Oracle® Retail Extract, Transform, and Load

Programmer's Guide

Release 13.0.2

January 2009

ORACLE®

Oracle Retail Extract, Transform, and Load Programmer's Guide, Release 13.0.2

Primary Author: Susan McKibbon

Value-Added Reseller (VAR) Language

Oracle Retail VAR Applications

The following restrictions and provisions only apply to the programs referred to in this section and licensed to you. You acknowledge that the programs may contain third party software (VAR applications) licensed to Oracle. Depending upon your product and its version number, the VAR applications may include:

(i) the software component known as **ACUMATE** developed and licensed by Lucent Technologies Inc. of Murray Hill, New Jersey, to Oracle and imbedded in the Oracle Retail Predictive Application Server - Enterprise Engine, Oracle Retail Category Management, Oracle Retail Item Planning, Oracle Retail Merchandise Financial Planning, Oracle Retail Advanced Inventory Planning, Oracle Retail Demand Forecasting, Oracle Retail Regular Price Optimization, Oracle Retail Size Profile Optimization, Oracle Retail Replenishment Optimization applications.

(ii) the **MicroStrategy** Components developed and licensed by MicroStrategy Services Corporation (MicroStrategy) of McLean, Virginia to Oracle and imbedded in the MicroStrategy for Oracle Retail Data Warehouse and MicroStrategy for Oracle Retail Planning & Optimization applications.

(iii) the **SeeBeyond** component developed and licensed by Sun MicroSystems, Inc. (Sun) of Santa Clara, California, to Oracle and imbedded in the Oracle Retail Integration Bus application.

(iv) the **Wavelink** component developed and licensed by Wavelink Corporation (Wavelink) of Kirkland, Washington, to Oracle and imbedded in Oracle Retail Mobile Store Inventory Management.

(v) the software component known as **Crystal Enterprise Professional and/or Crystal Reports Professional** licensed by SAP and imbedded in Oracle Retail Store Inventory Management.

(vi) the software component known as **Access Via™** licensed by Access Via of Seattle, Washington, and imbedded in Oracle Retail Signs and Oracle Retail Labels and Tags.

(vii) the software component known as **Adobe Flex™** licensed by Adobe Systems Incorporated of San Jose, California, and imbedded in Oracle Retail Promotion Planning & Optimization application.

(viii) the software component known as **Style Report™** developed and licensed by InetSoft Technology Corp. of Piscataway, New Jersey, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.

(ix) the software component known as **DataBeacon™** developed and licensed by Cognos Incorporated of Ottawa, Ontario, Canada, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.

You acknowledge and confirm that Oracle grants you use of only the object code of the VAR Applications. Oracle will not deliver source code to the VAR Applications to you. Notwithstanding any other term or condition of the agreement and this ordering document, you shall not cause or permit alteration of any VAR Applications. For purposes of this section, "alteration" refers to all alterations, translations, upgrades, enhancements, customizations or modifications of all or any portion of the VAR Applications including all reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The VAR Applications contain trade secrets of Oracle and Oracle's licensors and Customer shall not attempt, cause, or permit the alteration, decompilation, reverse engineering, disassembly or other reduction of the VAR Applications to a human perceivable form. Oracle reserves the right to replace, with functional equivalent software, any of the VAR Applications in future releases of the applicable program.

Contents

3 RETL Interface

4 RETL Program Flow

5 RETL Schema Files

6 Database Operators

7 RETL Parallel Processing

8 Input and Output Operators

9 Join Operators

10 Sort, Merge, and Partitioning Operators

G Appendix: Database Connections Quick Reference

Preface

The Oracle Retail Extract Transform and Load (RETL) Programmer's Guide contains a complete description of the RETL programming language, as well as installation, configuration, and maintenance instructions.

Audience

The RETL Programmer's Guide is intended for all RETL flow developers.

Related Documents

For more information, see the following document in the Oracle Retail Extract, Transform, and Load Release 13.0.2 documentation set:

- *Oracle Retail Extract, Transform, and Load Release Notes*

Customer Support

- https://metalink.oracle.com

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

Review Patch Documentation

If you are installing the application for the first time, you install either a base release (for example, 13.0) or a later patch release (for example, 13.0.2). If you are installing a software version other than the base release, be sure to read the documentation for each patch release (since the base release) before you begin installation. Patch documentation can contain critical information related to the base release and code changes that have been made since the base release.

Oracle Retail Documentation on the Oracle Technology Network

In addition to being packaged with each product release (on the base or patch level), all Oracle Retail documentation is available on the following Web sit e (with the exception of the Data Model, which is only available with the release packaged code):

http://www.oracle.com/technology/documentation/oracle_retail.html

Documentation should be available on this Web site within a month after a product release. Note that documentation is always available with the packaged code on the release date.

Conventions

Navigate: This is a navigate statement. It tells you how to get to the start of the procedure and ends with a screen shot of the starting point and the statement "the Window Name window opens."

Note: This is a note. It is used to call out information that is important, but not necessarily part of the procedure.

```
This is a code sample
     It is used to display examples of code
```

A hyperlink appears like this.

1
Introduction

Oracle Retail Extract Transform and Load (RETL) is a high-performance, scalable, platform-independent, parallel processing data movement tool. RETL addresses several primary needs:

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

To provide for these needs, 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 RETL supports.

RETL is a cross-platform development tool supported on Oracle Enterprise Linux. The XML definitions do not change on a per-platform basis, so moving between hardware platforms is simplified.

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

This guide shows the application developer how to rapidly deploy RETL in an application, and how to use it to manage your parallel processing system. This guide provides information on how 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 platform, and on any other system that serves as a source or target of data.

Technical Specifications

RETL is certified on the platform listed in this section. For historical reasons, the executable for RETL is called rfx. That executable name is used in this document.

The current configuration included with the RETL install package is documented in the Release Notes. If you have a configuration that is not included in the Release Notes, verify with Oracle Retail Customer Support whether your configuration is supported.

Supported Operating Systems

Supported On	Versions Supported
Database OS	■ Oracle Enterprise Linux 4 patch 5 ■ HP Itanium 11.31 ■ Solaris 10 ■ AIX 5.3
Database Server	Oracle Database 10g (10.2.0.3)

Supported Oracle Retail Products

Supported On	Versions Supported
RMS 13.0.2	RETL 13.0.2
RDW 13.0.2	RETL 13.0.2
RPM 13.0.2	RETL 13.0.2
ReIM 13.0.2	RETL 13.0.2

Data Integration

Integrate external data of these forms:

- UNIX flat files
- Oracle database tables

2

Installation and System Configuration

Installation

Install RETL on each server system that will be involved in input, output, 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.

Follow these steps to install RETL:

1. Log in as the `root` user on the host.
2. Create a UNIX group for the 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 Oracle Retail 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 the UNIX prompt, enter:

   ```
   >./install.sh
   ```

   Note: You must be in the *<cdrom_path>/<rfx_dir>* for the installation to complete successfully.
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...
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. Installation of RETL 13.0.2 version on Sun Solaris, IBM AIX and HP-UX platforms requires the installation of Java Runtime Environment 1.5 (JRE). JRE 1.5 is required for RETL to function properly. The Java Runtime Environment 1.5 (JRE) can be downloaded for the corresponding platforms at the websites listed below.

    For Sun Solaris: `http://java.sun.com/javse/downloads/index_jdk5.jsp`

    For IBM AIX: `http://www.ibm.com/developerworks/java/jdk/`

    For HP-UX: `http://h18012.www1.hp.com/java/download/`

    However, installation of RETL 13.0.2 version on Linux platform does not require the installation of Java Runtime Environment 1.5 (JRE). You do need not install JRE.

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

13. Set up your environment. See the "Setup" section that follows.

14. Verify the installation and setup by running the "verify_retl" script (see "RETL Package Verification Tool (verify_retl)").

Note: Creating Symbolic Link for Gsort under Linux OS.

On the Linux platforms, a link is to be created in the $RFX_HOME/bin that links the default linux sort to gsort. This can be done as below.

- The user can find where sort resides on his Linux box using "which sort" command.
- Then create a symolic link for gsort in $RFX_HOME/bin folder pointing to $RFX_HOME/bin/gsort.

Example: Lets us assume that on Linux box the sort resides under /usr/bin/sort. Hence the user can create the symbolic link for gsort pointing to $RFX_HOME/bin/gsort as

ln -s /usr/bin/sort $RFX_HOME/bin/gsort under $RFX_HOME/bin folder.

Setup

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 have selected an installation with database support, set up database and environment variables required for basic database setup. Please refer to the Appendix B, "Appendix: 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 the following:

   ```
   >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 can be made to `rfx.conf` as DEFAULTS for convenience. See 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 Field Descriptions" 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 has the following format:

   ```
   <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 as follows:

   ```
   <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:** If you have problems, see the troubleshooting information in Appendix C, "Appendix: Troubleshooting Guide".

Upgrading from Earlier Releases of RETL

RETL releases are required to be backward compatible with the XML flow interface of earlier releases. As a result, minimal changes to a setup should be necessary to upgrade. These are the steps to follow:

1. Choose a new (and different) location to install the new version of RETL.
2. Install the new version of RETL using the previous installation instructions.
3. Change the environment in which RETL runs so that it refers to the new RFX_HOME (for example, change within .kshrc or .cshrc).
4. Double check the environment variables to make sure that you were not explicitly referring to old RETL directories.
5. Read the "Backward Compatibility Notes" section to determine if you need to change any flows or scripts.
6. Make any changes necessary to correct your flows, so that you can run in the new version.
7. Verify that your flows and scripts run as expected. If any scripts or flows fail in running with the new release, but pass in running with the old release, and changes are not otherwise noted in the "Backward Compatibility Notes" section, Appendix D, "Appendix: FAQ", or RETL Release Notes, notify Customer Support.

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

RETL Package Verification Tool (verify_retl)

When setting up RETL in a new environment (or if you would 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. (The `verify_retl` script 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 stand-alone 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.

Backward Compatibility Notes

A major requirement for RETL releases is that they be backward-compatible in the XML interface with earlier releases. There are a few small changes that must be made for certain operators when upgrading to RETL 11 or RETL 12 from 10.x releases of RETL.

XML Flow Interface/Operator Differences Between 10.x and Later Releases

Operator	Property	Backward Compatibility Notes
FILTER	filter	DEPRECATED SYNTAX—RETL 10.x versions produced warning messages to correct 'filter' syntax. Later versions do not accept the following syntax in the filter property : >, <, >=, <=, =. These operations are replaced by GT, LT, GE, LE, EQ, respectively.

ORAREAD	query	INVALID SYNTAX—RETL 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 later 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 later 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.

ORAREAD/ ORAWRITE	dbname	NEW REQUIRED PROPERTY—The database to which to connect. This property should be used instead of sid from now on.
	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. This may need to be specified only once in the `rfx.conf` configuration file for convenience.
ORAWRITE	jdbcdriver	NEW PROPERTY—The type of JDBC driver required to connect to database. **Note:** By default, the operator creates the URL for database connection as a thin client. This can be overriden to create the URL as an OCI client by adding this property in the XML flow file. An example follows: `<PROPERTY name="jdbcdriver" value="oci" />`
	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. This may need to be specified only once in the `rfx.conf` configuration file for convenience.

Examples (in operator for ORAWRITE):

Connection to database through THIN database driver:

```
<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>
```

Connection to database through OCI database driver:

```
<OPERATOR type="orawrite">
    <INPUT name="test.v" />
         <PROPERTY name="threadModel" value="start_thread" />
         <PROPERTY name="dbuserid" value="username/password" />
         <PROPERTY name="dbname" value="databasename" />
         <PROPERTY name="jdbcdriver" value="oci" />
         <PROPERTY name="tablename" value="mytable" />
         <PROPERTY name="createtablemode" value="recreate" />
</OPERATOR>
```

Example (in `rfx.conf`):

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

Hardware Requirements Differences Between RETL 10 and Later Versions

In general, more physical memory is required in order to run the later versions of RETL than in RETL 10. There is no general formula or guideline for the additional memory requirement, because it strongly correlates to the flow, data, configuration, and so on.

rfx Command Line Options

You can get help on rfx options on the command line by entering the following on the command line:

```
rfx -h
```

The output is 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 described in the following table:

Option	Default Value	Description
`-h`		Shows the help message shown above.
`-oOPNAME`		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 in Chapter 4, "RETL Program Flow" for more information about this option.
`-e`		Prints RETL environment variables that can be used for things such as turning on verbose debugging, setting JVM parameters, and so on. See "RETL Environment Variables" for more information.
`-V`		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 "Logger 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. See "Producing Graphical Output of Flows with RETL" in Chapter 4, "RETL Program Flow" for more information on how to use this option.

For more information on DOTTY, see the following:
http://www.research.att.com/sw/tools/graphviz/download.html

RETL Environment Variables

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

```
export VARIABLE=VALUE
```

Option	Description
RFX_DEBUG	Option to turn on verbose RETL debugging to standard output. Set to "1" to turn on. Default is "0".
RFX_SHOW_SQL	Option to turn on verbose RETL database debugging to standard output. Set to "1" to turn on. Default is "0".
RETL_ENABLE_ASSERTIONS	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.

Option	Description
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 Oracle Retail Support, or if the user is aware of the implications of setting JVM parameters and has tested the results of making any changes.
RETL_TMP	Set for RETL flow temporary files (for example, temporary flow files from 'here' documents). Defaults to "." or the local directory where the script is run from. This can be either an absolute path name or relative to your current directory.
RETL_ENABLE_64BIT_JVM	Option to enable 64bit JVM. Set to 1 to enable a 64bit JVM if available for the particular platform.

Configuration

You can specify a configuration file to control how RETL uses system resources, where to store temporary files, and to set default values for operators.

Configuration Field Descriptions

These descriptions will assist you in modifying your configuration file. The RETL configuration file `rfx.conf` is located in the *<base_directory>*`/etc` directory.

Element Name	Attribute Name	Attribute Value/Description
CONFIG		CONFIG is the root element of the RETL Configuration file. This element can have either NODE or DEFAULTS elements.
NODE		NODE is a child element of the CONFIG element.
	hostname	The name of the UNIX server where RETL is installed.
	bufsize	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	Optional value, defaults to 1. See the following sections on partitioning.
TEMPDIR		TEMPDIR is a child element of the NODE element. This element can be specified more than one time. RETL will use the temporary file directories in a round-robin fashion. See "Temporary Space Configuration".
	path	Path to the directory where the RETL writes temporary files.
GLOBAL		The GLOBAL element is a child of CONFIG and specifies values for global settings within RETL.
	bytes_per_character	This setting specifies the maximum number of bytes that are in a character. Setting this value is necessary for allowing RETL to work with UNICODE data. See "Multibyte Character Support".

Element Name	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	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.
	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.

Element Name	Attribute Name	Attribute Value/Description
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	Operator type Name of the operator to assign default values. Refer to the following chapters for the operators that are available.
PROPERTY		The PROPERTY element is a child of the DEFAULTS element.
	name	The name of the operator property to assign a default value. See the following chapters for the property names for each operator.
	value	The value assigned as the default for the specified operator property. See the following chapters for the 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>
```

Temporary Space Configuration

The best performance can probably be attained when the number of temporary directories specified in the configuration file is equal to the number of partitions. Ideally, each temp directory should be on a separate disk controller.

> **Note:** These directories must always be local to the host where RETL is running. Use of network drives can have a drastic impact on performance.

By default, TEMPDIR is set to `/tmp`. This should be changed to be a local disk after installation, because `/tmp` on many platforms is a memory device used for O/S swap space. If this is not changed, the system could be exhausted of physical memory.

Take care to protect the files within this directory, because they can contain userid and password information. Check with your system administrator about setting the permissions so that only the appropriate personnel can access these files (for example, by setting: umask 077).

Temporary files should be removed from temporary directories on a daily or weekly basis. This important server maintenance task aids in RETL debugging, reviewing database-loading utility log files, and other activities. If a RETL module fails, you should not rerun the module until after removing the temporary files that were generated by the failing module.

Logger Configuration

Logging in to RETL is performed using the log4j logging facility. log4j is a 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. For more information about log4j, refer to the following:

`http://logging.apache.org/log4j/docs/documentation.html`

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 `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 is 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. You can configure the output logger to change the destination for these messages.

If you want to use any of the advanced features of log4j, change the output logger 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 display them 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 above example 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.

Multibyte Character Support

The bytes_per_character setting in the configuration file allows RETL to provide multibyte character support. While this variable is optional, it is required to ensure proper processing of data containing multibyte characters.

The setting of bytes_per_character is used mainly in the parsing of the schemas used by IMPORT and EXPORT. The schema files allow field length specification in the number of characters; RETL uses the bytes_per_character setting to convert the field length from a number of characters into a number of bytes.

The field length is used in several other operators, so the setting of bytes_per_character indirectly affects those operators as well. Most notably, it affects ORAWRITE. When ORAWRITE is configured to create or recreate the target table, RETL uses the field lengths to define the length of the table columns. For example, if bytes_per_character is set to 3, and a field is specified with a length of 10 characters in the IMPORT schema, the resulting database column will be 30 bytes wide.

No corresponding conversion takes place when data is read from an Oracle database, because the database reports the field lengths in bytes.

Note: The Oracle database reports the length of string constants in a query in the number of characters, instead of the number of bytes as it does with the table columns. RETL does not correctly compensate for the different units of measurement. Using a string constant with multibyte characters in a query is therefore highly discouraged.

For example, the example query below contains a string constant with multibyte characters in FIELD1. The Oracle database reports the length of the FIELD1 field as 2 characters instead of 6 bytes, so RETL has an incorrect field length for FIELD1.

```
SELECT
    '店舗' FIELD1,
    FIELD2
FROM
      TABLE1
```

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 that allows 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 their 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 Web site 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 the name 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 the type attribute specifies what type of operator is being specified. The operator type is not case-sensitive, although all lowercase characters are recommended. The different types of operators are described 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 attribute specifies the name of the property, and the value attribute specifies the value of the property. The valid properties for each operator are detailed later in this document.
- The input datasets 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 datasets 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 together logically related operators, thus reducing 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 does 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 (that is, 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, and 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 run-time 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 run-time parameters. See the Javadoc for more information.

Flows

Just like the XML interface, the Java interface runs flows. The difference is that in the Java interface, a flow is an instance of the Flow class. Operators are added to the flow using the 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 functionality is contained in a class named Import. The operators are documented in Chapters 7 through 13 of this guide.

The class name is always an uppercase first letter followed by lowercase 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 that has 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()`. There are only setter methods; no getter methods are 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);
    }
```

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

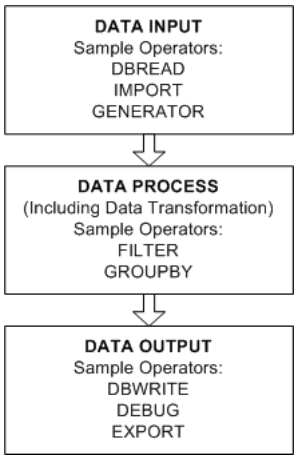
Data process operators commonly include those that transform data in the dataset being processed, including GROUPBY, which can be used to sum values in a table column.

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

General Flow

The following diagram 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.



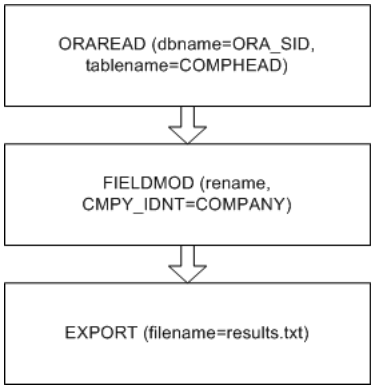
Some things to note about all flows:

- 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 such as an Oracle database or flat file.
- Similarly, DATA OUTPUT operators are terminal points within the flow. They do not generate output datasets. They export records to an external repository (for example, Oracle databases or flat files).

The next sections describe specific flows.

A Simple Flow

The following diagram 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 operator. Likewise, the arrow from FIELDMOD to EXPORT indicates that the records from the FIELDMOD operator are passed onto the EXPORT operator.



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

Here is XML for the above flow:

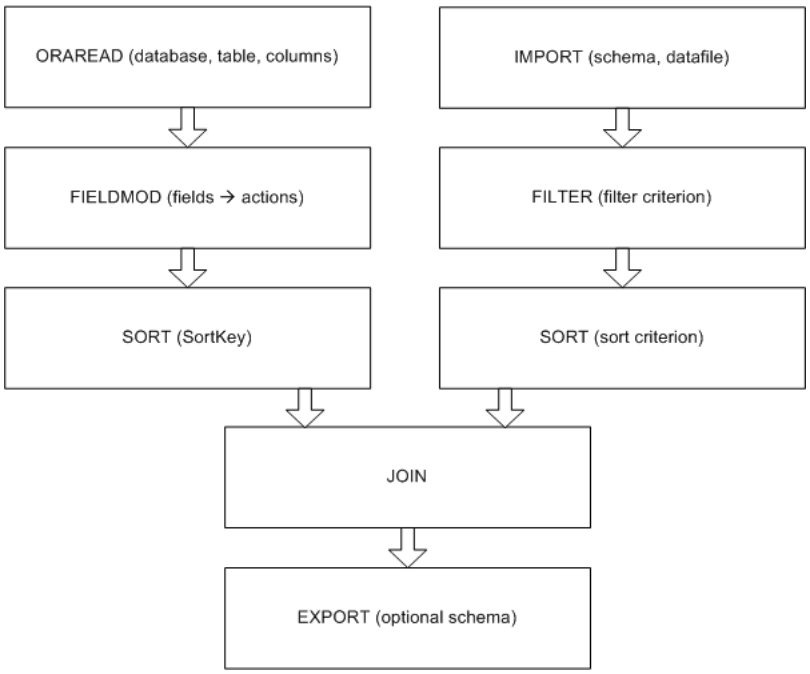
```
<FLOW name="cmpy.flw">
    <OPERATOR type="oraread">
    <PROPERTY name="dbname"        value="ORA_SID"/>
    <PROPERTY name="connectstring" value="userid/passwd"/>
    <PROPERTY name="query">
         <![CDATA[
              select * from COMPHEAD
         ]]>
    </PROPERTY>
    <OUTPUT name = "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 previously.

The following flow pulls in data simultaneously from a flat file and an 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 using the FIELDMOD operator, and then sorted. The resulting datasets are joined, and the results are output to a flat file using the EXPORT operator.



Note the following:

- RETL can deal with multiple data sources within a single flow.
- RETL can output results to multiple files or database tables.

Online Help

RETL has an online help system you can use to obtain operator syntax and usage information directly from the rfx binary. The following help is available:

- 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 the property is optional)

For more information about this feature, see information about the -o option in the "rfx Command Line Options" section in Chapter 2, "Installation and System Configuration".

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 using the following variables:

  ```
  export RFX_DEBUG=1
  export RFX_SHOW_SQL=1
  ```

- RFX_DEBUG turns on additional informational messages about RETL operations to standard output.
- RFX_SHOW_SQL turns on additional logging of database operations to standard output.

For more information about printing in schema file format, see "Configure RETL to Print Schema Output in Schema File Format" in Chapter 5, "RETL Schema Files".

For more information about printing graphical output of flows, see "Producing Graphical Output of Flows with RETL".

For more information about performance logging, see "Logger Configuration" in Chapter 2, "Installation and System Configuration".

Producing Graphical Output of Flows with RETL

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

1. Enter the following:

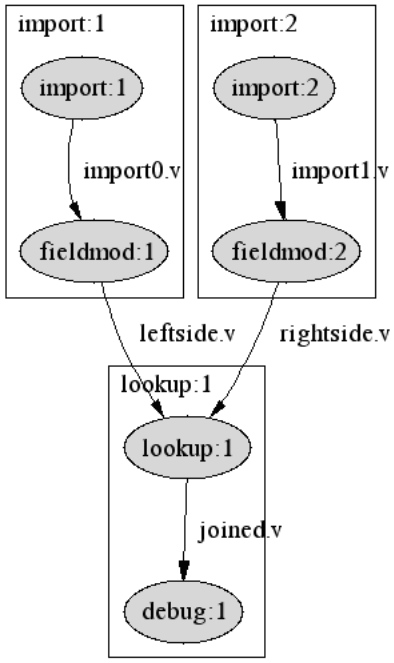
```
rfx -f <flow name.xml> -g
```

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" />
   </OPERATOR>
   <OPERATOR type="debug">
     <INPUT name="joined.v"/>
   </OPERATOR>
</FLOW>
```



This is an invaluable tool when debugging a RETL flow. For example, RETL might return the following error:

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

To determine which lookup operator the error came from, follow these steps:

1. Print a RETL flow graph (see RETL syntax above).
2. Locate the operator with the name lookup:1 in the flow graph. (This can also 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, lookup:1 has links named leftside.v and rightside.v as INPUTs, and joined.v as OUTPUT. You can use any of these to search for the appropriate lookup in the flow, because 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:

- Processing time for each operator
- Count of processed records for each operator
- Throughput in records per second for each operator
- 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. Also, 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:

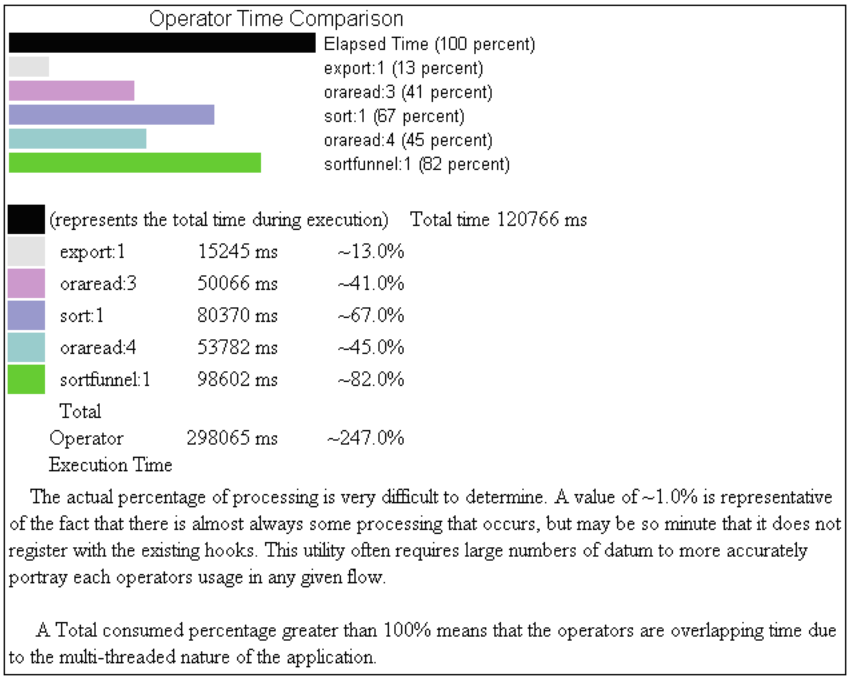
```
All threads complete

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
```

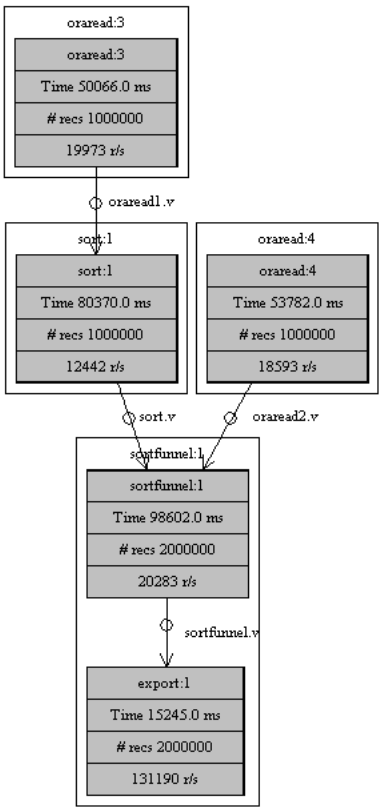
Sample Log File

```
[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



5

RETL Schema Files

RETL can process datasets directly from a database or from file-based systems. Depending on the data source, one of two operators can perform this function:

- For reading data directly from database tables, use the ORAREAD operator. See Chapter 6, "Database Operators" for more information about how to use ORAREAD.
- For situations in which 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 of 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. 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 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 Name	Name	Value/Description
RECORD		RECORD is the root tag of the schema file.
	type	"delimited" or "fixed" Required field. Specified whether the schema is fixed records or delimited records.
	final_ delimiter	Any 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 if the type attribute is "fixed". 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 what fields the records are sorted by. This should be used only when it is known that the records will always be sorted by specified keys. 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. See the "sortkeys" attribute.

Element Name	Name	Value/Description
FIELD		FIELD is a child element of RECORD.
	name	The name of the field.
	delimiter	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 ('|').
		The delimiter can be specified in hexadecimal notation by prefixing "0x" to the ASCII value of the character. This allows specification of a delimiter that is highly unlikely to appear in the data, such as 0x02.
	datatype	Any one of: "int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64", "dfloat", "sfloat", "string", "date", "time", or "timestamp"
		See Appendix E, "Appendix: RETL Data Types", especially the note in RETL data type to Oracle data type (ORAWRITE).
	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	Required if nullable is "true". This is the text string that an IMPORT operator uses when reading flat files to determine whether a field is null.
		This should be a unique value that is 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 ("") can 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	Specifies the maximum allowable length of the string field in characters. Required only for delimited string fields.
	len	Specifies the length of the field in characters. Required for all fixed length fields.
	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 that 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.

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 corrupt or invalid.

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

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, take care when selecting values for a field's nullvalue. Unless you are assigning a default value to a null field on export, do not use a value that can 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, null values precede all other values in ascending order and follow all 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 schema files are the interface between flows that tell RETL how to read and write files. Writing schema files manually is tedious and error-prone. It is highly recommended that developers use the -sSCHEMAFILE command line option to speed up development. For more information about this feature, see the "rfx Command Line Options" section in Chapter 2, "Installation and System Configuration".

6

Database Operators

ORAREAD

ORAREAD performs a read from Oracle databases. RETL 12 uses JDBC technology to read data out of Oracle databases.

ORAWRITE

ORAWRITE performs a load to Oracle databases. RETL 12 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 do not exist in the database table. Also, the incoming record schema determines what is set for nullabilities 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 relies on SQL*Loader to throw errors or reject inconsistent records. This only happens when appending to an existing table and does not affect database writes when specifically creating or 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. For instance:

- 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 that 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).

ORAREAD

Element Name	Name	Value
PROPERTY	dbname	Name of the Oracle database.
PROPERTY	connectstring	username/password
PROPERTY	maxdescriptors	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.
PROPERTY	datetotimestamp	This is an optional property. When it is 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 can 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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.

Element Name	Name	Value
PROPERTY	sp_prequery	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, and it does not allow data to be returned.
PROPERTY	sp_postquery	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, and it does not allow data to be returned.
PROPERTY	numpartitions	Optional property indicating the number of data partitions to create. A query must be specified for data partition.
PROPERTY	arraysize	Optional property indicating the number of rows to pre-fetch. 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.

PROPERTY	useString	Optional property used for testing the latin type characters.
		Note: When this property is added with value as true, the method used will be `getstring( )`. The default method used is `getbytes( )`. In the flow XML, add the property as follows.
		`<PROPERTY name="useString" value="true" />`
		`//`
		By default it is false.

OUTPUT		The output dataset name.

Note: You can connect to database using the ORAREAD operator as either a thin or oci client. To connect to the database as an OCI client, add the following URL in the flow XML.

```
<PROPERTY name="jdbcdriverstring" value="oracle.jdbc.driver.OracleDriver" />
<PROPERTY name="jdbcconnectionstring" value="jdbc:oracle:oci:@" />
```

ORAWRITE

Element Name	Name	Value
PROPERTY	Dbname	Name of the Oracle database.
PROPERTY	dbuserid	username/password
PROPERTY	maxdescriptors	Maximum number of columns allowed to be written to the database.
PROPERTY	schemaowner	The database owner of the table.
PROPERTY	tablename	Name of the Oracle table to which the data is written.

Element Name	Name	Value
PROPERTY	Method	"direct" or "conventional" This is an optional property to specify the loading method. The default is "conventional". "direct"—Load the data using the 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 does not exist. It will create the table before writing the data only if the table does not exist.
PROPERTY	jdbcdriver	Connects to the database through the specified database driver, for example, Oracle Call Interface (OCI). This is an optional property that defaults to THIN.
PROPERTY	allowedrejects	A value greater than zero that indicates how many errors are allowed before SQL*Loader exits with a critical error. This is an optional property that defaults to 50.
PROPERTY	exit_threshold	"warning", "exceeded_rejects", or "fail_or_fatal" Optional property. Allows configuration of the level of errors or warnings from SQL*Loader that 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".

Element Name	Name	Value
PROPERTY	numloaders	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 to maximize performance. For example, if running 8 ways parallel, 8 distinct temporary directories should be specified to maximize performance.
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 it 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 (for example, if numpartitions is set to 4, then 4 SQL*Loaders will be run in parallel). See the "Configuration" section in Chapter 2, "Installation and System Configuration" for more on setting numpartitions. To override parallelism in ORAWRITE, set parallel to "false" and set numloaders to the desired number of concurrent SQL*Loaders. When running ORAWRITE in parallel mode, there are restrictions imposed by SQL*Loader. Refer to Oracle SQL*Loader syntax for more information on these limitations.
PROPERTY	tempdir	The temporary directories to use for the SQL*Loader data files. May be specified multiple times by specifying separate properties. **Note:** To maximize performance, temporary directories should reside on separate disk controllers when specifying tempdir with parallel SQL*Loaders through the numloaders property.
PROPERTY	partition	Optional name of the partition of the Oracle table to which data is written.
PROPERTY	outputdelimiter	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 are not populated with data. The outputdelimiter should never be set to a value that can be part of the data.

Element Name	Name	Value
PROPERTY	loaderoptions	Optional. Name-value pair options to be passed to SQL*Loader through 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 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	Number of rows per commit. **Deprecated:** Use rows=numrows in the loaderoptions property.
PROPERTY	controlfileload options	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	Name of an existing database index. 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.
PROPERTY	singlerow	Specify "yes" to update indexes as each new row is inserted. Default value is "no". **Note:** The singlerow property 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 SQL*Loader documentation for more information about the SINGLEROW option and when it makes sense to use it.
PROPERTY	preload	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, and it does not allow data to be returned. This is useful to drop indexes prior to a load, for example.

Element Name	Name	Value
PROPERTY	postload	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, and it does not allow data to be returned. This is useful to rebuild indexes after a load.
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 can be specified as a default in rfx.conf for convenience.
PROPERTY	port	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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
PROPERTY	tablespace	Tablespace name
PROPERTY	control_file	File name of a valid a SQL*Loader control file that RETL should specify to SQL*Loader, instead of generating a control file dynamically. Using this property provides the most control over the SQL*Loader process. See the SQL*Loader documentation for information on the control file options. The file specified by this property is not deleted when the load is complete, unlike the dynamically generated control file used when this property is not specified.
PROPERTY	data_file	File name to use for the SQL*Loader data file. Using this property allows you to control where the file resides.
INPUT		The input dataset name.

UPDATE

Element Name	Name	Value
PROPERTY	dbname	Name of the Oracle database.
PROPERTY	sid	Alias for dbname.
PROPERTY	dbuserid	username/password
PROPERTY	maxdescriptors	Maximum number of columns allowed to be written to the database.
PROPERTY	schemaowner	The database owner of the table.

Element Name	Name	Value
PROPERTY	tablename	Name of the Oracle table to which the data is written.
PROPERTY	table	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 can be specified as a default in rfx.conf for convenience.
PROPERTY	port	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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
PROPERTY	fields	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.
PROPERTY	batchcount	Number of records to process between commit calls. Default value is 1000.
INPUT		The input dataset name.

DELETE

Element Name	Name	Value
PROPERTY	dbname	Name of the Oracle database.
PROPERTY	sid	Alias for dbname.
PROPERTY	dbuserid	username/password
PROPERTY	maxdescriptors	Maximum number of columns allowed to be written to the database.
PROPERTY	schemaowner	The database owner of the table.
PROPERTY	tablename	Name of the Oracle table to which the data is written.
PROPERTY	table	Alias for tablename.

Element Name	Name	Value
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 can be specified as a default in rfx.conf for convenience.
PROPERTY	port	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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
PROPERTY	fields	Space separated list of fields to use for selecting records to delete.
PROPERTY	batchcount	Number of records to process between commit calls. Default value is 1000.
INPUT		The input dataset name.

INSERT

Element Name	Name	Value
PROPERTY	dbname	Name of the Oracle database.
PROPERTY	sid	Alias for dbname.
PROPERTY	dbuserid	username/password
PROPERTY	maxdescriptors	Maximum number of columns allowed to be written to the database.
PROPERTY	schemaowner	The database owner of the table.
PROPERTY	tablename	Name of the Oracle table to which the data is written.
PROPERTY	table	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 can be specified as a default in rfx.conf for convenience.

Element Name	Name	Value
PROPERTY	port	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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
PROPERTY	fields	Space-separated list of fields to insert into the database table. Defaults to all fields.
PROPERTY	batchcount	Number of records to process between commit calls. Default value is 1000.
INPUT		The input dataset name.

PREPAREDSTATEMENT

Element Name	Name	Value
PROPERTY	dbname	Name of the Oracle database.
PROPERTY	sid	Alias for dbname.
PROPERTY	dbuserid	username/password
PROPERTY	maxdescriptors	Maximum number of columns allowed to be written to the database.
PROPERTY	schemaowner	The database owner of the table.
PROPERTY	tablename	Name of the Oracle table to which the data is written.
PROPERTY	table	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 can be specified as a default in rfx.conf for convenience.
PROPERTY	port	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 preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can be used to obtain the port number. Verify with your database administrator the proper port for your database installation.
PROPERTY	statement	SQL to execute for each input record. Use ? to denote a value to be replaced by a field from the current record.
PROPERTY	fields	Space separated list of fields to use as the values for the ? placeholders in the statement property.

Element Name	Name	Value
PROPERTY	batchcount	Number of records to process between commit calls. Default value is 1000.
INPUT		The input dataset name.

Database Operator Examples

ORAREAD

Connection to database through THIN database driver:

```
<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>
```

Connection to database through OCI database driver:

```
<OPERATOR type="oraread">
    <OUTPUT name="oraread.v" />
    <PROPERTY name="connectstring" value=" username/password" />
    <PROPERTY name="dbname" value=" RETLdb " />
    <PROPERTY name="jdbcdriverstring"value="oracle.jdbc.driver.OracleDriver" />
    <PROPERTY name="jdbcconnectionstring" value="jdbc:oracle:oci:@" />
    <PROPERTY name="query" value="select * from test_jdbc"/>
</OPERATOR>
```

ORAWRITE

Connection to database through THIN database driver:

```
<OPERATOR type="orawrite">
    <INPUT name="import0.v" />
    <PROPERTY name="threadModel" value="start_thread" />
    <PROPERTY name="dbuserid" value="username/password" />
    <PROPERTY name="dbname" value="RETLdb" />
    <PROPERTY name="tablename" value="ORG_LOC_DM" />
    <PROPERTY name="createtablemode" value="recreate" />
</OPERATOR>
```

Connection to database through OCI database driver:

```
<OPERATOR type="orawrite">
    <INPUT name="import0.v" />
    <PROPERTY name="threadModel" value="start_thread" />
    <PROPERTY name="dbuserid" value="username/password" />
```

```
    <PROPERTY name="dbname" value="RETLdb" />
    <PROPERTY name="jdbcdriver" value="oci" />
    <PROPERTY name="tablename" value="ORG_LOC_DM" />
    <PROPERTY name="createtablemode" value="recreate" />
</OPERATOR>
```

> **Note:** The steps to connect to an Oracle database using the Oracle THIN/OCI driver are described in Appendix G, "Appendix: Database Connections Quick Reference".

UPDATE

Suppose that you have dataset with records reflecting new employee salaries with the following fields:

- EMPLOYEE_ID
- SALARY
- EFFECTIVE_DATE

To update a table named EMPLOYEE_SALARY with the new values using EMPLOYEE_ID as the key field, 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 that 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. 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 that you have a dataset with records containing purchase order information that you need inserted into the PURCHASE_ORDER table. 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 as follows:

```
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 as follows:

```
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 preceding example 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>
```

7

RETL Parallel Processing

RETL Parallelism Overview

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

- Pipeline parallelism
- Framework parallelism and data partitioning

Pipeline Parallelism

Pipeline parallelism can be demonstrated in manufacturing by an assembly line, where each worker can operate on 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 (after each worker has product to work on). This is similar RETL pipeline parallelism. 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, or merge), and they execute these tasks independently of other operators' tasks.

Framework Parallelism

While pipeline parallelism is a means of increasing throughput, it only works to the point at which each worker reaches capacity. Some workers may be slower at their tasks than others, resulting in bottlenecks in the process. In the manufacturing world, a way of solving this dilemma is to increase the number of assembly lines. Now each assembly line can work independently of other assembly lines, uninhibited by bottlenecks. This is similar to RETL's framework-based parallelism. RETL implements framework-based parallelism in a concept called data partitioning. Data partitioning splits data from the source and passes each chunk of data to separate and concurrent pipelines.

Where can bottlenecks arise? In 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 to fully maximize data partitioning and be uninhibited by task bottlenecks. Starting with RETL release 11.2 and higher, RETL supports this additional enhancement to parallelize input and output tasks, by multithreading file and database reads and writes.

RETL uses both framework data partitioning and pipeline parallelism to achieve overall speed improvements 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 that you are familiar with the graphical representation of a flow. (See Chapter 4, "RETL Program Flow" for more information.)

RETL data partitioning, or simply partitioning, is the process RETL uses to increase the number of operators that are processing records, in an attempt to increase overall throughput. Partitioning consists of splitting 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

You enable partitioning in one of two ways:

- By 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.
- By 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 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.

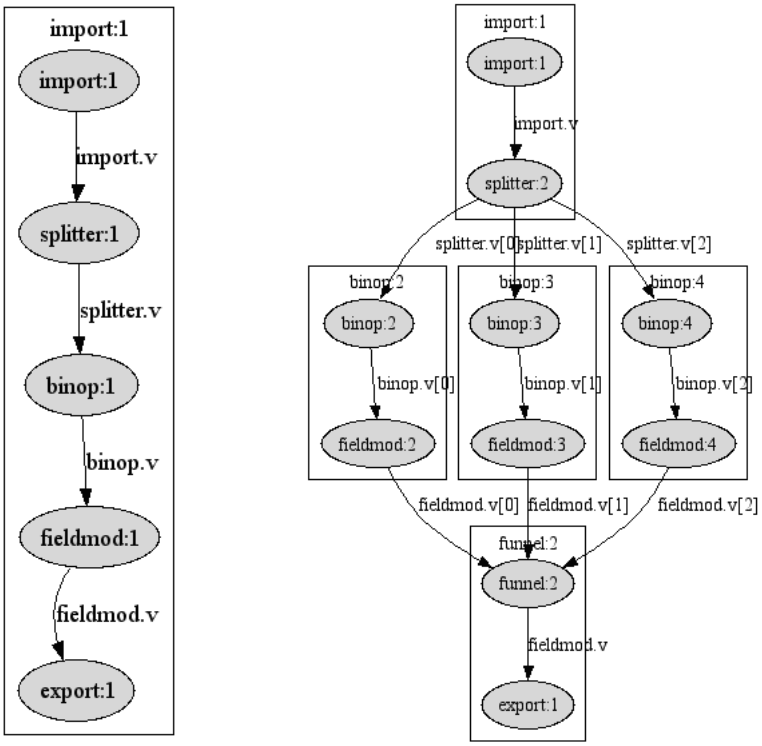
A partition pipeline ends when an operator is encountered that does not support partitioning, or does not support the type of partitioning performed by the partitioner. If necessary, RETL inserts a funnel in order to "unpartition" the data subsets back into a single dataset.

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 like this example:

```
<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.
- The 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.

Nonkeyed Partitioning

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

Partitioners

The following are the 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:** 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.

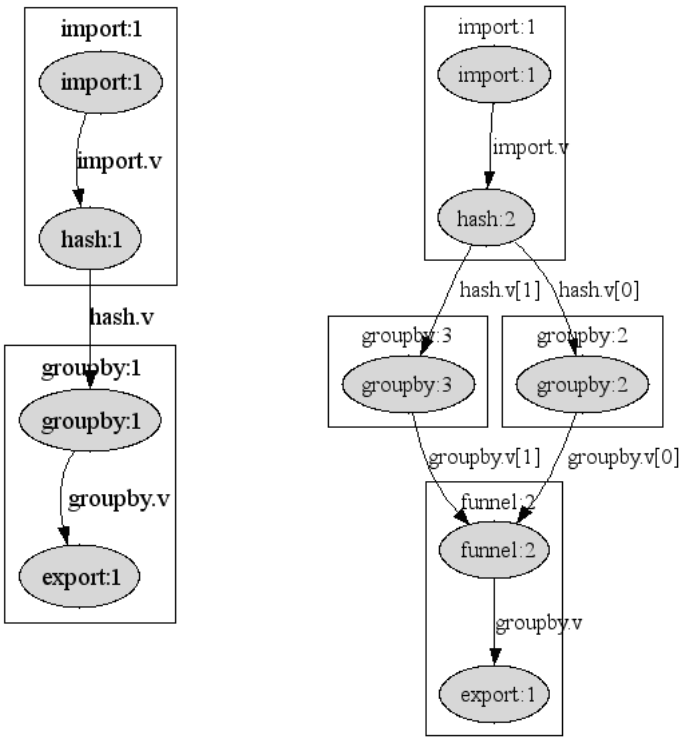
If the numpartitions property is not specified in the HASH operator, RETL uses the value from the -n command line option, if specified. If the -n command line option is not specified, the value from the configuration file is used.

In the following example, 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

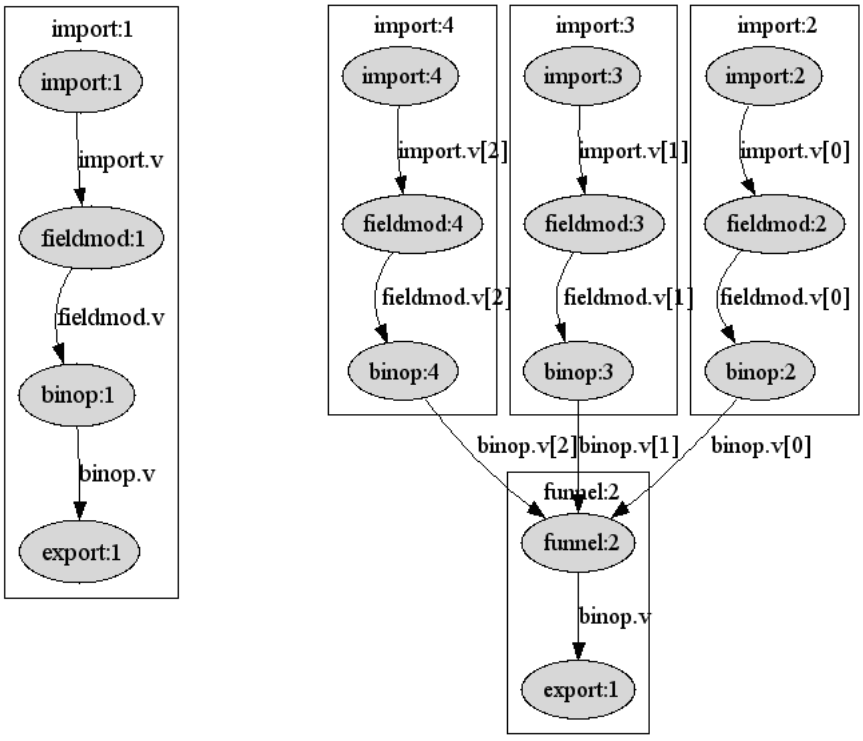
IMPORT partitioning performs nonkeyed partitioning. If one input file is specified, then roughly equal portions of the file are handled by each partition. If more than one file is specified, the number of partitions must equal the number of files, and each file is handled by one partition.

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 following example, 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 nonkeyed partitioning by sending records to the different partitions in round-robin fashion.

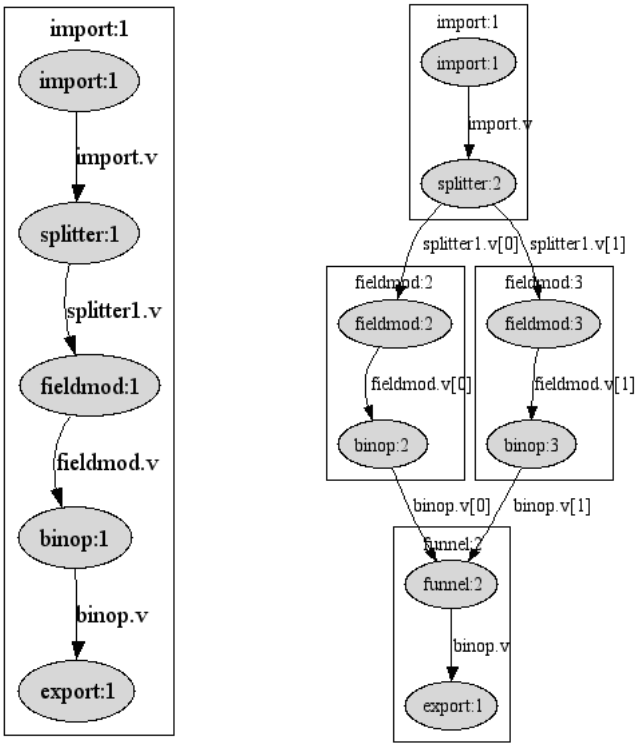
> **Note:** Although SPLITTER will be supported in future versions of RETL, it is not guaranteed that it will use the round-robin algorithm. Flows should not rely on SPLITTER partitioning records in a round robin manner.

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 following example, 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 nonkeyed 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, the result sets are funneled together into one dataset, instead of feeding into multiple partitions.

For example, suppose that you want to read all of the records from the table named sourcetable. The unpartitioned 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, negating 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 Oracle database documentation for information on partitioned tables.

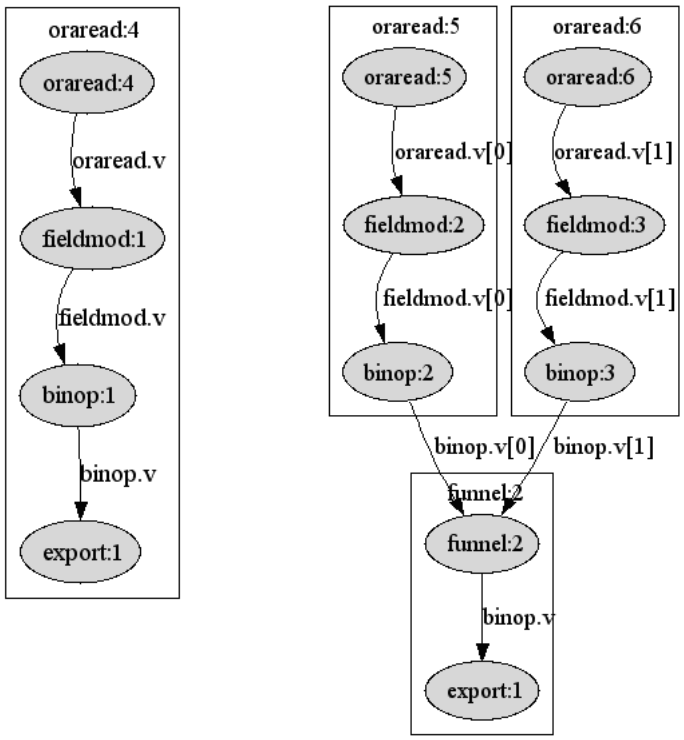
The stored procedure specified by the sp_prequery property is run before any of the queries are executed. Likewise, the stored procedure specified by the sp_postquery property is run after the last query completes.

In the following example, 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 HASH to partition the data, RETL ends 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.

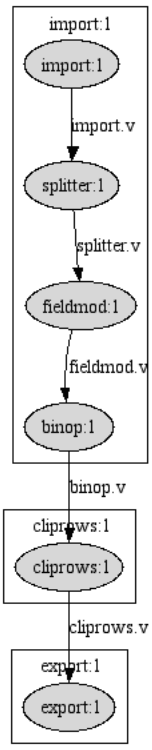
- If a database read operator is used to partition the data, replace the partitioned database read operator with multiple database read operators, and insert a SORTFUNNEL and a HASH, or combine all of the queries into one and insert a HASH after the DBREAD. In either case, make sure that all queries 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.

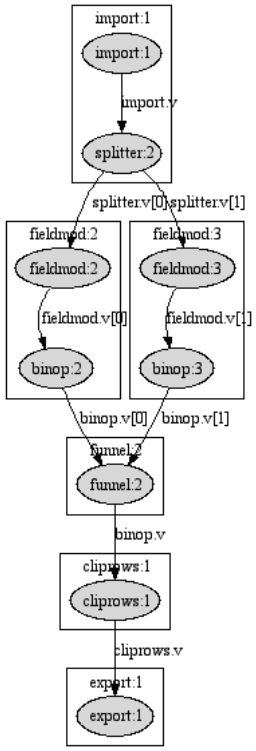
As an example, suppose that a SPLITTER is used to partition imported data and a CLIPROWS operator is encountered. RETL inserts a FUNNEL before the CLIPROWS and displays the following warning message:

```
[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:

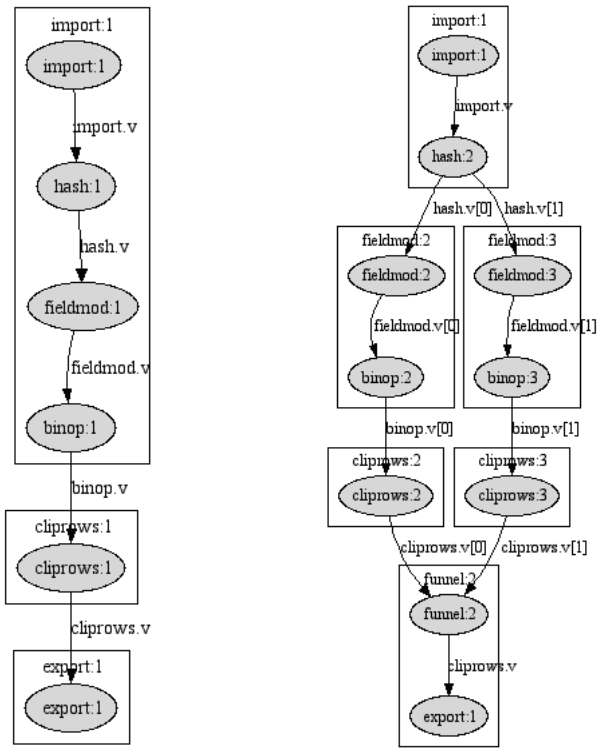
Unpartitioned	Partitioned





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 in any of the following cases:

- Partitioning the operator may sometimes produce incorrect results, and so the default is not to partition the operator.
- Partitioning the operator may improve performance on some platforms and not others. For example, using a partitioned EXPORT may not always improve performance.
- 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 ends the partition before the operator by inserting a funnel.

Partitioned EXPORT

When the parallel property of an EXPORT operator is set to "true", RETL includes 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.

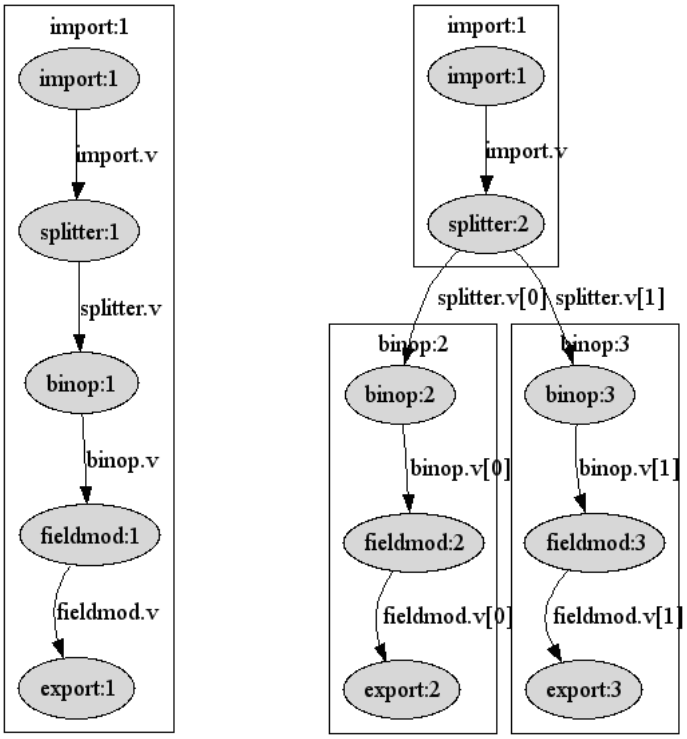
Note: Partitioning the EXPORT operator may or may not improve performance and should be tested before implementation.

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 uses 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.

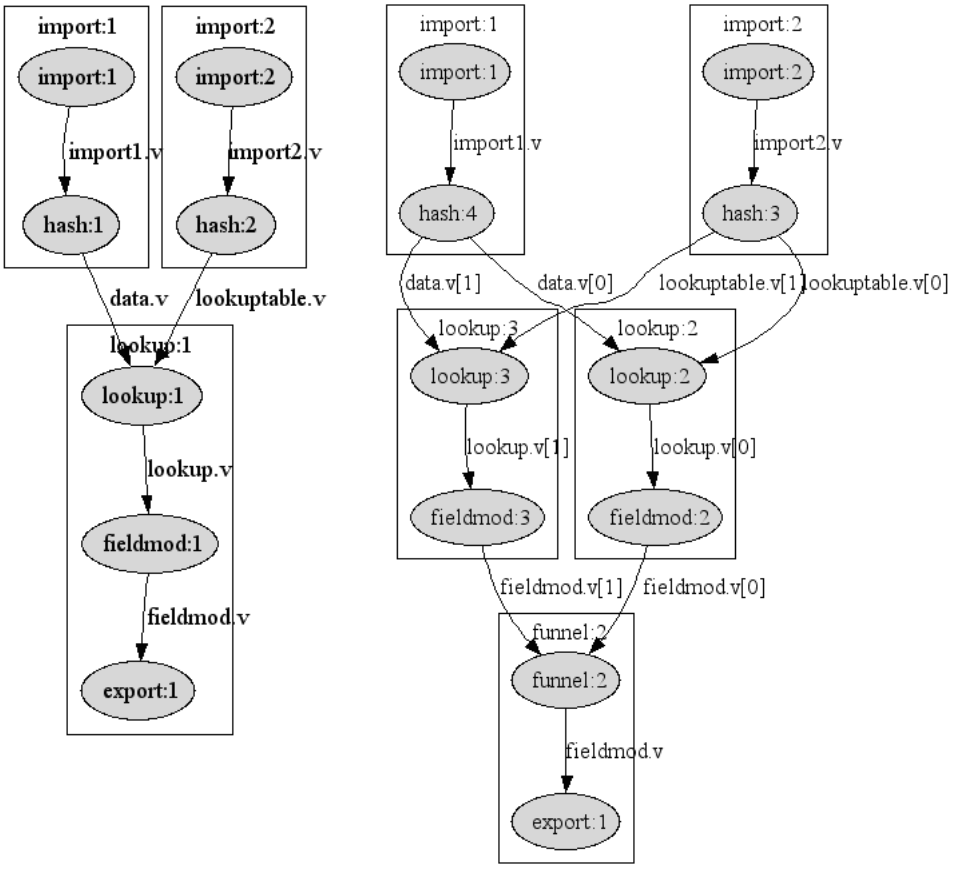
For example, if there are three partitions, and the sequence field's initial value is specified as 1, and the increment value is specified as 1, then the first generator will generate values 1, 4, 7, …; the second 2, 5, 8, …; and the third 3, 6, 9, …

A suggestion 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 uses a subset of the lookup dataset and a subset of the source dataset.

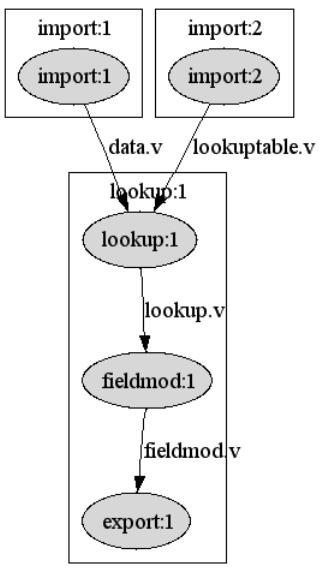
Unpartitioned	Partitioned

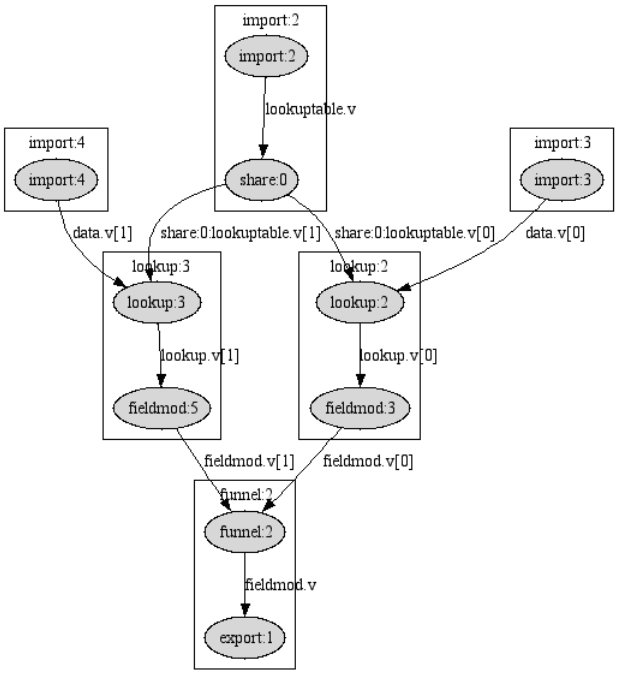


However, if the source dataset is partitioned by any other partitioner, or the lookup table is not partitioned at all, then the LOOKUP or CHANGECAPTURELOOKUP operators in each partition will share the same lookup table. A SHARE operator is inserted into the flow to accomplish the sharing and can be ignored.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

From the graph, it appears that the inserted HASH operator is partitioning the data; however, 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 partitioned ORAWRITE 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 can be partitioned.

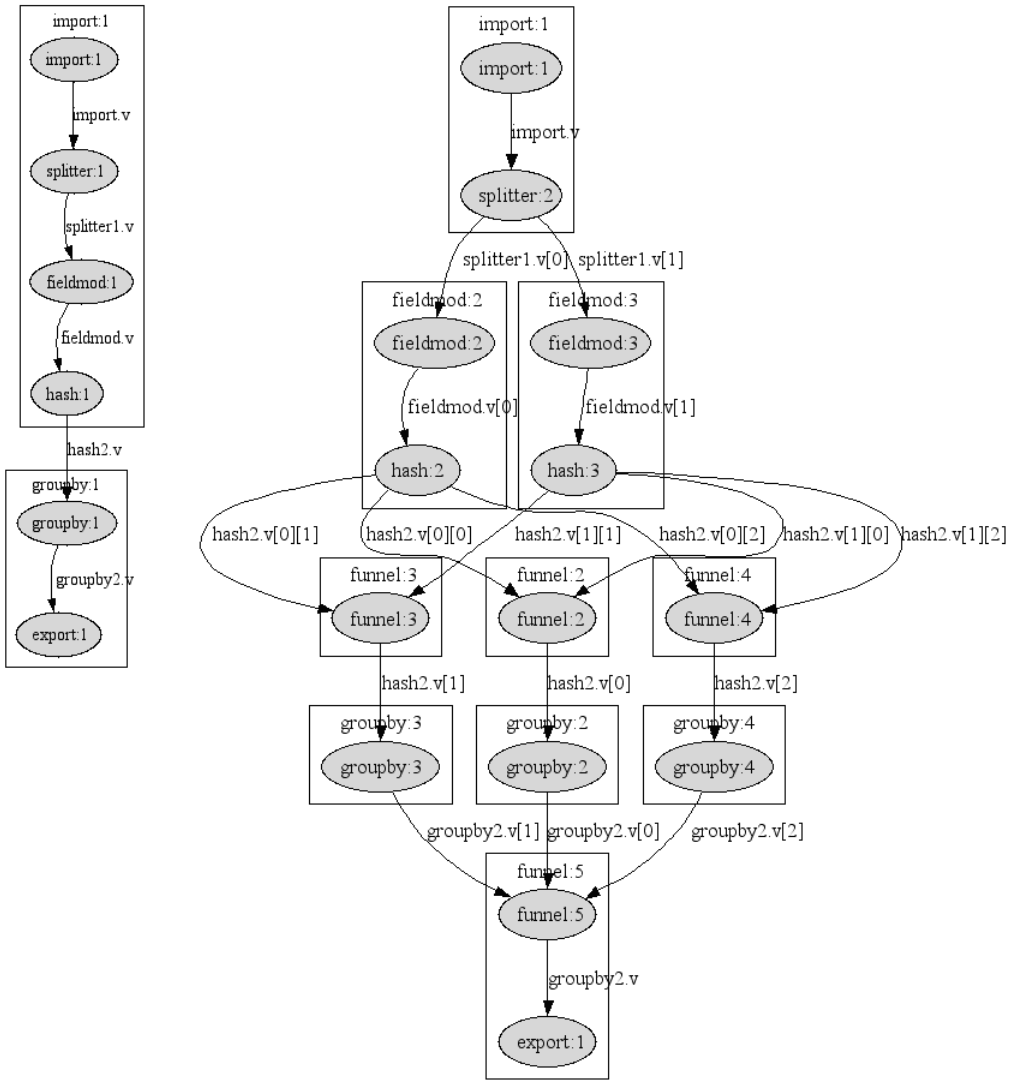
HASH Within a Partition

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 the appropriate sort operator if the data needs to be sorted.

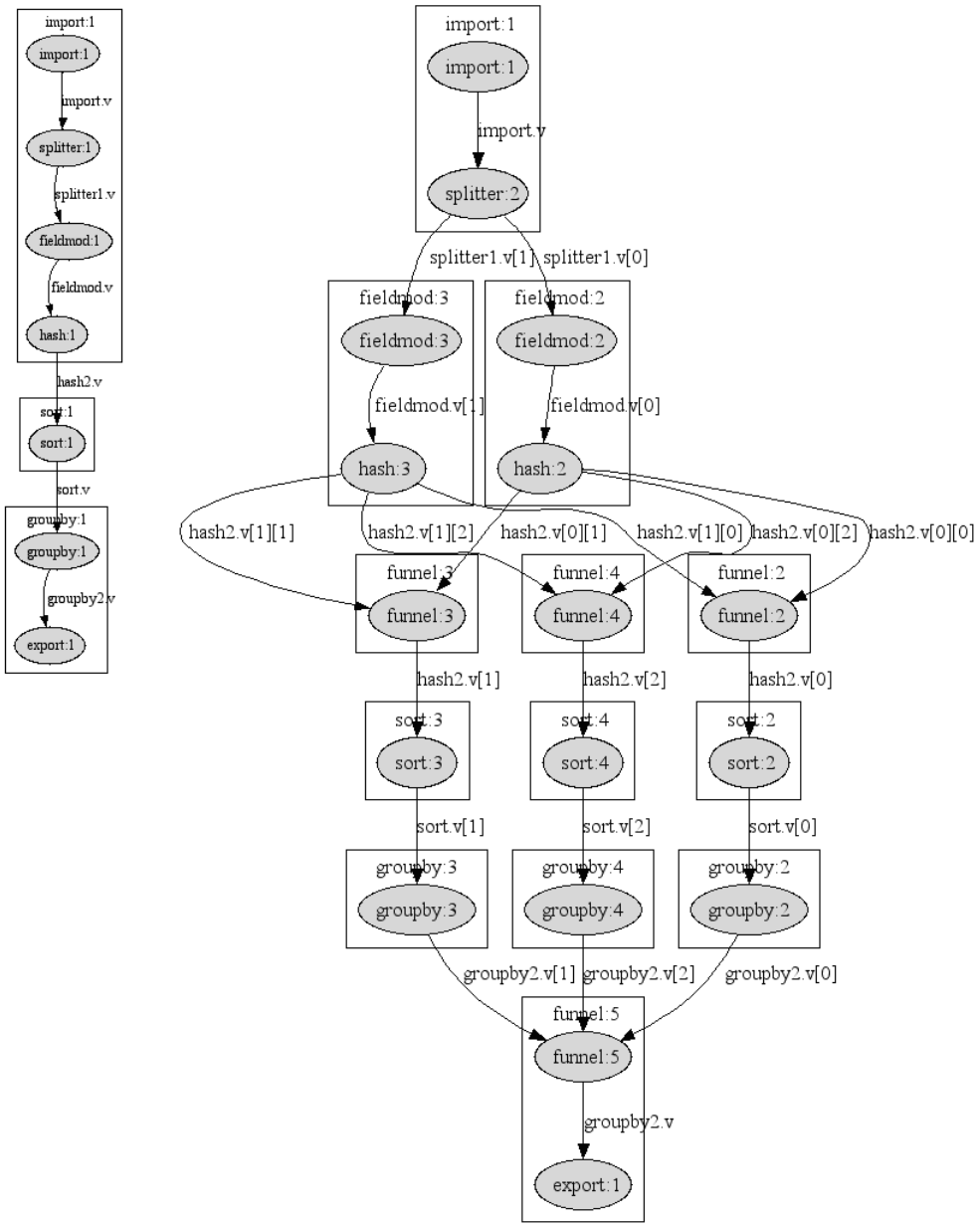
The following graph shows a flow that produces invalid results. The funnels inserted after the HASH operators lose the sorted order required by GROUPBY.

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**

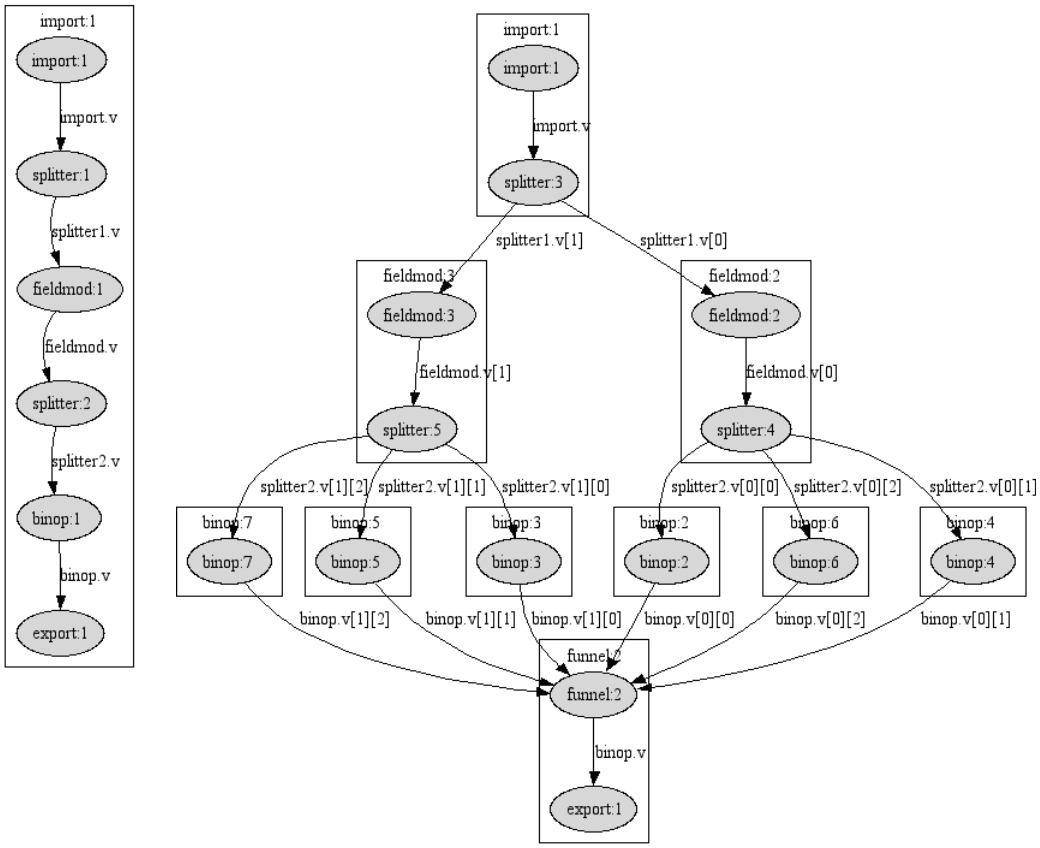


SPLITTER Within a SPLITTER Partition

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

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

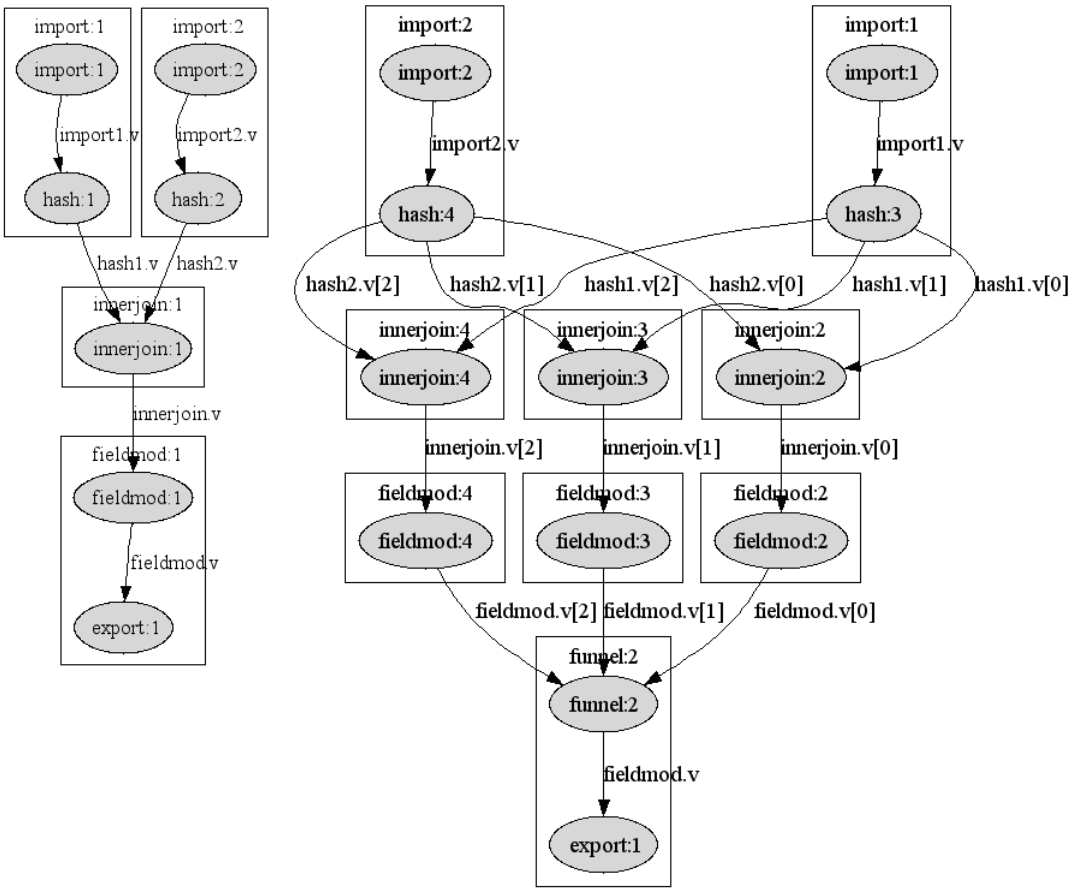
Unpartitioned Partitioned



Two HASH Operators Joined

The join operators (such as INNERJOIN, LEFTOUTERJOIN) support keyed partitioning. RETL ensures 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 are used for both HASH operators.

Unpartitioned	Partitioned

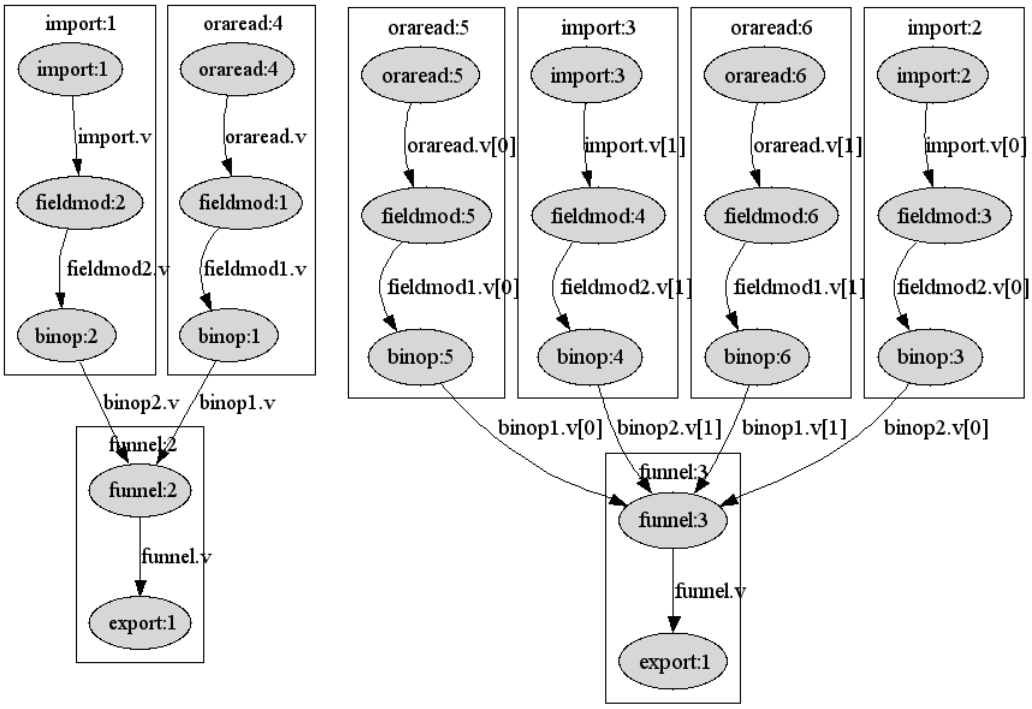


Funneled Partitions

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 Name	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: 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 does not necessarily 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.

Partitioning is not guaranteed to improve performance. In fact, carelessly adding partitioners likely degrades 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.)

8

Input and Output Operators

The following input and output operators are 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 that does little more than purge data.

EXPORT

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

> **Note:** By default, EXPORT exports records in pipe-delimited format (|). To change the output format, an export schema file should be used to specify 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 Tables

DEBUG

Element Name	Name	Value
INPUT		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

Element Name	Name	Value
INPUT		The input dataset name.

IMPORT

Element Name	Name	Value
PROPERTY	inputfile	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. This property can be specified as one of the following: a filename in the current working directory, a filename with a relative path to the current working directory, or an absolute path to a file.
PROPERTY	schemafile	The name of the schema file describing the layout of the data file. This property can be specified as one of the following: a filename in the current working directory, a filename with a relative path to the current working directory, or an absolute path to a file.
PROPERTY	rejectfile	Optional property. The name of the file where records that do not match the input schema are deposited. This property can be specified as one of the following: a filename in the current working directory, a filename with a relative path to the current working directory, or an absolute path to a file.

Element Name	Name	Value
PROPERTY	allowedrejects	Optional property. A value greater than zero indicates how many errors are allowed before RETL exits with a critical error. **Note:** A zero indicates that any number of errors will be accepted. This is the default setting. 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 prints 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	Optional property indicating the number of threads used to read each file. Defaults to 1. Increasing this value may increase performance.
PROPERTY	numpartitions	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 is created for each input file.
OUTPUT		The output dataset name.

EXPORT

Element Name	Name	Value
PROPERTY	outputfile	The output text file name. This property can be specified as one of the following: a filename in the current working directory, a filename with a relative path to the current working directory, or an absolute path to a file.
PROPERTY	outputmode	"overwrite" or "append" Optional property that specifies whether EXPORT overwrites an existing file or appends to it. Default is to "overwrite".
PROPERTY	schemafile	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. This property can be specified as one of the following: a filename in the current working directory, a filename with a relative path to the current working directory, or an absolute path to a file. **Note:** This property can be used to reorder and drop fields from the output and reformat the fixed length, delimiter, and 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.

Element Name	Name	Value
PROPERTY	parallel	"true" or "false" 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) Optional property. Ignored if not partitioning.
OUTPUT		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>
```

9

Join Operators

RETL provides the following join operators, 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 with key fields that 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 manner similar to 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 as needed. 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 favor data fields from the "dominant" side (usually, the first INPUT to the operator) when joining datasets that contain the same nonkey fields on each side. For example:

Left Dataset			Right Dataset		
Key	Same Nonkey Field	Nonkey field	Key	Same Nonkey Field	Nonkey field
A	B	C	A	B	C

Joined Dataset				
Operator	Key	Same Nonkey Field	Nonkey Field	Nonkey 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 note 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, to make the functionality more explicit to those maintaining your flows.

Join Operators XML Specification Tables

INNERJOIN

Element Name	Name	Value
INPUT		Input dataset 1 - sorted on the key columns.
INPUT		Input dataset 2 - sorted on the key columns.
PROPERTY	key	Key columns to be joined.
OUTPUT		The output dataset name.

LEFTOUTERJOIN, RIGHTOUTERJOIN, FULLOUTERJOIN

Element Name	Name	Value
INPUT		Input dataset 1. (left dataset) - sorted on the key columns
INPUT		Input dataset 2. (right dataset) - sorted on the key columns
PROPERTY	key	Key columns to be joined.
PROPERTY	nullvalue	column name=<null value> The default null value that is assigned to the null column.
OUTPUT		The output dataset name.

LOOKUP

Element Name	Name	Value
INPUT		The first input dataset is always the data that needs to be processed.
INPUT		The second input dataset is always the "lookup" dataset.
PROPERTY	tablekeys	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 defaults 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.
OUTPUT		The first output is the successful lookup output dataset name.
OUTPUT		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

Element Name	Name	Value
INPUT		The data to be processed.
PROPERTY	dbname	Required property. Name of the database.
PROPERTY	userid	Required property. Database login name.
PROPERTY	dbtype	"oracle" or "jdbc" Required property. The type of database to connect to.
PROPERTY	tablekeys	Required property. Comma separated key columns to be looked up in the database table or select_sql result set.
PROPERTY	table	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	Simple SQL SELECT statement used instead of table. Either table or select_sql must be specified.
		Note: select_sql can only contain a SELECT statement without the WHERE clause. RETL throws 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`

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 can be specified as a default in rfx.conf for convenience.
PROPERTY	port	The port on which the database resides. This defaults to 1521 if not specified in an operator or in rfx.conf. **Note:** There are preset defaults in rfx.conf for the necessary port numbers. The default for Oracle is 1521. The Oracle utility tnsping can 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 that 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.

Element Name	Name	Value
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 whether the results from the database lookup should be cached.
OUTPUT		The first output is the successful lookup output dataset name.
OUTPUT		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>
```

10

Sort, Merge, and Partitioning Operators

RETL provides the following sort and merge operators, 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

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

MERGE

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 input 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 Tables

COLLECT/FUNNEL

Note: These operators are aliases for each other. FUNNEL is used as a general rule, but either operator will work.

Element Name	Name	Value
INPUT		Input dataset name. This can be specified 1 or more times.
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		The output dataset name.

HASH

Element Name	Name	Value
INPUT		Input dataset.
PROPERTY	key	Key columns to be hashed.
PROPERTY	numpartitions	Number of data partitions to create. Defaults to the number of partitions specified in rfx.conf.
OUTPUT		The output dataset name.

SPLITTER

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	numpartitions	Optional property indicating the number of data partitions to create. Defaults to the number of partitions specified in rfx.conf.
OUTPUT		The output dataset name.

SORT

Element Name	Name	Value
INPUT		Input dataset name.
PROPERTY	key	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" Set to "desc" for descending sort order and "asc" for ascending sort order. Default value is "asc".
PROPERTY	removedup	"true" or "false" If set, then duplicate records will be removed. Default is "false".
PROPERTY	tmpdir	Name of directory to use for temporary files. **Use of this property is not recommended.** 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, "Installation and System Configuration")
PROPERTY	delimiter	Character to use as a delimiter when writing out the records to a temporary file. Defaults to "|" (pipe character). If your data contains the pipe character, set this property to a character that does not occur in your data.
PROPERTY	numsort	Integer greater than 0, indicating the 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	Alias for numsort.
OUTPUT	name.v	The output dataset name.

SORTCOLLECT/SORTFUNNEL

Note: These operators are aliases for each other. SORTFUNNEL is used as a general rule, but either operator will work.

Element Name	Name	Value
INPUT		Input dataset name. This can be specified one or more times.
PROPERTY	key	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.
OUTPUT		The output dataset name.

MERGE

Element Name	Name	Value
INPUT		Input dataset name. This can be specified one or more times.
OUTPUT		The output dataset name.

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>
```

11

Mathematical Operators

RETL provides the following mathematical operators, 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 can 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 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 provides a null column total. GROUPBY requires sorted input only when 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 is not spread to multiple partitions. If the next operator is a serial operator, thereis 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 does not call a FUNNEL operator, and the previous HASH/SORT is maintained. If you are 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, 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 is not maintained, because a serial GROUPBY was performed after that.

Mathematical Operators XML Specification Tables

BINOP

Element Name	Name	Value
INPUT		Input dataset name.
PROPERTY	left	Column name indicating the left operand. Only one of "left" and "constleft" can be specified.
PROPERTY	constleft	Constant value indicating the left operand. Only one of "left" and "constleft" can be specified.
PROPERTY	right	Column name indicating the right operand. Only one of "right" and "constright" can be specified.
PROPERTY	constright	Column name indicating the right operand. Only one of "right" and "constright" can be specified.
PROPERTY	dest	The result destination field name. This can be the same field name as was specified in left or right properties, to reuse existing fields.
PROPERTY	desttype	Optional field. The type of the destination field. The default value is the type of the left field.
PROPERTY	operator	"+" for addition "-" for subtraction "*" for multiplication "/" for division
OUTPUT		The output dataset name.

GROUPBY

Element Name	Name	Value
INPUT		Input dataset name.
PROPERTY	parallel	"true" or "false" "true"—If running in multiple partitions. "false"—If running in a single partition (default).
PROPERTY	key	Key columns to group by. If key is not specified, groupby considers all input data as one group. **Note:** If the key property is not specified, GROUPBY does not require sorted input.
PROPERTY	reduce	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	Name of the new column that will hold the minimum of the values in the column specified in the preceding reduce property.
PROPERTY	max	Name of the new column that will hold the maximum of the values in the column specified in the preceding reduce property.
PROPERTY	sum	Name of the new column that will hold the sum of the values in the column specified in the preceding reduce property.
PROPERTY	first	Name of the new column that will hold the first value in the column specified in the preceding reduce property.
PROPERTY	last	Name of the new column that will hold the last value in the column specified in the preceding reduce property.
PROPERTY	count	Name of the new column that will hold the count of the values in the column specified in the preceding reduce property.
OUTPUT		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 from 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>
```

12

Structures and Data Manipulation Operators

RETL provides the following operators for the manipulation of structures and data, 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 data type

- The <CONVERTFUNCTION> tag allows conversion of a column from one data type to a different data type and has the following attribute:

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, "Appendix: Default Conversions"). When using a default conversion, you simply specify 'default' as the name of the conversion.

Here are the other (nondefault) 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 "Structures and Data Manipulation Operators Examples".
`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 "Structures and Data Manipulation Operators Examples".
`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 reports invalid conversions, but RETL does not handle certain conversions well. Take care to ensure that the data is well-formed by the time a conversion takes place. Overflow and underflow conversion errors are not detected by RETL. For example, converting the INT16 "-12" to a UINT16 will result in undefined behavior.

FIELDMOD

FIELDMOD is 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 do not know about (allowing your flows to be more resilient to change). Any columns that are not specified in the keep property are dropped. The column names are separated by spaces. Multiple keep properties can be specified.
- If you want to keep most of columns and just drop a few columns, use the drop property. The columns to drop are separated by spaces. The drop property is processed after the keep property, which means that a column is dropped if it is specified on both the keep and drop properties. 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 generates an error (for example,." Error: the column 'Emp_Name' has been dropped and can not be duplicated").

FILTER

FILTER is used to filter a dataset into two categories, those that meet the filter criterion and those that do not.

Possible uses are as follows:

- To filter records where a field must be equivalent to a certain string,
- To filter on records where a certain field is less than 10, or filter on records where two fields are equal (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 stand-alone 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 following example 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 it can contain exactly one field generation tag. Field generation tags include:
  - <SEQUENCE> tag 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) Adds the partition number to the init value for parallel operation. The default is false.
partcount_incr	(true/false) Multiplies 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 Tables

CONVERT

Element Name	Name	Value
INPUT		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		The output dataset name.

FIELDMOD

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	keep	Space-separated list of the columns to be kept. The named fields are retained and all others are dropped. This property can be specified multiple times. Optional
PROPERTY	drop	Space-separated list of the columns to be dropped. The named fields are dropped and all other fields are retained. This property can be specified multiple times. Optional
PROPERTY	rename	Column to be renamed. The format is: new_column_name=existing_column_name
PROPERTY	duplicate	Column to be duplicated. The format is: new_column=existing_column Duplicate one column to another column. The source column is separated by an equal sign from the target column name.
OUTPUT		The output dataset name.

FILTER

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	filter	Filter expression The following operations are supported. See the "Filter Expressions" section for more detail.

Operation	Description
NOT	Logical NOT
AND	Logical AND
OR	Logical OR
IS_NULL	True if field is null.
IS_NOT_NULL	True if field is not null.
GE	Logical >=
GT	Logical >
LE	Logical <=
LT	Logical <
EQ	True if right and left fields are equal.
NE	True if right and left fields are not equal.

PROPERTY	rejects	"true" or "false" Optional filter value, default value "false". If "true", then the operator will expect a second OUTPUT to be specified into which rejected (filtered) records will be deposited.
OUTPUT		The first output dataset specified contains all of the records for which the filter expression evaluates to "true".
OUTPUT		The second output is required if the rejects property (above) is specified. This output dataset is the set of records for which the filter expression evaluates to "false".

GENERATOR

Element Name	Name	Value
INPUT		This property is optional. If specified, this is the input dataset. If not specified, numrecords must be specified.
PROPERTY	parallel	"true" or "false" "true"—GENERATOR should be run in parallel. "false"—GENERATOR should be run in a single partition (default).

Element Name	Name	Value
PROPERTY	numpartitions	Optional property indicating the number of data partitions to create.
PROPERTY	numrecords	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

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	key	Column names of columns to use to detect duplicate records. 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 defaults to 'first'. **Note:** Specify "first" as the value for the keep property when sort order makes it possible. This increases REMOVEDUP performance.
OUTPUT		The output dataset name.

Filter Expressions

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

Simple filter expressions are in one of the following forms:

operand comparison-operator operand

or

field null-comparison-operator

where:

operand can be the name of a field or a constant.

comparison-operator is one of the following:

- GE (greater than or equal to)
- GT (greater than)
- LE (less than or equal to)
- LT (less than)
- EQ (equal to)

- NE (not equal to)

null-comparison-operator is one of the following:

- IS_NULL
- 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 from the order used by 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>
```

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.

COMPARE appends a compare result to the beginning of each record with a value set as follows:

Code	Other
-1	First dataset FIELD is LESS THAN second dataset FIELD.
0	First dataset FIELD is the SAME AS second dataset FIELD.
-2	Record does not exist in the first dataset.
1	First dataset FIELD is GREATER THAN second dataset FIELD.
2	Record does not 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 does not exist in the compare dataset.

Note that the Compare Result column above is only 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			**Changed Dataset**		
Key	**Value**	**Other**	**Key**	**Value**	**Other**
John	5	corn	John	5	corn
Jill	10	beans	George	7	olives
Allie	2	pizza	Allie	5	pizza
Frank	14	42fas	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 is just for illustration. Only the CodeField 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 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 backward 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 returns 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 (for example, BINOP, FILTER, COPY, NOOP)
- Improve performance by reducing the number of operators in the flow
- Improve maintenance costs by decreasing complexity of flows

Additionally, flows with the PARSER operator are more readily sizable 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.) Additionally, 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 by using the * designation.
  - RECORD.OUTPUT[n] can be used as a BOOLEAN value in comparisons, so that you can check whether 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 semicolon.

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 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 Specifications

COPY

Element Name	Name	Value
INPUT		The input dataset name.
OUTPUT		The output dataset name. OUTPUT can be specified multiple times.

COMPARE

Element Name	Name	Value
INPUT		Input dataset 1.
INPUT		Input dataset 2.
PROPERTY	key	Column name to be compared between the two datasets defined above.
OUTPUT	name.v	Output dataset.

CLIPROWS

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	key	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	Required property. Indicates the number of rows to clip.
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		The output dataset name.

DIFF

Element Name	Name	Value
INPUT		First input dataset name. This dataset represents the original dataset.
INPUT		Second input dataset name. This dataset represents the modified dataset.
PROPERTY	key	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, 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	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", DIFF acts as if allvalues is set to "true".
PROPERTY	value	After the key property has been used to join datasets, the field specified by 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.
PROPERTY	codefield	The field name to set the change code field to. Defaults to 'codefield'.
PROPERTY	copycode	The code that is written to the change code field for a copy. The default value is 2.
PROPERTY	editcode	The code that is written to the change code field for an edit. The default value is 3.
PROPERTY	deletecode	The code that is written to the change code field for a delete. The default value is 1.
PROPERTY	insertcode	The code that is written to the change code field for an insert. The default value is 0.
PROPERTY	dropcopy	"true" or "false" If "true", copy records are dropped from the output. The default is "false".
PROPERTY	dropedit	"true" or "false" If "true", edit records are dropped from the output. The default is "false".

Element Name	Name	Value
PROPERTY	dropdelete	"true" or "false" If "true", delete record are dropped from the output. The default is "false".
PROPERTY	dropinsert	"true" or "false" If "true", insert records are dropped from the output. The default is "false".
PROPERTY	sortascending	"true" or "false" **Deprecated.**
OUTPUT		The output dataset name.

CHANGECAPTURE and CHANGECAPTURELOOKUP

Element Name	Name	Value
INPUT		First input dataset name. This dataset represents the original dataset.
INPUT		Second input dataset name. This dataset represents the modified dataset.
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 and CHANGECAPTURELOOKUP act as if allvalues is set to "true".
PROPERTY	key	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, 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	This property indicates the fields to use 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".

Element Name	Name	Value
PROPERTY	dropcopy	"true" or "false" If "true", copy records are dropped from the output. The default is "true".
PROPERTY	dropedit	"true" or "false" If "true", edit records are dropped from the output. The default is "false".
PROPERTY	dropdelete	"true" or "false" If "true", delete record are dropped from the output. The default is "false".
PROPERTY	dropinsert	"true" or "false" If "true", insert records are dropped from the output. The default is "false".
PROPERTY	copycode	The code that is written to the change code field for a copy. The default value is 0.
PROPERTY	insertcode	The code that is written to the change code field for an insert. The default value is 1.
PROPERTY	deletecode	The code that is written to the change code field for a delete. The default value is 2.
PROPERTY	editcode	The code that is written to the change code field for an edit. The default value is 3.
PROPERTY	codefield	The field name to set the change code field to.
PROPERTY	sortascending	"true" or "false" **Deprecated.**
OUTPUT		The output dataset name.

SWITCH

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	switchfield	Field name 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.
PROPERTY	discard	Value to be discarded or dropped.

Element Name	Name	Value
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 do not 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		The valid output dataset. Number of outputs should correspond to number of casevalues. For example, "value1=0, value2=1" should have two outputs (or three if discard is specified)
OUTPUT		The reject output dataset.

PARSER

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	expression	Language expression the operator should use to process records. Supports general Java-like syntax and must be surrounded by CDATA tags. See the examples that follow.
PROPERTY	allowcopy	"true" or "false" Set 'allowcopy' to "true" if the expression can copy the same record to more than one OUTPUT.
OUTPUT		First output dataset name. At least one OUTPUT is required.
OUTPUT		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

Element Name	Name	Value
INPUT		The input dataset name.
PROPERTY	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.
OUTPUT		The output dataset name.

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>
```

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>

Enclose date and string constants in double quotation marks:
<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) sends 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 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 are sent to the first OUTPUT and some are sent to the second OUTPUT. All
records are 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 are sent to the first OUTPUT, some will be sent to both OUTPUTs, and
some are 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>
```

14

Common Operator Properties

This chapter describes properties common to all operators.

Common Operator XML Specification

Element Name	Name	Value
PROPERTY	progind	Progress indicator string displayed as records are processed.
PROPERTY	progfreq	How often to display the progind string. The progind string is displayed every progfreq records.

15

Best Practices

Introduction and Objectives

Traditional application integration has been done in a point-to-point manner. Developers are given an integration tool and told to integrate to a defined file/database specification or to an API. The end result may be functionally valid according to the single specification, but the means to get there, the implementation, may be cumbersome, non-reusable, non-performant, and subject to heavy maintenance costs. Worse yet, the design itself may not have accounted for all the business process needs, invalidating the entire implementation.

This is why there is a need for best practice guidance when using any tool—so that the tool can be used and the interface developed and deployed to the maximum advantage in terms of costs and benefits. This chapter will cover simple and proven practices that can be used when developing and designing integration flows using the RETL tool.

RETL is a simple tool doing a very simple job—moving large amounts data in bulk. However, managing the variety of disparate systems and pulling data together to form a complete 'picture' can be very complex.

As a result of business process, integrated system, database and interface complexity, it is imperative that the RETL tool be used correctly so that the end deliverable of RETL flows is performant, maintainable, and of high quality.

This chapter will describe the following activities and what best practices should be followed before, during, and after each activity:

1. Project Initiation/Design/functional specification—The best practices to follow during requirements elicitation, interface analysis, and design.
2. Code/Implementation/Test—The best practices to follow when setting up the environment, during flow construction and all phases of testing. This will be the bulk of the chapter.
3. Review/product handoff—Activities to follow when reviewing the project and its deliverables. This also provides guidance for handing off interface deliverables to operations and support personnel.

This is a living chapter that will grow over time as we learn more about our customers needs and can fill it with tips from experts from the different domains that RETL touches upon.

Prerequisites

There are two core skill sets needed when designing and developing RETL integration flows:

1. Interface design skills—a functional/technical architect should design the interfaces. The designer should have knowledge of the following:
   a. Understand the source and target data sources. They should have in-depth knowledge of the data models for each application, and how a transformation may be performed in order to integrate the two applications
   b. Can understand how each application and business process works and can quickly interpret the data needs for an application based on the business process(es) that need to be supported.
   c. Has a general understanding of the schedule, data dependencies, batch jobs, and volume of the application in question.
2. ETL technical skills—The ETL coder should have knowledge of the following:
   a. Have strong Unix experience—in-depth Korn shell scripting and Unix in general are a must
   b. Have previous RETL Experience—familiarity with the RETL Programmer's Guide is a must and previous experience with writing RETL flows is strongly recommended
   c. Have strong database Experience—familiarity with SQL statements and database operations and utilities (e.g. SQL*Loader) is strongly recommended.

Project Initiation/Design/Functional Specification Best Practices

Before any development can be done, it is imperative to have a solid interface design that is generic and re-usable for future applications. The following practices will help ensure this:

Ask Discovery Questions First

An important part of the functional design is to ask pointed and relevant questions that can answer the following:

Generic Integration Questions

- *What type of data integration is needed?* Is there a potential for needing real-time or near-real-time integration? If there is potentially a need for more synchronous/real-time integration, perhaps consider using a different technology such as the Oracle Retail Integration Bus (RIB) or direct access.
- *What business process is to be integrated?* For any project manager or flow developer to understand how they must construct the data flow, they must understand at a high-level, what business process it is that they will support. Why do the users in system x need the data created in system y? Without this understanding, RETL users may not get a complete picture of what it is they need to integrate, nor would they have the opportunity to ask questions that might reveal additional interface capabilities, variations, or options that are needed by the business user community.

Application Domain Questions

- *What are the targeted versions for each application involved?* This is important to establish as a basis for integration development against each product and the business features and functions available in a particular version.
- *What is the source and target for each module?* For example, database to file, file to file, file to database, etc.
- *On what DBMS and version does the source and target application reside, if any?* If the source/target database isn't supported by RETL, then consider using a different technology or provide feedback to the RETL team to include that database support in a future release.
- *What types of transformations might be expected for each module?* This will affect the complexity of each module.
- *Are there any constraints on the source(s) and target(s)?* For example, does the target database table need to maintain indexes, etc? Are there any referential integrity issues to be aware of? There are implications when developing a flow against a transaction database or a table that needs to maintain indexes or /referential integrity.
- *What future applications might 're-use' these interfaces?* Designs should be generic and promote easy modification should any future applications need to re-use the same interfaces.

Data-related and Performance Questions

- *How much data is passing through each module?* High volume modules will need to be further scrutinized when coding and performance testing.
- *What is the size of the source and target database tables/files?* This is to raise any flags around performance early in the project.
- *What is the frequency in which each module will be run?* (e.g. nightly)
- *What is the time frame in which each module is expected to run in?* For the entire set of modules? This will need to be realistic as it will provide a baseline for performance.
- *Is data going to need to be transferred over the network?* There will be a negative effect on performance if massive amounts of data will need to be transferred over the network.
- *Are there potentially any high volume transformations that could be easily done inside the database?* If there are, these transformations may be done inside the database to start with so as to eliminate rewrites later on. The idea here is to maximize and balance each technology to what it does best.

Map Out The Movement of Data Visually

Use a tool to visually map out the movement of data from each source to each target. Use a tool such as Visio or even simply Word to diagram the source, transformation processing/staging, and target layers. (Use a database template or the 'basic flowchart' in Visio to map these layers out). This should serve as a basis for design discussions among application groups.

Define Concrete Functional Requirements for Each Module

As part of the design process, a logical description of how each module will extract, transform, and load its data should be completed. Again, all stakeholders should sign off on this.

Define Concrete Functional Designs for Each Module

As part of the design process, the following should be clearly defined:

- Source and Target locations (e.g. table name, filename, etc)
- Source and Target sizes/volumes
- Designs and metadata definitions for each input and output
- Mapping of the transformation process on how to get from input format to output format
- Name of the script/module

Design a Test Plan Early in the Process

Designs are not complete until the unit, system, integration and performance test plan designs are complete. This can be completed once the data and test environment needs are identified. Activities to complete for the test plan include the following:

- Identification of the different combinations of data and the expected results.
- Identification of degenerate cases and how errors should be handled.
- Identification of how many and what type of test cases that need to be built.
- Identification of configuration management and environment migration procedures to ensure that quality control individuals know what code set it is that they are testing.
- Identification of test environment needs to ensure that unit, system, integration and performance test can be done without negatively impacting each other.
- Prioritization of relative importance of execution for each test case based on most likely scenarios, highest risk code, etc.
- Determine a 'go/no-go' threshold for which tests must pass in order to release.

Design for Future Usage and Minimize Impact of Potential Changes

The design should take into account future applications that may need to live off the same interfaces. Relevant questions to ask are the following: What is the superset of data that an application needs or may need in the future? How can this flow be designed so that any future changes will not break the interface specification?

Agree on Acceptance Criteria

The overall project should set requirements based on a number of acceptance criteria set by the customer. A partial list of these may involve the following:

- *Functional criteria*—the inputs, outputs, and expected functionality of each module
- *Test criteria*—the types and how intensive must the testing be, and the threshold in acceptance of defects.

- *Performance criteria*—the performance requirements for each module, and the overall batch window in which the entire suite of modules must run in.
- *Documentation*—Identify clearly the end deliverables needed to hand off the interface deliverable to operations and support staff. A well-constructed interface cannot be used and maintained successfully without documentation or communication deliverables.

Document Design Assumptions, Issues, and Risks

Any design assumptions, issues and risks that have been purposely un-addressed should be documented in an 'Assumptions/Issues/Risks' section of the design documentation deliverables.

Code/Implementation/Test Best Practices

This section will break best practices to follow into these functional areas:

1. Korn shell module best practices—the .ksh module containing the RETL flow
2. RETL flow best practices—the XML flow that RETL runs
3. Database best practices
4. Testing best practices

Following this core set of best practices when developing and testing flow modules will provide the basis for high quality, maintainable, performant, and re-usable flows.

Korn Shell Best Practices

Execute Commands Using $(command) and Not 'command'

This reduces the likelihood of typographical errors when entering backspaces versus single quotes (` vs. ') and promotes better readability.

Ensure 'set -f' is set in a Configuration File

'set -f' disables filename generation. This should be set in the application's config file (e.g. rdw_config.env). If filename generation isn't turned off, a flow that contains a '*' in it (for example, a flow that contains a query 'select * from <table>') may end up causing the module to incorrectly expand * to include filenames and incorrectly pass these to RETL. This is because many modules may pass RETL a flow through standard input. See best practice "Write flow to intermediate file and then call RETL on that file".

Write Flow to an Intermediate File and then Call RETL on that File

It is recommended to 'cat' or write to a flow file first before calling RETL, rather than calling RETL and passing a flow through standard input. This aids in debugging, helps support when it is necessary to recreate an issue, and also prevents various nuances that may exist when sending the flow to RETL via standard input.

However, care should also be taken to protect source directories since these generated .xml flow files may contain sensitive information such as database login information. See the "Secure/Protect Files and Directories that may Contain Sensitive Information" best practice for more on the topic of security.

Good

```
cat > ${PROGRAM_NAME}.xml << EOF
…
EOF
${RFX_EXE} ${RFX_OPTIONS} -f ${PROGRAM_NAME}.xml
```

Bad

```
$RFX_HOME/bin/$RFX_EXE -f - << EOF
…
EOF
```

Secure/Protect Files and Directories that may Contain Sensitive Information

Any directories that may contain sensitive information such as database logins and passwords should be protected by the proper UNIX permissions. Namely, any temporary directories that RETL uses, any source directories in which flows may be written to, and any configuration files and directories should have strict permissions on them. Only the userid that runs the flows should have access to these files and directories.

Make Often-used Portions of the Module Parameters or Functions

Make Function Calls Only a Few Layers Deep

Use variables for portions of code that can be re-used (e.g. ${DBWRITE}), or create entire shell script functions for methods that can be re-used. (e.g. simple_extract "${QUERY}" "${OUTPUT_FILE}").

This can be taken to the extreme however, when functions call functions, which call functions. In general, function calls should go only a few layers deep at most, and only more when it makes sense from an abstraction and overall maintainability point of view.

Separate Environment Data from the Flow

Variables that can change (for example, database login information/etc) should go in a configuration file (for example, rdw_config.env). The idea here is to separate the configurable environment data from the generic flow, which lends to more maintainable and configurable code base.

Enclose Function Parameters in Double Quotes

When calling a function in ksh, care should be taken to place all arguments in double quotes. For example,

```
do_delete "${TABLE1}" "${TABLE2}" "${JOIN_KEYS}"
```

If function parameters aren't placed in quotes, there is the potential for misalignment of positions for variables and the function will likely not operate properly, if at all.

Set Environment Variable Literals in Double Quotes

When setting an environment variable, always use double quotes. For example,

```
export TARGET_TABLE="INV_ITEM_LD_DM"
```

If environment variable values aren't placed in quotes, there is the potential for losing the portion of the value after a space. or having the shell interpret it as a command.

Use Environment Variables as ${VARIABLE} Rather than $VARIABLE

Environment variables should be used as ${VARIABLE} so as to promote readability, especially when they exist in the context of other non-variables and text.

Follow Module Naming Conventions

There are a number of naming conventions to follow, and in general, this should be the standard for consistency among applications:

1. RETL modules/scripts (.ksh) should generally conform to the following convention:

   <application name><script type>_<program name>.ksh,

   where

   <application name> is the abbreviated application name,

   <script type> is 'e' for an extract program, 't' for a transform program, or 'l' for a load program. A program that contains all of the above will not have a script type associated with it,

   <program name> is the name of the program.

   For example:

   rmse_daily_sales.ksh (<rms><e>_<daily_sales>.ksh)

2. The program name in general doesn't have strict naming conventions, but it should be representative of what the program actually does. Each word should be separated by an underscore (_).
3. Custom modules/flows should be prefixed by the short client name.
4. Application log files should follow the format:

   <application name><date>.log

5. Error files should follow the format:

   <application name>.<module name>.<unique id>.<date>

   where

   <application name> is the abbreviated application,

   <module name> is name of the module,

   <unique id> is optional. If multiple copies of the module can be run concurrently, this should be a reasonable unique identifier. For example, it might be the input filename, thread #, etc.

   For example:

   rdw.slsildmdm.sls_100_1.txt.20020401

   (<rdw>.<slsildmdm>.<datafile #>.<date>)

Log Relevant Events in Module Processing

As a general rule, it is better to log too much verbosity than too little. The following should be completed:

1. Log the start and finish times for any significant processing events that occur in a module.
2. Log the number of records in the input and output files.
3. Redirect standard output and standard error to the log/error file. For example:

```
exec 1>>$ERR_FILE 2>&1
```

Place Relevant Log Files in Well-known Directories

Each application log file should be placed in a well-known directory for consistency and ease of accessibility. For example, RETL performance log files (as specified in rfx.conf) are placed in temporary directories by default. rfx.conf should be changed so that RETL performance log files places these log files in well-known directories rather than in temporary directories.

Use .ksh Templates

Insert template on a standard flow (e.g. library sourcing first, then variable definitions, function initionsdefs, and so on)

Document Each Flow's Behavior

Each flow should be documented as to its required inputs, required outputs, and behavior in between.

RETL Flow Best Practices

- **Parameterize the Call to and any Options Passed to the RETL Binary**

  Use the ${RETL_EXE} environment variable to replace the call to the rfx executable and use ${RETL_OPTIONS} to replace the command line option '-f -'. "rfx -f - " should not be used to call RETL - it should be "${RETL_EXE} ${RETL_OPTIONS}". $RETL_EXE and $RETL_OPTIONS should be defined in the configuration file (e.g. rdw_config.env). The default call in the configuration file should be rfx - f -

- **Perform all Field Modifications as Soon as Possible in the Flow**

  Field modifications such as renaming and type conversion should be performed as soon as possible in the flow to aid in tracking a field through the flow.

- **Use Care when Choosing and Managing your Temp Spa**ce

  RETL temp space should be spread out on multiple disks and, as part of a normal maintenance schedule, should be cleaned periodically as RETL may leave miscellaneous files in temp space for tracking and debugging purposes.

- **Turn on Debug Code When in Development**.

- **Turn Off Debug Code When in Production**

  RETL will print out extra messages in debug mode. The following environment variables can be set to produce extra verbosity:

  export RFX_DEBUG=1

  export RFX_SHOW_SQL=1

Additionally, there are certain warnings that may be printed to the screen in this mode that will not be printed to the screen when verbose debugging is turned off.

Do not run with these options on in production as they produce additional files and may leave around temporary/intermediate files.

- **Make Use of RETL Visual Graphs**

  In RETL 10.x versions and 11.2+, RETL offers a graphing option (--graphviz-files in 10.x or -g in 11.2+). This produces a visual graph of the flow and shows how each operator is connected inside the flow. For more information, see "Producing Graphical Output of Flows with RETL" in Chapter 4, "RETL Program Flow".

- **Delineate Input records from a File by a Newline ('\n')**

  Although RETL doesn't mandate having a newline record delimiter, it is recommended to use this for debugging and supportability purposes. However, it is also important to make sure that input data cannot contain the newline character. See "DON'T - Choose a delimiter that can be part of your data" for more information.

- **Run RETL With the '-s SCHEMAFILE' Option When in Development**

  The '-s SCHEMAFILE' command-line argument will tell RETL to print input and output schemas of each operator in a schemafile format that can be used in an import or export operator. This can be very useful when debugging and 'breaking up' flows into smaller portions. For example, the call to RETL would be as follows:

  ```
  rfx -f theFlow.xml -s SCHEMAFILE
  ```

- **Use the Latest Version of RETL Available**

  Future versions of RETL are always improving in error handling, debugging, logging, and performance. It is recommended to use the latest version of RETL that has been certified with the product being used.

- **Develop the RETL Flow First, Outside of the Shell Module**

  It is easier to develop the RETL flow outside the Korn shell module (.ksh) first, and then when a stable cut of the flow has been produced, the flow can be parameterized and retrofitted back into the Korn shell module (.ksh)

- **Specify Inputs, Properties, and Outputs, in that Order**

  In order to conceptually map out the flow of data from one operator to another, place the INPUTs first, the OUTPUTs last, and the properties in between.

  For example:

  ```
  <OPERATOR type="changecapture">
   name="before_dataset.v"/
  <INPUT name="after_dataset.v"/> name="key" value="CMPY_IDNT"/ name="value"
  value="CMPY_DESC"/ name=:"changes.v"/</OPERATOR>
  ```

- **Group Business Logic Together with Nested Operators**
- **Keep Connected Operators in Close Physical Proximity to Each Other in a Flow**
- **Nest Operators No More Than a Few Layers Deep**

  When multiple operators are used to perform a logical step in the flow, nest the operators together.

In the example below, the operators split the after_dataset.v dataset into separate datasets for inserts, deletes, and edits. The switch operator is embedded within the changecapture.

```
<!-Split up after_dataset.v into inserts, deletes, and edits. -->
<OPERATOR type="changecapture" >
<INPUT name = "before_dataset.v"/>
<INPUT name = "after_dataset.v"/>
<PROPERTY name="codefield" value="change_code" />
<PROPERTY name = "key" value = "CMPY_IDNT"/>
<PROPERTY name = "value" value = "CMPY_DESC"/>
<OPERATOR type="switch">
<PROPERTY  name="switchfield" value="change_code"/>
<PROPERTY  name="casevalues" value="1=0, 2=1, 3=2"/>
<OUTPUT  name="inserts.v"/>
<OUTPUT  name="deletes.v"/>
<OUTPUT  name="edits.v"/>
</OPERATOR>
</OPERATOR>
```

As the operator nesting gets deeper, the flow becomes unreadable, so don't nest operators more than a few layers deep.

- **Document Fixed Schema File Positions**

  In order to debug flows and to aid in documenting interface fixed-length files, fixed-length schema files should include positions as a comment before each field definition. However, this also adds additional overhead to schema file maintenance, and could prompt for typographical errors. This should be taken into account when making a decision to document a module's schema files.

  Example:

```
<RECORD type="fixed" len="9" final_delimiter="0x0A">
<!-- start pos 1  --> <FIELD name="FIELD1" len="4" datatype="int16" …/>
<!-- start pos 5  --> <FIELD name="FIELD2" len="4" datatype="int16" …/>
<!-- end pos 9  -->
</RECORD>
```

- **Use Valid XML Syntax**

  10.x versions of RETL allowed users to enter invalid XML syntax. Later versions are more strict about valid XML syntax, and will throw errors if syntax is invalid. In particular, when calling RETL's filter operator, utilize 'GT' instead of '>', 'LT' instead of '<', 'LE' instead of '<=', 'GE' instead of '>=', 'NOT … EQ …' instead of '<>' and 'EQ' instead of '='. Also see the best practice Wrap dbread 'query' property in a CDATA element.

- **Wrap dbread 'query' Properties in a CDATA Element**

  This is important to make sure flows are using valid XML syntax. In general, ALL 'query' properties of the dbread operators should be wrapped in a CDATA element. This tells the XML parser to treat the enclosed text as a single chunk of text rather than to parse it as XML. If this rule is not followed, RETL may not run (see best practice 'Use valid XML syntax' for more info).

  Bad

```
${DBREAD}
<PROPERTY name="query" value="SELECT * FROM TABLE WHERE COL1 > COL2"/>
...
```

Good

```
${DBREAD}
<PROPERTY name="query">
<![CDATA[
SELECT * FROM TABLE WHERE COL1 > COL2
]]>
</PROPERTY>
```

- **Write Flows to be Insulated from Changes to a Database Table**

  In general, flows should be written so that any changes to database tables don't affect the functionality of the flow. For example, the FIELDMOD operator can be used with the "keep" property to keep only fields that are needed in the flow.

- **Avoid Implicit Schema Changes**

  Certain operators that take more than one input (such as funnel) can implicitly change schemas for a field's nullability, max length, etc. Any desired schema changes should be explicitly made with the CONVERT operator otherwise undefined behavior may result.

- **Test Often**

  Testing often minimizes integration issues, increases quality, aids in easier defect diagnosis, and improves morale.

- **Do RETL Performance Tuning**

  In general, there are a few practices to follow as a guideline:

  – Achieve the optimal balance between the tools, hardware, and database.

  – Use the database for what it does best - significant data handling operations such as sorts, inter-table joins, and groupby's should be handled inside the database when possible.

  – Use RETL for what it does best - non-heterogeneous (for example, file, different db vendor) joins, data integration and movement from one platform to another, multiple source locations, etc.

  – Performance test and profile early and continuously in the development cycle. Don't delay this activity until the end of the project.

  – Don't performance tune a flow before the flow functions according to design.

  – Design a flow to make full use of parallelism. RETL internal parallelism coupled with standard Oracle Retail 'multi-threading', or running multiple RETL processes concurrently, can maximize the use of parallelism.

  – Develop for clarity, maintenance, and re-use, but keep in mind performance at all times. Other factors may be sacrificed if performance is of the utmost importance.

- **Name Flows Appropriately in the <FLOW> Element**

  Flows should include a 'name' attribute in the FLOW element of the XML. This eases debugging in RETL's logging mechanism. In general, the flow should be named the same as the module in which the flow is run. For example,

```
<FLOW name="modulename.flw">
...
</FLOW>
```

- **Use SQL MERGE Statements When Doing Updates**

  As of this writing, RETL does not have update capabilities built in. The recommended approach when doing updates is to use the Oracle MERGE statement. This statement is akin to a delete followed by insert for each updated record, although MERGE is much faster. The flow's processing would be as follows: insert records to a temp table, and then MERGE into the target table.

- **Document Complex Portions of Code**

  As with general programming best practices, any complex portions of code that could be misunderstood should be thoroughly documented. This aids in ease of maintenance and helps future personnel to understand the original intent of the code.

- **Define Field Names Without White Spaces**

  Many RETL operator properties allow specification of a list of fields. White space within a field name will cause RETL to misinterpret the field list.

- **Choose a Delimiter That Cannot be Part of the Data**

  If a data can even possibly contain a delimiter, a different delimiter should be chosen. Even to risk being redundant, it should be repeated - NEVER EVER choose a delimiter that can be part of the data. For the most part, pipe-delimiters ('|') and semi-colon delimiters (';') should work. If no reasonable delimiter can be found, then a fixed format file should be used.

Review/Product Handoff

A final product review and handoff to support is necessary to complete a successful RETL project.

Involve Support Personnel Early in the Project

This will help the learning curve and make the impact less dramatic when the project must be transitioned to support.

Identify clearly the end deliverables needed to hand off the interface deliverable to operations and support staff. A well-constructed interface cannot be used and maintained successfully without documentation or communication deliverables.

Assign a Long-term Owner to the Project/Product/Interface

Assigning ownership will allow the product to mature and develop properly and it will reduce internal contention about areas of responsibility when multiple applications/domains are involved.

Assigning ownership of the interface following handoff will also ensure that appropriate operations and support personnel are involved and trained throughout the life of the development effort.

A

Appendix: Default Conversions

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

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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.

Default Conversions 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 truncate decimal values.

Default Conversions from DEFLOAT

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 truncate decimal values.

B

Appendix: 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 specific tools 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 set up to support RETL activities. The following 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 or database operators, take a look at the log files or control files generated by the database operators (see the topic "How do I tell what commands are being sent to the database?" in Appendix C, "Appendix: Troubleshooting Guide"). 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 configuration error, 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. To 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 with 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.

In the following list, note the 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 be 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 is not 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 the following command:

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

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

5. Verify database login id/password.

   Log in to the database by using the following:

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

   Contact to your DBA if this step fails; your userid/password are not set up correctly.

6. Check database user ID privileges.

   Contact your DBA to ensure that the RETL user has privileges to select, insert, update, create table, and so on. Because RETL database operators invoke database utilities to extract and load (see Chapter 6, "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

It is recommended that the RETL/database login id is set up 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 set up properly for RETL.

7. Run RETL/database testing scripts.

   Note that these scripts require that the rfx userid has full privileges to run. If you do not 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:

   - 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. Debug 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 have probably typed an invalid database name, userid, or password (see "When running rfx I get the following error: 'Error connecting to the database'" in Appendix C, "Appendix: Troubleshooting Guide"). 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. Debug 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 would 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 have probably typed an invalid database name, userid, or password (see "When running rfx I get the following error: 'Error connecting to the database'" in Appendix C, "Appendix: Troubleshooting Guide"). Verify that these parameters are correct (see step 4) 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. Run 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 Oracle Retail Support:

    https://metalink.oracle.com

C

Appendix: Troubleshooting Guide

This troubleshooting guide provides 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 have tested the smaller flow. Back up your work often, so that you can look at the differences between flows to narrow problems to smaller segments.
3. As you are coding, copy and paste from examples that you know work, to avoid syntax and XML errors if you are using a text editor.
4. When you run into problems, try breaking up your flows. Insert DEBUG 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 the flow 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" in Chapter 2, "Installation and System Configuration" 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 the "Configuration" in Chapter 2, "Installation and System Configuration" section for more details).

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

You have probably typed an invalid database name, userid, or password. Verify that these parameters are correct (See steps 1-4 of "Troubleshooting RETL with Your Database" in Appendix B, "Appendix: Database Configuration and Troubleshooting Guide") 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 on the position within the flow and other factors.

There are two ways to determine which 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 that 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, or do not 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?

SORTFUNNEL 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 to find the source of the problem.

If you are using partitioning and are seeing operators that you do not expect, refer to the preceding topic: '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 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.

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 (for example, NUMBER(12,4)) to RETL types?

See Appendix E, "Appendix: RETL Data Types" for 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 (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 have chosen.

See Appendix E, "Appendix: RETL Data Types" for 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" in Appendix E, "Appendix: RETL Data Types".) 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 to remain consistent with each of the database vendors' terminology. However, this made it 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, 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 the 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
```

This is because SQL*Loader cannot maintain the index that is in IU state prior to the beginning of a direct path load. Fix this problem by re-creating the table index.

How do I request help when all else fails?

You should contact Oracle Retail Support at https://metalink.oracle.com when requesting help with a flow. Follow these guidelines to get a faster 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 or the data involved, allowing us to focus on the true problem.
2. If you have a database operator (for example, 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 (for example, ORAREAD, ORAWRITE) with an IMPORT or EXPORT operator. Use the exact same schema as 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 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 have 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 that 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 or values do not 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 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.

D

Appendix: FAQ

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. 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 (for example, between EBCDIC and ASCII)?

Currently, RETL does not do textual translation of this type.

Can RETL handle data translations (for example, between numbers and strings)?

Data translations between different fields can be done with 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, because 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 numbers specific to your group. The RETL group does benchmark internal performance results, and these are made available through the Product Enhancement Review Board. Contact your product group for inclusion to this board.

Is it possible to join data from two different databases?

Yes. This is generally done with two ORAREAD 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.

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 these scenarios:

- Schemas—Mismatched input schemas (for example, for FUNNEL), invalid schema specified (for example, record length does not equal sum of field lengths).
- Mismatched datasets—RETL enforces one-to-one 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, through 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 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 (for example, /tmp/rfx) directory, so that they can be easily purged on a regular basis.

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 three 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 a "diamond" flow that contains circular loops.

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 Oracle Retail product strategy in defining future enhancements to RETL. If you are not part of the PERB, contact your point person or team lead to inquire about joining.

I have been getting warnings about sorting/sort order such as the following. How do I remove these warnings?

```
[<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!

[<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!
```

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, 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, because if data turns out to not be in the presumed sorted order, certain operators may produce unexpected results.

Which operators require sorted input?

You should read this document 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 or nonexistent. Make sure that 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.

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.

Since RETL is 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. If these types of requests happen with much frequency or urgency, Oracle Retail will take this into account when setting priorities for efforts for platform proliferation in future releases. Note that additional platform support comes at much lower 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 and writing to databases. Does this mean I can connect to any database?

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.

What changes do I need to make to my RETL 10.x flows in order for RETL 11.x and 12.x to run them?

RETL requirements specify that it must be backward-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.

E

Appendix: 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.

Note: For integral RETL data types, not all possible values of the RETL data type can be stored in the mapped Oracle data type. For example, int8 can hold the value 100, but the mapped Oracle data type, NUMBER(2,0) cannot.

When the data source is an Oracle table (directly or indirectly) the incomplete range handling does not pose a problem because ORAREAD chooses RETL data types that map back to the same or larger Oracle data types in ORAWRITE.

Take care, however, if the data source is a flat file and the destination is an Oracle table. Make sure that you choose the RETL data type that maps to an Oracle data type that can hold the entire range of values. Otherwise, ORAWRITE (through SQL*Loader) will reject a record if it contains a field with a value outside of the Oracle data type's range.

RETL Data Type to Oracle Data Type (ORAWRITE)

RETL Data Type	Oracle Data Type	Range
int8	NUMBER(2,0)	[-99, 99]
int16	NUMBER(4,0)	[-9999, 9999]
int32	NUMBER(9,0)	[-999999999, 999999999]
int64	NUMBER(18,0)	[-999999999999999999, 999999999999999999]
uint8	NUMBER(2,0)	[0, 99]
uint16	NUMBER(4,0)	[0, 9999]
uint32	NUMBER(9,0)	[0, 999999999]
uint64	NUMBER(19,0)	[0, 9999999999999999999]
dfloat	NUMBER	[-1.7976931348623157E+308, 1.7976931348623157E+308]
sfloat	NUMBER	[-3.40282347e+38, 3.40282347e+38]
string	VARCHAR	N/A
date	DATE	[1/1/0001, 12/31/9999]
time	DATE	[00:00:00, 23:59:59]
timestamp	DATE	[00:00:00 1/1/0001, 23:59:59 12/31/9999]

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

F

Appendix: Data Partitioning Quick Reference

This appendix describes how to set up a flow to have multiple data partitions. For more detailed information, see Chapter 7, "RETL Parallel Processing".

Partitioning the Data

Data partitioning does not happen automatically in RETL; you must 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.

G

Appendix: Database Connections Quick Reference

This appendix is a quick reference on how to set up a flow to have database connections using thin and oci clients for ORAREAD/ORAWRITE operators. For more detailed information, see Chapter 6, "Database Operators".

Setting the Environment Variables

The variables that must be configured to connect RETL 13.0.2 to an Oracle database using Oracle Thin/OCI driver are as follows.

- Set ORACLE_HOME environment variable.
- Add ORACLE_HOME/lib to LD_LIBRARY_PATH env variable.
- Set CLASSPATH environment variable to use Oracle JDBC driver from ORACLE_HOME/bin.

> **Note:** ORACLE_HOME refers to the folder where Oracle Client or Database has been installed.
>
> Example: /home/oracle/product/10.2.0. This folder will have sub-folders such as bin, sqlj, network, rdbms, jdbc, jlib, and lib.

ORAREAD

ORAREAD performs a read from Oracle databases. RETL 13.0.2 uses JDBC technology to read data out of Oracle databases. RETL 13.0.2 can connect to Oracle database as either thin client or Oracle Call Interace (OCI) client. By default, operator will be creating the URL for database connection as a thin client. This can be overriden to create the URL for database connection as oci client.

The following examples illustrate the connection to the database using thin and OCI database drivers in case of ORAREAD operator.

Connection to database through thin database driver:

```
<OPERATOR type="oraread">
    <PROPERTY name="sp_prequery" value="exec pre_storedproc"/>
    <PROPERTY name="dbname"          value="RETLdb"/>
    <PROPERTY name="connectstring"   value="username/password"/>
    <!--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>
```

Connection to database through OCI database driver:

```
<OPERATOR type="oraread">
    <OUTPUT name="oraread.v" />
    <PROPERTY name="connectstring" value=" username/password" />
    <PROPERTY name="dbname" value=" RETLdb " />
    <PROPERTY name="jdbcdriverstring"value="oracle.jdbc.driver.OracleDriver" />
    <PROPERTY name="jdbcconnectionstring" value="jdbc:oracle:oci:@" />
    <PROPERTY name="datetotimestamp" value="true"/>
    <PROPERTY name="query" value="select * from test_jdbc"/>
</OPERATOR>
```

ORAWRITE

ORAWRITE performs a load to Oracle databases. RETL 13.0.2 uses SQL*Loader (sqlldr) to load the records. RETL 13.0.2 can connect to Oracle database using either thin client or Oracle Call Interface (OCI) client. By default, the operator will be creating the URL for database connection as a thin client. This can be overriden to create the URL for database connection as and OCI client by adding the jdbcdriver property in the XML flow file.

The following examples illustrate the connection to the database using thin and OCI database drivers in case of ORAWRITE operator.

Connection to database through thin database driver:

```
<OPERATOR type="orawrite">
    <INPUT name="import0.v" />
    <PROPERTY name="threadModel" value="start_thread" />
    <PROPERTY name="dbuserid" value="username/password" />
    <PROPERTY name="dbname" value="RETLdb" />
    <PROPERTY name="tablename" value="ORG_LOC_DM" />
    <PROPERTY name="createtablemode" value="recreate" />
</OPERATOR>
```

Connection to database through OCI database driver:

```
<OPERATOR type="orawrite">
    <INPUT name="import0.v" />
    <PROPERTY name="threadModel" value="start_thread" />
    <PROPERTY name="dbuserid" value="username/password" />
    <PROPERTY name="dbname" value="RETLdb" />
    <PROPERTY name="jdbcdriver" value="oci" />
    <PROPERTY name="tablename" value="ORG_LOC_DM" />
    <PROPERTY name="createtablemode" value="recreate" />
</OPERATOR>
```