and you need to add two fields and write the sum to another file. If you wanted to use data partitioning, the flow would look something like this:

```
<FLOW name=”sumvalues”>
   <!-- Read in the values. -->
   <OPERATOR type=”import”>
      <PROPERTY name=”inputfile” value=”values.txt”/>
      <PROPERTY name=”schemafile” value=”values-schema.xml”/>
      <OUTPUT name=”import.v”/>
   </OPERATOR>
   <!-- Partition into 3 partitions. -->
   <OPERATOR type=”splitter”>
      <INPUT name=”import.v”/>
      <PROPERTY name=”numpartitions” value=”3”/>
      <OUTPUT name=”splitter.v”/>
   </OPERATOR>
   <!-- Add the values. -->
   <OPERATOR type=”binop”>
```

```
        <INPUT name=”splitter.v”/>
        <PROPERTY name=”left” value=”VALUE1”/>
        <PROPERTY name=”right” value=”VALUE2”/>
        <PROPERTY name=”operator” value=”+”/>
        <PROPERTY name=”dest” value=”SUM”/>
        <OUTPUT name=”binop.v”/>
    </OPERATOR>
    <!—Keep only the SUM field. -->
    <OPERATOR type=”fieldmod”>
        <INPUT name=”binop.v”/>
        <PROPERTY name=”keep” value=”SUM”/>
        <OUTPUT name=”fieldmod.v”/>
    </OPERATOR>
    <!-- Write the sum to sum.txt. -->
    <OPERATOR type=”export”>
        <INPUT name=”fieldmod.v”/>
        <PROPERTY name=”outputfile” value=”sum.txt”/>
        <PROPERTY name=”schemafile” value=”sum-schema.xml”/>
    </OPERATOR>
</FLOW>
```

The graphs of the unpartitioned and partitioned flow look like this:

Unpartitioned	Partitioned
	

In this example, the SPLITTER operator is the partitioner. It distributes the records read by the IMPORT operator across three partitions. The BINOP and FIELDMOD operators are duplicated across all three partitions by RETL. Finally, a funnel is inserted before the EXPORT operator to gather the partitioned records back into one dataset.

Partitioning Types

The partitioners provide two different types of partitioning: keyed and non-keyed.

Keyed Partitioning

Keyed partitioning provides two guarantees:

- Records with the same key are processed in the same partition.
- Order of records is retained within the data subset. That is, any two records in a partition are in the same order as they were in the original dataset.

Keyed partitioning is required to partition operators that require sorted input, such as groupby and cliprows.

Non-Keyed Partitioning

Unlike keyed partitioning, non-keyed partitioning makes no guarantee about which partition will process a particular record. Thus, it is inappropriate for any operator that requires sorted input.

Partitioners

There are seven partitioners:

- HASH
- IMPORT
- SPLITTER
- DBREAD
- ORAREAD
- GENERATOR

The number of partitions created by a partitioner is specified in the ‘numpartitions’ property. The ‘numpartitions’ property is common to all of the partitioners, although some of the partitioners require specification of other properties.

Note that data partitioning must be enabled for the ‘numpartitions’ property to have any affect. If data partitioning is not enabled, the property is ignored.

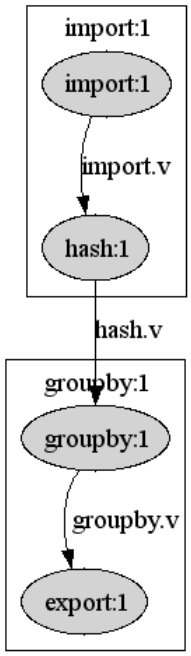
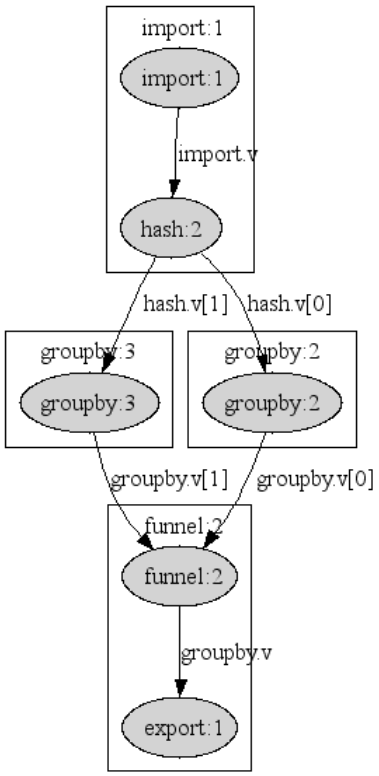
HASH

The HASH operator is the only partitioner that performs keyed partitioning. The key fields are specified in the 'key' property. As each record is processed, the hash code of the key fields is calculated and records with the same hash code are sent to the same partition.

If the 'numpartitions' property is not specified in the HASH operator, RETL uses the value from the –n command line option, if specified. If the –n command line option is not specified, the value from the configuration file is used.

In the example below, the HASH operator splits the data into two partitions. RETL automatically duplicates the groupby operator and inserts the funnel. (The EXPORT operator in this example is not configured to be partitionable.)

```
<FLOW name="hash-example">
   <OPERATOR type="import">
      <PROPERTY name="inputfile" value="input.txt"/>
      <PROPERTY name="schemafile" value="in-schema.xml"/>
      <OUTPUT name="import.v"/>
   </OPERATOR>
   <OPERATOR type="hash">
      <INPUT name="import.v"/>
      <PROPERTY name="numpartitions" value="2">
      <PROPERTY name="key" value="KEY_FIELD">
      <OUTPUT name="hash.v"/>
   </OPERATOR>
   <OPERATOR type="groupby">
      <INPUT name="hash.v"/>
      <PROPERTY name="key" value="KEY_FIELD">
      <PROPERTY name="reduce" value="AMOUNT">
      <PROPERTY name="sum" value="TOTAL_AMOUNT">
      <OUTPUT name="groupby.v"/>
   </OPERATOR>
   <OPERATOR type="export">
      <INPUT name="groupby.v"/>
      <PROPERTY name="outputfile" value="output.txt">
      <PROPERTY name="schemafile" value="out-schema.xml">
   </OPERATOR>
```

Unpartitioned	Partitioned
	

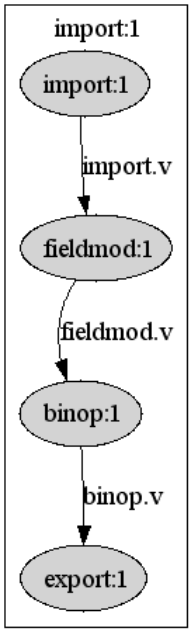
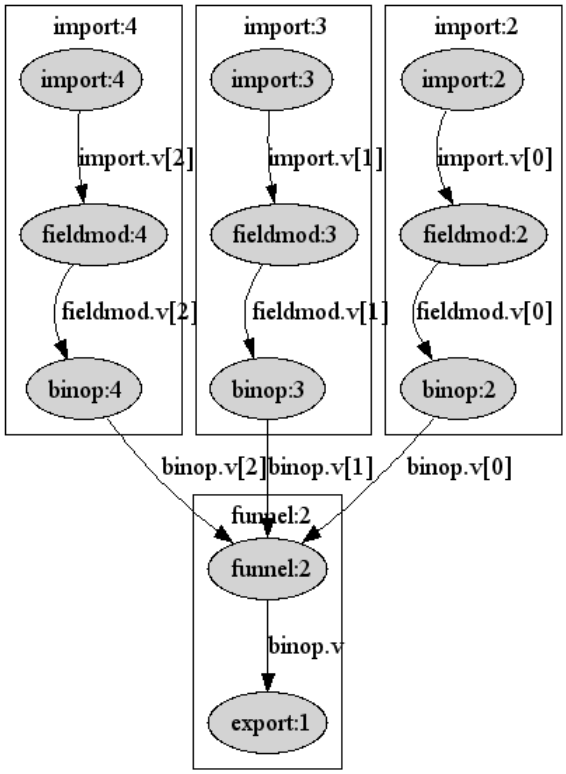
IMPORT

Import partitioning performs non-keyed partitioning. If one input file is specified, then roughly equal portions of the file are handled by each partition. If more than one file is specified, the number of partitions must equal the number of files and each file is handled by one partition.

To enable IMPORT partitioning, the 'numpartitions' property must be specified. Because the main purpose of the IMPORT operator is not data partitioning, the default number of partitions does not come from the configuration file or from the command line.

In the example below, three IMPORT operators are created by RETL to read records. RETL automatically duplicates the BINOP and FIELDMOD operators and inserts the FUNNEL. (The EXPORT operator in this example is not configured to be partitionable.)

```
<FLOW name=”import-example”>
   <OPERATOR type=”import”>
      <PROPERTY name=”numpartitions” value=”3”>
      <PROPERTY name=”inputfile” value=”input.txt”/>
      <PROPERTY name=”schemafile” value=”in-schema.xml”/>
      <OUTPUT name=”import.v”/>
   </OPERATOR>
   <OPERATOR type=”fieldmod”>
      <INPUT name=”import.v”/>
      <PROPERTY name=”drop” value=”UNNEEDED_FIELD”>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”binop”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name="left" value="VALUE1"/>
      <PROPERTY name="operator" value="+"/>
      <PROPERTY name="right" value="VALUE2"/>
      <PROPERTY name="dest" value="SUM"/>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”export”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name=”outputfile” value=”output.txt”>
      <PROPERTY name=”schemafile” value=”out-schema.xml”>
   </OPERATOR>
```

Unpartitioned	Partitioned
	

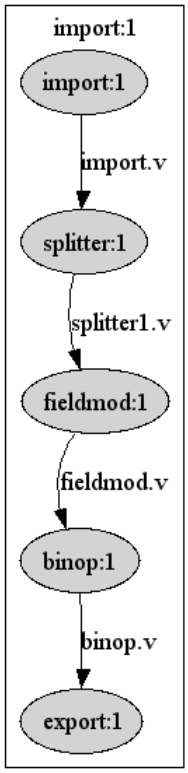
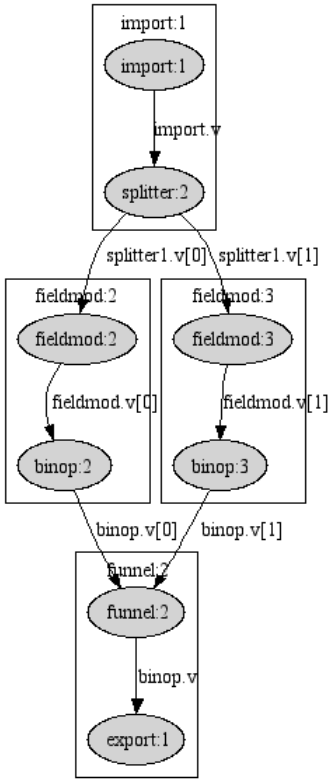
SPLITTER

The SPLITTER operator performs non-keyed partitioning by sending records to the different partitions in a round-robin fashion. Although SPLITTER will be supported in future versions of RETL, it is not guaranteed that it will use the round-robin algorithm. Flows should not rely on SPLITTER partitioning records in a round robin manner.

If the 'numpartitions' property is not specified in the SPLITTER operator, RETL uses the value from the –n command line option. If the –n command line option is not specified, the value from the configuration file is used.

In the example below, the SPLITTER operator splits the data into two partitions. RETL automatically duplicates the FIELDMOD and BINOP operators and inserts the FUNNEL. (The EXPORT operator in this example is not configured to be partitionable.)

```
<FLOW name=”splitter-example”>
   <OPERATOR type=”import”>
      <PROPERTY name=”inputfile” value=”input.txt”/>
      <PROPERTY name=”schemafile” value=”in-schema.xml”/>
      <OUTPUT name=”import.v”/>
   </OPERATOR>
   <OPERATOR type=”splitter”>
      <INPUT name=”import.v”/>
      <PROPERTY name=”numpartitions” value=”3”>
      <OUTPUT name=”splitter.v”/>
   </OPERATOR>
   <OPERATOR type=”fieldmod”>
      <INPUT name=”splitter.v”/>
      <PROPERTY name=”drop” value=”UNNEEDED_FIELD”>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”binop”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name="left" value="VALUE1"/>
      <PROPERTY name="operator" value="+"/>
      <PROPERTY name="right" value="VALUE2"/>
      <PROPERTY name="dest" value="SUM"/>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”export”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name=”outputfile” value=”output.txt”>
      <PROPERTY name=”schemafile” value=”out-schema.xml”>
   </OPERATOR>
```

Unpartitioned	Partitioned
	

DBREAD, ORAREAD

The database read partitioners implement non-keyed partitioning by allowing specification of multiple queries. Each query result set provides the data for one partition.

The 'numpartitions' property must be set to enable partitioning and the value must equal the number of queries. If the 'numpartitions' property is not specified, then the result sets will be funneled together into one dataset instead of feeding into multiple partitions.

For example, suppose you want to read all of the records from the table 'sourcetable'. The non-partitioned dbread operator would have a query property like this:

```
<PROPERTY name="query">
   <![CDATA[
   select * from sourcetable
   ]]>
</PROPERTY>
```

If you wanted to partition the database read into three, the flow would have something like this:

```
<PROPERTY name="numpartitions" value=”3”>
<PROPERTY name="query">
   <![CDATA[
   select * from sourcetable where keyfield between 0 and 100000
   ]]>
</PROPERTY>
<PROPERTY name="query">
   <![CDATA[
   select * from sourcetable where keyfield between 100001 and
200000
   ]]>
</PROPERTY>
<PROPERTY name="query">
   <![CDATA[
   select * from sourcetable where keyfield > 200000
   ]]>
</PROPERTY>
```

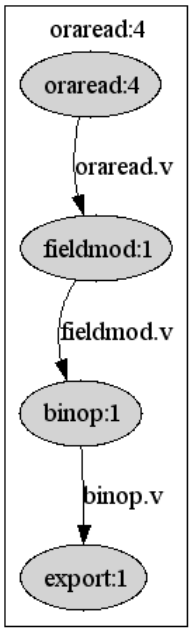
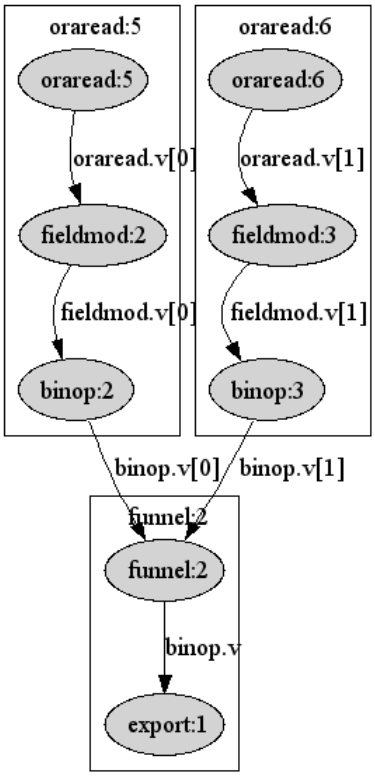
It is important to perform performance analysis on each of the queries. Adding a where clause to the unpartitioned SQL query may change the execution plan and cause it to run many times slower, eating up any time saved by partitioning.

Data partitioning with ORAREAD works particularly well with Oracle partitioned tables when each query reads from a different table partition. Consult the Oracle documentation for information on partitioned tables.

The stored procedure specified by the ‘sp_prequery’ property is run before any of the queries are executed. Likewise, the stored procedure specified by the ‘sp_postquery’ property is run after the last query completes.

In the example below, two queries are specified so RETL creates two ORAREAD operators to read records from two different tables in an Oracle database. RETL automatically duplicates the fieldmod and binop operators and inserts the funnel.

```
<FLOW name=”oraread-example”>
   <OPERATOR type=”oraread”>
      <!-- connection properties not shown -->
      <PROPERTY name=”numpartitions” value=”2”>
                     <PROPERTY name="query">
         <![CDATA[
         select * from table_a
         ]]>
                     </PROPERTY>
                     <PROPERTY name="query">
         <![CDATA[
         select * from table_b
         ]]>
      </PROPERTY>
      <OUTPUT name=”oraread.v”/>
   </OPERATOR>
   <OPERATOR type=”fieldmod”>
      <INPUT name=”splitter.v”/>
      <PROPERTY name=”drop” value=”UNNEEDED_FIELD”>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”binop”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name="left" value="VALUE1"/>
      <PROPERTY name="operator" value="+"/>
      <PROPERTY name="right" value="VALUE2"/>
      <PROPERTY name="dest" value="SUM"/>
      <OUTPUT name=”fieldmod.v”/>
   </OPERATOR>
   <OPERATOR type=”export”>
      <INPUT name=”fieldmod.v”/>
      <PROPERTY name=”outputfile” value=”output.txt”>
      <PROPERTY name=”schemafile” value=”out-schema.xml”>
   </OPERATOR>
```

Unpartitioned	Partitioned
	

Operators and Partitioning Types

When a flow uses an operator other than a HASH to partition the data, RETL will end the partition before any operator that requires sorted input, such as cliprows or groupby. This is because only the HASH operator guarantees that records with the same key are handled in the correct order by the same partition.

RETL ends the partition by inserting a funnel. Because a funnel does not put records back into a sorted order, the operator that requires sorted input will display a warning message about not having sorted input and the results will be incorrect.

There are several ways to fix this problem. The best solution depends on the specifics of the flow. In general:

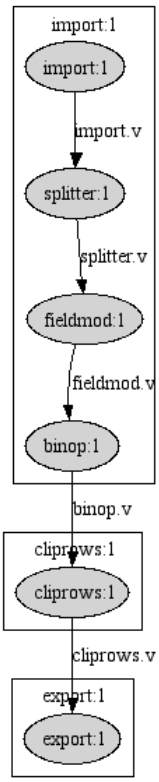
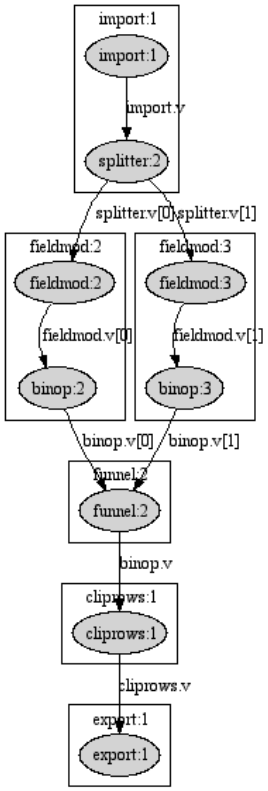
- If a SPLITTER operator is used to partition the data, change it to a HASH.
- If an IMPORT operator is used to partition the data, change the IMPORT to be unpartitioned and insert a HASH operator after the IMPORT.
- If a database read operator is used to partition the data, replace the partitioned database read operator with multiple database read operators and insert a sortfunnel and a HASH or combine all of the queries into one and insert a HASH after the dbread. In either case make sure that all queries return sorted records.

- If the operator requiring sorted order is a join, consider using a LOOKUP operator instead. The LOOKUP operator does not require sorted input. Keep in mind that the LOOKUP operator requires more memory than a join and is only appropriate for small lookup table sizes.
- If the operator requiring sorted order is CHANGECAPTURE, consider using a CHANGECAPTURELOOKUP operator instead. The CHANGECAPTURELOOKUP operator does not require sorted input. Keep in mind that the CHANGECAPTURELOOKUP operator requires more memory than a CHANGECAPTURE and is only appropriate for small lookup table sizes.

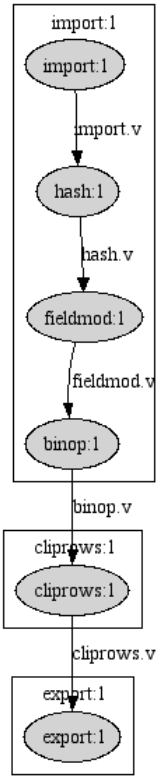
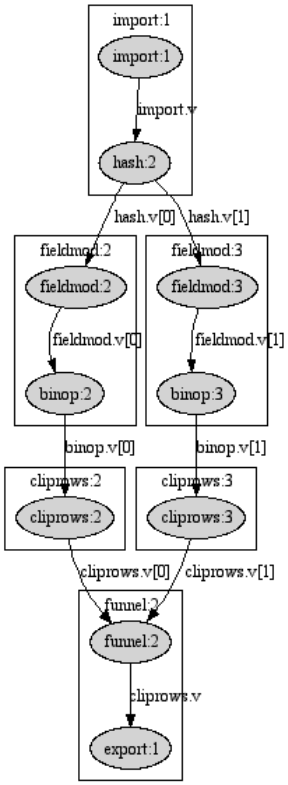
As an example, suppose a SPLITTER is used to partition imported data and a CLIPROWS operator is encountered. RETL inserts a FUNNEL before the CLIPROWS and displays the following warning message:

[cliprows:1]: Warning: Input to cliprows:1 does not seem to be sorted! Data should be sorted according to the proper keys or you may get unexpected results!

The flow graph looks like this.

Unpartitioned	Partitioned
	

To correct this flow, change the SPLITTER to a HASH:

Unpartitioned	Partitioned
	

The following operators require sorted input and support only keyed partitioning:

- CHANGECAPTURE
- CLIPROWS
- COMPARE
- DIFF
- FULLOUTERJOIN
- GROUPBY
- INNERJOIN
- LEFTOUTERJOIN
- MERGE (MERGE does not require sorted input, but it does require that records be processed in the same order, so HASH partitioning is required)
- REMOVEDUP
- RIGHTOUTERJOIN

‘parallel’ Property

To be included in a partition, some operators require that the ‘parallel’ property be set to true. The ‘parallel’ property is typically required of an operator if

- Partitioning the operator may sometimes produce incorrect results and so the default is not to partition the operator, or
- Partitioning the operator may improve performance on some platforms and not others. For example, using a partitioned EXPORT may not always improve performance. or
- The operator did not partition in earlier versions of RETL.

The following operators require the “parallel” property to be set to “true” to allow the operator to be portioned.

- CLIPROWS
- DBWRITE
- DEBUG
- EXPORT
- GENERATOR
- GROUPBY
- ORAWRITE

If the ‘parallel’ property is not specified, RETL will end the partition before the operator by inserting a funnel.

Partitioned EXPORT

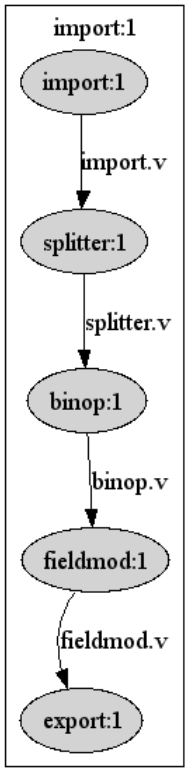
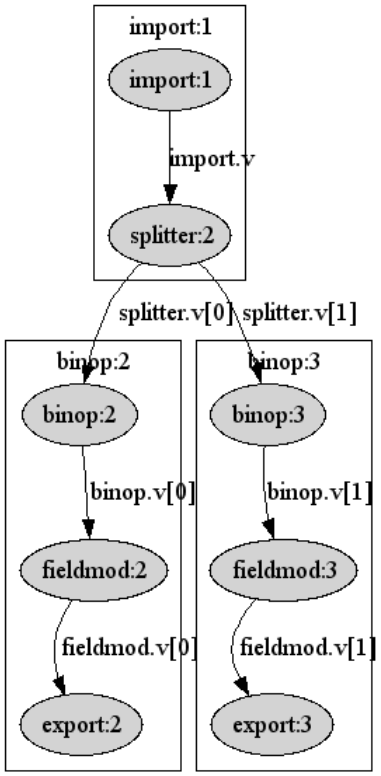
When the ‘parallel’ property of an EXPORT operator is set to true, RETL will include the EXPORT in the partition. The output from each partition is written to separate temporary files. When all partitions have completed processing, the temporary files are concatenated into the specified output file.

Note: Partitioning the EXPORT operator may or may not improve performance and should be tested before implementation.

The temporary files are created in the TEMPDIR directories specified in the configuration files. Best results are obtained when there is a temporary directory for each partition and each directory is on a separate disk controller.

The following graph is for a flow identical to the simple flow described at the beginning of this section except that a partitioned EXPORT is specified. Note that two EXPORT operators are created but that there is only one file exported.

```
<FLOW>
   <!-- Other operators same as in original example. -->
   <!-- Write the sum to sum.txt. -->
   <OPERATOR type="export">
      <INPUT name="fieldmod.v"/>
      <PROPERTY name="parallel" value="true"/>
      <PROPERTY name="outputfile" value="sum.txt"/>
      <PROPERTY name="schemafile" value="sum-schema.xml"/>
   </OPERATOR>
</FLOW>
```

Unpartitioned	Partitioned
	

Partitioned GENERATOR

The GENERATOR operator utilizes two partitioning-related attributes when generating a sequence field within a partition. Generally, these two attributes are used together to allow the sequence field to contain unique values across all partitions.

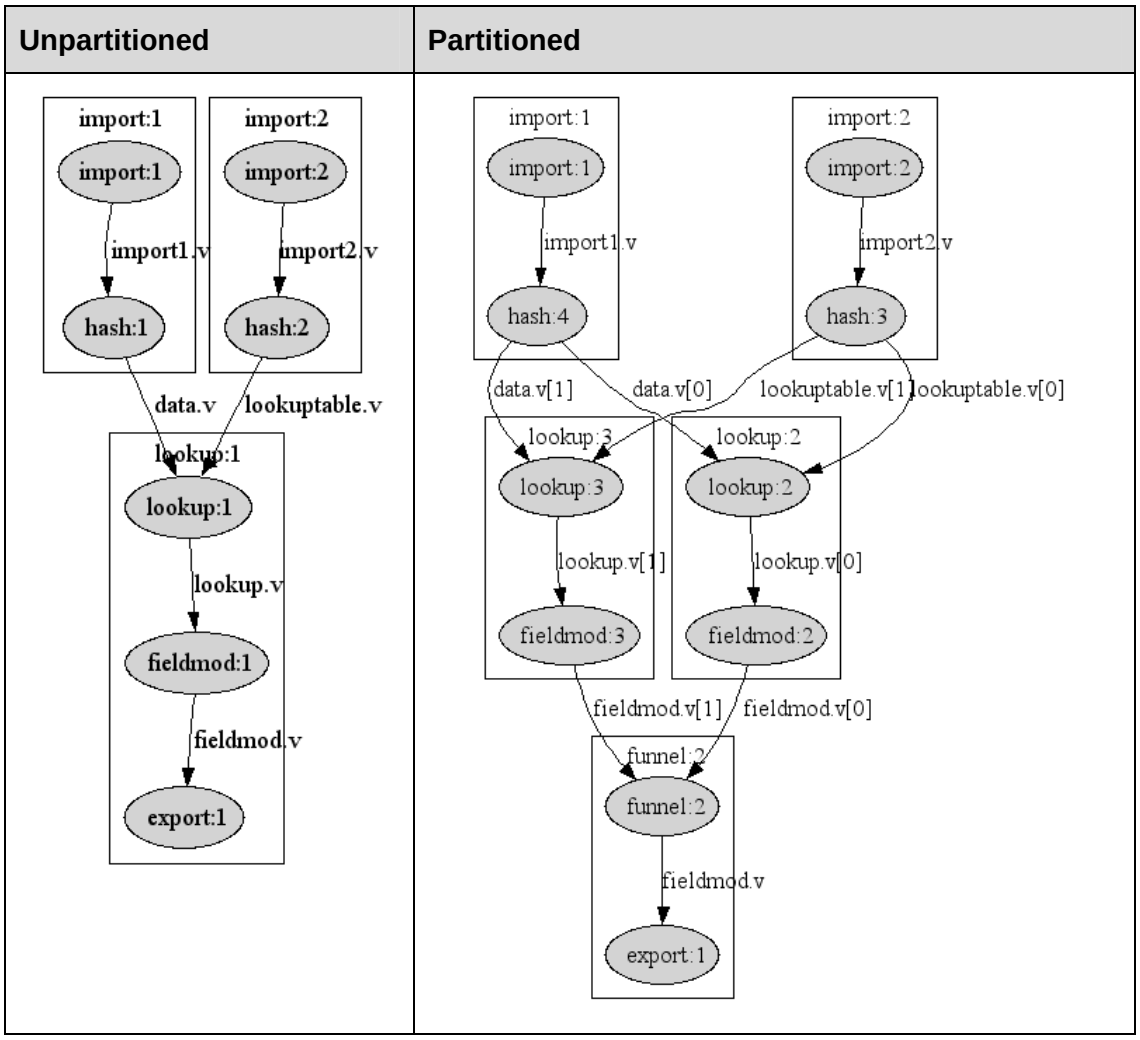
- When the 'partnum_offset' property is set to "true", the initial value is incremented by the zero-based partition number.
- When the 'partnum_incr' property is set to "true", the increment value is multiplied by the number of partitions.

For example, if there are three partitions and the sequence field's initial value is specified as 1 and the increment value is specified as 1, then the first generator will generate values 1, 4, 7, …; the second 2, 5, 8, …; and the third 3, 6, 9, …

A handy trick for debugging purposes is to set 'init' to 0, 'incr' to 0, 'partnum_offset' to true and 'partnum_incr' to false. The generated field indicates the partition that processed the record.

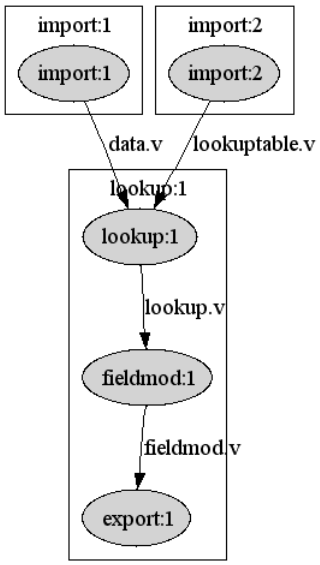
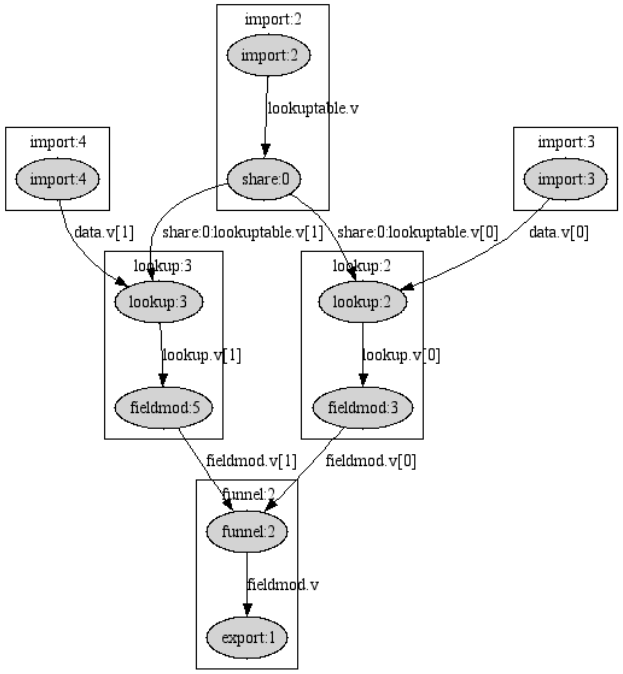
Partitioned LOOKUP and CHANGECAPTURELOOKUP

When both the source and lookup datasets of a LOOKUP or CHANGECAPTURELOOKUP operator are partitioned by a HASH operator, each partitioned LOOKUP or CHANGECAPTURELOOKUP operator will use a subset of the lookup dataset and a subset of the source dataset.



However, if the source dataset is partitioned by any other partitioner or the lookup table is not partitioned at all, then the LOOKUP or CHANGECAPTURELOOKUP operators in each partition will share the same lookup table. A SHARE operator is inserted into the flow to accomplish the sharing and can be ignored. (The SHARE operator is a special operator used for LOOKUP partitioning.)

Although from the graph it looks like the inserted HASH operator is partitioning the data, it actually is not. All of the lookup table records are sent to the first partition's LOOKUP or CHANGECAPTURELOOKUP operator, which loads the shared lookup table.

Unpartitioned	Partitioned
	

Partitioned ORAWRITE

The stored procedure specified in the ‘preload’ property is run before any of the partitioned orawrites has started loading data. Likewise, the stored procedure specified in the ‘postload’ property is run after the last partition completes.

Flows with Multiple Partitioners

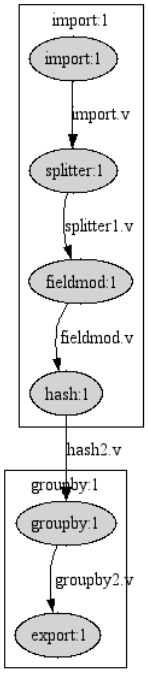
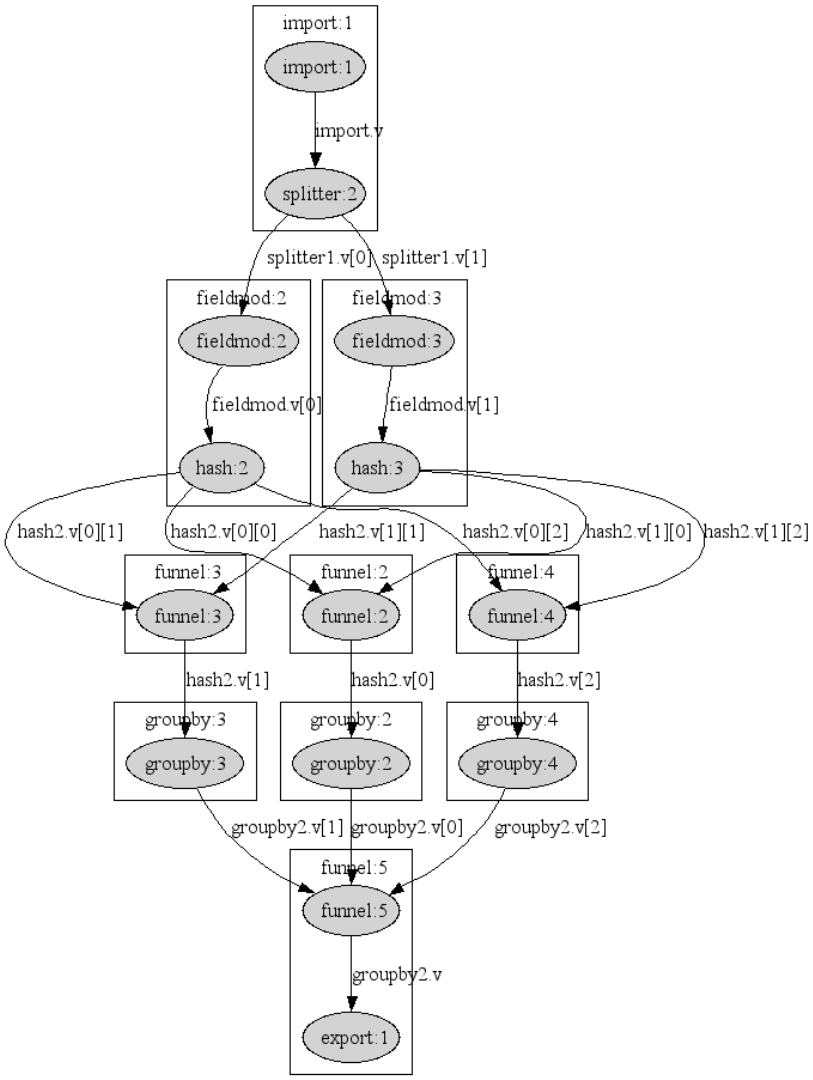
It is possible to use more than one partitioner in a flow. For example, both an IMPORT and a database read may be partitioned.

HASH within a partition

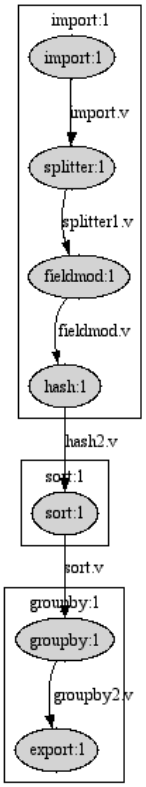
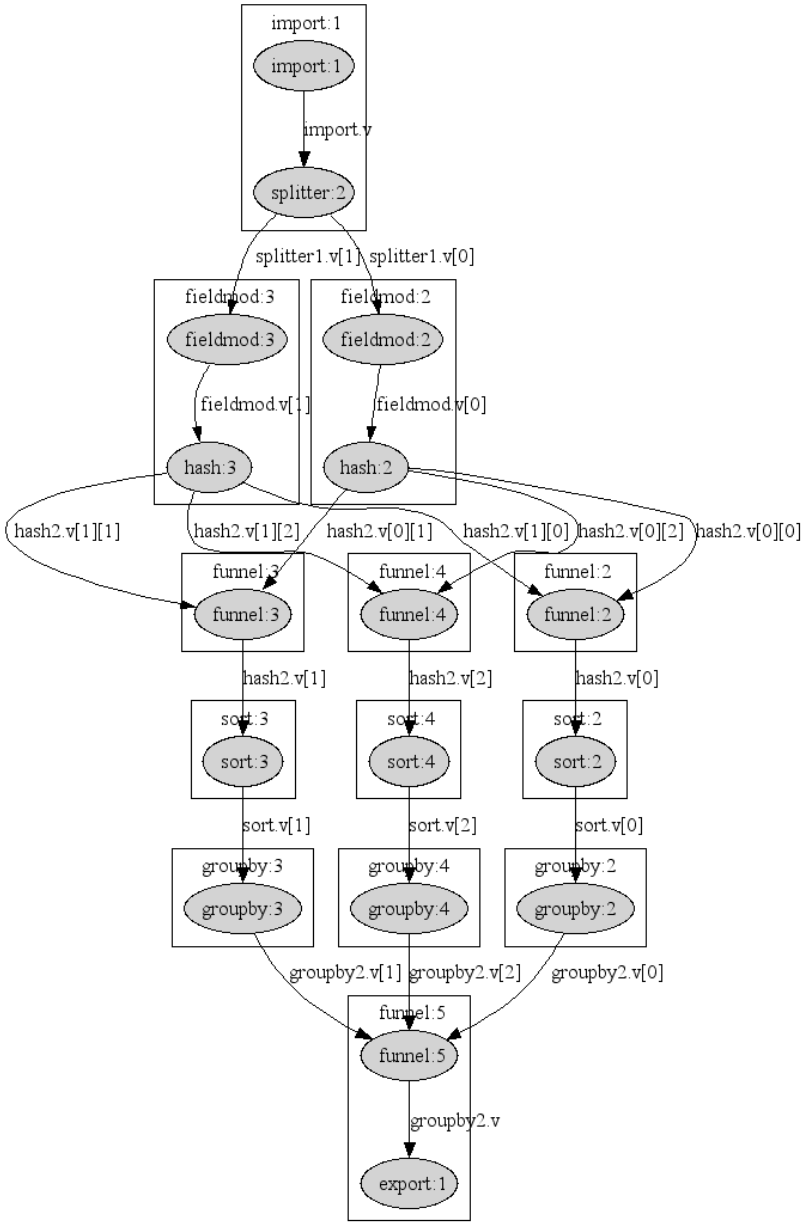
If a HASH is found within a partition, a funnel is automatically inserted after the HASH to gather records with the same keys into the same data subset.

Note: This will not leave the data sorted by the new key fields. You must enter in the appropriate sort operator if the data needs to be sorted.

The following graph shows a flow that will produce invalid results. The funnels inserted after the HASH operators lose the sorted order required by groupby.

Unpartitioned	Partitioned
	

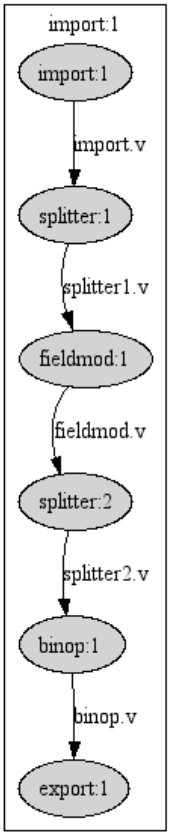
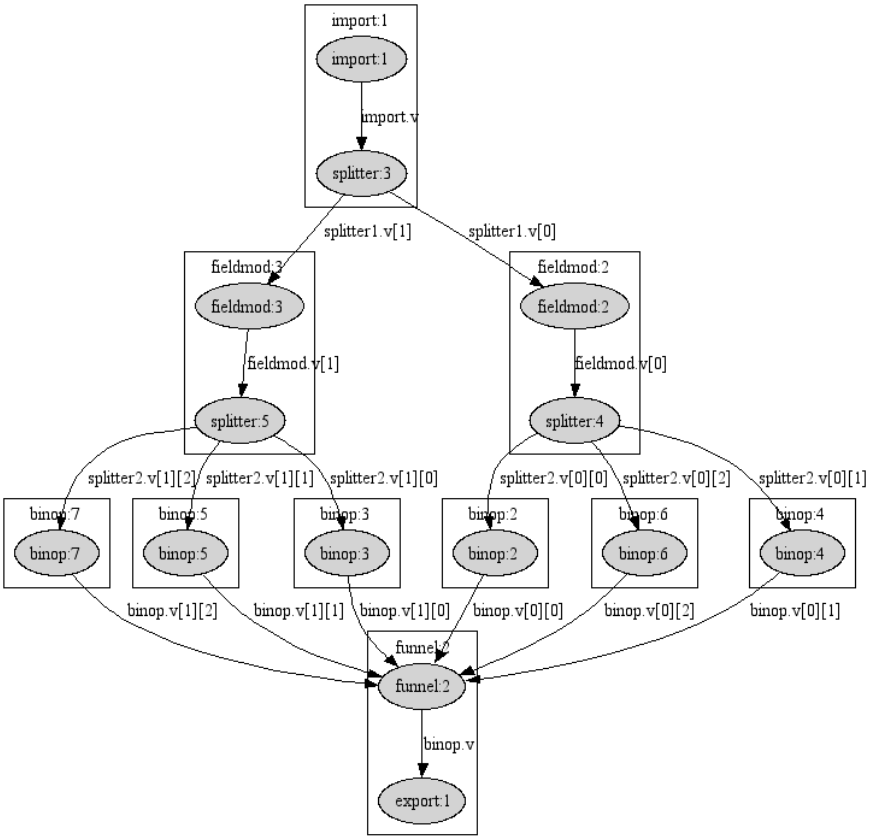
The following graph shows a corrected version of the above flow. Note that there is a sort in the unpartitioned flow immediately after the HASH.

Unpartitioned	Partitioned
	

SPLITTER within a SPLITTER partition

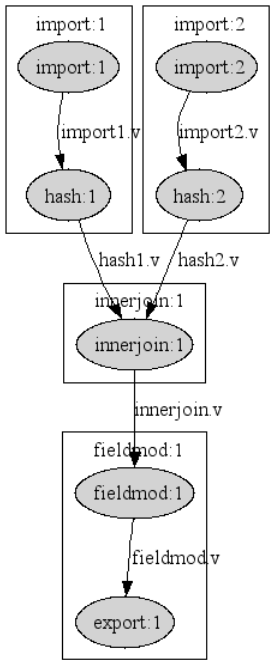
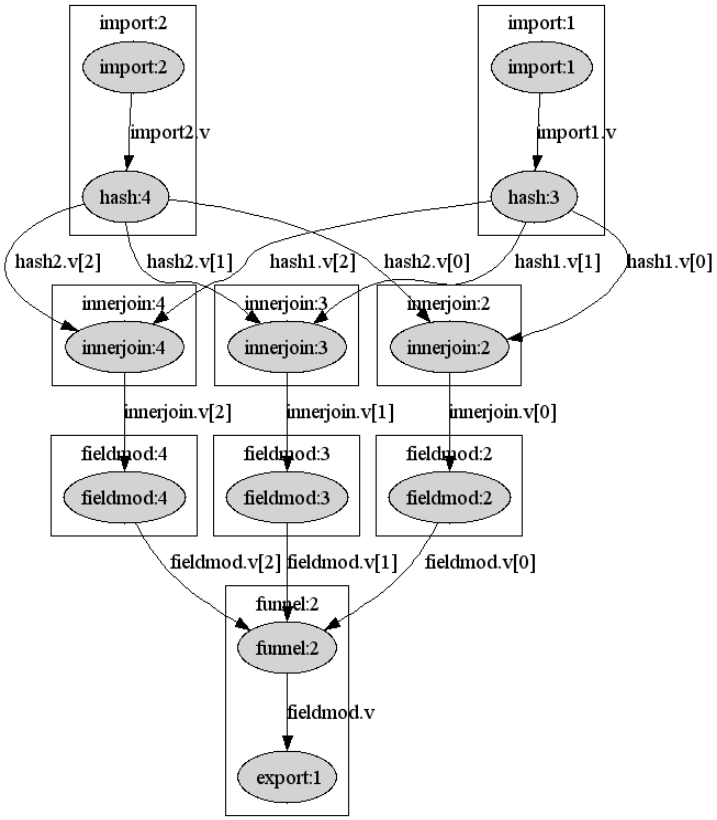
If a SPLITTER is found within a partition, the partitioned data is re-partitioned. For example, if the first partitioner specifies two partitions and the SPLITTER specifies three partitions, there will be six partitions after the SPLITTER.

The following graph shows a flow with two partitions started by a SPLITTER expanding to six partitions.

Unpartitioned	Partitioned
	

Two HASH operators joined

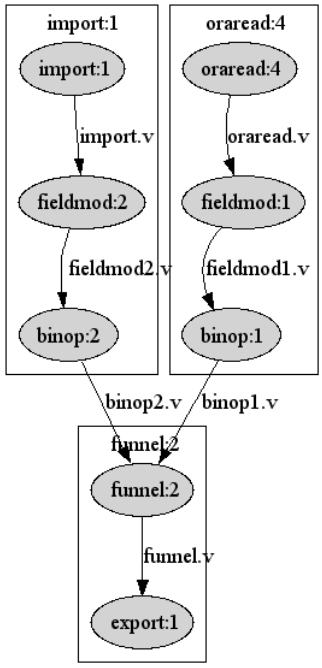
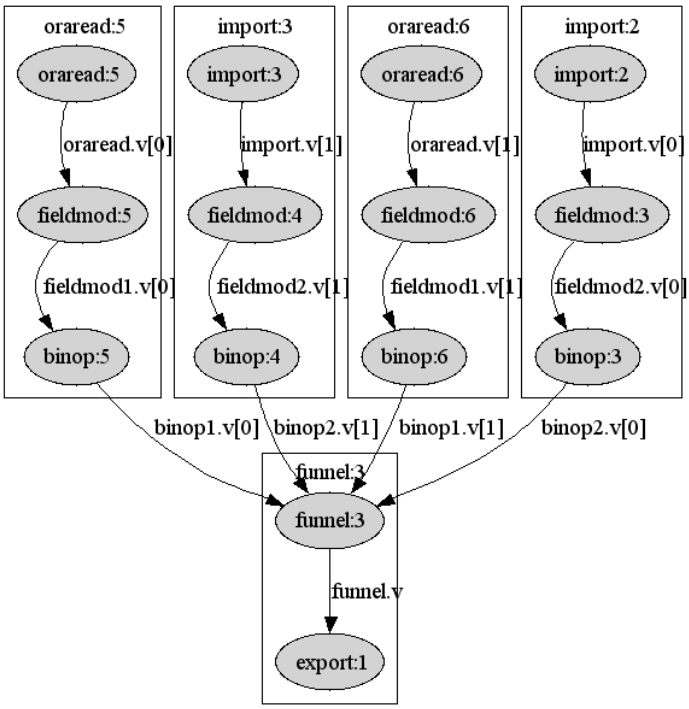
The join operators (innerjoin, leftouterjoin, etc.) support keyed partitioning. RETL will ensure that records with identical keys are sent to the same partitioned join operator. If the HASH operators specify a different number of partitions, then the smaller numpartitions will be used for both HASH operators.

Unpartitioned	Partitioned
	

Funneled Partitions

The data from partitions started by multiple partitioners can be funneled using a single funnel operator. As of version 11.2, RETL requires that both partitioners have the same number of partitions. This restriction will be lifted in a later release.

In the following example, an IMPORT creates two partitions and an ORAREAD creates two partitions. The four partitions are funneled together using the same funnel.

Unpartitioned	Partitioned
	

Data Partitioning Guidelines

Keep these guidelines in mind when designing your flow:

- Performance improvement is more likely when the inputs to the flow are partitioned than when a HASH or a SPLITTER is used to partition. The input to HASH or SPLITTER is serial, whereas IMPORT and database reads are parallel.
- Because of the expense in funneling, performance improvement is more likely when parallel database writes or EXPORT are performed than when the data is unpartitioned using a funnel or sortfunnel before the output.
- More partitions are not always better. More partitions require more threads and there is a limit to the optimum number of threads used to process the flow. This limit depends on the hardware and the specifics of the flow. Thus, finding the optimum number of partitions can be a difficult task.

Operator Configuration for Data Partitioning

The following table shows each operator that requires modifications to its properties to allow it to create or process partitioned data. If an operator is not listed, it either supports partitioned data without any configuration or it does not support data partitioning at all.

Operator	Property	Property Value
CLIPROWS	“parallel”	“true”
DBREAD	“numpartitions”	greater than “1”
	“query”	Specify a query for each partition.
DBWRITE	“parallel”	“true”
DEBUG	“parallel”	“true”
EXPORT	“parallel”	“true”
GENERATOR (generating fields for pre-exiting records)	“parallel”	“true”
GENERATOR (generating new records)	“numpartitions”	greater than “1”
GROUPBY	“parallel”	“true”
IMPORT	“numpartitions”	greater than “1”
	“inputfile”	Specify either one input file or “numpartitions” input files.
ORAREAD	“numpartitions”	greater than “1”
	“query”	Specify a query for each partition.
ORAWRITE	“parallel”	“true”
	“mode”	“append” Note that changing the mode property to “append” from “truncate” may require changes to the batch job because the existing database rows will no longer be deleted by RETL.

Additionally, some operators only work correctly when keyed (hash) partitioning is used. For a description of keyed partitioning, see "Partitioning Types". The following operators require keyed partitioning:

- CHANGECAPTURE
- CLIPROWS
- COMPARE
- DIFF
- FULLOUTERJOIN
- GROUPBY
- INNERJOIN
- LEFTOUTERJOIN
- MERGE
- REMOVEDUP
- RIGHTOUTERJOIN

A Final Word on Data Partitioning

Data partitioning is not a silver bullet that can magically correct performance problems in a flow. Just as performance tuning cannot significantly improve a poorly designed application, data partitioning cannot significantly improve a poorly designed flow.

Nor is partitioning guaranteed to improve performance. In fact, blindly tossing in partitioners will likely degrade performance because of the overhead of the partitioning, funneling, and additional threads.

If your flow is experiencing performance problems, do not look to partitioning first. Perform a thorough analysis of your flow to determine where the performance problem is taking place and take appropriate steps. (See the RETL Performance Tuning Guide for more information.)

Chapter 7 – Database Operators

Previous version of RETL supported Oracle, Teradata, and DB2 databases. Support for Teradata and DB2 has been dropped starting in RETL 11.3.

ORAREAD

ORAREAD performs a read from Oracle databases. RETL 11.x uses JDBC technology to read data out of Oracle databases.

ORAWRITE

ORAWRITE performs a load to Oracle databases. RETL 11.x uses SQL*Loader (sqlldr) to load the records.

Note: The ORAWRITE operator may implicitly drop fields when writing to the database, if fields in the incoming record schema don't exist in the database table. Also, the incoming record schema will determine what is set for nullabilties and maxlengths of fields as well. If there are any inconsistencies between the incoming record and the format of the database table, RETL may not flag this as a problem; instead it will rely on SQL*Loader to throw errors or reject inconsistent records. This will only happen when appending to an existing table, and doesn't affect database writes when specifically creating/recreating a table.

UPDATE

The UPDATE operator updates records in an Oracle database. Fields in the records are used to provide updated values and also select records for updating.

DELETE

The DELETE operator deletes records from an Oracle database table. Fields in the records are used to select records for deleting.

INSERT

The INSERT operator inserts records into an Oracle database table. This is the same functionality that ORAWRITE performs, only using a different technology (JDBC instead of SQL*Loader). ORAWRITE is preferred from a performance standpoint.

PREPAREDSTATEMENT

The PREPAREDSTATEMENT allows you to execute SQL commands using values from the current record as input into the SQL command.

The PREPAREDSTATEMENT can perform inserts, deletes, and updates like the INSERT, UPDATE, and DELETE operators, only there is much more flexibility:

- Record field names can be used more than once
- Complex WHERE clause logic can be used
- SQL calculations can be used

The "statement" property specifies the SQL to execute. The SQL specified can contain question marks (“?”) that act as placeholders for values from the current record. The “fields” property contains the names of the fields that will be used to provide the values. The fields must be specified in the same order as the question mark placeholders, but the same field can be specified multiple times.

Note: One important restriction is there must be a database column with the same name for any field specified in the “fields” property.

All of the following are valid values for the “statement” property:

- `UPDATE EMPLOYEES SET SALARY = ? WHERE EMPLOYEE_ID = ? AND JOB_TITLE = ‘SOFTWARE ENGINEER’`

  This statement applies the new salary only if the employee is a software engineer. The JOB_TITLE field does not need to be in the RETL dataset.

- `UPDATE EMPLOYEES SET SALARY = SALARY * 1.10 WHERE EMPLOYEE_ID = ?`

  This statement gives a 10% raise to an employee. The 10% calculation is performed by the database, not RETL.

- `DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = ? AND LOCKED = ‘FALSE’`

  This statement deletes a record from the EMPLOYEES table provided its LOCKED column is set to “FALSE”.

- `INSERT INTO EMPLOYEES (LAST_NAME, FIRST_NAME, SALARY, JOB_TITLE) VALUES (?, ?, 0, ‘UNKNOWN’)`

  This statement inserts a record into the employees table with a salary of 0 and a title of “UNKNOWN”. The input dataset does not need to have fields named SALARY and JOB_TITLE.

Database Operators XML Specification Table

The following table outlines database read/write operators.

Note: Throughout the RETL Programmer's Guide, 'schema' can apply to the data structure that the RETL applies to a dataset (such as a 'schema file'), or 'schema' can apply to the database owner of a table (the 'schemaowner' property of a database write operator).

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
ORAREAD				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	connectstring	username/password	Database login name and password separated by a forward slash - "/".
	PROPERTY	maxdescriptors	number	Maximum number of columns allowed to be returned from the database. This is an optional property. The default value is 70.
	PROPERTY	query		The SQL query (select statement) used to extract the data from the database. SQL syntax must be executable for the database environment.  **Note:** This property must be enclosed in a CDATA tag in order to allow all SQL query types. See examples for more information. If any kind of function is applied to a column in the select statement, the column will be extracted as a DFLOAT even if the column in the database is defined as an int or string type. You may have to perform a CONVERT to change datatypes in order to compensate for this behavior.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	datetotimestamp	"true" or "false"	This is an optional property. When set to "true", ORAREAD returns DATE columns with the time included. When false, the time is not included. The default for this property is "false".
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified.  **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf.  **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	sp_prequery	string	This is a special query that allows the flow developer to specify a stored procedure to be run prior to the main query that ORAREAD executes. The current implementation does not allow variables to be specified nor does it allow data to be returned.
	PROPERTY	sp_postquery	string	This is a special query that allows the flow developer to specify a stored procedure to be run after the main query that ORAREAD executes. The current implementation does not allow variables to be specified nor does it allow data to be returned.
	PROPERTY	numpartitions	1..n	Optional property indicating the number of data partitions to create. A query must be specified for data partition.
	PROPERTY	arraysize	1..n	Optional property indicating the number of rows to prefetch. The default is 100. Increasing this value may improve performance because fewer trips are made to the database. However, memory usage increases because more records are being held in memory at one time.
	OUTPUT	name.v		The output dataset name.
ORAWRITE				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	dbuserid	username/ password	Database login name and password separated by a forward slash - "/".
	PROPERTY	maxdescriptors	number	Maximum number of columns allowed to be written to the database.
	PROPERTY	schemaowner	schema name	The database owner of the table.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	tablename	table name	Name of the Oracle table to which the data is written.
	PROPERTY	method	“direct” or “conventional”	This is an optional property to specify the loading method. The default is “conventional”. Direct - load the data using SQL*Loader utility with direct=true. Conventional - Load the data using SQL*Loader utility with direct=false.
	PROPERTY	mode	“append” or “truncate”	This is an optional property. Append – add the new data in addition to existing data on the target table. Truncate – delete the data from the target table before loading the new data. The default value is “append.”
	PROPERTY	createtablemode	“recreate” or “create”	This is an optional property that allows the developer to specify table creation mode. If this property is not given, the flow will fail if the specified table does not already exist. recreate – drops the table and recreates the same table before writing the data. create – use this option when the target table being written to does not exist. It will create the table before writing the data only if the table doesn’t exist.
	PROPERTY	allowedrejects	number >= 0	Optional property. A value greater than zero indicates how many errors are allowed SQL*Loader exits with a critical error. If not specified, the Oracle default of 50 rejects will be used

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	exit_threshold	"warning", "exceeded_rejects", or "fail_or_fatal"	Optional property. Allows configuration as to which level of errors or warnings from SQL*Loader will cause RETL to abort. Valid values are as follows, in order of most strict to least strict in causing RETL to abort: 'warning' – any SQL*Loader warnings or fatal errors will cause RETL to abort 'exceeded_rejects' – SQL*Loader rejects exceeding the 'allowedreject' property or SQL*Loader fatal errors will cause RETL to abort 'fail_or_fatal' – only SQL*Loader failures or fatal errors will cause RETL to abort The default value is 'fail_or_fatal'
	PROPERTY	numloaders	1…n	Specifies the degree of parallelism by executing a number of SQL*Loaders running in parallel. May be specified in either conventional or direct mode, although there are certain restrictions for each (See Oracle SQL*Loader syntax for more information)  **Note**: Multiple 'tempdir' properties should be used in conjunction with the 'numloaders' property in a one-to-one mapping in order to maximize performance. For example, if running 8 ways parallel, 8 distinct temporary directories should be specified to maximize performance.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	parallel	true or false	Optional property that allows RETL to execute one SQL*Loader per RETL partition. This behaves similar to the 'numloaders' property but connects SQL*Loader sessions directly to each parallel RETL partition. **Note**: 'parallel' ORAWRITE automatically inherits parallelism from the 'numpartitions' property in rfx.conf, in that one SQL*Loader session is spawned for each partition (e.g. 'numpartitions' set to 4, then 4 SQL*Loaders will be run in parallel). See the "Configuration" section of this document for more on setting 'numpartitions'. To override parallelism in ORAWRITE, set 'parallel' to 'false' and then set 'numloaders' to be the desired number of concurrent SQL*Loaders. When running ORAWRITE in parallel mode, there are restrictions imposed by SQL*Loader. Refer to Oracle's SQL*Loader syntax for more information on these limitations.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	tempdir	temporary directory	The temporary directories to use for the SQL*Loader data files. May be specified multiple times by specifying separate properties  **Note**: In order to maximize performance, temporary directories should reside on separate disk controllers when specifying tempdir with parallel SQL*Loaders via the 'numloaders' property.
	PROPERTY	partition	partition name	Optional Name of the partition of the Oracle table to which data is written.
	PROPERTY	outputdelimiter	output delimiter (e.g. '|', ';', etc)	Optional. Output delimiter to be used when sending data to SQL*Loader  **Note:** By default, SQL*Loader is sent fixed-format data. The outputdelimiter property can significantly speed up the performance of flows that have records with many fields that aren't populated with data. The outputdelimiter should never be set to a value that can be part of the data.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	loaderoptions	options to pass directly to SQL*Loader	Optional. Name-value pair options to be passed to SQL*Loader via the SQL*Loader parfile. In particular, the rows setting can be tweaked to improve performance. The setting specifies the number of rows per commit and defaults to 5000. Increasing this value will decrease the number of commits per SQL*Loader session. **Note:** See Oracle's documentation on SQL*Loader for more information about valid options. RETL does not validate any options passed through this parameter. If these options are incorrect, SQL*Loader will cause RETL to fail.
	PROPERTY	rows	any number	**Deprecated:** Use "rows=*numrows*" in the loaderoptions property.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	controlfileloadoptions	options to pass directly to SQL*Loader	Optional. Options to pass in the control file LOAD command. RETL handles the following: • LOAD keyword • INFILE clause • APPEND, TRUNCATE, REPLACE keywords • INTO TABLE clause In particular, the 'controlfileloadoptions' property can be set to "PRESERVE BLANKS" so that trailing blanks are not stripped by SQL*Loader. The 'outputdelimiter' property should also be specified in this case so that the spaces used for padding a fixed-width data file are not preserved.
	PROPERTY	sortedindexes	database index name	Indicates data sent to SQL*Loader exists in the same order as the specified database index. This eliminates the need to sort the new index entries and can give considerable performance increases when the sorted order of the records is known to match the index. Multiple sort indexes can be separated by commas. May be used in direct mode only

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	singlerow	yes or no	Specify 'yes' to update indexes as each new row is inserted. Default value is 'no'. 📖 **Note**: 'singlerow' should only be specified as 'yes' when it has been shown to increase performance. In most cases, it will make the load slower. Consult the Oracle's SQL*Loader documentation for more information about the SINGLEROW option and when it makes sense to use.
	PROPERTY	preload	string	This is a special query that allows the flow developer to specify a stored procedure to be run prior to SQL*Loader execution. The current implementation does not allow variables to be specified nor does it allow data to be returned. This is useful, for example, to drop indexes prior to a load.
	PROPERTY	postload	string	This is a special query that allows the flow developer to specify a stored procedure to be run after SQL*Loader execution. The current implementation does not allow variables to be specified nor does it allow data to be returned. This is useful, for example, to rebuild indexes after a load.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. 📖 **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. 📖 **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	tablespace	tablespace name	
	INPUT	name.v		The input dataset name.
UPDATE				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	sid	database name	Alias for "dbname"
	PROPERTY	dbuserid	username/ password	Database login name and password separated by a forward slash - "/".

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	maxdescriptors	Number	Maximum number of columns allowed to be written to the database.
	PROPERTY	schemaowner	Schema name	The database owner of the table.
	PROPERTY	tablename	table name	Name of the Oracle table to which the data is written.
	PROPERTY	table	table name	Alias for "tablename"
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	fields	Field name(s)	Space separated list of fields to use for selecting records to update. Fields in the input schema but not in the "fields" property are the fields that are updated.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	batchcount	Number of records	Number of records to process between commit calls. Default value is 1000.
DELETE				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	sid	database name	Alias for “dbname”
	PROPERTY	dbuserid	username/ password	Database login name and password separated by a forward slash - “/”.
	PROPERTY	maxdescriptors	Number	Maximum number of columns allowed to be written to the database.
	PROPERTY	schemaowner	Schema name	The database owner of the table.
	PROPERTY	tablename	table name	Name of the Oracle table to which the data is written.
	PROPERTY	table	table name	Alias for “tablename”
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	fields	Field name(s)	Space separated list of fields to use for selecting records to delete.
	PROPERTY	batchcount	Number of records	Number of records to process between commit calls. Default value is 1000.
INSERT				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	sid	database name	Alias for "dbname"
	PROPERTY	dbuserid	username/ password	Database login name and password separated by a forward slash - "/".
	PROPERTY	maxdescriptors	Number	Maximum number of columns allowed to be written to the database.
	PROPERTY	schemaowner	Schema name	The database owner of the table.
	PROPERTY	tablename	table name	Name of the Oracle table to which the data is written.
	PROPERTY	table	table name	Alias for "tablename"

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	fields	Field name(s)	Space separated list of fields to insert into the database table. Defaults to all fields.
	PROPERTY	batchcount	Number of records	Number of records to process between commit calls. Default value is 1000.
PREPAREDSTATEMENT				
	PROPERTY	dbname	database name	Name of the Oracle database.
	PROPERTY	sid	database name	Alias for "dbname"
	PROPERTY	dbuserid	username/ password	Database login name and password separated by a forward slash - "/".

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	maxdescriptors	Number	Maximum number of columns allowed to be written to the database.
	PROPERTY	schemaowner	Schema name	The database owner of the table.
	PROPERTY	tablename	table name	Name of the Oracle table to which the data is written.
	PROPERTY	table	table name	Alias for "tablename"
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	statement	SQL statement	SQL to execute for each input record. Use ? to denote a value to be replaced by a field from the current record.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	fields	Field name(s)	Space separated list of fields to use as the values for the ? place holders in the "statement" property.
	PROPERTY	batchcount	Number of records	Number of records to process between commit calls. Default value is 1000.

Database Operator Examples

ORAREAD

```
<OPERATOR type="oraread">
   <PROPERTY name=”sp_prequery” value=”exec pre_storedproc”/>
   <PROPERTY name="dbname"          value=“RETLdb"/>
   <PROPERTY name="connectstring"   value=“RETLdb/rpassword"/>
   <!--Note: query must be enclosed in CDATA element otherwise -->
   <!--      this query will contain invalid XML!              -->
   <PROPERTY name="query">
   <![CDATA[
      select * from rtbl where col > 1
   ]]>
   </PROPERTY>
   <PROPERTY name="maxdescriptors"  value="100"/>
   <PROPERTY name="datetotimestamp" value="false"/>"
   <PROPERTY name=”sp_postquery” value=”exec post_storedproc”/>
   <OUTPUT name="test.v"/>
</OPERATOR>
```

ORAWRITE

```
<OPERATOR type="orawrite">
   <INPUT name="test.v"/>
   <PROPERTY name=”preload” value=”exec pre_storedproc”/>
   <PROPERTY name="dbname"    value=“RETLdb"/>
   <PROPERTY name="dbuserid"  value="username/password"/>
   <PROPERTY name=“tablename”    value="ORG_LOC_DM"/>
   <PROPERTY name="mode"      value="append"/>
   <PROPERTY name="method"    value="conventional"/>
   <PROPERTY name=”preload” value=”exec pre_storedproc”/>
</OPERATOR>
```

UPDATE

Suppose you have dataset with records reflecting new employee salaries with the fields below.

- EMPLOYEE_ID
- SALARY
- EFFECTIVE_DATE

To update a table named EMPLOYEE_SALARY with the new values using EMPLOYEE_ID as the key field, you'd use an UPDATE operator similar to the following:

```
<OPERATOR type="update">
   <INPUT name="salary_updates.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="EMPLOYEE_SALARY"/>
   <PROPERTY name="fields" value="EMPLOYEE_ID"/>
</OPERATOR>
```

The SQL executed is

```
UPDATE EMPLOYEE_SALARY SET SALARY = salary, EFFECTIVE_DATE =
effective_date WHERE EMPLOYEE_ID = employee_id
```

salary, *effective_date*, and *employee_id* are the values of the SALARY, EFFECTIVE_DATE, and EMPLOYEE_ID fields of the current record being processed by RETL.

DELETE

Suppose you have a dataset with information on stores that are being closed keyed on STORE_ID and AREA and you want to delete the corrsponding records in the STORES table. You'd use a DELETE operator similar to the following:

```
<OPERATOR type="delete">
   <INPUT name="closing_stores.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="STORES"/>
   <PROPERTY name="fields" value="STORE_ID AREA"/>
</OPERATOR>
```

The SQL executed is

```
DELETE FROM STORES WHERE STORE_ID = store_id AND AREA = area
```

store_id and *area* are the values of the STORE_ID and AREA fields of the current record being processed by RETL.

INSERT

Suppose you have a dataset with records containing purchase order information that you need inserted into the PURCHASE_ORDER table. You'd use an INSERT operator similar to the following:

```
<OPERATOR type="insert">
   <INPUT name="purchase_orders.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="PURCHASE_ORDERS"/>
</OPERATOR>
```

The SQL executed is

```
INSERT INTO PURCHASE_ORDERS (PO_NUMBER, ORDER_DATE, BILL_TO_ADDRESS,
COMMENT, ...) VALUES (po_number, order_date, bill_to_address,
comment, ...)
```

po_number, *order_date*, *bill_to_address*, and *comment* are the values of the PO_NUMBER, ORDER_DATE, BILL_TO_ADDRESS, and COMMENT fields of the current record being processed by RETL.

To leave out fields, specify only the fields you want in the "fields" property:

```
<OPERATOR type="insert">
   <INPUT name="purchase_orders.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="PURCHASE_ORDER_DATES"/>
   <PROPERTY name="fields" value="PURCHASE_ORDER ORDER_DATE"/>
</OPERATOR>
```

The SQL executed is

```
INSERT INTO PURCHASE_ORDER_DATES (PO_NUMBER, ORDER_DATE) VALUES
(po_number, order_date)
```

PREPAREDSTATEMENT

The example for UPDATE above can be performed by a PREPAREDSTATEMENT similar to the following:

```
<OPERATOR type="preparedstatement">
   <INPUT name="salary_updates.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="EMPLOYEE_SALARY"/>
   <PROPERTY name="statement"
      <![CDATA[
         UPDATE EMPLOYEE_SALARY
         SET SALARY = ?,
         EFFECTIVE_DATE = ?
         WHERE EMPLOYEE_ID = ?
      ]]>
   <PROPERTY name="fields" value="EMPLOYEE_ID SALARY
EFFECTIVE_DATE"/>
</OPERATOR>
```

The example above for DELETE can be performed by a PREPAREDSTATEMENT similar to the following

```
<OPERATOR type="preparedstatement">
   <INPUT name="closing_stores.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="STORES"/>
   <PROPERTY name="statement" value="DELETE FROM STORES WHERE
STORE_ID = ? AND AREA = ?"/>
   <PROPERTY name="fields" value="STORE_ID AREA"/>
</OPERATOR>
```

The example above for INSERT can be performed by a PREPAREDSTATEMENT similar to the following

```
<OPERATOR type="preparedstatement">
   <INPUT name="purchase_orders.v"/>
   <PROPERTY name="dbname" value="dbname"/>
   <PROPERTY name="schemaowner" value="schemaowner"/>
   <PROPERTY name="dbuserid" value="username/password"/>
   <PROPERTY name="tablename" value="PURCHASE_ORDERS"/>
   <PROPERTY name="statement">
      <![CDATA[
         INSERT INTO PURCHASE_ORDERS
            (PO_NUMBER, ORDER_DATE, BILL_TO_ADDRESS, COMMENT)
         VALUES (?, ?, ?, ?)
      ]]>
   </PROPERTY>
   <PROPERTY name="fields" value="PO_NUMBER ORDER_DATE
      BILL_TO_ADDRESS COMMENT"/>
</OPERATOR>
```

Chapter 8 – Input and Output Operators

Each of the following input and output operators is described in this chapter, along with tag usage example code:

- DEBUG
- NOOP
- EXPORT
- IMPORT

DEBUG

The DEBUG operator prints all records to standard output.

NOOP

The NOOP operator acts as a terminator for datasets. In short it does nothing with the data – just throws it away.

EXPORT

The EXPORT operator writes a RETL dataset to a flat file, either as delimited or fixed-length records, depending upon the schema.

Note: By default, EXPORT will export records in pipe-delimited format ('|'). To change the output format, an export schema file should be used to explicitly use a different format.

IMPORT

The IMPORT operator imports a flat file into RETL, translating the data file into a RETL dataset.

Input and Output Operators XML Specification Table

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
DEBUG				
	INPUT	name.v		The input dataset name.
	PROPERTY	parallel	true or false	Optional property. When set to 'true', the DEBUG operator is included in a partition. When set to 'false', the DEBUG operator is not included in a partition and a funnel is inserted before the DEBUG operator to unpartition the data (default). Ignored if not partitioning.
NOOP				
	INPUT	name.v		The input dataset name.
IMPORT				
	PROPERTY	inputfile	input data file	The name of the text file to import into RETL. This property can be specified multiple times. If numpartitions is not specified or is set to 1, records are funneled into one dataset. Otherwise, a data partition is created for each input file.
	PROPERTY	schemafile	schema file	The name of the schema file describing the layout of the data file.
	PROPERTY	rejectfile	reject file	Optional property. The name of the file where records that do not match the input schema are deposited.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	allowedrejects	number >= 0	Optional property. A value greater than zero indicates how many errors are allowed before rfx exits with a critical error. A zero indicates that any number of errors will be accepted (this is the default setting). 📖 **Note:** The allowedrejects property is only valid when the 'rejectfile' property is specified.
	PROPERTY	verboserejects	'true' or 'false'	Optional property. A value of 'true' means that RETL will print granular warning messages about rejected records to standard error. The default is 'true'. 📖 **Note:** Displaying rejected records to standard error will degrade performance, but setting verboserejects to 'false' without specifying a rejectfile will result in lost records.
	PROPERTY	degreesparallel	1..n	Optional property indicating the number of threads used to read each file. Defaults to 1. Increasing this value may increase performance.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	numpartitions	1..n	Optional property indicating the number of data partitions to create. Defaults to 1. If multiple input files are specified and numpartitions is greater than 1, then numpartitions must equal the number of input files. One data partition will be created for each input file.
	OUTPUT	name.v		The output dataset name.
EXPORT				
	PROPERTY	outputfile	output data file	The output text file name.
	PROPERTY	outputmode	“overwrite” or “append”	Optional property specifies whether EXPORT overwrites an existing file or appends to it. Default is to overwrite.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	schemafile	schema file	Optional property. The name of the schema file that describes the layout of the output file. If not specified, uses current schema (as output from the previous dataset). The default delimiter ('|') is used if no schema file is specified. **Note:** This property may be used to reorder and/or drop fields from the output and reformat the fixed length, delimiter and/or nullvalue characteristics of the schema. However, if there are incompatible changes between the incoming schema and output schemafile, RETL may print warning messages or throw errors.
	PROPERTY	parallel	"true" or "false"	Optional property. When set to 'true', each partition writes its data to a temporary file, then the temporary files are concatenated into the destination file. When set to 'false', the EXPORT operator is not included in a partition and a funnel is inserted before the EXPORT operator to unpartition the data. (default) Ignored if not partitioning.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	INPUT	name.v		The output dataset name.

Input and Output Operators Examples

IMPORT

```
<OPERATOR type="import" name="import1,0">
   <PROPERTY name="inputfile"    value="import.dat"/>
   <PROPERTY name="schemafile"   value="ImportOp.schema"/>
   <OUTPUT name=”test.v"/>
</OPERATOR>

<OPERATOR type="import" name="import1,0">
   <PROPERTY name="inputfile"      value="import.dat"/>
   <PROPERTY name="schemafile"     value="ImportOp.schema"/>
   <PROPERTY name="allowedrejects" value="100"/>
   <PROPERTY name="rejectfile"     value="import.rej"/>
   <OUTPUT name=”test.v"/>
</OPERATOR>
```

EXPORT

```
<OPERATOR type="export" name="export1,0">
   <PROPERTY name="outputfile"   value="output.dat"/>
   <PROPERTY name="schemafile"   value="outputOp.schema"/>
   <INPUT name=“test.v"/>
</OPERATOR>
```

Chapter 9 – Join Operators

RETL provides the following join operators, each of which is described in this chapter:

- INNERJOIN
- LEFTOUTERJOIN
- RIGHTOUTERJOIN
- FULLOUTERJOIN
- LOOKUP

Note: For INNERJOIN, LEFTOUTERJOIN, RIGHTOUTERJOIN and FULLOUTERJOIN the input datasets must be sorted on the key. LOOKUP does not require sorted input.

INNERJOIN

INNERJOIN transfers records from both input datasets whose key fields contain equal values to the output dataset. Records whose key fields do not contain equal values are dropped.

LEFTOUTERJOIN

LEFTOUTERJOIN transfers all values from the left dataset, and transfers values from the right dataset only where key fields match. The operator drops the key field from the right dataset. Otherwise, the operator writes default values.

RIGHTOUTERJOIN

RIGHTOUTERJOIN transfers all values from the right dataset, and transfers values from the left dataset only where key fields match. The operator drops the key field from the left dataset. Otherwise, the operator writes default values.

FULLOUTERJOIN

For records that contain key fields with identical and dissimilar content, the FULLOUTERJOIN operator transfers records from both input datasets to the output dataset.

LOOKUP

LOOKUP is similar to the INNERJOIN operator. However, there is no need to do a sort on the input datasets. Looks up a value from one dataset whose key fields match the lookup dataset and outputs the matching values.

Remember that the lookup dataset must be small enough to fit in memory otherwise severe performance problems may result. If in doubt as to whether the lookup dataset will fit in memory always use one of the join operators.

DBLOOKUP

DBLOOKUP behaves in a similar manner as the LOOKUP operator, but it is used to look up records directly in a database table. This is useful when a flow needs to join a relatively small number of records with a relatively large database table. In this sense, it behaves similar to a DBREAD with a LOOKUP. However DBLOOKUP only pulls records from the database on an as-needed basis. DBLOOKUP also attempts to maximize performance by caching database results and thus minimizing database accesses on key-matched records.

Special notes about Join operators

Join operators will 'favor' data fields from the 'dominant' side (usually, the first INPUT to the operator) when joining datasets that contain the same non-key fields on each side. For example,

Left Dataset

Key	Same Non-Key field	Non-Key field
A	B	C

Right Dataset

Key	Same Non-Key field	Non-Key field
A	B	C

"Joined" dataset

Operator	Key	Same Non-Key field	Non-Key field	Non-Key field
Innerjoin	A	B (from LHS)	C	D
Leftouterjoin	A	B (from LHS)	C	D
Rightouterjoin	A	B (from RHS)	C	D
Fullouterjoin	A	left_B, right_B (from LHS)	C	D
Lookup	A	B (from LHS)	C	D

It is important to notice that fullouterjoin will rename the data fields that have the same name on each dataset, to left_<shared column name> and right_<shared column name>. It is recommended to drop the unneeded same-name fields from each dataset prior to joining in order to make the functionality more explicit to those maintaining your flows.

Join Operators XML Specification Table

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
INNERJOIN				
	PROPERTY	key	key field	Key columns to be joined.
	INPUT	inputname1.v		Input dataset 1 – sorted on the key columns.
	INPUT	inputname2.v		Input dataset 2 – sorted on the key columns.
	OUTPUT	outputname.v		The output dataset name.
LEFTOUTERJOIN RIGHTOUTERJOIN FULLOUTERJOIN				
	PROPERTY	key	key field	Key columns to be joined.
	PROPERTY	nullvalue	column name=<null value>	The default null value that is assigned to the null column.
	INPUT	inputname1.v		Input dataset 1. (left dataset) – sorted on the key columns
	INPUT	inputname2.v		Input dataset 2. (right dataset) – sorted on the key columns
	OUTPUT	name.v		The output dataset name.
LOOKUP				
	PROPERTY	tablekeys	key field	Comma separated key columns to be looked up.
	PROPERTY	ifnotfound	“reject”, “continue”, “drop”, or “fail”	What to do with the record if the result did not match. If not specified, this option will default to “fail”.
	PROPERTY	allowdups	“true” or “false”	Optional property defaults to “true”. This property allows the lookup to return more than 1 record if the lookup dataset has more than 1 matching set of keys.
	INPUT	inputname1.v		First input dataset is always the data that needs to be processed.
	INPUT	inputname2.v		Second input dataset is always the “lookup” dataset.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	resdataset.v		The first output is the successful lookup output dataset name.
	OUTPUT	rejdataset.v		The second will contain all records not matching the "tablekeys" property. This needs to be specified if the "ifnotfound" option is set to "reject".
DBLOOKUP				
	PROPERTY	dbname	database name	Required property. Name of the database.
	PROPERTY	userid	database username	Required property. Database login name
	PROPERTY	password	database password	Required property. Database login password.
	PROPERTY	dbtype	database type	Required property. The type of database to connect to. Valid values are as follows: oracle, jdbc
	PROPERTY	tablekeys	Database table keys to join on	Required property. Comma separated key columns to be looked up in the database 'table' or 'select_sql' result set.
	PROPERTY	table	table name	The table to look up in. This property can be used generically instead of 'select_sql'. Either 'table' or 'select_sql' must be specified.
	PROPERTY	select_sql	SQL SELECT statement	Optional property. Simple SQL SELECT statement used instead of 'table'. Either 'table' or 'select_sql' must be specified. **Note:** 'select_sql' may only contain a SELECT statement without the WHERE clause. RETL will throw an error on an invalid query. Valid 'select_sql' : "SELECT colA,colB from table" Invalid 'select_sql': "SELECT colA,colB from table WHERE colA > colB"

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	hostname	Hostname or IP address	This is an optional property. The fully-specified hostname or IP address where the database resides. This defaults to the localhost if not specified. **Note:** This property should only be specified when it is known that connections are being made to a remote database. This may be specified as a default in rfx.conf for convenience.
	PROPERTY	port	Port number	This is a required property. The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are defaults that come preset in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility 'tnsping' may be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
	PROPERTY	datetotimestamp	"true" or "false"	This is an optional property defaults to "false". When set to "true", ORAREAD returns DATE columns with the time included. When false, the time is not included.
	PROPERTY	ifnotfound	"reject", "continue", "drop", or "fail"	Optional property defaults to "fail". What to do with the record if the result did not match.
	PROPERTY	allowdups	"true" or "false"	Optional property defaults to "true". This property allows the lookup to return more than 1 record if the lookup dataset has more than 1 matching set of keys.
	PROPERTY	usecache	"yes" or "no"	Optional property defaults to "yes". Specifies if the results from the database lookup should be cached or not.
	INPUT	inputname1.v		The data that needs to be processed.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	resdataset.v		The first output is the successful lookup output dataset name.
	OUTPUT	rejdataset.v		The second will contain all records not matching the “tablekeys” property. This needs to be specified if the “ifnotfound” option is set to “reject”.

Join Operators Examples

INNERJOIN

```
<OPERATOR type="innerjoin" name="innerjoin,0">
   <INPUT name=“test_1.v"/>
   <INPUT name=“test_2.v"/>
   <PROPERTY name="key" value="LOC_KEY"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

LEFTOUTERJOIN

```
<OPERATOR type="leftouterjoin" name="leftouterjoin,0">
   <INPUT name="left.v"/>
   <INPUT name="right.v"/>
   <PROPERTY name="key" value="LOC_KEY"/>
   <PROPERTY name="nullvalue" value="PROD_SEASN_KEY=-1"/>
   <OUTPUT name="result.v"/>
</OPERATOR>
```

RIGHTOUTERJOIN

```
<OPERATOR type="rightouterjoin" name="rightouterjoin,0">
   <INPUT name=“right.v"/>
   <INPUT name=“left.v"/>
   <PROPERTY name="key" value="LOC_KEY"/>
   <PROPERTY name="nullvalue" value="PROD_SEASN_KEY=-1"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

FULLOUTERJOIN

```
<OPERATOR type="fullouterjoin" name="fullouterjoin,0">
   <INPUT name=“right.v"/>
   <INPUT name=“left.v"/>
   <PROPERTY name="key" value="LOC_KEY"/>
   <PROPERTY name="nullvalue" value="PROD_SEASN_KEY=-1"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

LOOKUP

```
<OPERATOR type="lookup">
   <INPUT name=“dataset.v"/>
   <INPUT name=“lookupdataset.v"/>
   <PROPERTY name="tablekeys" value="LOC_KEY,SUPP_KEY"/>
   <PROPERTY name="ifnotfound" value="reject"/>
   <OUTPUT name=”result.v"/>
   <OUTPUT name=”reject.v"/>
</OPERATOR>

<OPERATOR type="lookup">
   <PROPERTY name="tablekeys" value="LOC_KEY"/>
   <PROPERTY name="ifnotfound" value="continue"/>
   <PROPERTY name="allowdups" value="true"/>
   <INPUT name="dataset.v"/>
   <INPUT name="lookupdataset.v"/>
   <OUTPUT name="result.v"/>
</OPERATOR>
```

DBLOOKUP

```
<OPERATOR type="dblookup">
   <INPUT name="dataset.v" />
   <PROPERTY name="table" value="mytab"/>
   <PROPERTY name="tablekeys" value="D"/>
   <PROPERTY name="ifnotfound" value="fail"/>
   <PROPERTY name="hostname" value="dbhostname"/>
   <PROPERTY name="userid" value="myuserid" />
   <PROPERTY name="password" value="mypassword" />
   <PROPERTY name="dbname" value="mydatabase" />
   <PROPERTY name="dbtype" value="oracle"/>
   <OUTPUT name="joined.v" />
</OPERATOR>
```

Chapter 10 – Sort, Merge, and Partitioning Operators

RETL provides the following sort and merge operators, each of which is described in this chapter:

- COLLECT and FUNNEL
- SORTCOLLECT and SORTFUNNEL
- HASH
- SORT
- MERGE

COLLECT and FUNNEL

Both operators combine records from input datasets as they arrive. This can be used to combine records that have the same schema from multiple sources. Note that these operators are sometimes used implicitly by RETL to rejoin datasets that have been divided during partitioning in parallel processing environments (where the number of RETL partitions is greater than one).

SORTCOLLECT and SORTFUNNEL

Like COLLECT and FUNNEL these operators combine records from input datasets. These operators maintain sorted order of multiple datasets already sorted by the key fields. Records are collected in sorted order using a merge sort algorithm. If incoming records are unsorted, FUNNEL followed by the SORT operator should be used instead.

Note: The sort order (ascending or descending) of the input datasets to SORTFUNNEL must be the same and must match the 'order' property of SORTFUNNEL. Otherwise, SORTFUNNEL may produce unexpected results.

HASH

The HASH operator examines one or more fields of each input record, called "hash key" fields, to assign records to a processing node. Records with the same values for all hash key fields are assigned to the same processing node. This type of partitioning method is useful when grouping or sorting data to perform a processing operation.

SPLITTER

The round-robin partitioning operator splits records among outputs in an ordered record-by-record fashion. This is an alternative to the HASH operator and distributes records more evenly than HASH.

SORT

Sorts the records in the RETL dataset based on one or more key fields in the dataset.

MERGE

Merges input dataset columns record by record. Each output record is the union of the fields for each incoming record in the INPUT datasets. The number of records contained in the inputs' datasets must match. Columns are ordered by the order of the input datasets, first input dataset, second input dataset, and so on.

Sort and Merge Operators XML Specification Table

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
COLLECT FUNNEL				**Note:** These operators are aliases for each other. We use FUNNEL as a general rule but either operator will work.
	INPUT	inputname1.v		Input dataset 1.
	INPUT	inputname2.v		Input dataset n. There may be 0 or more additional datasets.
	PROPERTY	method	"monitor", "poll", or "cycle"	Optional property. Defaults to "monitor". Specifies the method used to determine which INPUT has a record ready to be funneled. Changing this property from the default can have major performance and CPU usage implications.
	OUTPUT	outputname.v		The output dataset name.
HASH				
	PROPERTY	key	key field	Key columns to be hashed.
	PROPERTY	numpartitions	1..n	Optional property indicating the number of data partitions to create. Defaults to the number of partitions specified in rfx.conf.
	INPUT	inputname.v		Input dataset.
	OUTPUT	name.v		The output dataset name.
SPLITTER				
	PROPERTY	numpartitions	1..n	Optional property indicating the number of data partitions to create. Defaults to the number of partitions specified in rfx.conf.
	INPUT	inputname.v		Input dataset.
	OUTPUT	name.v		The output dataset name.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
SORT				
	PROPERTY	key	key field	Key columns to sort by. When specifying multiple sort keys, the order that they are specified is very important. The records are ordered by the first key field. If there is more than one record with the same first key field, those records are ordered by the second key field, and so on. This behavior replicates the sorting that happens in an SQL ORDER BY clause.
	PROPERTY	order	"desc" or "asc"	Optional property. Set to "desc" for descending sort order and "asc" for ascending sort order. Default value is "asc".
	PROPERTY	removedup	"true"	Optional value. If set then duplicate records will be removed. Default is false.
	PROPERTY	tmpdir	Directory	Optional property. We recommend that this property not be specified. Temporary directories should be specified in the rfx.conf. This property allows one to override the temporary directory to use for the sort command.  **Note:** If used, the directory should be periodically cleaned as the TEMPDIR from the rfx.conf file (see Chapter 2 above).
	PROPERTY	delimiter	Any character	Optional property. Delimiter used when writing out the records to a temporary file. Defaults to '|'. If your data contain '|', set this property to a character that does not occur in your data.
	PROPERTY	numsort	Integer greater than 0	Number of threads used to sort the data. Defaults to 1. Increasing the number of threads used to sort the input may improve performance.
	PROPERTY	numpartitions	Integer greater than 0	Alias for "numsort".
	INPUT	inputname.v		Input dataset.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	name.v		The output dataset name.
SORTCOLLECT SORTFUNNEL				**Note:** These operators are aliases for each other. We use SORTFUNNEL as a general rule but either operator will work.
	PROPERTY	key	key field	Key columns to sort by. Specify multiple keys by using multiple 'key' property instances. For example: <PROPERTY name="key" value="k1"/> <PROPERTY name="key" value="k2"/>
	PROPERTY	order	"desc" or "asc"	Optional property that describes the sort order of the incoming key fields. Default value is "asc" for ascending order. Specify "desc" if keys are sorted in descending order.
	INPUT	inputname1.v		Input dataset 1.
	INPUT	inputname_n.v		Input dataset n – can be 0 or more datasets.
	OUTPUT	resultdataset.v		Output dataset.
MERGE				
	INPUT	inputname1.v		Input dataset 1.
	INPUT	inputname_n.v		Input dataset n – this can be more than 2 datasets.
	OUTPUT	resultdataset.v		The output dataset.

Sort, Merge, and Partitioning Operators Tag Usage Examples

COLLECT

```
<OPERATOR type="collect">
   <INPUT name=“test_1.v"/>
   <INPUT name=“test_2.v"/>
   <INPUT name=“test_2.v"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

HASH

```
<OPERATOR type="hash">
   <INPUT   name=“left.v"/>
   <PROPERTY name="key" value="LOC_KEY"/>
      <OPERATOR type=”sort” >
      <PROPERTY  name="key"   value="LOC_KEY"/>
      <OUTPUT name=”result.v"/>
   </OPERATOR>
</OPERATOR>
```

SORTCOLLECT

```
<OPERATOR type="sortcollect">
   <PROPERTY name="key" value="LOC_KEY"/>
   <INPUT name=“test_1.v"/>
   <INPUT name=“test_2.v"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

MERGE

```
<OPERATOR type="merge">
   <INPUT name=“test_1.v"/>
   <INPUT name=“test_2.v"/>
   <OUTPUT name=”output.v"/>
</OPERATOR>
```

Chapter 11 – Mathematical Operators

RETL provides the following mathematical operators, each of which is described in this chapter:

- BINOP
- GROUPBY
- GROUPBY on multiple partitions

BINOP

BINOP performs basic algebraic operations on two fields. Addition, subtraction, multiplication and division are supported for all numeric types. The left and right operands may also be defined as constants. Note that division by zero produces a null value. Only addition is supported for string fields; this becomes concatenation. Binop is not currently supported for date fields.

GROUPBY

The input to the GROUPBY operator is a dataset to be summarized; the output is a dataset containing one record for each group in the input dataset. Each output record contains the fields that define the group and the output summaries calculated by the operator. One simple summary is a count of the number of records in each group. Another kind of summary is a statistical value calculated on a particular field for all records in a group.

Note: If all rows in a column are null, the GROUPBY operator will provide a null column total.

Groupby requires sorted input only in the case where keys are specified. Otherwise, sorted input is not necessary

GROUPBY on multiple partitions

When running RETL using multiple partitions, run the GROUPBY operator with the ‘parallel’ property set to ‘true’ to get the performance increase from the multiple partitions. A HASH/SORT needs to be performed before the parallel GROUPBY. If the same KEYS are used to GROUPBY that were used for the HASH/SORT, data with the same KEY set will not be spread to multiple partitions. If the next operator is a serial operator, there will be an implied FUNNEL operator called by RETL after the GROUPBY to gather the data from the different partitions together into a single dataset. If the next operator is a parallel operator, RETL will not call a FUNNEL operator and the previous HASH/SORT will be maintained. If not using the same KEYS to GROUPBY, a SORTFUNNEL is required to collect the data from the multiple partitions back into a single, sorted dataset. A final serial GROUPBY must then be performed because data may have been spread across multiple partitions and not included in the GROUPBY calculation. If the next operator is a serial operator, then nothing more needs to be done. If the next operator is a parallel operator, another HASH/SORT must be performed because the HASH/SORT from the parallel GROUPBY will not be maintained because a serial GROUPBY was performed after that.

Mathematical Operators XML Specification Table

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
BINOP				
	INPUT	inputname.v		Input dataset.
	PROPERTY	left / constleft	field name or constant value	If property name = 'left' then the column name for the left operand must be specified. If property name = 'CONSTLEFT' then a constant value must be specified.
	PROPERTY	right / constright	field name or constant value	If property name = 'right' then the column name for the right operand must be specified. If property name = 'CONSTRIGHT' then a constant value must specified.
	PROPERTY	dest	destination field	The result destination field name. This can be the same field name as was specified in 'left' or 'right' properties in order to re-use existing fields.
	PROPERTY	desttype	type	Optional field. The type of the destination field. The default value is the type of the left field.
	PROPERTY	operator	'+' or '-' or '*' or '/'	'+' for addition '-' for subtraction '*' for multiplication '/' for division
	OUTPUT	outputname.v		The output dataset name.
GROUPBY				
	INPUT	input.v		Input dataset.
	PROPERTY	parallel	"true" or "false"	True – If running in multiple partitions. False – If running in a single partition (default).

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	key	key field	Optional property. Key columns to groupby. If key is not specified, groupby considers all input data as one group. **Note**: If 'key' property isn't specified, groupby does not require sorted input
	PROPERTY	reduce	column name	Column to be used in the calculation specified in the following "min", "max", "sum", "first", "last", or "count" properties. More than one operation ("min", "max", etc.) property can be specified after a reduce property. The column specified is used in all of the operations until the next reduce property is specified.
	PROPERTY	min	destination column name	Name of the new column that will hold the minimum of the values in the column specified in the preceding reduce property.
	PROPERTY	max	destination column name	Name of the new column that will hold the maximum of the values in the column specified in the preceding reduce property.
	PROPERTY	sum	destination column name	Name of the new column that will hold the sum of the values in the column specified in the preceding reduce property.
	PROPERTY	first	destination column name	Name of the new column that will hold the first value in the column specified in the preceding reduce property.
	PROPERTY	last	destination column name	Name of the new column that will hold the last value in the column specified in the preceding reduce property.
	PROPERTY	count	destination column name	Name of the new column that will hold the count of the values in the column specified in the preceding reduce property.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	name.v		The output dataset name.

Mathematical Operators Examples

BINOP

```
<OPERATOR type="binop">
   <INPUT name="import.v"/>
   <PROPERTY name="left" value="field1"/>
   <PROPERTY name="operator" value="+"/>
   <PROPERTY name="constright" value="2"/>
   <PROPERTY name="dest" value="destfield"/>
   <OUTPUT name="binop.v"/>
</OPERATOR>

<OPERATOR type="binop">
   <INPUT name="import.v" />
   <PROPERTY name="left" value="field1"/>
   <PROPERTY name="operator" value="-"/>
   <PROPERTY name="right" value="field2"/>
   <PROPERTY name="dest" value="destfield"/>
   <OUTPUT name="binop.v"/>
</OPERATOR>
```

GROUPBY

```
<OPERATOR type="groupby">
   <INPUT name="input_1.v"/>
   <PROPERTY name=”parallel" value="true "/>
   <PROPERTY name="key" value="sex"/>
   <PROPERTY name="reduce" value="age"/>
   <PROPERTY name="min" value="min_age"/>
   <PROPERTY name="max" value="max_age"/>
   <PROPERTY name="first" value="first_age"/>
   <PROPERTY name="last" value="last_age"/>
   <PROPERTY name="count" value="count_age"/>
   <PROPERTY name="sum" value="sum_age"/>
```

```
      <PROPERTY name="reduce" value="birthday"/>
      <PROPERTY name="min" value="min_birthDay"/>
      <PROPERTY name="max" value="max_birthDay"/>
      <PROPERTY name="first" value="first_birthDay"/>
      <PROPERTY name="last" value="last_birthday"/>
      <PROPERTY name="count" value="count_birthday"/>
      <OUTPUT name=”output.v"/>
   </OPERATOR>
```

The following example shows what happens if the dataset was previously hashed and sorted on a different key than the group by key in the GROUPBY operator, and if the GROUPBY was run in multiple partitions.

In this case additional SORTCOLLECT and GROUPBY operators need to be used to guarantee the correct result.

```
   <OPERATOR type="hash">
      <PROPERTY name="key" value="ZIP_CODE"/>
      <PROPERTY name="key" value="NAME"/>
      <OPERATOR type="sort">
         <PROPERTY name="key" value="ZIP_CODE"/>
         <PROPERTY name="key" value="NAME"/>
         <OUTPUT name="output1.v"/>
      </OPERATOR>
   </OPERATOR>

   <OPERATOR type="groupby">
      <INPUT name="output1.v"/>
      <PROPERTY name="parallel" value="true"/>
      <PROPERTY name="key" value="sex"/>
      <PROPERTY name="reduce" value="age"/>
      <PROPERTY name="count" value="count_age"/>
      <PROPERTY name="reduce" value="birthday"/>
      <PROPERTY name="max" value="max_birthday"/>
      <OPERATOR type="sortcollect">
         <PROPERTY name="key" value="sex"/>
         <OPERATOR type="groupby">
            <PROPERTY name="parallel" value=”true”/>
            <PROPERTY name="key" value="sex"/>
            <PROPERTY name="reduce" value="age"/>
```

```
            <PROPERTY name="count" value="count_age"/>
            <PROPERTY name="reduce" value="birthday"/>
            <PROPERTY name="max" value="max_birthday"/>
            <OUTPUT name="output2.v"/>
         </OPERATOR>
      </OPERATOR>
   </OPERATOR>
```

Chapter 12 – Structures and Data Manipulation Operators

RETL provides the following operators for the manipulation of structures and data, each of which is described in this chapter:

- CONVERT
- FIELDMOD
- FILTER
- GENERATOR
- REMOVEDUP

CONVERT

The convert operator is used to convert an existing datatype to a new datatype from an input dataset. The following paragraph describes the syntax of the CONVERT property.

- The root tag <CONVERT>, containing one or more <CONVERTFUNCTION> or <TYPEPROPERTY> tags
- The <CONVERT> tag requires the following attributes:
  - ***destfield***: destination field name or new name.
  - ***sourcefield***: original field name or old field name.
  - ***newtype***: new datatype
  - The <CONVERTFUNCTION> tag allows conversion of a column from one data type to a different data type and has the following attributes:
    - name: The value of the “name” attribute of the conversion function determines what conversion is done. The name generally follows the form: <typeTo>_from_<typeFrom>. Conversions are defined between all numeric values and from numbers to strings as well. The table below shows some example conversions.
  - The <TYPEPROPERTY> tag allows turning the column into nullable or not null and has the following attributes:
    - name: nullable
    - value: “true” or “false”

Conversion functions

There are many conversion functions to allow conversion between types. Much of this is done with default conversions between types (see "Appendix A – Default Conversions"). When using a default conversion you simply specify "default" as the name of the conversion.

Here are the other (non-default) conversion functions:

Conversion Name	Description
make_not_nullable	Converts nullable field to not null. Requires specification of a functionarg tag. The value specified in the "nullvalue" functionarg is used to replace any null fields that are found. See the "Tag usage examples," section later in this chapter.
make_nullable	Converts non-nullable field to nullable field. Requires specification of a functionarg tag. The value specified in the "nullvalue" functionarg is used as the field's null value. See the "Tag usage examples," section later in this chapter.
string_length	Converts a string to a UINT32 with a value equivalent to the string's length.
string_from_int32	Converts to STRING from INT32.
string_from_int64	Converts to STRING from INT64.
string_from_int16	Converts to STRING from INT16.
string_from_int8	Converts to STRING from INT8.
int32_from_dfloat	Converts to INT32 from DFLOAT rounding the result to the nearest integer value.
int64_from_dfloat	Converts to INT64 from DFLOAT rounding the result to the nearest integer value.
string_from_date	Converts to STRING from DATE.
dfloat_from_string	Converts to DFLOAT from STRING.
string_from_dfloat	Converts to STRING from DFLOAT.

Note: RETL will report invalid conversions, but RETL doesn't handle certain conversions well. Care should be taken to ensure that the data is well formed by the time a conversion takes place. Currently overflow and/or underflow conversion errors are not detected by RETL. For example, converting the INT16 "-12" to a UINT16 will result in undefined behavior.

FIELDMOD

Used to remove, duplicate, drop and rename columns within RETL. There are several optional parameters that you can use in the fieldmod operator. Here are some notes on how to use the fieldmod operator:

- Use the keep property when you need to drop a large number of columns and/or ignore columns that you don't know about (allowing your flows to be more resilient to change). Any columns that are not specified in the "keep" property will be dropped. The column names are separated by a space. Multiple keep properties can be specified.
- If you want keep most of columns and just drop a few columns, use the "drop" property. The columns to drop are separated by a space. The drop property is processed after the keep property, which means a column is dropped if it is specified on both keep and drop property. Multiple drop properties can be specified. Duplicated columns in single or multiple drop properties will generate an error, for example, "Delete of field failed (field 'Emp_Age' not found)" since the column has been dropped.
- The "rename" and "duplicate" properties affect the dataset after keep and drop properties have been processed. If you attempt to rename or duplicate a column that has been dropped, RETL will generate an error (e.g. Error: the column 'Emp_Name' has been dropped and can not be duplicated).

FILTER

Used to filter a dataset into two categories:

- Those that meet the filter criterion and,
- Those that do not.

Some uses might be to: filter records where a field must be equivalent to a certain string, filter on records where a certain field is less than 10, or filter on records where two fields are equal. This is somewhat like the WHERE clause of a SQL statement.

GENERATOR

GENERATOR is used to create new datasets for testing or for combining with other data sources. It can act as a standalone operator that generates the dataset from scratch, or as an operator that can add fields to an existing dataset. The following paragraph describes the syntax of the "SCHEMA" property, which in the example below, is represented within the **CDATA** tag.

The SCHEMA property should specify the XML that will generate the fields. The basic outline is:

- The root tag <GENERATE>, containing one or more <FIELD> tags
- The <FIELD> tag specifies the "name" and "type" of the field as attributes, and can contain exactly one field generation tag. Field generation tags include:
  - <SEQUENCE> with the following attributes:
    - init: starting number in sequence. The default is "0".
    - incr: increment to add for each record in sequence. The default is "1".
    - limit: max number, once reached, will start from beginning. Optional, unlimited if not specified.
    - partnum_offset: (true/false) will add the partition number to the init value for parallel operation. The default is false.
    - partcount_incr: (true/false) will multiply the incr value by the number of partitions. The default is false.
  - The <CONST> tag allows the specification of a single attribute, value, which is the value that the field will take on.
  - The <VALUELIST> tag has no attributes, but has multiple <CONST> tags within it, specifying the list of values that should be cycled through

The following are the only data types that can be used within the GENERATOR operator:

- int8
- int16
- int32
- int64
- dfloat
- string
- date

REMOVEDUP

The REMOVEDUP operator performs a record-by-record comparison of a sorted dataset to remove duplicate records. Duplicates are determined based upon the key fields specified.

Structures and Data Manipulation Operators XML Specification Table

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
CONVERT				
	INPUT	input.v		The input dataset name.
	PROPERTY	convertspec		Describes the new column data structure to be converted. See the convert specification and the example for more detail.
	OUTPUT	name.v		The output dataset name.
FIELDMOD				
	INPUT	input.v		The input dataset name.
	PROPERTY	Keep	column_name	This optional property is a space-separated list of the columns to be kept - the named fields will be retained and all others will be dropped. This property may be specified multiple times.
	PROPERTY	Drop	column_name	This optional property is a space-separated list of the columns to be dropped - the named fields will be dropped and all other fields will be retained. This property may be specified multiple times.
	PROPERTY	Rename	new_column_name =old_column_name	Column to be renamed. The new column name is separated by an equal sign from the old column name.
	PROPERTY	Duplicate	column_name =target_colmn_name	Duplicate one column to another column. The source column is separated by an equal sign from the target column name.
	OUTPUT	name.v		The output dataset name.

<table>
<tr><th>OPERATOR Type
(attribute)</th><th>Sub-Tag Name</th><th>Property Name</th><th>Property Value</th><th colspan="2">Description</th></tr>
<tr><td>FILTER</td><td></td><td></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td>INPUT</td><td>inputname.v</td><td></td><td colspan="2">The input dataset name.</td></tr>
<tr><td rowspan="13"></td><td rowspan="13">PROPERTY</td><td rowspan="13">filter</td><td rowspan="13">Filter expression</td><td colspan="2">The following operations are supported. See the “Filter Expressions” section below for more detail.</td></tr>
<tr><td>Operation</td><td>Description</td></tr>
<tr><td>"NOT"</td><td>Logical NOT. Only one NOT is allowed per filter.</td></tr>
<tr><td>"AND"</td><td>Logical AND</td></tr>
<tr><td>"OR"</td><td>Logical OR</td></tr>
<tr><td>"IS_NULL"</td><td>True if field is null.</td></tr>
<tr><td>"IS_NOT_NULL"</td><td>True if field is not null.</td></tr>
<tr><td>"GE"</td><td>Logical >=</td></tr>
<tr><td>"GT"</td><td>Logical ></td></tr>
<tr><td>"LE"</td><td>Logical <=</td></tr>
<tr><td>"LT"</td><td>Logical <</td></tr>
<tr><td>"EQ"</td><td>True if right and left fields are equal.</td></tr>
<tr><td>"NE"</td><td>True if right and left fields are not equal.</td></tr>
<tr><td></td><td>PROPERTY</td><td>rejects</td><td>“true” or “false”</td><td colspan="2">Optional filter value whose default value is false. If true then the operator will expect a second OUTPUT to be specified into which rejected (i.e. filtered) records will be deposited.</td></tr>
<tr><td></td><td>OUTPUT</td><td>name.v</td><td></td><td colspan="2">The first output dataset specified contains all of the records for which the filter expression was true.</td></tr>
</table>

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	reject.v		This property is required if the "rejects" property (above) is specified. This output dataset is the set of records that did not meet the filter condition.
GENERATOR				
	INPUT	input.v		This property is optional. If specified, this is the input dataset.
	PROPERTY	parallel	"true" or "false"	True – if generator option should be run in parallel. False – if generator option should be run in a single partition (default).
	PROPERTY	numpartitions	1..n	Optional property indicating the number of data partitions to create.
	PROPERTY	numrecords	1..n	This optional field is used when no INPUT is specified to determine how many records are generated. If an input is specified this value is ignored. The default value is 1.
	PROPERTY	schema		Describes the new column data structure to be generated. See the generator specification and the example for more detail.
	OUTPUT	name.v		The output dataset name.
REMOVEDUP				
	INPUT	input.v		The input dataset name.
	PROPERTY	key	column_name	Columns that have the same data to be removed. More than one column property can be specified.
	PROPERTY	keep	first or last	Either keeps the first or last data if key fields are the same. It will default to first. **Note**: 'first' should be specified as the value for the 'keep' property when sort order makes it possible. This will increase the speed upon which removedup operates.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	name.v		The output dataset name.

Filter Expressions

Filter expressions are made up of fields, constants, and comparison operators, much like the WHERE clause of a SQL SELECT statement.

Simple filter expressions are in the form

```
operand comparison-operator operand
```

or

```
field null-comparison-operator
```

`operand` can be the name of a field or a constant. `comparison-operator` is one of GE (greater than or equal to), GT (greater than), LE (less than or equal to), LT (less than), EQ (equal to), and NE (not equal to). `null-comparison-operator` is IS_NULL or IS_NOT_NULL.

For example,

- CUST_KEY EQ -1
- CUST_DT_OF_BIRTH IS_NOT_NULL

Simple filter expressions can be chained together using AND and OR to form more complex expressions. The complex expression is always evaluated from left to right. That is, AND and OR are at the same order of evaluation.

Note: This order of evaluation is different than SQL.

Starting with RETL 11.3, parentheses are supported to alter the order of evaluation. Subexpressions within parentheses are evaluated first.

For example,

- TRAN_CODE EQ 30 OR TRAN_CODE EQ 31
- START_DATE LE DAY_DT AND END_DATE GE DAY_DT
- CUST_DT_OF_BIRTH EQ ‘20050101’ OR CUST_DT_OF_BIRTH IS_NULL
- (TRAN_CODE EQ 30 AND TRAN_TYPE EQ ‘A’) OR (TRAN_CODE EQ 31 AND TRAN_TYPE EQ ‘B’)

  Note that without the parentheses, the filter expression above would not evaluate as intended because the order of evaluation is left to right and would be equivalent to

  (((TRAN_CODE EQ 30 AND TRAN_TYPE EQ ‘A’) OR TRAN_CODE EQ 31) AND TRAN_TYPE EQ ‘B’)

  In this case, a record with TRAN_CODE = 30 and TRAN_TYPE = ‘A’ would evaluate to false.

Structures and Data Manipulation Operators Examples

CONVERT

```
<OPERATOR type="convert">
   <INPUT name="input.v"/>
   <PROPERTY name="convertspec">
      <![CDATA[
         <CONVERTSPECS>
            <CONVERT destfield="SUPP_IDNT" sourcefield="SUPPLIER"
               newtype="string">
               <CONVERTFUNCTION name="string_from_int64"/>
            </CONVERT>
            <CONVERT destfield="LOC_IDNT" sourcefield="LOCATION"
               newtype="string">
               <CONVERTFUNCTION name="string_from_int32"/>
               <TYPEPROPERTY name="nullable" value="true"/>
            </CONVERT>
         </CONVERTSPECS>
      ]]>
   </PROPERTY>
   <OPERATOR type="debug"/>
</OPERATOR>

<OPERATOR type="convert">
   <INPUT name="inv_sbc_lw_dm.v"/>
   <PROPERTY name="convertspec">
      <![CDATA[
         <CONVERTSPECS>
            <CONVERT destfield="SBCLASS_KEY"
               sourcefield="SBCLASS_KEY">
               <CONVERTFUNCTION name="make_not_nullable">
                  <FUNCTIONARG name="nullvalue" value="-1"/>
               </CONVERTFUNCTION>
            </CONVERT>
         </CONVERTSPECS>
      ]]>
```

```
      </PROPERTY>
      <OUTPUT name="converted.v"/>
   </OPERATOR>
```

FIELDMOD

```
   <OPERATOR type="fieldmod">
      <INPUT name="input.v"/>
      <PROPERTY name="keep" value="Emp_Name Emp_Age"/>
      <PROPERTY name="keep" value="Emp_ID"/>
      <OUTPUT name="output.v"/>
   </OPERATOR>

   <OPERATOR type="fieldmod">
      <INPUT name="input.v" />
      <PROPERTY name="rename" value="Emp_Name=EmpName"/>
      <OUTPUT name="output.v"/>
   </OPERATOR>

   <OPERATOR type="fieldmod">
      <INPUT name="input.v"/>
      <PROPERTY name="drop" value="Emp_Title"/>
      <OUTPUT name="output.v" />
   </OPERATOR>

   <OPERATOR type="fieldmod">
      <INPUT name="input.v" />
      <PROPERTY name="duplicate" value="Emp_Name=Name"/>
      <OUTPUT name="output.v"/>
   </OPERATOR>
```

FILTER

```
<OPERATOR type="filter">
   <INPUT name="input.v"/>
   <PROPERTY name="filter" value="SSN EQ ‘123456789’ AND
      SALARY GT ‘124.22’ AND DOB LT ‘19981010’"/>
   <PROPERTY name="rejects" value=”true”/>
   <OUTPUT name="valid.v"/>
   <OUTPUT name="reject.v"/>
</OPERATOR>
```

GENERATOR

```
<OPERATOR type="generator">
   <PROPERTY name="numrecords" value="5"/>
   <PROPERTY name="schema">
      <![CDATA[
         <GENERATE>
            <FIELD name="DM_RECD_LOAD_DT" type="date"
               nullable="false">
               <CONST value="19800101"/>
            </FIELD>
            <FIELD name="LOC_KEY" type="int8"  nullable="true">
               <SEQUENCE init="1" incr="1"/>
            </FIELD>
         </GENERATE>
      ]]>
   </PROPERTY>
</OPERATOR>

<OPERATOR type="generator">
   <INPUT name="input.v"/>
   <PROPERTY name="schema">
      <![CDATA[
         <GENERATE>
            <FIELD name="F_FULL_PO_COUNT" type="int64"
                nullable="true">
               <CONST value="1"/>
            </FIELD>
```

```
            <FIELD name="F_PART_PO_COUNT" type="int64"
               nullable="true">
               <CONST value="0"/>
            </FIELD>
         </GENERATE>
      ]]>
   </PROPERTY>
   <OPERATOR type="debug"/>
</OPERATOR>
```

REMOVEDUP

```
<OPERATOR type="removedup">
   <INPUT name="input.v"/>
   <PROPERTY name="key" value="SKU_KEY”/>
   <PROPERTY name="key" value="LOC_KEY”/>
   <PROPERTY name="keep" value="LAST”/>
   <OUTPUT name="output.v"/>
</OPERATOR>
```

Chapter 13 – Other Operators

This chapter describes these RETL operators and their usage:

- COMPARE
- SWITCH
- CHANGECAPTURE
- COPY
- DIFF
- CLIPROWS
- PARSER
- EXIT

COMPARE

COMPARE performs a field-by-field comparison of records in two presorted input datasets. This operator compares the values of top-level, non-vector data types such as strings. All appropriate comparison parameters are supported (for example, case sensitivity and insensitivity for string comparisons). The assumption is that the original dataset is used as a basis for the comparison to the second dataset. The comparison results are recorded in the output dataset.

Appends a compare 'result' to beginning of each record with a value set as follows depending upon the outcome of the comparison:

Code	Other
-1	first dataset FIELD is LESS THAN second dataset FIELD
0	first dataset FIELD is the SAME AS second dataset FIELD
1	first dataset FIELD is GREATER THAN second dataset FIELD
-2	record doesn't exist in the first dataset
2	record doesn't exist in the second dataset

Here is an example to illustrate:

Original Key		Compare Key
A		A
B		A
C		E
D		

The resulting output would look like this:

Key	result	Compare Result
A	0	Keys match
B	1	Original key is greater than compare key
C	-1	Original key is less than compare key
D	2	Original key doesn't exist in the compare dataset

Note that the "Compare result" column above is just for illustration. Only the "result" above would be added to the actual output dataset.

SWITCH

SWITCH assigns each record of an input dataset to an output dataset, based on the value of a specified field.

CHANGECAPTURE and CHANGECAPTURELOOKUP

Use CHANGECAPTURE or CHANGECAPTURELOOKUP to compare two datasets. The assumption is that the flow is comparing an original dataset with a modified copy (changed) dataset. The output is a description of what modifications have been made to the original dataset. Here are the changes that are captured:

- Inserts: records added to the original dataset.
- Deletes: records deleted from the original dataset.
- Edits: records matching the given keys fields whose other fields have values have been modified from the original records.
- Copies: records not changed from the original.

The operator works based on these conditions:

- Two inputs (the original and changed datasets), one output containing the combination of the two (edits are taken from the modified side) and a new field to show what change (if any) occurred between the original and the changed dataset.
- The two input datasets must have the same schemas
- For CHANGECAPTURE, the two datasets must be previously sorted on specified key fields. CHANGECAPTURELOOKUP does not require sorted input.

CHANGECAPTURE and CHANGECAPTURELOOKUP identify duplicate records between two datasets by comparing specified key fields. It then compares the value of the fields between the two datasets. The operator adds a field to the output records that contain one of four possible values, representing the conditions noted earlier (inserts, deletes, edits, or copies). The actual values assigned to each of the possibilities can be changed from the default if desired. It is possible to assign the name of this field and to indicate that records assigned into one or more of the above categories be filtered from the output dataset, rather than being passed along with the change code.

To eliminate the need for sorted input, CHANGECAPTURELOOKUP reads the entire changed dataset into memory into a lookup table. Thus, the memory requirements of CHANGECAPTURELOOKUP are much higher than for CHANGECAPTURE.

Here is an example to illustrate:

Original Dataset		
Key	**Value**	**Other**
John	5	corn
Jill	10	beans
Allie	2	pizza
Frank	14	42fas

Changed Dataset		
Key	**Value**	**Other**
John	5	corn
George	7	olives
Allie	5	pizza
Frank	14	Bogo

The resulting OUTPUT would look like this:

Key	Value	Other	Change	CodeField
John	5	Corn	Copy	0
Jill	10	Beans	deletion	2
George	7	Olives	insertion	1
Allie	5	Pizza	Edit	3
Frank	14	Bogo	Copy	0

Note that the “Change” column above is just for illustration. Only the “codefield” above would be added to the actual output dataset.

CHANGECAPTURE is often used in conjunction with the SWITCH operator.

COPY

The COPY operator copies a single input dataset to one or more output datasets.

DIFF

The DIFF operator performs a record-by-record comparison of two versions of the same dataset (the ‘before’ and ‘after’ datasets) and outputs one dataset that contains the difference between the compared datasets. DIFF is very similar to CHANGECAPTURE – the only differences are the default change codes and default drops. CHANGECAPTURE is the recommended operator to use for these types of operations, and DIFF is provided simply for backwards compatibility.

CLIPROWS

The CLIPROWS operator performs a record-by-record comparison of a sorted dataset to “clip” a number of rows from a group of records. For each group value of the input keys it will return up to the first or last N entries in the group. It sends to its OUTPUT dataset the given records for further processing and discards the rest.

PARSER

The PARSER operator allows business logic to be coded in a Java-like script. As a result, it is possible to encapsulate several operators’ functionality into one operator (e.g. binop, filter, copy, noop), improve performance by reducing the number of operators in the flow, and improve maintenance costs by decreasing complexity of flows. Additionally, flows with the PARSER operator are more readily sizeable with partitioning.

Supported Parser constructs:

Conditional operators:

==, !=, >, !>, >=, !>=, <, <=, !<, !<=

Assignment operators:

=, +=, -=, /=, *=, %=

Mathematic operators:

+, -, *, /, %

Statements:

if/else

Datatypes and representations:

Strings: "test1", "-345"

Longs: 1, 2, 1000, -10000

Doubles: 1.0, 1e2

Nulls: null, ""

Fields: RECORD.FIELDNAME (e.g. RECORD.ITEM_LOC)

(RECORD must be in all capitals.)

Dataset output:

A record can be sent to the nth output by using RECORD.OUTPUT[n-1] = true;. (The OUTPUTs are indexed starting at 0. Just like Java arrays.) Additionaly, RECORD.OUTPUT[n-1] = false; allows the nth output to be turned off.

RECORD.OUTPUT[*] = true; sends the record to all defined OUTPUTs. While RECORD.OUTPUT[*] = false; will turn all outputs off.

A record can be conditionally sent to an OUTPUT, making parser act like a filter or switch operator.

A record can be sent to more than one OUTPUT, making parser act like a copy operator. When this is done, the “allowcopy” property must be set to “true”.

A record can be sent to no OUTPUT and consumed like noop. All OUTPUT settings must be set to false individually or using the * designation.

RECORD.OUTPUT[n] can be used as a BOOLEAN value in comparisons, so that you can check if a RECORD is set to go to a specific OUTPUT.

```
if ( RECORD.OUTPUT[0] == true ) { …
```

Comparisons:

Comparisons are made based upon the left-hand-side datatype.

Assignments:

In an assignment, the right-hand-side will be converted to the left-hand-side datatype. An error will be thrown for invalid conversions.

Strings can be concatenated with the + and += operators.

Assignments must end with a semi-colon.

Field usage:

Fields can be used in assignments and comparisons by using RECORD.FIELDNAME to refer to the FIELDNAME field of the current record.

Fields can only be used in an assignment to set other fields. For example, -1 = RECORD.ITEM_LOC is an invalid assignment.

Multiple fields can be assigned or compared, but all fields come from the current record.

Code Blocks:

All statements and expressions appear within a code block, which must be surrounded by curly braces ({}). The curly braces are not optional for single line code blocks as they are in Java.

Comments:

PARSER supports both single line comments starting with // and block comments delimited by /* and */.

EXIT

The EXIT operator aborts the RETL flow after a specified number of records have been processed by the operator.

Other Operators XML Specification

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
COPY				
	INPUT	inputname.v		The input dataset.
	OUTPUT	output1.v		The output dataset 1.
	OUTPUT	output2.v		The output dataset 2.
COMPARE				
	INPUT	input_1.v		Input dataset 1.
	INPUT	input_2.v		Input dataset 2.
	PROPERTY	key	column_name	Column name to be compared between the two datasets defined above.
	OUTPUT	name.v		Output dataset.
CLIPROWS				
	INPUT	input.v		Input dataset.
	PROPERTY	key	column_name	Required property. Indicates that the given column is part of the group definition. Multiple keys can be specified. If you are using partitioning with cliprows, make sure you are using a HASH operator with the same key values. That is, the cliprows keys and hash keys must be the same otherwise incorrect results will be returned.
	PROPERTY	which	“first” or “last”	Required property. Indicates that the given column is part of the group definition. Multiple keys can be specified.
	PROPERTY	count	number >= 1	Required property. Indicates the number of rows to clip.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	parallel	“true” or “false”	Optional property. When set to ‘true’, the cliprows operator will be included in a partition. When set to ‘false’, the cliprows operator will not be included in a partition and a funnel will be inserted before the cliprows operator to unpartition the data.(default) Ignored if not partitioning.
	OUTPUT	name.v		Output dataset.
DIFF				
	INPUT	input_1.v		Input dataset 1.
	INPUT	input_2.v		Input dataset 2.
	PROPERTY	key	column_name	The key field to perform the comparison. Can be multiple columns.
	PROPERTY	allvalues	‘true’ or ‘false’	True – compares all the values between the datasets. False – Does not compare all values. If this property is ‘false’ then the ‘VALUE’ property must be set (default).
	PROPERTY	value	column_name	After the “key” property has been used to join datasets, the “value” property is used to determine differences among the records in the joined datasets. Must be present in both before and after schemas. Determines if the record is a copy or a delete. The fields are processed one by one in the order they are present in the record.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	copycode	2	The code that is written to the first field of the output dataset. The default is 2 if it is a copy.
	PROPERTY	editcode	3	The code that is written to the first field of the output dataset. The default value is 3.
	PROPERTY	deletecode	1	The code that is written to the first field of the output dataset. The default for a delete is 1.
	PROPERTY	insertcode	0	The code that is written to the first field of the output dataset. The default for an insert is 0.
	PROPERTY	dropcopy	True	To drop a copy record from the output. The default is true.
	PROPERTY	dropedit	True	To drop an edit record from the output. The default is true.
	PROPERTY	dropdelete	True	To drop a delete record from the output. The default is true.
	PROPERTY	dropinsert	True	To drop an insert record from the output. The default is true.
	PROPERTY	sortascending	True	The operator assumes that the records are sorted in ascending order by default; use this field to change the sort order.
	OUTPUT	name.v		Output dataset.
CHANGE-CAPTURE and CHANGE-CAPTURE-LOOKUP				
	INPUT	original.v		Input dataset 1. This is the original dataset.
	INPUT	changed.v		Input dataset 2. This is the modified dataset.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	allvalues	'true' or 'false'	True – compares all the values between the datasets. False – Does not compare all values. If this property is 'false' then the 'VALUE' property should be set. **Note**: If the 'values' property is not set when 'allvalues' is 'false', changecapture acts as if 'allvalues' is set to 'true'.
	PROPERTY	key	column_name	Records that match key fields are considered to be the same when determining existence in one dataset versus the other. After the two records are matched via the keys basis the operator can determine if there are copies or edits. If a record exists in the original dataset but not the changed dataset then it is considered to be a delete. Likewise if a record exists in the changed dataset but not the original it is an insertion. There can be multiple instances of this property, allowing the use of composite keys.
	PROPERTY	value	column_name	This property indicates which fields to base the comparisons after records have been matched on key fields. When the given values match the records are considered to be copies. When the values differ the records are considered to be edits. There can be multiple instances of this property, allowing the use of composite values. This is an optional property. When not specified, all common fields are used in the comparison. This is identical to setting the 'allvalues' property to 'true'.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	dropcopy	“true” or “false”	Indicates whether records determined to be copies should be filtered out of the output stream. The default is true.
	PROPERTY	dropedit	“true” or “false”	Similar to dropcopy, but with regard to edits. The default is false.
	PROPERTY	dropdelete	“true” or “false”	Similar to dropedit, but with regard to deletes.
	PROPERTY	dropinsert	“true” or “false”	Similar to dropedit, but with regard to inserts.
	PROPERTY	copycode	0	The value to set the change code field to for records that are copies. The default is 0.
	PROPERTY	insertcode	1	Similar to copycode, but for inserts. The default is 1.
	PROPERTY	deletecode	2	Similar to copycode, but for deletes. The default is 2.
	PROPERTY	editcode	3	Similar to copycode, but for edits. The default is 3.
	PROPERTY	codefield	column_name	The field name to set the change code field to.
	PROPERTY	sortascending	“true” or “false”	A Boolean property to indicate whether the incoming records are sorted in ascending order (true) on the key fields, or in descending order (false). The default is true.
	OUTPUT	name.v		Output dataset name.
SWITCH				
	INPUT	input.v		Input dataset name.
	PROPERTY	switchfield	field_name	Fieldname to determine the switch.
	PROPERTY	casevalues	“value1 = 0, value2 = 1”	This property assigns the switch value base on a comma separated list of mappings from value to output dataset. Note: output datasets are numbered starting from 0 starting from first specified to the last. Note: string values must be enclosed in single quotes. (e.g. “value1=’str0’, value2=’str1’)

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	PROPERTY	discard	“value”	Value to be discarded or dropped.
	PROPERTY	ifnotfound	“allow”, “fail”, “ignore”	This property determines what to do with the record if the casevalues are not found. ‘allow’ – the record is put on the last output dataset. If the number of output datasets matches the number of values in the "casevalues" property, then records that match the last case value and those that don't match any of the case values will share the same dataset. ‘fail’ – halts the process (default). ‘ignore’ – Drops the record. The record is not put on any output dataset.
	OUTPUT	valid.v		The valid output dataset. Number of outputs should correspond to number of casevalues. E.g. “value1=0, value2=1” should have two outputs (or three if ‘discard’ is specified)
	OUTPUT	reject.v		The reject output dataset.
PARSER				
	INPUT	input.v		Input dataset name.
	PROPERTY	expression	any	Language expression the operator should use to process records. Supports general ‘java-like’ syntax and must be surrounded by CDATA tags. See the examples below.
	PROPERTY	allowcopy	“true” or “false”	Set “allowcopy” to “true” if the expression can copy the same record to more than one OUTPUT.
	OUTPUT	output.v		First output dataset name. At least one OUTPUT is required.

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
	OUTPUT	output.v		Multiple OUTPUTs can be specified. To copy the current record to a specific OUTPUT, use RECORD.OUTPUT[n] = true;, where n is between 0 and the number of OUTPUTs minus 1.
EXIT				
	PROPERTY	records	Number of records	When the number of records processed by the EXIT operator reaches the number of records specified in the “records” property, RETL aborts the flow. An error message is displayed indicating that the EXIT operator terminated the flow. The default value is “0”, which is interpreted to mean that the EXIT operator should **not** terminate the flow.

Other Operators Examples

COPY

```
<OPERATOR type=”copy”>
   <INPUT name=”input.v”/>
   <OUPUT name=”copy1.v”/>
   <OUTPUT name=”copy2.v”/>
</OPERATOR>
```

SWITCH

```
<OPERATOR type="switch">
   <INPUT name="import1.v" />
   <PROPERTY name="switchfield" value="COLOR" />
   <PROPERTY name="casevalues" value=" RED=0, BLUE=1" />
   <PROPERTY name="discard" value="YELLOW" />
   <PROPERTY name="ifnotfound" value="allow" />
   <OUTPUT name="red.v" />
   <OUTPUT name="blue.v" />
   <OUTPUT name="not_red_blue_or_yellow.v " />
</OPERATOR>
```

COMPARE

```
<OPERATOR type="compare">
   <INPUT name="import1.v"/>
   <INPUT name="import2.v"/>
   <PROPERTY name="key" value="NAME"/>
   <OUTPUT name="compare.v"/>
</OPERATOR>
```

CHANGECAPTURE

```
<OPERATOR type="changecapture">
   <INPUT name="import1.v" />
   <INPUT name="import2.v" />
   <PROPERTY name="key"           value="emp_age" />
   <PROPERTY name="value"         value="emp_name" />
   <PROPERTY name="codefield"     value="change_code" />
   <PROPERTY name="copycode"      value="0" />
   <PROPERTY name="editcode"      value="3" />
   <PROPERTY name="deletecode"    value="2" />
   <PROPERTY name="insertcode"    value="1" />
   <PROPERTY name="dropcopy"      value="false" />
   <PROPERTY name="dropedit"      value="true" />
   <PROPERTY name="dropdelete"    value="true" />
   <PROPERTY name="dropinsert"    value="true" />
   <PROPERTY name="allvalues"     value="true" />
   <PROPERTY name="sortascending" value="false" />
   <OUTPUT name="changecapture.v" />
</OPERATOR>
```

CHANGECAPTURELOOKUP

```
<OPERATOR type="changecapturelookup">
   <INPUT name="import1.v" />
   <INPUT name="import2.v" />
   <PROPERTY name="key"           value="emp_age" />
   <PROPERTY name="value"         value="emp_name" />
   <PROPERTY name="codefield"     value="change_code" />
   <OUTPUT name="changecapture.v" />
</OPERATOR>
```

CLIPROWS

```
<OPERATOR type="cliprows">
   <INPUT name="import1.v"/>
   <PROPERTY name="key" value="LOC"/>
   <PROPERTY name="which" value="first"/>
   <PROPERTY name="count" value="2"/>
   <OUTPUT name="cliprows.v"/>
</OPERATOR>
```

DIFF

```
<OPERATOR type="diff">
   <INPUT name="import1.v"/>
   <INPUT name="import2.v"/>
   <PROPERTY name="key" value="emp_name"/>
   <PROPERTY name="allvalues" value="true"/>
   <OUTPUT name="diff.v"/>
</OPERATOR>
```

PARSER

```
<OPERATOR type="parser">
   <INPUT name="sales.v"/>
      <PROPERTY name="expression">
         <![CDATA[
            if (RECORD.SALES_AMT == null)
            {
               RECORD.SALES_AMT = 0;
            }
         ]]>
      </PROPERTY>
      <OUTPUT name="newsales.v"/>
   </OPERATOR>
```

EXIT

```
<!-- Validates that CODE > 0.  Terminates the flow on the first
record found that has an invalid CODE. -->
<OPERATOR type="filter">
   <INPUT name="data.v"/>
   <PROPERTY name="filter" value="CODE GT 0"/>
   <PROPERTY name="rejects" value="true"/>
   <OUTPUT name="valid_codes.v"/>
   <OPERATOR type="exit">
      <PROPERTY records="1"/>
   </OPERATOR>
</OPERATOR>
```

Examples for the expression property of PARSER

The expression value must be surrounded by `CDATA` tags:

```
<PROPERTY name="expression">
<![CDATA[
   if (RECORD.D > 500000)
   {
      RECORD.D *= 10;
   }
   else
   {
      RECORD.D *= 5;
   }
]]>

</PROPERTY>
```

Surround date and string constants with double-quotes:

```
<PROPERTY name="expression">
   <![CDATA[
      if (RECORD.A == "20020101")
      {
         RECORD.A = "20030101";
      }
      if (RECORD.B == "abc")
```

```
        {
           RECORD.B = "DWS";
        }
        else if (RECORD.B == "def")
        {
           RECORD.B = "CP";
        }
        else
        {
           if (RECORD.B == "hij")
           {
              RECORD.B = "2uy";
           }
           RECORD.B = "JC";
        }
     ]]>
  </PROPERTY>
```

Example of ==, <, <=, >, >=, +=, -=, *=, /=, and %= operators:

```
  <PROPERTY name="expression">
     <![CDATA[
        if (RECORD.D == 1)
        {
           RECORD.D += 1;
        }

        if (RECORD.D < 10)
        {
           RECORD.D -= 2;
        }

        if (RECORD.D <= 1)
        {
           RECORD.D *= 3;
        }

        if (RECORD.D > 1)
```

```
        {
           RECORD.D /= 4;
        }

        if (RECORD.D >= 1)
        {
           RECORD.D %= 5;
        }

        if (RECORD.DAY_DT !> "20030629")
        {
           RECORD.DAY_DT = "20030725";
        }
     ]]>
  </PROPERTY>
```

Setting `RECORD(`*n*`)` will send the record to the `n+1` OUTOUT, making parser act like a filter or switch operator.

```
  <PROPERTY name="expression">
     <![CDATA[
        if (RECORD.A == 1)
        {
           RECORD.OUTPUT[0] = true;
        }
        else
        {
           RECORD.OUTPUT[1] = true;
        }
     ]]>
  </PROPERTY>
```

Sending the record to multiple outputs makes parser act like a copy operator. The “allowcopy” property must be set to “true” or you will receive an error.

```
<PROPERTY name="allowcopy" value=”true”/>
<PROPERTY name="expression">
   <![CDATA[
      // Either the code below
      RECORD.OUTPUT[0] = true;
      RECORD.OUTPUT[1] = true;

      // Or the simpler syntax in the commented code below
      //RECORD.OUTPUT[*] = true;
   ]]>
</PROPERTY>
```

Parser can act like a filter and a copy at the same time. In this example, some records will be sent to the first OUTPUT and some will be sent to the second OUTPUT. All records will be sent to the third OUTPUT.

```
<PROPERTY name="allowcopy" value=”true”/>
<PROPERTY name="expression">
   <![CDATA[
      if (RECORD.A == 1)
      {
         RECORD.OUTPUT[0] = true;
      }
      else
      {
         RECORD.OUTPUT[1] = true;
      }
      RECORD.OUTPUT[2];
   ]]>
</PROPERTY>
```

Parser can act like a FILTER, COPY and a NOOP at the same time. In this example, some records will be sent to the first OUTPUT, some will be sent to both OUTPUTs, and some will be sent to no OUTPUTs.

```
<PROPERTY name="allowcopy" value=”true”/>
<PROPERTY name="expression">
      <![CDATA[
      if (RECORD.A == 1)
      {
         RECORD.OUTPUT[0] = true;
      }
      else if (RECORD.A == 2)
      {
         RECORD.OUTPUT[*] = true;
      }
      else
      {
         RECORD.OUTPUT[*] = false;
      }
   ]]>
</PROPERTY>
```

Chapter 14 – Common Operator Properties

This chapter describes properties common to all operators.

Common Operator XML Specification

OPERATOR Type (attribute)	Sub-Tag Name	Property Name	Property Value	Description
ALL				
	PROPERTY	progind	any string	Progress indicator string displayed as records are processed.
	PROPERTY	progfreq	number	How often to display the progind string. The progind string will be displayed every progfreq records.

Appendix A – Default Conversions

These are the default conversions for use by the CONVERT operator.

From UINT8

Conversion Name	Description
default	Converts to INT8 from UINT8.
default	Converts to UINT16 from UINT8.
default	Converts to INT16 from UINT8.
default	Converts to UINT32 from UINT8.
default	Converts to INT32 from UINT8.
default	Converts to UINT64 from UINT8.
default	Converts to INT64 from UINT8.
default	Converts to FLOAT from UINT8.
default	Converts to DFLOAT from UINT8.

From INT8

Conversion Name	Description
default	Converts to UINT8 from INT8.
default	Converts to UINT16 from INT8.
default	Converts to INT16 from INT8.
default	Converts to UINT32 from INT8.
default	Converts to INT32 from INT8.
default	Converts to UINT64 from INT8.
default	Converts to INT64 from INT8.
default	Converts to FLOAT from INT8.
default	Converts to DFLOAT from INT8.

From UINT16

Conversion Name	Description
default	Converts to UINT8 from UINT16.
default	Converts to INT8 from UINT16.
default	Converts to INT16 from UINT16.
default	Converts to UINT32 from UINT16.
default	Converts to INT32 from UINT16.
default	Converts to UINT64 from UINT16.
default	Converts to INT64 from UINT16.
default	Converts to FLOAT from UINT16.
default	Converts to DFLOAT from UINT16.

From INT16

Conversion Name	Description
default	Converts to UINT8 from INT16.
default	Converts to INT8 from INT16.
default	Converts to UINT16 from INT16.
default	Converts to UINT32 from INT16.
default	Converts to INT32 from INT16.
default	Converts to UINT64 from INT16.
default	Converts to INT64 from INT16.
default	Converts to FLOAT from INT16.
default	Converts to DFLOAT from INT16.

From UINT32

Conversion Name	Description
default	Converts to UINT8 from UINT32.
default	Converts to INT8 from UINT32.
default	Converts to UINT16 from UINT32.
default	Converts to INT16 from UINT32.
default	Converts to INT32 from UINT32.
default	Converts to UINT64 from UINT32.
default	Converts to INT64 from UINT32.
default	Converts to FLOAT from UINT32.
default	Converts to DFLOAT from UINT32.

From INT32

Conversion Name	Description
default	Converts to UINT8 from INT32.
default	Converts to INT8 from INT32.
default	Converts to UINT16 from INT32.
default	Converts to INT16 from INT32.
default	Converts to UINT32 from INT32.
default	Converts to UINT64 from INT32.
default	Converts to INT64 from INT32.
default	Converts to FLOAT from INT32.
default	Converts to DFLOAT from INT32.

From UINT64

Conversion Name	Description
default	Converts to UINT8 from UINT64.
default	Converts to INT8 from UINT64.
default	Converts to UINT16 from UINT64.
default	Converts to INT16 from UINT64.
default	Converts to UINT32 from UINT64.
default	Converts to INT32 from UINT64.
default	Converts to INT64 from UINT64.
default	Converts to FLOAT from UINT64.
default	Converts to DFLOAT from UINT64.

From INT64

Conversion Name	Description
default	Converts to UINT8 from INT64.
default	Converts to INT8 from INT64.
default	Converts to UINT16 from INT64.
default	Converts to INT16 from INT64.
default	Converts to UINT32 from INT64.
default	Converts to INT32 from INT64.
default	Converts to UINT64 from INT64.
default	Converts to FLOAT from INT64.
default	Converts to DFLOAT from INT64.

From SFLOAT

Conversion Name	Description
default	Converts to UINT8 from SFLOAT.
default	Converts to INT8 from SFLOAT.
default	Converts to UINT16 from SFLOAT.
default	Converts to INT16 from SFLOAT.
default	Converts to UINT32 from SFLOAT.
default	Converts to INT32 from SFLOAT.
default	Converts to UINT64 from SFLOAT.
default	Converts to INT64 from SFLOAT.

Note: Conversions to integer types will truncate decimal values.

From DFLOAT

Conversion Name	Description
default	Converts to UINT8 from DFLOAT.
default	Converts to INT8 from DFLOAT.
default	Converts to UINT16 from DFLOAT.
default	Converts to INT16 from DFLOAT.
default	Converts to UINT32 from DFLOAT.
default	Converts to INT32 from DFLOAT.
default	Converts to UINT64 from DFLOAT.
default	Converts to INT64 from DFLOAT.

Note: Conversions to integer types will truncate decimal values.

Appendix B – Troubleshooting Guide

This Troubleshooting Guide has been developed to provide RETL installation, development and operation suggestions for common support issues. The suggested resolutions are in response to product and documentation feedback from RETL users to the RETL product team.

When trying to run rfx I get this error: ksh: rfx: cannot execute.

RETL is not installed or configured properly. Try running verify_retl from the command line to get more information about the problem. Reinstalling RETL may also resolve the problem.

Are there any suggestions for how best to develop and debug flows?

Yes, there are several important steps you should take when developing and debugging to keep you on the right track.

1 Do not optimize until you have a working flow. Specifically, set numpartitions to 1. Do not use the HASH operator. Worry about tuning for performance after your initial development is complete.

2 When building new flows, start small and work your way up. Add another operator only after you've tested the smaller flow. Back up your work often so that you can look at the differences between flows in order to narrow problems to smaller segments.

3 As you are coding, copy and paste from examples that you know work to avoid syntax/xml errors if you are using a text editor.

4 When you run into problems, try breaking up your flows. Insert DEBUG and/or EXPORT operators to show that your flow is working to a certain point (think binary search here). Move these operators further down the flow until you figure out what is causing it to break. A feature that has been added since version 10.3 is the ability of RETL to print schema files for each dataset connecting operators (by adding '-sSCHEMAFILE' to the command line options). This greatly aids when breaking up and debugging flows, by printing out each input/output schema in schema file format. See the section RFX Command line options for more information about the '-sSCHEMAFILE' option.

5 As of release 10.3, you can look at the log file to see when operators start and stop and how many records are flowing through them. Turn on more details by specifying a higher "level" in the LOGGER section of the configuration (see Configuration section for more details).

When running rfx I get the following error: Error connecting to the database.

You've probably typed an invalid database name, userid, or password. Verify that these parameters are correct (See steps 1-4 of Appendix B – Database Troubleshooting) and try again.

How do I tell what commands are being sent to the database?

Previous versions of RETL streamed database commands to the console by default. To see these commands starting from version 1.7, you should ensure that the environment variable "RFX_SHOW_SQL" is defined within your environment. Assuming that you are using ksh, you can do this by putting the following statement into your .profile or .kshrc, for example:

```
export RFX_SHOW_SQL=1
```

I got an error message about operator "sort:3". How do I figure out which operator "sort:3" is?

RETL names operators internally based upon the type and their position within the specified XML flow. Assuming that numpartitions is set to 1, sort:3 is the fourth sort operator in the XML flow (RETL starts counting at 0).

This becomes a bit more complex if you are using partitioning (numpartitions>1). In short, partitioning splits data and creates parallel data streams within the flow. In effect this multiplies the number of operators being used depending upon the position within the flow and other factors.

There are two ways to determine what operator RETL is referring to in this case. The easiest way is to set numpartitions to 1 and count the operators of that type within the flow. If you cannot do this, then run the flow again except specify the "-g" option. This generates two ".dot" files which contain the flow before and after partitioning. In order to view these you need to download a copy of "dotty" which is included with the graphviz package available from AT&T Bell Labs: (http://www.research.att.com/sw/tools/graphviz/download.html).

I've hashed and sorted my data, but for some reason my output data is not sorted. What is wrong?

Your partitioning is probably set to a value greater than 1. Hashing your data allows RETL to break up your data and parallelize your flow into separate data streams. Later in the flow RETL will rejoin these separate data streams. By default RETL will do this by using the COLLECT or FUNNEL operator. However, these operators do not keep the dataset in sorted order, hence your unsorted output. You can correct this problem by explicitly using a SORTCOLLECT to rejoin the data whenever you are rejoining sorted data.

This problem is also avoided by not partitioning (set partitioning to 1 and/or don't specify a HASH). As always, we recommend that you keep flows as simple as possible until or unless you need the performance boost that partitioning can give you.

I'm using sortcollect (or sortfunnel), but my data is not sorted! What is wrong?

Sort funnel performs a merge sort. It requires sorted input and can merge multiple inputs maintaining sorted order. It cannot perform a full sort of the data. You should use sort to do a full sort of a single data stream. Sortfunnel can be used to merge two data sources that are already sorted on the same keys – the result is a single data stream sorted on the same key.

I am missing fields within my flow. Where did they go?

Often the cause of this problem is that a schema file is incorrect (search for missing quotes, extra quotes) or the database schema is not what you expected. If your flow is not working correctly, take a look at the schemas that are output to stdout when RETL first starts up (specify the "-s" option if necessary to make RETL output the schemas). Look carefully at the schemas displayed for each operator within the flow. Make sure that these display ALL of the fields that you expect to see for this part of the flow. If you are missing fields or fields are of the wrong type, trace backwards to where the schema originates to try and find the source of the problem.

If you are using partitioning and are seeing operators that you do not expect, refer to I got an error message about operator "sort:3". How do I figure out which operator "sort:3" is? This section helps you understand exactly what operators RETL is seeing so that you can more easily find the source of your problem.

How do I filter values from two different tables?

Currently filter assumes that all values required for the filter are in one record. This makes it a bit more difficult to compare values from multiple tables. The way to handle this is to JOIN the two tables and then filter the values after the join.

How do I filter out a set of dynamic values from a data stream?

How do I filter out a set of values? For example – I want to remove any records whose "FIRST_NAME" column does not have a value in the set {Joe, Susan, Mary} and keep all the rest.

This is easily done with RETL using the LOOKUP operator. Simply use the filter set as the key for the lookup. Import a flat file and or table with a single column describing the values to filter – FIRST_NAME = {Joe, Susan, Mary} and use that as the input for the lookup table of the lookup. Any matches are discarded and any misses are processed.

So the actual lookup would look something like this:

```
<OPERATOR type="lookup">
<PROPERTY name="tablekeys"    value="FIRST_NAME"/>
<PROPERTY name="ifnotfound"   value="continue"/>
<INPUT name="records_to_filter.v"/>
<INPUT name="set_names_to_drop.v"/>
<OUTPUT name="records_in_set.v"/>
<OUTPUT name="records_not_in_set.v"/>
</OPERATOR>
```

How do I translate from database types (e.g. NUMBER(12,4)) to RETL types?

"Appendix F – RETL Data Types" contains tables that show the conversion from database data types to RETL data types and vice versa.

What RETL data types should I use when importing a file?

We recommend that you use the biggest numeric type you can (i.e. int64, uint64 or dfloat) to ensure that data is properly preserved and the flow works for as many different sets of data as possible. If you know your data very well and are sure that you can use a small type, you may get a bit of a performance boost by making this a restriction. RETL warns you on the IMPORT if your data cannot be represented by the type that you've chosen.

"Appendix F – RETL Data Types" contains a table that shows the range, size and precision for each datatype.

Why do I receive a warning about the output schema length being less than the input schema length?

For example:

WARN - W115: export:1: Output schema length for field LOC_IDNT (10) is less than the input schema length (25).

Data WILL NOT be truncated to match the output schema and the output data file may be improperly formatted.

This warning is saying that the entire range of values that can be stored in the RETL data type requires more characters than are specified in the export schema. To prevent data loss, the entire value will be written to the file, thereby breaking the file's fixed-length format.

The warning can be encountered when exporting NUMBER fields read from a database. If you have a column defined as NUMBER(10), you need 11 characters (10 + 1 for the negative sign) to hold all possible values. However, RETL stores NUMBER(10) values internally as int64. (See "RETL Data Type/Database Data Type Mapping".) The range of values for an int64 (-9223372036854775808 to 9223372036854775807) requires 25 characters.

This warning can be safely ignored if you know that the data length will not exceed the width specified in the export schema. Exporting NUMBER columns from a database as outlined above is one example.

Is there a common set of properties for all database operators?

Originally, RETL was developed to try and stay consistent with each of the database vendor's terminology. This, however, made it very difficult to write code that was portable between the different databases and to understand how to write flows for different databases. Starting in release 10.2 the following common properties have been added to each of the database operators:

- userid
- password
- tablename

Over time we will attempt to merge the database operators so that RETL is database independent without the use of shell variables.

In my query, I am explicitly selecting the timestamp field as 'YYYYMMDDHH24MISS', but the data in the output file reads "invalid date". What am I doing wrong?

Currently RETL cannot read DATE, TIME and TIMESTAMP types directly out of the database. Because there is no conversion from string to these fields if you must access these fields as DATE, TIME and/or TIMESTAMP you will have to export your data to a file and then import it as a date field.

When attempting an ORAWRITE I get the following error: SQL*Loader-951: Error calling once/load initialization

When you use ORAWRITE operator with the “method” property set to "direct,” you may get the following error message in your SQL*Loader log file:

```
SQL*Loader-951: Error calling once/load initialization
ORA-26028: index name.name initially in unusable state
```

It is because SQL*Loader cannot maintain the index which is in IU state prior to the beginning of a direct path load. Fix this problem by recreating the table index.

How do I request help when all else fails?

You should contact Retek Support at support@retek.com. When requesting help with a flow follow these guidelines to get a more rapid response:

1 Minimize the scope of the problem. Submit the minimal flow with the minimal set of data and minimal schema that still reproduces the problem. This helps to ensure that the support team does not waste time trying to understand a long complicated flow and/or the data involved – allowing us to focus on the true problem.

2 If you have a database operator, (e.g. ORAREAD, ORAWRITE) separate the data from the database. Usually the database is not the problem; therefore removing the database operator from the equation makes our job easier. Replace the DB operator (e.g. ORAREAD, ORAWRITE) with an IMPORT or EXPORT operator. Use the exact same schema as is in the database. Extract a sample set of data into a flat file for the IMPORT operator. This should allow the support team to reproduce your problem here and debug it if necessary. This drastically reduces the amount of time it would take us to look into the problem otherwise.

3 If removing the database operator removes the problem then look very carefully to make sure that there is not a syntax error in the database operator. If you cannot find anything there, look at the IMPORT schema and make sure that it matches the database schema exactly (including null fields, etc). Finally, look at the flat file that the IMPORT operator is using. Make sure that you are using the correct delimiter, that the delimiter is not within the data and that the data is valid. Try setting the “rejectfile” property to see if there are any rejected records. If all this looks good then there may be a problem with the DB operator. Contact us and let us know what you’ve found.

4 Submit the flow and all data files associated with the flow so that the support team can run “rfx –f <yourflowname>” to reproduce the problem along with the log file from “verify_retl” so they can reproduce your environment.

I’ve upgraded and now I get WARNINGS when I never did before! What’s wrong!?

Each version of RETL brings better identification of errors and what might have been overlooked previously is now being caught. “WARNING” messages are generated if the property names and/or values don’t match the spelling/case noted in the Programmer’s Guide. It is very important to run the latest version of RETL and correct the warnings since they indicate that parameters are incorrect. If incorrect, they can produce unexpected results and/or corrupt data. In the future, RETL will stop processing and display error messages rather than just providing warnings to stop the flow until the problem is fixed. Warnings are being used as an intermediate step to minimize the impact to existing product.

Appendix C – Database configuration and Troubleshooting Guide

Since RETL database operators work with database severs and utilities, validating RETL requires some background in database setup and administration. Specifically this involves database setup, RETL database login id privileges, and database connectivity, etc.

RETL Database configuration and maintenance notes

RETL uses very specific tools in order to access the different databases. RETL developers should be sure that their Database Administrator is aware of what RETL does, and how the database needs to be setup to support RETL activities. Below are notes particular to certain databases and platforms.

Debugging database ETL utilities

The DBWRITE operators use the database specific import/export utilities. If you run into problems specific to the database and/or database operators, take a look at the log files and/or control files generated by the database operators (see How do I tell what commands are being sent to the database?). You may be able to better diagnose these problems by checking the utility's limitations, restrictions, environment, and privileges to see if there is a misconfiguration either with the database or within the RETL flow.

Database semaphore problems

Semaphores from database utilities and ODBC instances invoked by rfx may not be properly cleaned up if an rfx process or job is killed from UNIX. In order find and clean up semaphores use “ipcs –s” and “ipcrm –s” respectively.

Runaway loader processes.

Sometimes when rfx is killed some ancillary database utilities are left running. These can be killed via the normal UNIX kill command. The name of the Oracle utility is “sqlldr”.

Troubleshooting RETL with your database

The following RETL/ database setup validation steps will help you work with your DBA and System Administrator to verify and troubleshoot the RETL/database installation.

Note below that we include commands that should be executed from the command line. These are written in `Courier` font. Additionally, variables such as machine names and IP addresses that should be replaced by values particular to your environment are surrounded in <> like this: `<variable name>`.

1 Run verify_retl.

   If you have not already done so run the verify_retl script and correct any problems as directed by the script. When this is running properly proceed to step 2.

2 Verify that your databases are setup properly.

   For all databases go through RETL Database Configuration Notes to make sure that the database is setup properly.

3 Verify database connectivity

   Check the connection between the rfx local machine and the remote database machine if the database and rfx are not on the same machine. We recommend that RETL is located on the database machine for better performance.

   ```
   ping <database machine name>
   ```

   If this fails, try pinging the IP address directly:

   ```
   ping <database machine IP address>
   ```

   If this works then your DNS isn't set up properly. Contact your Network Administrator to fix the problem by adding the IP address to the DNS.

   If this second ping command fails then you need to contact your Network Administrator to determine why you cannot contact the machine. Until this problem is resolved RETL will not work.

4 Check Oracle database server connectivity using

   ```
   tnsping <Oracle Server Name>
   ```

   If this fails contact to your DBA for TNS name setup.

5 Verify database login id/password.

Login to the database by using

```
sqlplus userid/password@Oracle_database_server
```

Contact to your DBA if this step fails – your userid/password is not setup correctly.

6 Check database user id privileges.

Contact your DBA to ensure that the RETL user has privileges to: select, insert, update, create table, etc. Since RETL database operators invoke database utilities to extract and load (see Chapter 5 Database Operators for details), your RETL DB user needs privileges to meet all of the utilities requirements.

These are the privileges that each of the operators requires:

Operators	Privilege
ORAREAD	select
ORAWRITE	select, insert, update, create table, drop table, load

We recommend setting up the RETL/database login id with create/drop table privileges during RETL/database setup phase so that we can run the RETL/database test flows to verify that the database is setup properly for RETL.

7 Run RETL/database testing scripts.

Note these scripts require that the rfx userid has full privileges to run. If you don't have the appropriate privileges to change tables you can still run the read scripts to verify that you can read from the database without problems go to step 10.

```
$RFX_HOME/samples/verify_db/oracle.ksh <database name> <userid>
<password>
```

These scripts will:

- Create a "RETL_test" table with two columns and two rows on the database owned by the given userid.
- Read the two rows from the database and verify that the rows contain the correct information.
- Compare the results from read and verify what was supposed to be written.

In short, this test verifies both RETL read operator and RETL write operator appropriate to the database type.

If the scripts are working properly you see something like the following:

```
Testing a write to etlsun9i
Reading data back out of etlsun9i
Comparing data received from etlsun9i and expected output...
Test Passed!
```

If both tests succeed - Congratulations your database seems to be properly configured! Skip the following steps.

If the scripts are not working properly you will see the following:

```
Testing a write to etlsun9i
Exception in operator [oraread:1]
Error Message
Reading data back out of etlsun9i
Exception in operator [orawrite:1]
Error Message
Comparing data received from <database name> and expected output...
diff: filename: No such file or directory
Changes were detected! Test failed!
```

If either test fails you need to move onto the following steps to try and diagnose the problem. If the ORAWRITE operator passed but the ORAREAD operator did not, move onto step 9. If both of the tests failed, then move onto step 8.

8 Debugging an ORAWRITE failure.

This step verifies that the database write operator is working properly.

```
orawrite.ksh <database_name> <userid> <password>
```

This flow works if the last line of output shows "Flow ran successfully". Go to step 9.

Here are some common failures and their causes:

```
'Error Connecting to the database()'
```

You've probably typed an invalid database name, userid, or password (see "When running rfx I get the following error: 'Error connecting to the database'"). Verify that these parameters are correct (See step 4 above) and try again.

If you still cannot find the problems you can try running step 10 on a table that you know already exists.

9 Debugging an ORAREAD failure.

This step will verify that the database read operator is working properly.

```
oraread.ksh <database name> <userid> <password>
```

If the last line of output shows "Flow ran successfully" your RETL/database setup is working for reads – Congratulations! You can run step 10 if you'd like to try extracting data from your own tables.

If it fails there are several possible reasons (generally the same as those for the write operator) – here are some common errors:

```
'Error Connecting to the database()'
```

You've probably typed an invalid database name, userid, or password (see "When running rfx I get the following error: 'Error connecting to the database'"). Verify that these parameters are correct (See step 4 above) and try again.

If you are still running into problems and there are valid tables that you know work, try step 10. Otherwise review the RETL Database Configuration Notes, to verify that everything is set up correctly and try again.

10 Running the ORAREAD script against user tables.

The database read scripts could be used to access any table to which you have read privileges by simply specifying the table name as the last parameter.

```
oraread.ksh <database> <userid> <password> <table>
```

If the last line of output shows “Flow ran successfully” your RETL/database setup is working for reads on this table.

If you still cannot find out the problems, contact to Retek Support (support@retek.com).

Appendix D – FAQs

Does RETL call stored procedures?

RETL can call stored procedures only within the ORAWRITE operator via the sp_prequery and sp_postquery, which run before and after the ORAWRITE operation respectively. These stored procedures can take static variables (not data from the flow) as parameters. We anticipate expanding on this feature in the future to allow lookups based upon flow data and support for other database operators.

Can RETL handle text translations (e.g. between EBCDIC and ASCII)?

Currently RETL does not do textual translation of this type.

Can RETL handle data translations (e.g. between numbers and strings, etc)?

Data translations between different fields can be done via the LOOKUP and JOIN operators.

Are clients able to obtain RETL performance metrics on RETL?

Generally clients are interested in how a particular RETL flow runs – rather than how RETL itself performs at a fundamental level. These are generally handled on a case-by-case basis for the different product groups since flows are very different from product to product and the environment is different for hardware and configuration. Contact your product group to determine if they have benchmarking number specific to your group. The RETL group does benchmark internal performance results and these are made available via the Product Enhancement Review Board. Contact your product group for inclusion to this board.

Is it possible to join data from 2 different databases?

Yes. This is generally done via two DBREAD operators and then a JOIN operator.

What error handling capabilities does RETL have?

There are many other areas where RETL does error handling and this capability has been expanding.

Error Handling – flow related problems. These are problems that an RETL flow developer might run into while developing flows using RETL. There are many different areas that RETL detects – because these are generally found only during development RETL usually exits with an error message describing one of the scenarios below:

- Schemas: mismatched input schemas (e.g. for FUNNEL), invalid schema specified (e.g. record length does not equal sum of field lengths).
- Mismatched datasets – RETL enforces 1-to-1 correspondence between inputs and outputs.
- Validation of operators, properties and property values.

Data related problems. These are found after the flow is parsed and execution begins. In these cases invalid or non-conforming data may be siphoned into a separate file for later processing:

- Improperly formatted fields via the IMPORT operator (invalid types, lengths, etc).
- Improperly formatted records via the IMPORT operator (missing fields, etc).
- Invalid schemas.

Additionally, flow developers can insert data validation to verify that various database constraints are upheld via a combination of LOOKUP, JOIN and FILTER operators.

What temp files does RETL create and how much space will they take up?

RETL generates temporary files dynamically in order to sort and store intermediate data. These files are generally prefixed with "rfx", but this is not always the case. Therefore, place separate rfx files into their own tmp (e.g. /tmp/rfx) directory so that they can be easily purged on a regular basis (as per the Programmer's Guide).

How much space rfx will consume depends upon the complexity of the flows and how large the source system is. A good starting point is twice the size of the data being manipulated. A more conservative number is 3 times the size of the data being moved. Many people start with 10GB and move up from there if needed.

However, a more precise way of calculating the minimum size of temporary space for a particular flow would be as follows:

Total temp space for a particular flow =

((size of data file) * (# sort operators) * 2)

+ ((size of data file) * (# operators that need to expand datasets))

+ ((size of data file) * (# database write operators))

where # sort operators is the number of sort operators in a flow, # operators that need to expand datasets are the operators that 'page' to disk in 'diamond' flow that contains circular loops, and # database write operators is the number of database write operators such as ORAWRITE.

For example, on a flow IMPORT->SORT->ORAWRITE on a 2Gig input file would be as follows:

(2Gig*1*2) + (2Gig*0) + (2Gig*1) = 6 Gig

Note: It is recommended that system administrators clean up temporary files in the RETL temporary directory on a regular basis.

How do I participate in defining RETL requirements and future direction?

The Product Enhancement Review Board (PERB) is the forum to assist Retek product strategy in defining future enhancements to RETL. If you are not part of the PERB, contact your Retek point person or team lead to inquire about joining.

I have been getting warnings about sorting/sort order such as the following:

[<operator>]: Warning: mismatched sort order - <operator> specifies "ascending" sort order and for the first INPUT to <operator> records are in "descending". These should match - otherwise you may get unexpected results!

OR

[<operator>]: Warning: Input to <operator> does not seem to be sorted! Data should be sorted according to the proper keys or you may get unexpected results!

OR

Various other warning messages dealing with sort order or sort keys

How do I remove these warnings?

Certain operators require that incoming data be sorted. To remove these error messages, place a sort operator that sorts by the required keys in the required order, before the operator in question. If you know that your data is always going to be sorted according to a particular set of keys, then you can specify this in the schema file (See the schema file documentation for more details on this). This last feature should be used with care since if data turns out to not be in the presumed sorted order, certain operators may produce unexpected results.

Which operators require sorted input?

This document should be read in its entirety to assess the need for sorted input. However, the following is a list of operators that do currently require sorted input:

- innerjoin
- leftouterjoin
- rightouterjoin
- fullouterjoin
- sortfunnel
- removedup
- compare
- changecapture
- diff
- groupby (only if 'key' properties are specified)

I have been getting errors when RETL tries to connect to the database such as the following:

--- SQLException caught ---

java.sql.SQLException: Io exception: Connection refused(DESCRIPTION=(TMP=)(VSNNUM=153092608)(ERR=12505)(ERROR_STACK=(ERROR=(CODE=12505)(EMFI=4))))

Message: Io exception: Connection refused(DESCRIPTION=(TMP=)(VSNNUM=153092608)(ERR=12505)(ERROR_STACK=(ERROR=(CODE=12505)(EMFI=4))))

XOPEN SQL State: null

Database Vendor Driver/Source Error Code: 17002

The 'hostname' or 'port' of the database read/write operator may be incorrect or not specified, or the database is down/non-existent. Make sure the database instance is up and running and located on the hostname/port specified in the flow or rfx.conf. If you are using an Oracle database, check tnsnames.ora or use tnsping to figure out the hostname/port.

Why is my database field coming being translated to a RETL dfloat type?

If the SQL query contains a union, the precision of the NUMBER/DECIMAL fields is reported as 0. To ensure that the RETL data type is large enough to hold the data, dfloat is used.

I see an OPERATOR within another OPERATOR. What is that?

This is called "Operator Nesting". See the "**Error! Reference source not found.**" section **Error! Reference source not found.**.

RETL isn't handling international data correctly. What's wrong?

RETL relies on the locale being set correctly in the environment. Make sure that all locale-related environment variables are set and exported correctly.

What is taking up so much memory?

The LOOKUP and CHANGECAPTURELOOKUP operators store the entire lookup table in memory. Thus, if the lookup table has a large number of rows, the memory requirements will be extensive. Consider sorting the data and using an INNERJOIN instead of the LOOKUP or a CHANGECAPTURE instead of the CHANGECAPTURELOOKUP.

Appendix E – RETL 11.x-specific FAQs

Why re-write RETL if it's not broken?

There are a number of reasons why the decision was made. Perhaps the biggest reason was that a huge portion of RETL development resources are required to support the 26 binaries on 3 different platforms for RETL 10.x. The maintenance costs here justify a port such that only one RETL package/binary would be required. Additionally, there are numerous other reasons, such as synergy with other Retek applications, congruency with Retek skill sets as a whole, speed of operator development, etc. that have driven 11.x's development.

Will pre-11.x versions of RETL still be supported?

Since RETL is a customer-driven product, support for pre-11.x versions is still considered top priority. Team RETL has built mini-milestones around consulting with customers and giving support to those upgrading to 11.x. While pre-11.x releases will be supported as long as it is necessary for customers to migrate to 11.x, support for pre-11.x versions may be dropped at some point in the future. The general guideline is that all fixes will be made to 11.x versions. 10.x versions may not be patched if he 11.x functions offer or provide a more appropriate solution.

Since RETL has been re-written in Java, can I run RETL on other platforms that have a JRE?

See the Compatibility Matrix in the Release Notes for supported platforms. While running RETL on platforms other than these is not supported, it is technically feasible that the core framework and operators should work on any platform that has a JRE version 1.4 or higher. We would certainly like to hear from you when/if you have successfully deployed on other platforms, and if these types of requests happen with much frequency/urgency, Team RETL will take this into account when prioritizing efforts for platform proliferation in future releases. Note that additional platform support comes at much less a cost than with previous versions of RETL (10.x).

With the introduction of RETL 11.x, JDBC technology has been introduced as a mechanism for connecting and reading from/writing to databases. Does this mean I can connect to any database (e.g. Sybase, MySQL, Acumate, etc)?

See the Supportability Matrix in the Release Notes for supported databases. While running RETL against databases other than these is not supported, it is possible that 'snapping in' a JDBC-compliant driver by using the generic 'dbread'/'dbwrite' operators will work. Please contact customer support if you are attempting to use a JDBC driver for an unsupported database. Note that additional database support in 11.x releases come at much less a cost than previous versions of RETL (10.x).

RETL 11.x has been re-written in Java. Is there a performance impact in doing this?

Part of the reason RETL was re-written was to address architectural models that have been shown to positively impact performance. Initial volume testing has shown that in many cases RETL 11.2 is significantly faster than pre-11.2 versions. Moreover, performance requirements on 11.x mandate that it come within 30% or better performance for the BETA release, within 15% in the 1st GA release, and meet pre-11.1 performance in the 2nd GA release.

What changes do I need to make to my flows in order for RETL 11.x to run them?

RETL 11.1 requirements specify that it must be backwardly-compatible in the XML flow interface. As a result, only minor changes will need to be made. Read this document in its entirety to assess the need for these changes.

What is the 'upgrade plan' for RETL 11.x?

The recommended upgrade plan is for product groups to start testing with RETL 11.1 as soon as possible. Once product groups determine that RETL 11.x is a feasible upgrade for customers, it is recommended that deployment to these customers be taken. A formal upgrade plan with timelines will be crafted for each interested party at some point in the near future.

Can I deploy RETL 11.x directly to my customers or in a production environment?

This recommendation should come from the product group whose product implements RETL. Only after each product that uses RETL is certified on RETL 11.x should a customer or production system receive the upgrade.

I have a backwards-compatibility problem or other bug with RETL 11.x. Where do I send my bug report?

Narrow down the problem as much as possible, including the flow, data, and a schema file in order to reproduce your problem. Then, please send everything needed to reproduce this problem along with a brief explanation to Retek customer support through an entry of the issue in the customer support ROCS system.

Appendix F – RETL Data Types

Numbers work differently within RETL than they do within databases. This can cause a good deal of confusion. In short, RETL does not currently support arbitrary precision math and therefore has a limited amount of precision to deal with.

RETL Data Type Properties

This table shows the size and precision that each RETL type has available:

RETL Type	Max Size	Max Precision	Validation/Numeric Range
DFLOAT	25	15	Min= -1.7976931348623157E+308 Max= 1.7976931348623157E+308
SFLOAT	25	6	Min=-3.40282347e+38 Max=3.40282347e+38
INT8	4	4	Min=-128 Max=127
INT16	6	6	Min=-32768 Max=32767
INT32	11	11	Min=-2147483648 Max=2147483647
INT64	20	20	Min=-9223372036854775808 Max=9223372036854775807
UINT8	3	3	Min=0 Max=255
UINT16	5	5	Min=0 Max=65535
UINT32	10	10	Min=0 Max=4294967295
UINT64	20	20	Min=0 Max=18446744073709551615
DATE	8	8	Is specified in YYYYMMDD where: YYYY=0000-9999 MM=01-12 DD=01-31

RETL Type	Max Size	Max Precision	Validation/Numeric Range
TIME	8	8	Is specified in HH24MISS format where: HH24=00-23 MI=00-59 SS=00-59
TIMESTAMP	16	16	Is specified in YYYYMMDDHH24MISS format where: MM=01-12 DD=01-31 YYYY=0000-9999 HH24=00-23 MI=00-59 SS=00-59

Note the “Max Precision” column above. For integral types the precision is roughly equivalent to the size (the caveat being the min and max values in the table); however for floating point types (sfloat and dfloat) the precision indicates how many total digits to the left of the decimal point (assuming that the float is written in scientific form) you can look at before rounding errors are encountered.

Thus, a database column defined as NUMBER(12,4) will have a maximum length of 12, with 4 digits of precision right of the decimal place. For this datatype, an SFLOAT field can be safely used to represent all numbers in that range.

RETL Data Type/Database Data Type Mapping

The following tables show the mapping RETL uses when reading from or writing to a database.

Database data types that are merely synonyms for other data types are supported. For example, a REAL in Oracle is actually a synonym for FLOAT(63), which is supported.

RETL Data Type to Oracle Data Type (ORAWRITE)

RETL Data Type	Oracle Data Type
int8	NUMBER(2,0)
int16	NUMBER(4,0)
int32	NUMBER(9,0)
int64	NUMBER(18,0)
uint8	NUMBER(2,0)
uint16	NUMBER(4,0)
uint32	NUMBER(9,0)
uint64	NUMBER(19,0)
dfloat	NUMBER
sfloat	NUMBER
string	VARCHAR
date	DATE
time	DATE
timestamp	DATE

Oracle Data Type to RETL Data Type (ORAREAD)

Oracle Data Type	RETL Data Type
CHAR, VARCHAR	string
DATE	date or timestamp, depending on value of "datetotimestamp" property
FLOAT, NUMBER, INTEGER, DECIMAL	precision = 0: dfloat scale > 0: dfloat precision > 19: dfloat precision = 19: uint64 precision >= 10, < 19: int64 precision >= 5, < 10: int32 precision >= 3, < 5: int16 otherwise: int8

Appendix G – Data Partitioning Quick Reference

This appendix is a quick reference on how to set up a flow to have multiple data partitions. For more detailed information, see “Chapter 6 – RETL Parallel Processing”.

Partitioning the Data

Data partitioning does not happen automatically in RETL; you need to configure rfx.conf and the operators in the flow.

In rfx.conf, set the “numpartitions” attribute in the NODE element to a value greater than 1. This tells RETL to allow partitioning and also provides a default “numpartitions” for HASH and SPLITTER.

The following table shows the operators that partition data and what must be done to configure the operator:

Operator	Configuration
DBREAD	Set the “numpartitions” property to a value greater than “1”. Specify a query for each partition.
GENERATOR (generating new records)	Set the “numpartitions” property to a value greater than “1”. Set “partnum_incr” and “partnum_offset” if appropriate.
HASH	No configuration required, but “numpartitions” can be used to override the “numpartitions” from rfx.conf.
IMPORT	Set the “numpartitions” property to a value greater than “1”. Specify either one input file or “numpartitions” input files.
ORAREAD	Set the “numpartitions” property to a value greater than “1”. Specify a query for each partition.
SPLITTER	No configuration required, but “numpartitions” can be used to override the “numpartitions” from rfx.conf.

Continuing the Partition

Some operators must be configured to continue a data partition. If the operator is not configured to continue the partition, RETL will end the data partition by inserting a FUNNEL to collect the partitioned records. To configure an operator to continue a data partition, set the “parallel” property to true”

The following operators require the “parallel” property to be set to “true”:

- CLIPROWS
- DBWRITE
- DEBUG
- EXPORT
- GENERATOR (generating new fields)
- GROUPBY
- ORAWRITE

Hash Partitioning

Some operators only work correctly if the data is partitioned using the HASH operator. These are typically operators that work with key fields. Using a HASH operator to perform the data partitioning ensures that records with the same key end up in the same data partition.

The following operators require a data partition started by a HASH operator:

- CHANGECAPTURE
- CLIPROWS
- COMPARE
- DIFF
- FULLOUTERJOIN
- GROUPBY
- INNERJOIN
- LEFTOUTERJOIN
- MERGE
- REMOVEDUP
- RIGHTOUTERJOIN

Ending the Data Partition

RETL will continue a data partition until an operator is encountered that does not support data partitioning (or is configured to not continue a data partition). At this point, RETL inserts a FUNNEL to unpartition the records back into a single dataset.

Because FUNNEL and SORTFUNNEL do not support partitioning, you can insert a FUNNEL or a SORTFUNNEL at any point to end a partition.