Oracle® Fusion Middleware

Administering Oracle GoldenGate Application Adapters

12*c* (12.2.0.1)

E76796-01

June 2016

This document explains how to configure, customize, and run the Oracle GoldenGate Adapters to produce flat files, capture JMS messages and deliver them as an Oracle GoldenGate trail, or read a trail and deliver transactions to a messaging system.

ORACLE

Oracle Fusion Middleware Administering Oracle GoldenGate Application Adapters, 12c (12.2.0.1)

E76796-01

Contents

Preface

This guide contains information about configuring, and running Oracle GoldenGate Adapters to extend the capabilities of Oracle GoldenGate instances.

Audience

This guide is intended for system administrators who are configuring and running Oracle GoldenGate Adapters.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

The Oracle GoldenGate Application Adapters documentation set includes the following components:

- *Release Notes for Oracle GoldenGate for Big Data*
- *Installing Oracle GoldenGate Big Data*
- *Administering Oracle GoldenGate for Big Data*
- *Integrating Oracle GoldenGate for Big Data*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, such as "From the File menu, select **Save**." Boldface also is used for terms defined in text or in the glossary.
italic *`italic`*	Italic type indicates placeholder variables for which you supply particular values, such as in the parameter statement: `TABLE` *`table_name`*. Italic type also is used for book titles and emphasis.
`monospace` `MONOSPACE`	Monospace type indicates code components such as user exits and scripts; the names of files and database objects; URL paths; and input and output text that appears on the screen. Uppercase monospace type is generally used to represent the names of Oracle GoldenGate parameters, commands, and user-configurable functions, as well as SQL commands and keywords.
UPPERCASE	Uppercase in the regular text font indicates the name of a utility unless the name is intended to be a specific case.
{ }	Braces within syntax enclose a set of options that are separated by pipe symbols, one of which must be selected, for example: `{`*`option1`*` | `*`option2`*` | `*`option3`*`}`.
[ ]	Brackets within syntax indicate an optional element. For example in this syntax, the `SAVE` clause is optional: `CLEANUP REPLICAT` *`group_name`* `[, SAVE` *`count`*`]`. Multiple options within an optional element are separated by a pipe symbol, for example: `[`*`option1 | option2`*`]`.

Part I

Understanding Oracle GoldenGate Application Adapters

This part of the book describes the concepts and basic structure of the Oracle GoldenGate Application Adapters.

Part I contains the following chapters:

1

Understanding Oracle GoldenGate Adapters

This chapter provides an overview of the Oracle GoldenGate Adapters that integrate with Oracle GoldenGate instances to bring in Java Message Service (JMS) information or to deliver information as JMS messages or files.

This chapter includes the following sections:

1.1 Oracle GoldenGate Application Adapters Overview

This section provides an overview of the Oracle GoldenGate Application Adapters.

1.1.1 Oracle GoldenGate Integration

Oracle GoldenGate Application Adapters integrate with core Oracle GoldenGate instances.

The core Oracle GoldenGate product:

- Captures transactional changes from a source database
- Sends and queues these changes as a set of database-independent files called the Oracle GoldenGate trail
- Optionally alters the source data using mapping parameters and functions
- Applies the transactions in the trail to a target system database

Oracle GoldenGate performs this capture and apply in near real-time across heterogeneous databases, platforms, and operating systems.

1.1.2 Oracle GoldenGate Application Adapter Integration Options

The Oracle GoldenGate Application Adapters integrate with installations of the Oracle GoldenGate core product to do one of the following:

- Read JMS messages and deliver them as an Oracle GoldenGate trail
- Read an Oracle GoldenGate trail and deliver transactions to a JMS provider or other messaging system or custom application
- Read an Oracle GoldenGate trail and write transactions to a file that can be used by other applications

1.1.2.1 Capturing Transactions to a Trail

Oracle GoldenGate message capture can be used to read messages from a queue and communicate with an Oracle GoldenGate Extract process to generate a trail containing the processed data.

The message capture processing is implemented as a Vendor Access Module (VAM) plug-in to a generic Extract process. A set of properties, rules and external files provide messaging connectivity information and define how messages are parsed and mapped to records in the target GoldenGate trail.

Currently this adapter supports capture from JMS text messages.

1.1.2.2 Applying Transactions from a Trail

Oracle GoldenGate delivery can be used to apply transactional changes to targets other than a relational database: for example, ETL tools (DataStage, Ab Initio, Informatica), JMS messaging, or custom APIs. There are a variety of options for integration with Oracle GoldenGate:

- Flat file integration: predominantly for ETL, proprietary or legacy applications, Oracle GoldenGate file writer can write micro batches to disk to be consumed by tools that expect batch file input. The data is formatted to the specifications of the target application such as delimiter separated values, length delimited values, or binary. Near real-time feeds to these systems are accomplished by decreasing the time window for batch file rollover to minutes or even seconds.
- Messaging: transactions or operations can be published as messages (e.g. in XML) to JMS. The JMS provider is configurable; examples include ActiveMQ, JBoss Messaging, TIBCO, WebLogic JMS, WebSphere MQ and others.
- Java API: custom event handlers can be written in Java to process the transaction, operation and metadata changes captured by Oracle GoldenGate on the source system. These custom Java handlers can apply these changes to a third-party Java API exposed by the target system.

All three options have been implemented as extensions to the core Oracle GoldenGate product using Oracle GoldenGate's user exit interface, a C API.

- For the flat file integration, Oracle GoldenGate File Writer provides a user exit library that is dynamically linked into the Oracle GoldenGate Extract process. Configuration is done using a properties file, and no programming is required.
- For Java integration using either JMS or the Java API, use Oracle GoldenGate for Java.

1.2 Using Oracle GoldenGate Application Adapters Properties

The Oracle GoldenGate Application Adapters are configured and controlled through predefined properties.

1.2.1 Values in Property Files

All properties in Oracle GoldenGate Application Adapter property files are of the form:

```
property.name=value
```

The value may be single or comma-delimited strings, an integer, or a boolean value.

1.2.2 Location of Property Files

Sample Oracle GoldenGate Application Adapter property files are installed to the `AdapterExamples` subdirectory of the installation directory. These files should be changed as needed and then moved to the `dirprm` subdirectory.

You must specify each of these property files through parameters or environmental variables as explained below. These settings allow you to change the name or location, but it is recommended that you do *not* change them unless there is an unavoidable requirement.

The following sample files are included:

- `ffwriter.properties`

  This stores the properties for the file writer. It is set with the `CUSEREXIT` Extract parameter.

- `jmsvam.properties`

  This stores properties for the JMS message capture VAM. This is set with the Extract `VAM` parameter.

- `javaue.properties`

  This stores properties for the combined user exit and Java application used for message delivery. It is set through the environmental variable:

  ```
  SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")
  ```

  Optionally, the java application properties and native user exit library properties can be in separate files. To do this set `GGS_USEREXIT_CONF` to the user exit property file and `GGS_JAVAUSEREXIT_CONF` to the Java application properties fie.

1.2.3 Using Comments in the Property File

Comments can be entered in the properties file with the # prefix at the beginning of the line. For example:

```
# This is a property comment
some.property=value
```

Properties themselves can also be commented. This allows testing configurations without losing previous property settings.

1.2.4 Variables in Property Names

Some properties have a variable in the property name. This allows identification of properties that are to be applied only in certain instances.

For example, you can declare more than one file writer using `goldengate.flatfilewriter.writers property` and then use the name of the file writer to set the properties differently:

1. Declare two file writers named `writer` and `writer2`:

   ```
   goldengate.flatfilewriter.writers=writer,writer2
   ```

2. Specify the properties for each of the file writers:

```
writer.mode=dsv
writer.files.onepertable=true
writer2.mode=ldv
writer2.files.onpertable=false
```

1.3 Oracle GoldenGate Documentation

For information on installing and configuring the core Oracle GoldenGate software for use with the Oracle GoldenGate File Writer or Java adapters, see the Oracle GoldenGate documentation:

- Installation and Setup guides: There is one such guide for each database that is supported by Oracle GoldenGate for Mainframe. It contains system requirements, pre-installation and post-installation procedures, installation instructions, and other system-specific information for installing the Oracle GoldenGate for Mainframe replication solution.
- : Explains how to plan for, configure, and implement the Oracle GoldenGate for Mainframe replication solution on the Windows and UNIX platforms.
- : Contains detailed information about Oracle GoldenGate for Mainframe parameters, commands, and functions for the Windows and UNIX platforms.

2

Introducing the File Writer

This chapter provides an overview of the Oracle GoldenGate Adapter for Flat Files. This adapter provides a user exit library that is dynamically linked into an Oracle GoldenGate Extract process. The library may be a `.ddl` or an `.so` format. It is configured using a properties file so no programming is required. The Oracle GoldenGate Adapter for Flat Files currently only works with an Oracle GoldenGate Extract process; it does not work a Replicat process.

This chapter includes the following sections:

2.1 Overview of the Adapter for Flat Files

Oracle GoldenGate Adapter for Flat Files outputs transactional data captured by Oracle GoldenGate to rolling flat files to be used by a third party product.

The user exit supports two modes of output:

- DSV – Delimiter Separated Values (commas are an example)
- LDV – Length Delimited Values

It can output data:

- All to one file
- One file per table
- One file per operation code

The user exit can roll over based on time and/or size criteria. It flushes files and maintains checkpoints whenever Oracle GoldenGate checkpoints to ensure recovery. It writes a control file containing a list of rolled over files for synchronization with the supported data integration product and can also produce a summary file for use in auditing.

Additional properties control formatting (delimiters, other values), directories, file extensions, metadata columns (such as table name, file position, etc.) and data options.

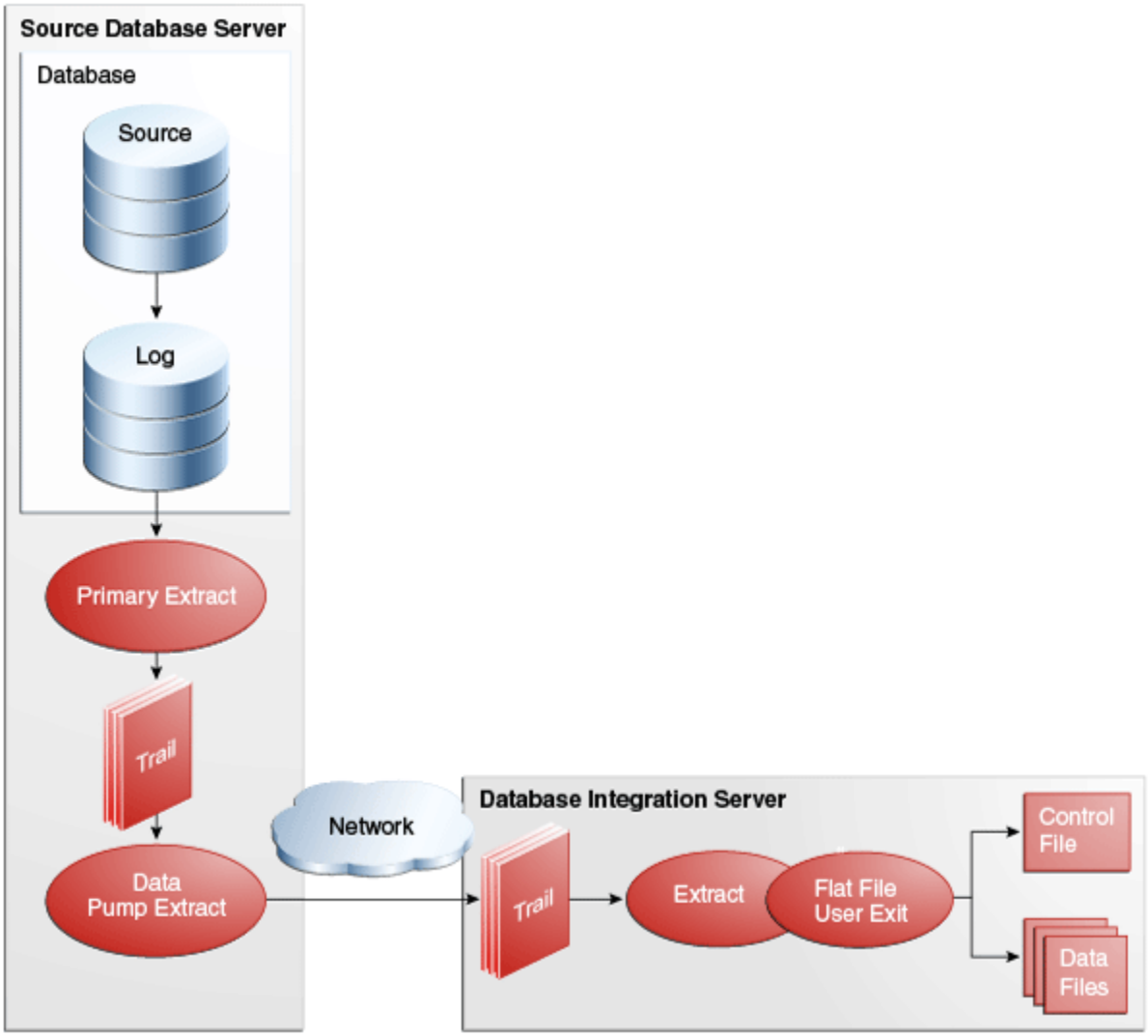
2.2 Typical Configuration

The following diagram shows a typical configuration for the Oracle GoldenGate Adapter for Flat Files.

In this configuration, transactions from the source database are captured by a Primary Extract process and written to an Oracle GoldenGate trail. A data pump Extract send

this trail to the Oracle GoldenGate Adapter Extract, whose associated user exit process writes the data to flat files formatted to suit a third party application.

Figure 2-1 Oracle GoldenGate for Flat File



3

Introducing the Java Adapter

This chapter describes the Oracle GoldenGate Adapter for Java. The Oracle GoldenGate Adapter for Java implements 1) the capture of Java Message Service (JMS) messages to send for processing into Oracle GoldenGate trail data, and 2) the processing of transactional data captured by Oracle GoldenGate to be delivered as JMS messages.

This chapter includes the following sections:

3.1 Oracle GoldenGate VAM Message Capture

Oracle GoldenGate message capture connects to JMS messaging to parse messages and send them through a VAM interface to an Oracle GoldenGate Extract that builds an Oracle GoldenGate trail of message data. This allows JMS messages to be delivered to an Oracle GoldenGate system running for a target database.

Using Oracle GoldenGate JMS message capture requires two components:

- The dynamically linked shared VAM library that is attached to the Oracle GoldenGate Extract process.
- A separate utility, Gendef, that uses the message capture properties file and parser-specific data definitions to create an Oracle GoldenGate source definitions file.

3.1.1 Message Capture Configuration Options

The options for configuring the three parts of message capture are:

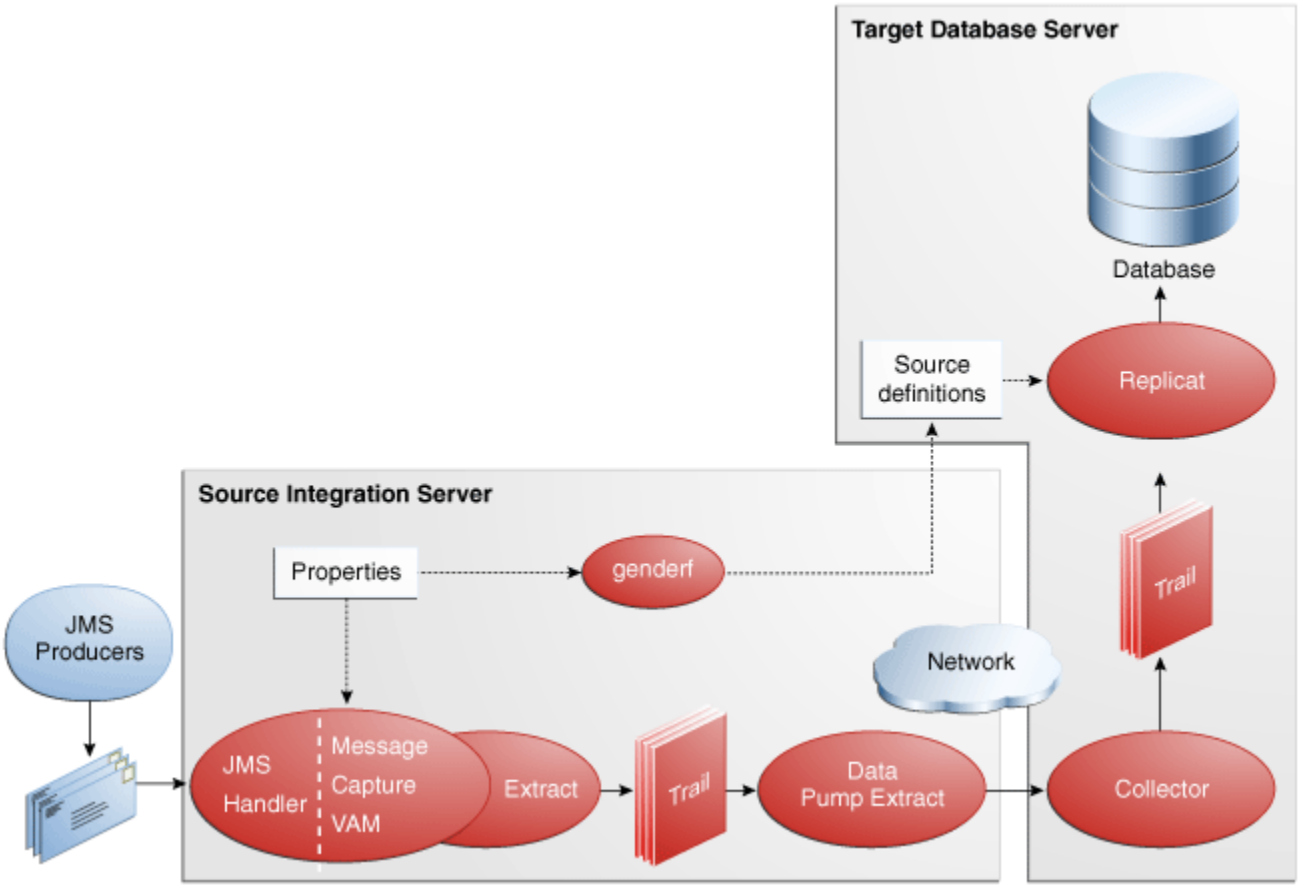
- Message connectivity: Values in the property file set connection properties such as the Java class path for the JMS client, the JMS source destination name, JNDI connection properties, and security information.
- Parsing: Values in the property file set parsing rules for fixed width, comma delimited, or XML messages. This includes settings such as the delimiter to be used, values for the beginning and end of transactions and the date format.
- VAM interface: Parameters that identify the VAM, `dll`, or `so` library and a property file are set for the Oracle GoldenGate core Extract process.

3.1.2 Typical Configuration

The following diagram shows a typical configuration for capturing JMS messages.

In this configuration, JMS messages are picked up by the Oracle GoldenGate Adapter JMS Handler and transferred using the adapter's message capture VAM to an Extract process. The Extract writes the data to a trail which is sent over the network by a Data Pump Extract to an Oracle GoldenGate target instance. The target Replicat then uses the trail to update the target database.

Figure 3-1 Configuration for JMS Message Capture



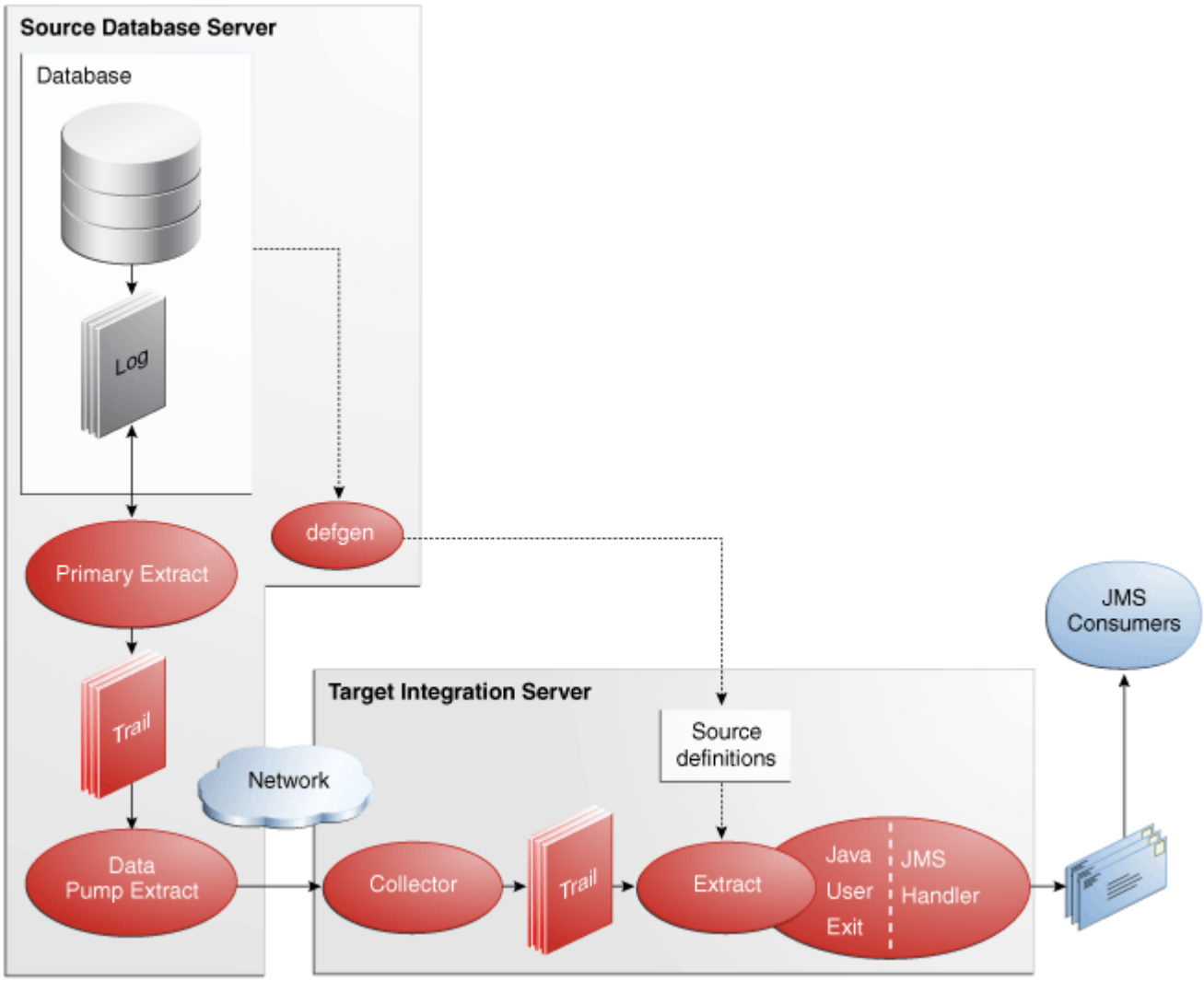
3.2 Oracle GoldenGate Java User Exit

Through the Oracle GoldenGate Java API, transactional data captured by Oracle GoldenGate can be delivered to targets other than a relational database, such as a JMS (Java Message Service), files written to disk, or an integration with a custom application Java API.

Oracle GoldenGate for Java provides the ability to execute Java code from an Oracle GoldenGate Extract process or Replicat process. Replicat is the recommended way to integrate Java delivery because of the transaction grouping functionality and improved checkpointing provided by Replicat. Using Oracle GoldenGate for Java requires two components:

- A dynamically linked or shared library implemented in C or C++, integrating as a User Exit (UE). The user exit shared libraries are different for Extract and Replicat. For Extract the library is `libggjava_ue.so` on Linux and UNIX; it is `gjava_ue.dll` for Windows. For Replicat, the library is `libggjava.so` for Linux and UNIX; it is `ggjava.dll` for Windows.
- A set of Java libraries (JARS) that comprise the Oracle GoldenGate Java API. This Java framework communicates with the user exit through the Java Native Interface (JNI). The interface of the Java layer is identical for running with either the Extract process or the Replicat process.

Figure 3-2 Configuration for Delivering JMS Messages



3.2.1 Delivery Configuration Options

The dynamically linked library is configurable using a simple properties file. The Java framework is loaded by the user exit and is also initialized by a properties file. Application behavior can be customized by:

- Editing the property files; for example to:

  - Set host names, port numbers, output file names, JMS connection settings;
  - Add/remove targets (such as JMS or files) by listing any number of active handlers to which the transactions should be sent;
  - Turn on/off debug-level logging, etc.
  - Identify which message format should be used.
- Customizing the format of messages sent to JMS or files. Message formats can be custom tailored by:
  - Setting properties for the pre-existing format process (for fixed-length or field-delimited message formats);
  - Customizing message templates, using the Velocity template macro language;
  - (Optional) Writing custom Java code.
- (Optional) Writing custom Java code to provide custom handling of transactions and operations, do filtering, or implementing custom message formats.

There are existing implementations (handlers) for sending messages via JMS and for writing out files to disk. There are several predefined message formats for sending the messages (e.g. XML or field-delimited); or custom formats can be implemented using templates. Each handler has documentation that describes its configuration properties; for example, a file name can be specified for a file writer, and a JMS queue name can be specified for the JMS handler. Some properties apply to more than one handler; for example, the same message format can be used for JMS and files.

3.3 Running with Extract

This section explains how to run Java Adapter with the Oracle GoldenGate Extract process.

3.3.1 Extract Configuration

The following

```
EXTRACT hdfs
discardfile ./dirrpt/avro1.dsc, purge
--SOURCEDEFS ./dirdef/dbo.def
CUSEREXIT libjavaue.so CUSEREXIT PASSTHRU, INCLUDEUPDATEBEFORES, PARAMS "dirprm/
hdfs.props"
GETUPDATEBEFORES
TABLE dbo.*;
```

The following is explanation of the Replicat configuration entries:

`EXTRACT hdfs` - The Extract process name.

`discardfile ./dirrpt/avro1.dsc, purge` - Set the discard file

`--SOURCEDEFS ./dirdef/dbo.def` - Source definitions are not required for 12.2 trial files.

`CUSEREXIT libjavaue.so CUSEREXIT PASSTHRU, INCLUDEUPDATEBEFORES, PARAMS "dirprm/hdfs.props"` - Set you exit shared library, and point to the Java Adapter Properties file

`GETUPDATEBEFORES` - Get update before images.

`TABLE dbo.*;` - Select which tables to replicate or exclude to filter.

3.3.2 Adding the Extract Process

```
ADD EXTRACT hdfs, EXTTRAILSOURCE ./dirdat/gg
START hdfs
```

3.3.3 Extract Grouping

The Extract process provides no functionality for transaction grouping. However, transaction grouping is still possible when integrating Java Delivery with the Extract process. The Java Delivery layer enables transaction grouping with configuration in the Java Adapter properties file.

1. `gg.handler.name.mode`

   To enable grouping, the value of this property must be set to tx.

2. `gg.handler.name.maxGroupSize`

   Controls the maximum number of operations that can be held by an operation group - irrespective of whether the operation group holds operations from a single transaction or multiple transactions.

   The operation group will send a transaction commit and end the group as soon as this number of operations is reached. This property leads to splitting of transactions across multiple operation groups.

3. `gg.handler.name.minGroupSize`

   This is the minimum number of operations that must exist in a group before the group can end.

   This property helps to avoid groups that are too small by grouping multiple small transactions into one operation group so that it can be more efficiently processed.

Note:

`maxGroupSize` should always be greater than or equal to `minGroupSize`; that is, `maxGroupSize >= minGroupSize`.

Note:

It is *not* recommended to use the Java layer transaction grouping when running Java Delivery with the Replicat process. If running with the Replicat process, you should use Replicat transaction grouping controlled by the `GROUPTRANSOPS` Replicat property.

3.4 Running with Replicat

This section explains how to run the Java Adapter with the Oracle GoldenGate Replicat process.

3.4.1 Replicat Configuration

The following is an example of a Replicat process properties file for Java Adapter.

```
REPLICAT hdfs
TARGETDB LIBFILE libggjava.so SET property=dirprm/hdfs.properties
--SOURCEDEFS ./dirdef/dbo.def
DDL INCLUDE ALL
GROUPTRANSOPS 1000
MAPEXCLUDE dbo.excludetable
MAP dbo.*, TARGET dbo.*;
```

The following is explanation of the Replicat configuration entries:

`REPLICAT hdfs` - The name of the Replicat process.

`TARGETDB LIBFILE libggjava.so SET property=dirprm/hdfs.properties` - Names the target database as you exit `libggjava.so` and sets the Java Adapters Property file to `dirprm/hdfs.properties`

`--SOURCEDEFS ./dirdef/dbo.def` - Sets a source database definitions file. Commented out because Oracle GoldenGate 12.2.0.1 trail files provide metadata in trail.

`GROUPTRANSOPS 1000` - To group 1000 transactions from the source trail files into a single target transaction. This is the default and improves the performance of Big Data integrations.

`MAPEXCLUDE dbo.excludetable` - To identify tables to exclude.

`MAP dbo.*, TARGET dbo.*;` - Shows the mapping of input to output tables.

3.4.2 Adding the Replicat Process

The command to add and start the Replicat process in `ggsci` is the following:

```
ADD REPLICAT hdfs, EXTTRAIL ./dirdat/gg
START hdfs
```

3.4.3 Replicat Grouping

The Replicat process provides the Replicat configuration property `GROUPTRANSOPS` to control transaction grouping. By default, the Replicat process implements transaction grouping of 1000 source transactions into a single target transaction. If you want to turn off transaction grouping then the `GROUPTRANSOPS` Replicat property should be set to `1`.

3.4.4 Replicat Checkpointing

`CHECKPOINTTABLE` and `NODBCHECKPOINT` are not applicable for Java Delivery with Replicat. Beside Replicat checkpoint file (`.cpr`), additional checkpoint file (`dirchk/<group>.cpj`) will be created that contains information similar to `CHECKPOINTTABLE` in Replicat for RDBMS.

3.4.5 Unsupported Replicat Features

The following Replicat features are not supported in this release:

- `BATCHSQL`
- `SQLEXEC`
- Stored procedure
- Conflict resolution and detection (CDR)

- REPERROR

3.4.6 Mapping Functionality

The Oracle GoldenGate Replicat process supports mapping functionality to custom target schemas. This functionality is not available using the Oracle GoldenGate Extract process. You must use the Metadata Provider functionality to define a target schema or schemas and then use the standard Replicat mapping syntax in the Replicat configuration file to define the mapping. Refer to the Oracle GoldenGate Replicat documentation to understand the Replicat mapping syntax in the Replication configuration file.

4

Configuring Logging

This chapter describes the default logging for the Oracle GoldenGate Adapters and explains how to configure a different logging option.

Application Adapters Default Logging

Logging is set up by default for the Oracle GoldenGate Application Adapters.

4.1.1 Default Implementation Type

The default type of implementation for the Oracle GoldenGate Adapters is the `JDK` option. This is the built-in Java logging called `java.util.logging` (`JUL`).

4.1.2 Default Message Logging

The default log file is created in the standard report directory. It is named for the associated Extract process. Problems are logged to the report file and the log file.

4.1.2.1 Logging Problems

An overview of a problem is written to the Extract Report file and the details of the problem are written to the log file.

4.1.2.2 Log File Name

By default log files are written to the *`installation_directory`*`/dirrpt` directory. The name of the log file includes the Extract *`group_name`* and it has an extension of `log`.

4.2 Changing the Default Logging

The logging for Oracle GoldenGate Adapters can be changed from `JUL` to another type, debug can be turned on, and the configuration file can be specified.

4.2.1 Changing the Logging Type

The Java-based logging utility, Log4j is the preferred method of logging for Oracle GoldenGate Application Adapters.

To change the logging implementation type, set the `gg.log` property to `log4j` or `logback`. For example set:

```
gg.log=log4j
```

4.2.2 Changing the Logging Configuration

To designate a specific configuration file, set `jvm.bootoptions` to the system property that defines it. This will implicitly set the implementation type and append

the appropriate binding to the class path. Contact Oracle Support for help using this option.

4.2.3 Enabling Debug

To enable debug logging, set the `gg.log.level` property to debug as shown below.

```
gg.log.level=debug
```

Part II

Creating Flat Files

This part explains how to configure and administer Oracle GoldenGate adapters that write to flat files.

Part III contains the following chapters:

- Configuring the Flat File Adapter
- Using the Flat File Adapter
- Using Predefined Defaults and Formats
- Flat File Properties

5

Configuring the Flat File Adapter

This chapter explains how to configure the Oracle GoldenGate Adapter for writing flat files by setting user exit parameters and file writer properties.

This chapter includes the following sections:

- Configuring the Adapter for Writing Flat Files
- Recommended Data Integration Approach
- Producing Data Files

5.1 Configuring the Adapter for Writing Flat Files

Figure Figure 5-1, shows a typical configuration for an that is writing flat files. Transactions are captured from the source database by a Primary Extract process that writes the data to an Oracle GoldenGate trail. A Data Pump Extract is then used to send the transactions to a trail that will be read by the Adapter Extract. The Oracle GoldenGate Application Adapters Flat File writer only integrates with the Oracle GoldenGate Extract process; it does not currently work with the Replicat process.

To configure the source database system:

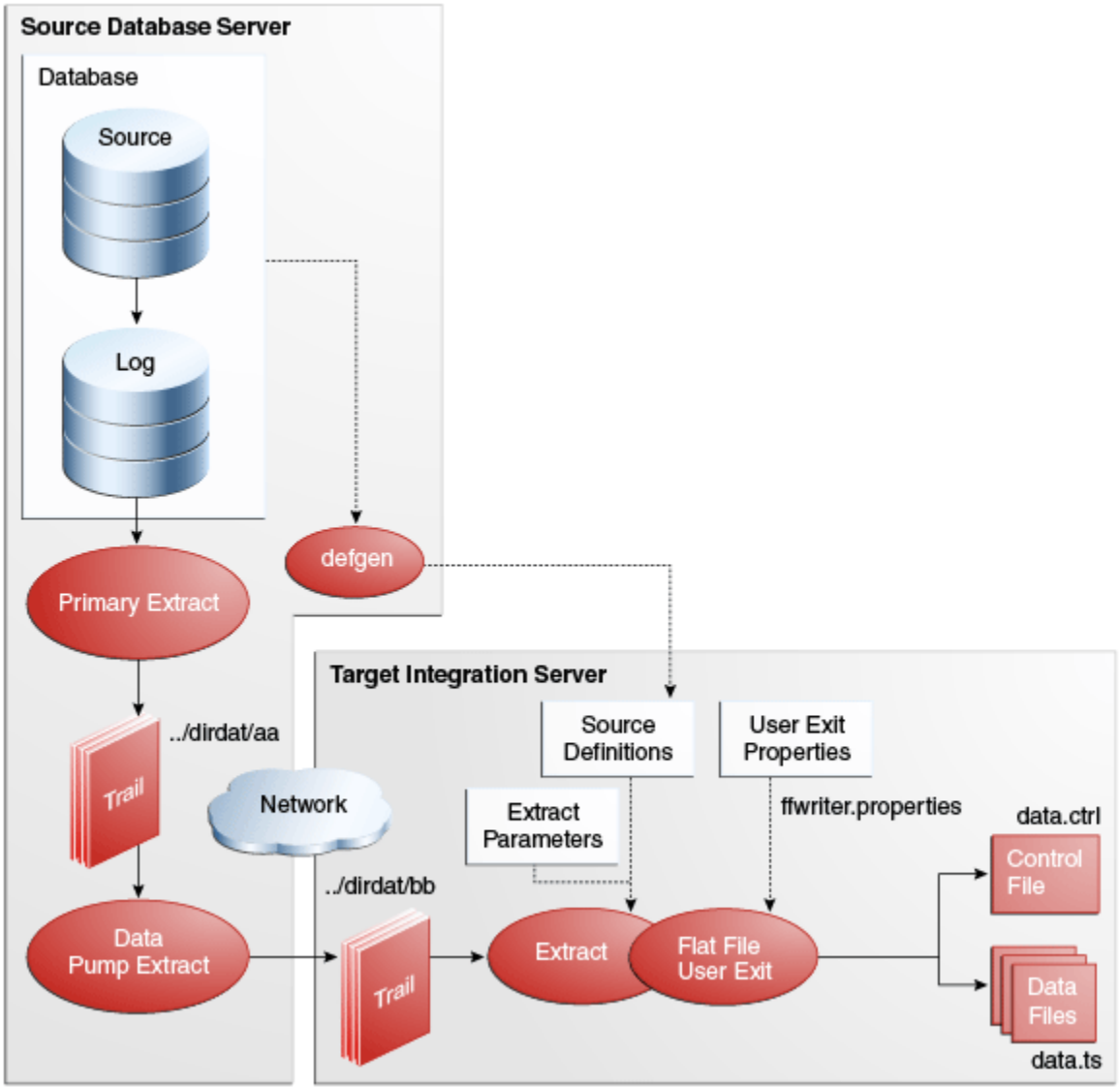
```
GGSCI > ADD EXTRACT pump, EXTTRAILSOURCE dirdat/aa
GGSCI > ADD RMTTRAIL dirdat/bb, EXTRACT pump, MEGABYTES 20
```

To configure the data integration:

```
GGSCI > ADD EXTRACT ffwriter, EXTTRAILSOURCE dirdat/bb
```

The sample process names and trail names used above can be replaced with any valid name. Process names must be 8 characters or less, trail names must be two characters.

Figure 5-1 Typical Configuration For Writing Flat Files



5.1.1 User Exit Extract Parameters

The user exit Extract parameters (`ffwriter.prm`) are as follows:

Parameter	Description
`EXTRACT FFWRITER`	All Extract parameter files start with the Extract name. In this case it is the user exit's file writer name.
`SOURCEDEFS dirdef/hr_ora.def`	A source definitions file to determine trail contents.

Parameter	Description
`CUSEREXIT flatfilewriter.dll CUSEREXIT PASSTHRU, INCLUDEUPDATEBEFORES, PARAMS ffwriter.properties`	The `CUSEREXIT` parameter options: • `flatfilewriter.dll` is the name of the user exit .dll or .so library. • `CUSEREXIT` is the name of the user exit routine that will be invoked (case sensitive). • `PASSTHRU` specifies that the Extract process does not need to write a trail. • `INCLUDEUPDATEBEFORES` allows both the before and after image to be included in the output. It is also required for consistency purposes and transaction tracking. • `PARAMS` allows you to specify the name of the user exit properties file.
`TABLE HR.*;`	Specifies a list of tables to process.

5.1.2 User Exit Properties

The user exit reads properties from the file identified in `CUSEREXIT PARAMS`. The default is to read from `ffwriter.properties`.

The properties file contains details of how the user exit should operate. For more information on individual properties see Flat File Properties .

5.2 Recommended Data Integration Approach

To take best advantage of the micro-batch capabilities, customers should do the following in their data integration tool:

1. Wait on the control file
2. Read list of files to process from the control file
3. Rename the control file
4. Iterate over the comma-delimited list of files read from the control file
5. Process each data file, deleting the data file when complete
6. Delete the renamed control file

On startup, the data integration tool should check for the renamed control file to see if it needs to recover from previously failed processing

When the control file is renamed, the user exit will write a new one on the first file rollover, which will contain the list of files for the next batch.

If the user exit has been configured to also output a summary file, the data integration tool can optionally also read that summary file and cross-check the number of operations it has processed with the data in the summary file for each processed data file.

5.3 Producing Data Files

Data files are produced by configuring a *writer* in the user exit properties. A single user exit properties file can have multiple writers, which allows for the generation of multiple differently formatted output data files for the same input data.

Writers are added by name to the `goldengate.flatfilewriter.writers` property. For example:

```
goldengate.flatfilewriter.writers=dsvwriter,diffswriter,binarywriter
```

The remainder of the properties file contains detailed properties for each of the named writers where the properties are prefixed by the writers name. For example:

```
dsvwriter.files.onepertable=true
binarywriter.files.onepertable=false
binarywriter.files.oneperopcode=true
```

Each writer can output all the data to a single (rolling) data file, or produce one (rolling) data file per input table or operation type. This is controlled by the `files.onepertable` and `files.oneperopcode` properties as shown in the example above.

The data written by each writer can be in one of two output formats controlled by the mode property. This can either be:

- DSV – Delimiter Separated Values
- LDV – Length Delimited Values

For example:

```
dsvwriter.mode=dsv
binarywriter.mode=ldv
```

When data files are first written to disk, they have a temporary extension. Once the file meets rollover criteria, the extension is switched to the rolled extension. If control files are used, the final file name is added to the list in the control file, creating the control file if necessary. Also, if a file level statistics summary is being generated, it will be created upon rollover of the file.

The output directory (for data files and control files separately), temporary extension, rolled extension, control extension and statistical summary extension can all be configured through properties. For output configuration details see Output File Properties .

Each data file that is written follows a naming convention which depends on the output style. For files written one per table, the name includes the table name, for example:

```
MY.TABLE_2013-08-03_11:30:00_data.dsv
```

For files written with all data in one file, the name does not include the table name, for example:

```
output_2013-08-03_11:30:00_data.dsv
```

In addition to the basic data contents, additional *metadata columns* can be added to the output data to aid in data consumption. This includes the schema (owner) and table

information, source commit timestamp, Oracle GoldenGate read position and more. For a detailed description of metadata columns see Metadata Columns .

The contents of the data file depend on the mode, the input data, and the various properties determining which (if any) metadata columns are added, whether column names are included, whether before images are included etc. For full details of all properties governing the output data see section Data Content Properties .

6

Using the Flat File Adapter

This chapter discusses how to manage on-going operation of your system by managing file rollover, gathering statistics on your Oracle GoldenGate adapter instance to help you tune your system, managing the processes, and handling errors.

This chapter includes the following sections:

- Working with Control Files
- Working with Statistical Summaries
- Managing Oracle GoldenGate processes
- Trail Recovery Mode
- Locating Error Messages

6.1 Working with Control Files

Control files store information on which data files have rolled over. If the control file exists, it will be appended to; if it does not exist it will be created. For writers that output all data to one file, a single control file will be created. If the writer is outputting to one file per table or operation type, a control file will also be created per table or operation type.

The generation of a control file, its output directory, prefix, and extension are controlled by the properties defined in Output File Properties .

Each control file contains a comma-delimited list of data files that have been rolled over since the control file was created. The files are listed in the order they were rolled over. This allows data integration tools to ensure that data files are read in the correct order and that they have all been consumed.

6.2 Working with Statistical Summaries

Summary statistics about the data production process can be collected. This statistical summary information can be written to the Oracle GoldenGate report file or individual summary files.

When writing to the report file, the user can decide if this information should be written when files are rolled over, or periodically based on a time period. Information written to the report file is output in a standard fashion, and contains total records, totals for each database operation type, deltas since the last report, rate information, and detail information for each table.

When writing to individual summary files, a file is created for each rolled-over file. The statistical information for the rolled-over file is listed separated by a delimiter. The extension of the summary file, the data to be output, data delimiter, and line delimiter can all be controlled.

Statistics and Reporting , contains detailed property information about statistics and summary files.

6.3 Managing Oracle GoldenGate processes

The processes involved in a typical data integration solution include:

- A primary Extract process, capturing transactional data from the source database
- A PASSTHRU data pump Extract moving the captured transactional data across the network from the source database machine to the data integration machine
- A delivery data pump Extract configured to run the user exit

Typically, the original capture and PASSTHRU data pump are part of one Oracle GoldenGate installation and the delivery data pump is part of a second installation. Both of these installations will also need to have an Oracle GoldenGate Manager process running.

Processes within these installations are managed through the Oracle GoldenGate GGSCI command line with simple commands like start and stop. Full details of managing these processes and their configuration can be found in the *Oracle GoldenGate Administrator's Guide*.

6.4 Trail Recovery Mode

The RECOVERYOPTIONS Extract parameter determines the restart behavior of an Extract that abends while writing to a trail. APPENDMODE is the default for release 10 trails and later. When an abended Extract restarts in append mode, it writes a recovery marker to the trail followed by the entire transaction that was interrupted.

When the Oracle GoldenGate Flat File Adapter file writer reads this trail, it receives the partial transaction followed by the recovery marker indicating the partial transaction should be discarded. The file writer then repositions itself in the output file to the beginning of the partial transaction and overwrites it with the next transaction from the trail file.

6.5 Locating Error Messages

There are three types of errors that may occur in the operation of the Oracle GoldenGate for Flat File:

1. The Extract process running the user exit does not start
2. The process starts, but abends at some point later
3. The process runs successfully, but the data is incorrect or non-existent

In the first two cases, there are a number of places to look for error messages:

- The standard ggserr.log file, which contains basic information about Oracle GoldenGate processes, their run history and a brief error message if any error occurred.
- The Oracle GoldenGate report file for the Extract process running the user exit, found in the `dirrpt` subdirectory. For example, if the process name is `ffwriter`, the report file would be ffwriter.rpt. This may contain more detailed information

about the error, especially if it is a problem in the Oracle GoldenGate core product rather than the user exit.

- In the user exits log file, the name of which depends on the `log.logname` property. If this file does not exist, the user exit most likely did not start up and the report file should help isolate that problem.

Troubleshooting the Flat File Adapter contains more information on error handling.

7

Using Predefined Defaults and Formats

This chapter explains the standard and application specific property defaults that are included with the Oracle GoldenGate Adapters.

This chapter includes the following sections:

7.1 Overview of Predefined Defaults and Formats

To make the task of setting the file writer properties easier, the Oracle GoldenGate Adapter:

- Sets defaults for some standard properties
- Includes predefined sets of properties that create a typical format for particular applications receiving the output.

  Using these predefined formats changes the standard defaults based on what certain applications typically expect. You can override a format property by manually setting it in the properties file. When processing a property from the format, the system first checks to see if that property is set in the properties file itself. If it is, the property file setting is used, otherwise the format setting is used.

7.1.1 Default Properties

All writers use the following properties. The values shown for each property are the defaults.

```
writer.files.data.rootdir=./out
writer.files.data.rollover.time=10
writer.files.data.rollover.size=100000
writer.files.data.norecords.timeout=10
writer.files.control.use=true
writer.files.control.ext=.ctrl
writer.files.control.rootdir=./out
```

7.1.2 Specifying Consumer Formats

Use the `template` property to specify the name of the format file that is to be used.

Syntax

writer.template=*format_name*

writer - Specifies the name of the flat file writer.

format_name - Specifies the name of an existing file of default property settings for a particular application. Valid sets include:

`SIEBEL` - Properties to create one DSV format output file with transaction information for Siebel Remote.

`ABINITIO` - Properties to create LDV format output for consumption by Ab Initio.

`NETEZZA` - Properties to create one DSV format output file per table for Netezza.

`GREENPLUM` - Properties to create one DSV format output file for Greenplum.

`COMMADELIM` - Properties to create one comma delimited output file per table.

7.2 Siebel Remote Format

```
goldengate.userexit.outputmode=txs
goldengate.userexit.buffertxs=true
goldengate.userexit.datetime.removecolon=true
goldengate.userexit.timestamp=utc
writer.mode=DSV
writer.rawchars=false
writer.includebefores=true
writer.includecolnames=true
writer.omitvalues=false
writer.diffsonly=false
writer.omitplaceholders=true
writer.files.onepertable=false
writer.files.data.ext=_data.csv
writer.files.data.tmpext=_data.csv.temp
writer.files.data.bom.code=efbbbf
writer.dsv.nullindicator.chars=NULL
writer.dsv.nullindicator.escaped.chars=
writer.dsv.fielddelim.chars=,
writer.dsv.fielddelim.escaped.chars=
writer.dsv.linedelim.chars=\n
writer.dsv.linedelim.escaped.chars=
writer.dsv.quotes.chars="
writer.dsv.quotes.escaped.chars=""
writer.dsv.quotealways=true
writer.groupcols=true
writer.afterfirst=true
writer.begintx.metacols="B","S",position,"GGMC",%LAST_UPD_BY,"1",
numops
writer.metacols="R",opcode,%ROW_ID,%LAST_UPD_BY,%LAST_UPD,
%MODIFICATION_NUM,%CONFLICT_ID,position,txoppos,table,"","","","","",
"",%DB_LAST_UPD,%DB_LAST_UPD_SRC,numcols
writer.metacols.DB_LAST_UPD.omit=true
writer.metacols.DB_LAST_UPD_SRC.omit=true
writer.metacols.opcode.updatepk.chars=U
```

```
writer.metacols.position.format=dec
writer.endtx.metacols="E"
```

7.3 Ab Initio Format

```
writer.mode=LDV
writer.files.onepertable=false
writer.files.data.ext=.data
writer.files.data.tmpext=.temp
writer.metacols=position,timestamp,opcode,txind,schema,table
writer.metacols.timestamp.fixedlen=26
writer.metacols.schema.fixedjustify=right
writer.metacols.schema.fixedpadchar.chars=Y
writer.metacols.opcode.fixedlen=1
writer.metacols.opcode.insert.chars=I
writer.metacols.opcode.update.chars=U
writer.metacols.opcode.delete.chars=D
writer.metacols.txind.fixedlen=1
writer.metacols.txind.begin.chars=B
writer.metacols.txind.middle.chars=M
writer.metacols.txind.end.chars=E
writer.metacols.txind.whole.chars=W
writer.metacols.position.format=dec
writer.ldv.vals.missing.chars=M
writer.ldv.vals.present.chars=P
writer.ldv.vals.null.chars=N
writer.ldv.lengths.record.mode=binary
writer.ldv.lengths.record.length=4
writer.ldv.lengths.field.mode=binary
writer.ldv.lengths.field.length=2
writer.statistics.period=onrollover
writer.statistics.tosummaryfile=true
writer.statistics.overall=true
writer.statistics.summary.fileformat=schema,table,schemaandtable,total,
gctimestamp,ctimestamp
writer.statistics.summary.delimiter.chars=|
writer.statistics.summary.eol.chars=\n
```

7.4 Netezza Format

```
writer.mode=DSV
writer.rawchars=false
writer.includebefores=false
writer.includecolnames=false
writer.omitvalues=false
writer.diffsonly=false
writer.omitplaceholders=false
writer.files.onepertable=true
writer.files.data.ext=_data.dsv
writer.files.data.tmpext=_data.dsv.temp
writer.dsv.nullindicator.chars=
writer.dsv.fielddelim.chars=;
writer.dsv.fielddelim.escaped.chars=
```

7.5 Greenplum Format

```
writer.mode=DSV
writer.rawchars=false
writer.includebefores=false
writer.includecolnames=false
writer.omitvalues=false
writer.diffsonly=false
writer.omitplaceholders=false
writer.files.onepertable=true
writer.files.data.ext=_data.dsv
writer.files.data.tmpext=_data.dsv.temp
writer.dsv.nullindicator.chars=
writer.dsv.fielddelim.chars=|
writer.dsv.fielddelim.escaped.chars=
writer.metacols=opcode,timestamp
writer.metacols.opcode.insert.chars=I
writer.metacols.opcode.update.chars=U
writer.metacols.opcode.delete.chars=D
```

7.6 Comma Delimited Format

```
writer.mode=DSV
writer.rawchars=false
writer.includebefores=false
writer.includecolnames=false
writer.omitvalues=false
writer.diffsonly=false
writer.omitplaceholders=false
writer.files.onepertable=true
writer.files.data.ext=_data.dsv
writer.files.data.tmpext=_data.dsv.temp
writer.dsv.nullindicator.chars=NULL
writer.dsv.fielddelim.chars=,
writer.dsv.linedelim.chars=\n
writer.dsv.quotes.chars="
writer.dsv.quotes.escaped.chars=""
writer.metacols=position,txind,opcode,timestamp,schema,table
writer.statistics.period=onrollover
writer.statistics.overall=true
```

8

Flat File Properties

This chapter describes properties that you can configure in the Oracle GoldenGate Flat File Adapter property file.

The chapter includes the following sections:

8.1 User Exit Properties

User exit properties include properties to control logging and general properties that control naming and handling of transactions.

8.1.1 Logging Properties

Logging is controlled by the following properties.

8.1.1.1 goldengate.log.logname

Specifies the prefix to the log file name. This must be a valid ASCII string. The log file name has the current date appended to it, in `yyyymmdd` format, together with the `.log` extension.

The following example will create a log file of name `writer_20140803.log` on August 3, 2014.

```
goldengate.log.logname=writer
```

8.1.1.2 goldengate.log.level

Specifies the overall log level for all modules. The syntax is:

```
goldengate.log.level=ERROR | WARN | INFO| DEBUG
```

The log levels are defined as follows:

- `ERROR` – Only write messages if errors occur
- `WARN` – Write error and warning messages
- `INFO` – Write error, warning and informational messages
- `DEBUG` – Write all messages, including debug ones.

The default logging level is `INFO`. The messages in this case will be produced on startup, shutdown, and periodically during operation. For example, the following sets the global logging level to `INFO`:

```
goldengate.log.level=INFO
```

Note:

If the level is switched to DEBUG, large volumes of messages may occur, which could impact performance.

8.1.1.3 goldengate.log.tostdout

Controls whether or not log information is written to standard out. This setting is useful if the Extract process is running with a VAM started from the command line or on an operating system where stdout is piped into the report file. However, Oracle GoldenGate processes generally run as background processes. The syntax is:

```
goldengate.log.tostdout={true | false}
```

The default is false.

8.1.1.4 goldengate.log.tofile

Controls whether or not log information is written to the specified log file. The syntax is:

```
goldengate.log.tofile={true | false}
```

The default is false. Log output is written to the specified log file when set to true.

8.1.2 General Properties

General properties control file writer names, check pointing, handling of transactions, representation of timestamps, and the format used for column and object names.

8.1.2.1 goldengate.flatfilewriter.writers

Specifies the name of the writer that will run within the user exit. Enter multiple string values to enable multiple named writers to run within the same user exit. For example:

```
goldengate.flatfilewriter.writers=dsvwriter,diffswriter,binwriter
```

Ensure there are no spaces before or after the equal sign or the commas. All other properties in the file should be prefixed by one of the writer names.

8.1.2.2 goldengate.userexit.buffertxs

Controls whether entire transactions are read before being output.When set to true, an entire transaction is read from the trail before being output. For example:

```
goldengate.userexit.buffertxs=true
```

The default is false. Setting this to true is useful only if the numops metadata column is used. Currently the only way to calculate the numops value is to buffer transactions and output one transaction at a time.

8.1.2.3 goldengate.userexit.chkptprefix

Specifies a string value as the prefix to be added to the checkpoint file name. When running multiple data pumps, the checkpoint prefix should be set to the name of the process. For example:

```
goldengate.userexit.chkptprefix=pump1_
```

8.1.2.4 goldengate.userexit.chkpt.ontxend

Controls whether the need to roll files over is checked after every transaction or only when the Extract process checkpoints. If set to `true`, the adapter checks if a file is due to be rolled over after it has processed a transaction. If due, the rollover is performed and the checkpoint file updated. This is useful if tight control over the contents of output files is required. For example, if all data up to midnight should be written to files before rolling over at midnight, it is important that the check occurs on every transaction. For example:

```
goldengate.userexit.chkpt.ontxend=true
```

The default is `false`. If set to `false`, the adapter will only check for rollover when Extract checkpoints (every 10 seconds by default).

8.1.2.5 goldengate.userexit.datetime.removecolon

Controls whether or not a colon is written between the date and time. When set to `false`, the date and time column values are written to the output files in the default format of the Oracle GoldenGate trail, `YYYY-MM-DD:HH:MI:SS.FFFF`. When set to `true`, the format is changed to `YYYY-MM-DD HH:MI:SS.FFF` with no colon between date and time. The default is `false`.

```
goldengate.userexit.datetime.removecolon=true
```

8.1.2.6 goldengate.userexit.timestamp

Controls whether the record timestamp is output as local time or Coordinated Universal Time (UTC). When this is not set to `utc` the record timestamp is output as local time using the local time zone. The default is local time.

```
goldengate.userexit.timestamp=utc
```

8.1.2.7 goldengate.userexit.datetime.maxlen

Controls the maximum output length of a date time column. Setting this to an integer value truncates the column value to that length. Since the date and time format is `YYYY-MM-DD:HH:MI:SS.F(9)` the maximum length of a date and time column is 29 characters.

For example:

```
goldengate.userexit.datetime.maxlen=19
```

Setting `goldengate.userexit.maxlen=19` truncates to date and time with no fractional seconds. Setting `goldengate.userexit.maxlen=10` truncates to date only. The default is to output the full date and time column value.

8.1.2.8 goldengate.userexit.utf8mode

Controls whether column data and table, file, and column names are returned in the UTF8 character set. When this is set to `false`, all data will be in the character set of the operating system. The default is `true`.

The syntax is:

```
goldengate.userexit.utf8mode=true|false
```

8.2 File Writer Properties

File writer properties control the format of the output file and how the files are written.

8.2.1 Output Format Properties

The following properties set the delimiter types of the values and the grouping of columns.

8.2.1.1 *writer*.mode

Controls whether the output format is DSV or LDV.

- **DSV** – Delimiter Separated Values, for example:

```
POSITION|OPCODE|TIMESTAMP|COLVALA|COLVALB|. . .
```

> **Note:**
>
> DSV is not limited to comma separated values (as is CSV).

- **LDV** – Length Delimited Values, for example:

```
0109TIMESTAMPI302MY05TABLEP042000P03ETC
```

> **Note:**
>
> Lengths can be ASCII or binary, some metadata columns can be fixed length (see Metadata Columns) and this format will support unicode multi-byte data.

For example:

```
writer.mode=dsv
writer2.mode=ldv
```

> **Note:**
>
> For backward compatibility, `csv` is accepted instead of `dsv`, binary instead of `ldv`. There is no difference in the output formats when using the alternate options.

8.2.1.2 *writer*.groupcols

Controls whether or not the column names, before values and after values are grouped together.

The syntax is:

```
writer.groupcols=true|false
```

The default is false. This results in a set of name, before value and after value listed together, as shown in this example for `COL1` and `COL2`:

```
"COL1", COL1_B4, COL1, "COL2", COL2_B4, COL2
```

With the property set to `true`, the columns are grouped into sets of all names, all before values, and all after values:

```
"COL1", "COL2", COL1_B4, COL2_B4, COL1, COL2
```

8.2.2 Output File Properties

The following properties control how files are written, where to, and what their extensions will be. This is independent of the writer mode and data contents.

8.2.2.1 *writer*.files.onepertable

Controls whether data is split over multiple rolling files (one per table in the input data) or all data is written to one rolling file. The default is true.

The syntax is:

```
writer.files.onepertable=true|false
```

In the following example the `writer` file writer will create one file per table, and `writer2` will write all data to one file.

```
writer.files.onepertable=true
writer2.files.onepertable=false
```

8.2.2.2 *writer*.files.oneperopcode

Controls whether or not data is split based on the insert, update, delete, or primary key operation codes.

For example, the following setting will create separate output files for inserts, updates, deletes, and primary key updates:

```
writer.files.oneperopcode=true
```

The default is false; output all records to the same files independent of the type of operation.

In addition to this property, you must also modify the `files.formatstring` property to accept the `%O` placeholder. This indicates the position to write the operation code when the file name is created if the `files.oneperopcode` property is set. The default filename should also include the operation code if that property is set.

8.2.2.3 *writer*.files.prefix

Specifies a value to be used as the prefix for data files and control files. This property only applies if the writer is not in one per table mode (`files.onepertable=true`). For data files, the prefix is ignored if the `property files.formatstring` is being used.

By default, the prefix is set to the string `output`. A file named `data1` will become `outputdata1`by default. The file name will be `test_data1` using the following example.

```
writer.files.prefix=test_
```

8.2.2.4 *writer.*files.data.rootdir, *writer.*files.data.ext, *writer.*files.data.tmpext

Specifies the location and extension of all data files. Before rolling over the files will have the `tmpext` extension, after rolling over they will have the `ext` extension. The extension does not have to be just an .`ext` format, additional characters can be appended to the file name before the extension to differentiate the data output. You should ensure the named output directory exists, and that the user running the Oracle GoldenGate processes has the correct permissions to write to that directory. For example:

```
# specify the root directory for outputting data files
writer.files.data.rootdir=./out

# determine the extension for data files when rolled over
writer.files.data.ext=_data.dsv

# determine the extension for data files before rolling over
writer.files.data.tmpext=_data.dsv.temp
```

8.2.2.5 *writer.*files.control.use, *writer.*files.control.rootdir, *writer.*files.control.ext

`writer.``files.control.use` is a boolean true or false value that defaults to true. The others are ASCII values. These properties determine the user, location and extension of control files. Control files will share the same name prefix as the data files they are related to, but will have the defined extension. By default `files.control.ext` is `.control`. For example:

```
# specify whether or not to output a control file
writer.files.control.use=true

# specify the extension to use for control files
writer.files.control.ext=_data.control

# directory in which to place control files, defaults to data directory
writer.files.control.rootdir=./out
```

8.2.2.6 *writer.*files.control.delim.chars/code, *writer.*files.control.eof.chars/code

Specifies the value in characters or hexadecimal code to be used as the data delimiter or the end-of-line indicator. The default for the delimiter is a comma (,) The default new line trigger is the `newline` character that is valid for the platform.

For example, to override the comma as the data delimiter:

```
writer.files.control.delim.chars=#
```

For example, to set the new line indicator:

```
writer.files.control.eol.chars=\n
```

8.2.2.7 *writer.*files.formatstring

Specifies the filename format string to be used in creating the filenames for data files. The format string overrides the `files.prefix` property. This filename format string is similar in syntax to standard C formatting except the following placeholders can be added to the filename:

- **%s** = schema
- **%t** = table

- **%n** = seqno
- **%d** = timestamp
- **%o** = opcode

The format of the `seqno` can be specified. For example %05n means 5 digits will be displayed and padded with 0's. The `seqno` starts at zero and is incremented by one each time a file rolls over. It is stored as a `long int` and therefore the maximum value is platform dependent. For example on a 64 bit machine the largest value is 2^64-1.

These placeholders can be intermingled with user specified text in any order desired. For example:

```
writer.files.formatstring=myext_%d_%010n_%s_%
```

8.2.2.8 *writer*.files.data.bom.code

Specifies a hexadecimal value as the byte order marker (BOM) to be written to the beginning of the file. The user is responsible for ensuring the BOM matches the data in the files. If no hexadecimal value is specified the marker is not written.

The following example results in the UTF8 BOM `efbbf` written as the first bytes of all output files.

```
writer.files.data.bom.code=efbbbf
```

8.2.2.9 *writer*.files.includeprocessname

Controls whether or not the name of the Extract process is included as part of the file name. The default is false.

The syntax is:

```
writer.files.includeprocessname=true|false
```

8.2.2.10 *writer*.files.useownerfiles

Controls whether or not hidden files are created to identify the Extract process that owns the file. This can be used to avoid overwriting files from different Oracle GoldenGate installations. The default is false.

The syntax is:

```
writer.files.useownerfiles=true|false
```

8.2.3 File Rollover Properties

The following properties determine the policies for rolling over files.

8.2.3.1 *writer*.files.data.rollover.time

Specifies the maximum number of seconds of elapsed time that must pass from the first record written to the file before the file is rolled over. For example:

```
# number of seconds before rolling over
writer.files.data.rollover.time=10
```

8.2.3.2 *writer*.files.data.rollover.size

Specifies the minimum number of kilobytes that must be written to the file before the file is rolled over.

This example sets the minimum to 10,000 KB:

```
# max file size in KB before rolling over
writer.files.data.rollover.size=10000
```

8.2.3.3 *writer*.files.data.norecords.timeout

Specifies the maximum number of elapsed seconds since data was written to a file to wait before rolling over the file. The default is 120 seconds.

This example sets the timeout interval to 10 seconds:

```
# roll over in case no records for a period of time
writer.files.data.norecords.timeout=10
```

8.2.3.4 *writer*.files.rolloveronshutdown

Controls the policy for roll over when the Extract process stops. If this value is false, all empty temporary files will be deleted, but any that have data will be left as temporary files. If this property is true, all non-empty temporary files will be rolled over to their rolled file name, a checkpoint written and empty temporary files deleted. For example:

```
# roll over non-empty and delete all empty files when Extract stops
writer.files.rolloveronshutdown=true
```

Note:

You can use time and/or size. If you use both, the first reached will cause a roll over. The time out interval ensures files are rolled over if they contain data, even if there are no records to be processed. If neither time or size are specified, files will roll over after a default maximum size of 1MB.

8.2.3.5 *writer*.files.data.rollover.timetype

Controls whether to use the Julian commit timestamp rather than the system time to trigger file roll over. The syntax is:

```
writer.files.data.rollover.timetype=commit|system
```

The following example will use the commit timestamp of the source trail records to determine roll over:

```
writer.files.data.rollover.timetype=commit
```

The default is to use the system time to determine when to roll over files.

8.2.3.6 *writer*.files.data.rollover.multiple

Controls whether or not all files will be rolled over simultaneously independent of when they first received records. Normally files are rolled over individually based on the time or size properties. The time is based on the roll over period, so it depends on the time records were first written to a particular file. In some cases, especially when outputting data with one file per table, you may want to roll over all currently open files at the same time, independent of when data was first written to that file.

The following example instructs the adapter to roll over all files simultaneously.

```
writer.files.data.rollover.multiple=true
```

The default value is `false`.

8.2.3.7 *writer*.files.data.rollover.attime

Specifies a time for the adapter to roll over files. Enter the specified time in 24 hour format (`HH:MM`). Only one value entry is supported. The wildcard (*) is supported for hours. The syntax is:

```
writer.files.data.rollover.attime=time_specifier
```

The following example will roll over to a new file every hour on the hour:

```
writer.files.data.rollover.attime=*:00
```

The following example will roll over every hour at fifteen minutes after the hour:

```
writer.files.data.rollover.attime=*:15
```

Note that the `writer.rollover.timetype` property determines whether the time to use is system or commit time.

8.2.3.8 *writer*.writebuffer.size

Specifies the write buffer chunk size. Use to reduce the number of system write calls. For example:

```
writer.writebuffer.size=36863
```

8.2.4 Data Content Properties

The following properties determine the data that is written to the data files. These properties are independent of the format of the output data.

8.2.4.1 *writer*.rawchars

Controls whether character data retains its original binary form or is output as ASCII. The default is false. This property should be set if the input data contains Unicode multibyte data that should not be converted to ASCII. For example:

```
# whether to output characters as ASCII or binary (for Unicode data)
writer.rawchars=false
writer2.rawchars=true
```

8.2.4.2 *writer*.includebefores

Controls whether or not both the before and after image of data is included in the output for update operations. The default is false. This is only relevant if the before images are available in the original data, and `getupdatebefores` is present in all Oracle GoldenGate parameter files in the processing chain. For example:

```
# whether to output update before images
writer.includebefores=true
```

This
produces `. . ."VAL_BEFORE_1","VAL_1","VAL_BEFORE_2","VAL_2". . .`

8.2.4.3 *writer*.afterfirst

Controls whether or not the after image is written before the before image when `includebefores` is set to `true.`

For example:

```
writer.afterfirst=true
```

This true setting results in the after image listed before the before image.

```
"VAL_1", "VAL_BEFORE_1", "VAL_2", "VAL_BEFORE_2"
```

The default is `false. In this case` the after image is written after the before image.

8.2.4.4 *writer*.includecolnames

Controls whether or not column names are output before the column values. The default is `false`. For example:

```
# whether to output column names
writer.includecolnames=true
```

This produces …`"COL_1","VAL_1","COL_2","VAL_2"`…

8.2.4.5 *writer*.omitvalues

Controls whether or not column values are omitted in the output files. The default is `false`. For example:

```
# whether to output column values
writer.omitvalues=false
```

This produces …`"COL_1","COL_2"`…, if `includecolnames` is also set to `true`.

8.2.4.6 *writer*.diffsonly

Controls whether all columns are output, or only those where the before image is different from the after image. The default is `false`. This only applies to updates and requires `GETUPDATEBEFORES` in all Oracle GoldenGate parameter files in the processing chain. This property is independent of the `includebefores` property. For example:

```
# whether to output only columns with differences between before and
# after images (deletes and inserts have all available columns)
writer.diffsonly=true
```

This produces `. . ."VAL_1",,,"VAL_4",,,"VAL_7". . .`

8.2.4.7 *writer*.omitplaceholders

Controls whether delimiters/lengths are included in the output for missing columns. The default is `false`. This applies to updates and deletes where the `COMPRESSUPDATES` or `COMPRESSDELETES` flag was present in a Oracle GoldenGate parameter file in the processing chain. In this case, values may be missing. Also, if *`writer`*`.diffsonly` is `true`, values that are not different are said to be missing. For example:

```
# whether to skip record delimiters if columns are missing
writer.omitplaceholders=true
```

This changes `. . ."VAL_1",,,"VAL_4",,,"VAL_7". . .`

`to . . ."VAL_1","VAL_4","VAL_7". . .`

8.2.4.8 Metadata Columns

Metadata columns are optional Extract columns that contain data about a record, not actual record data. These columns are written at the beginning of the output record, before any column values.

8.2.4.9 Valid Metadata Columns

Valid metadata columns are:

- **position** - A unique position indicator of records in a trail.
- **opcode** - `I`, `U`, `D` or `K` for Insert, Update, Delete, or Primary Key update records.
- **txind** - The general record position in a transaction (0 - begin, 1 - middle, 2 - end, 3 - only).
- **txoppos** - Position of record in a transaction, starting from 0.
- **schema** - The schema (owner) name of the changed record.
- **table** - The table name of the changed record.
- **schemaandtable** - Both the schema and table name concatenated as schema.table
- **timestamp** - The commit timestamp of the record.
- **@<token name>** - A token value defined in the Extract parameter file.
- **$getenv** - A `GETENV` value as documented in the *Oracle GoldenGate Reference Guide*; for example `$GGHEADER.OPCODE.`
- **%COLNAME** - The value of a data column.
- **numops** -The number of operations in the current transaction. This value will always be 1 if `goldengate.userexit.buffertxs` is not `true.`
- **numcols** - The number of columns to be output. This value is equal to the number of columns in the original record, minus the number of columns output as metadata columns up until the point this metadata column is used.
- **"<value>**" - Any literal value.

8.2.4.10 Using Metadata Columns

Some things to consider when using metadata columns:

- The ASCII values for `opcode` and `txind` can be overridden.
- For `LDV`, metadata columns can be variable or fixed length.
- The `position` can be written in hexadecimal or decimal.
- Any metadata column can be the internal value or it can be read from a column of the original data.
- A literal value is indicated by enclosing it in quotes. When a literal value is specified, that value will be output as a character string in the specified metadata column position using the appropriate quote policy.

- A column value is indicated by `%COLNAME`. When a column value is specified, that column value is output in the metadata section of the output record, rather than in the column values section. This may be used to ensure that the column is always output in the same position in the record, independent of the table being output.

The following properties apply to metadata columns.

8.2.4.11 *writer*.metacols

Specifies the metadata columns to output in the order of output. Enter multiple names as ASCII values separated by commas. For example:

```
# which metacols to output and in which order
writer.metacols=timestamp,opcode,txind,position,schema,table
```

8.2.4.12 *writer*.metacols.*metacol_name*.fixedlen

Specifies an integer value to determine the length of data to write for the metadata column specified by *`metacol_name`*. If the actual data is longer than the fixed length it will be truncated, if it is shorter the output will be padded. For example:

```
# timestamp is fixed length
writer.metacols.timestamp.fixedlen=23
```

This truncates `2011-08-03 10:30:51.123456` to `2011-08-03 10:30:51.123.`

8.2.4.13 *writer*.metacols.*metacol_name*.column

Specifies an ASCII value to use as the column name of data values instead of using the *`metacol_name`* value for a metadata column. If set, this column name must exist in all tables processed by the user exit. There is currently no way to override this column name on a per table basis. For example, to override the internal timestamp from a column:

```
# timestamp is read from a column
writer.metacols.timestamp.column=MY_TIMESTAMP_COL
```

8.2.4.14 *writer*.metacols.*token_name*.novalue.chars | *writer*.metacols.*token_name*.novalue.code

Specifies values to represent characters or hexadecimal code to be used when the value of *`token_name`* is not available. Use ASCII values for `chars and` hexadecimal values for `code`. The default value is `NO VALUE`. For example:

```
writer.metacols.TKN-SCN.novalue.chars=0
```

8.2.4.15 *writer*.metacols.*metacol_name*.fixedjustify

Controls whether the justification for the *`metacol_name`* column value is to the left or right. By default all metadata columns will be justified to the left. For example, to justify a token to the right:

```
writer.metacols.TKN-SCN.fixedjustify=right
```

8.2.4.16 *writer*.metacols.*metacol_name*.fixedpadchar.chars | *writer*.metacols.*metacol_name*.fixedpadchar.code

Specifies either a character or code value to be used for padding a metadata column. Use ASCII values for `chars and` hexadecimal values for `code`. The default character used for padding is a space (" "). For example:

```
writer.metacols.TKN-SCN.fixedpadchar.chars=0
```

8.2.4.17 *writer*.metacols.opcode.insert.chars | *writer*.metacols.opcode.insert.code

Specifies an override value for the default character `I` that identifies insert operations. Use ASCII values for `chars` and hexadecimal values for `code`.

The following example instructs the adapter to use `INS` for inserts:

```
writer.metacols.opcode.insert.chars=INS
```

8.2.4.18 *writer*.metacols.opcode.update.chars | *writer*.metacols.opcode.update.code

Specifies an override value for the default character `U` that identifies update operations. Use ASCII values for `chars` and hexadecimal values for `code`.

The following example instructs the adapter to use `UPD` for updates:

```
writer.metacols.opcode.update.chars=UPD
```

8.2.4.19 *writer*.metacols.opcode.delete.chars | *writer*.metacols.opcode.delete.code

Specifies an override value for the default character `D` that identifies delete operations. Use ASCII values for `chars` and hexadecimal values for `code`.

The following example instructs the adapter to use `DEL` for deletes:

```
writer.metacols.opcode.delete.chars=DEL
```

8.2.4.20 *writer*.metacols.opcode.updatepk.chars | *writer*.metacols.opcode.updatepk.code

Specifies an override value for the default character `K` that identifies primary key update operations. Use ASCII values for `chars` and hexadecimal values for `code`.

The following example instructs the adapter to use `PKU` for primary key updates:

```
writer.metacols.opcode.updatepk.chars=PKU
```

8.2.4.21 *writer*.metacols.txind.begin.chars | *writer*.metacols.txind.begin.code

Specifies the override values to use to identify the beginning, middle, end of transactions, or if an operation that is the whole transaction. Use ASCII values for `chars` and hexadecimal values for `code`. The default value is 0 for Begin.

The following example overrides the 0 with the letter B.

```
# tx indicator values is overridden
writer.metacols.txind.begin.chars=B
```

8.2.4.22 *writer*.metacols.txind.middle.chars | *writer*.metacols.txind.middle.code

Specifies the override value to use to identify the middle transactions. Use ASCII values for `chars` and hexadecimal values for `code`. The default value is 1 for Middle.

The following example overrides the1 with the letter M.

```
# tx indicator value is overridden
writer.metacols.txind.middle.chars=M
```

8.2.4.23 *writer*.metacols.txind.end.chars | *writer*.metacols.txind.end.code

Specifies the override value to use to identify the end transactions. Use ASCII values for `chars` and hexadecimal values for `code`. The default value is 2 for End.

The following example overrides the 2 with the letter E.

```
# tx indicator value is overridden
writer.metacols.txind.end.chars=E
```

8.2.4.24 *writer*.metacols.txind.whole.chars | *writer*.metacols.txind.whole.code

Specifies the override value to use to identify. if an operation that is the whole transaction. Use ASCII values for `chars` and hexadecimal values for `code`. The default value is 3 for Whole.

The following example overrides the 3 with the letter W.

```
# tx indicator value is overridden
writer.metacols.txind.whole.chars=W
```

8.2.4.25 *writer*.metacols.position.format

Controls whether the output of the of the `position` metadata column is in decimal or hexadecimal format. If hexadecimal, this will typically be a 16 character value; if decimal, the length will vary. Currently this contains the sequence number and RBA of the Oracle GoldenGate trail that the Extract process is reading from. For example:

```
# position is in decimal format (seqno0000000rba)
writer.metacols.position.format=dec
```

This produces `120000012345` for `seqno` 12, `rba` 12345

```
writer2.metacols.position.format=hex
```

This produces `0000000c00003039` for seqno 12, rba 12345.

8.2.4.26 *writer*.metacols.*colname*.omit

Controls whether the `COLNAME` column can be used as metadata but not output.

The following example specifies that `numcols` can be used as metadata, but not output.

```
writer.metacols.numcols.omit=true
```

8.2.4.27 *writer*.begintx.metacols, *writer*.endtx.metacols

Specifies the metadata columns to use to mark the beginning and end of a transaction. These marker records are written (with end of line delimiters) to the output files before and after the operation records that make up the transaction.

The syntax is:

```
writer.begintx.metacols=metacols_list
```

The following example specifies marking the beginning of a transaction with the letter `B` and the number of operations in the transaction.

```
writer.begintx.metacols="B",numops
```

In the following example, the end of the transaction marker will be the letter `E`.

```
writer.endtx.metacols="E"
```

Any of the existing metadata columns can be used in the transaction begin and end markers. If you specify a column value or specific property of a record (such as table

name) for `begintx.metacols`, the value for the first record in the transaction is used. For `endtx.metacols`, the value for the last record is used.

For example, if the transaction has the following records:

```
rec=0,table=tabA,operation=insert,col1=val1,col2=val2
rec=1,table=tabA,operation=update,col1=val3,col2=val4
rec=2,table=tabA,operation=delete,col1=val5,col2=val6
rec=3,table=tabB,operation=update,col1=val7,col2=val8
```

And the properties are set as follows:

```
writer.begintx.metacols="B",table,%col2
writer.endtx.metacols="E",table,%col2
```

Then the begin transaction marker will be `"B","tabA","val2"` and the end marker will be `"E","tabB","val8".`

If `numops` is used to output the number of operations in a transaction for either the begin or end markers, the user must also set:

```
goldengate.userexit.buffertxs=true
```

Note:

When this property is set, the adapter buffers transactions in memory, so care should be taken to limit the number of operations in the transactions being handled by the system.

8.2.5 DSV Specific Properties

DSV files have the following record format:

```
{[METACOL][FD]}n{[COL][FD]}m[LD]
```

Where:

- `METACOL` is any defined metadata column
- `COL` is any data column
- `FD` is the field delimiter
- `LD` is the line delimiter

Column values may be quoted, `e.g. "2013-01-10 10:20:31","U","MY.TABLE", 2000,"DAVE"`

8.2.5.1 *writer*.dsv.nullindicator.chars | *writer*.dsv.nullindicator.code

Specifies the characters to use for `NULL` values in delimiter separated files. These values override the default `NULL` value of an empty string. Use ASCII values for `chars and` hexadecimal values for `code`. For example:

```
writer.dsv.nullindicator.chars=NULL
writer.dsv.nullindicator.code=0a0a0a0a
```

8.2.5.2 *writer*.dsv.fielddelim.chars | *writer*.dsv.fielddelim.code

Specifies an override value for the field delimiter. The default is a comma (,). Use ASCII values for `chars and` hexadecimal values for `code`. For example:

```
# define the characters to use for field delimiters in DSV files
writer.dsv.fielddelim.chars=|
```

8.2.5.3 *writer*.dsv.linedelim.chars | *writer*.dsv.linedelim.code

Specifies an override value for the line delimiter. The default is a new line character appropriate to the operating system. Use ASCII values for `chars` and hexadecimal values for `code`. For example:

```
# define the characters to use for line delimiters in DSV files
writer.dsv.linedelim.chars=\n
```

8.2.5.4 *writer*.dsv.quote.chars | *writer*.dsv.quote.code

Specifies an override value for the quote character. The default is a double quote ("). Use ASCII values for `chars` and hexadecimal values for `code`.For example:

```
# define the characters to use for quotes in DSV files
writer.dsv.quotes.chars='
```

8.2.5.5 *writer*.dsv.quotes.policy

Controls the policy for applying quotes.

The syntax is:

```
writer.dsv.quotes.policy={default|none|always|datatypes}
```

Where:

- **default** – Only dates and chars are quoted
- **none** – No metadata column or column values are quoted
- **always** – All metadata columns and column values are quoted
- **datatypes** – Only specific data types are quoted

If this property is set it will override the `dsv.quotealways` property. Use the `dsv.quotes.datatypes` property to specify which data types should be quoted.

8.2.5.6 *writer*.dsv.quotes.datatypes

Controls whether integer, character, float, or datetime data types are to be quoted when `dsv.quotes.policy` is set to `datatype`.

The syntax is:

```
writer.dsv.quotes.datatypes=[char][,integer][,float][,date]
```

For example the following instructs the adapter to quote characters and date time values only.

```
writer.dsv.quotes.datatypes=char,date
```

If no data types are specified, the data types option defaults to all data types, which is equivalent to `always`.

8.2.5.7 *writer*.dsv.nullindicator.escaped.chars | *writer*.dsv.nullindicator.escaped.code

Specifies the escaped value for a null indicator. If set, all values will be checked for the null indicator value and replaced with the escaped value when output. Use ASCII values for `chars` and hexadecimal values for `code`. For example:

```
# (optionally) you can define the characters (or code) to use
# to escape these values if found in data values
writer.dsv.nullindicator.escaped.chars=NULL
```

This changes the null indicator to `NULL`.

8.2.5.8 *writer*.dsv.fielddelim.escaped.chars | *writer*.dsv.fielddelim.escaped.code

Specifies the escaped value for a field delimiter. If set, all values will be checked for the field delimiter value and replaced with the escaped value when output. Use ASCII values for `chars` and hexadecimal values for `code`. For example:

```
writer.dsv.fielddelim.escaped.chars=|
```

This changes the field delimiter to `|`.

8.2.5.9 *writer*.dsv.linedelim.escaped.chars | *writer*.dsv.linedelim.escaped.code

Specifies the escaped value for a line delimiter. If set, all values will be checked for the line delimiter value and replaced with the escaped value when output. Use ASCII values for `chars` and hexadecimal values for `code`. For example:

```
writer.dsv.linedelim.escaped.chars=\n
writer.dsv.linedelim.escaped.code=D
```

Both change the line delimiter to `\n`.

8.2.5.10 *writer*.dsv.quotes.escaped.chars | *writer*.dsv.quotes.escaped.code

Specifies the escaped value for a field delimiter. If set, all values will be checked for the field delimiter value and replaced with the escaped value when output. Use ASCII values for `chars` and hexadecimal values for `code`. For example:

```
writer.dsv.quotes.escaped.chars=""
```

This changes the "some text" to ""some text"".

8.2.5.11 *writer*.dsv.onecolperline

Controls whether or not each column value is forced onto a new line. Each line will also contain the metadata columns defined for this writer. The default is false. For example:

```
# Force each column onto a new line with its own meta cols
writer.dsv.onecolperline=true
```

This changes:`{metacols},val_1,val_2` to

```
{metacols},val1
{metacols},val2
```

8.2.5.12 *writer*.dsv.quotealways

Controls whether or not each column is surrounded by quotes, even if it is a numeric value. The default is false.

Note:

This property has been superseded by `dsv.quotes.policy` and is supported only for backward compatibility. The value set for `dsv.quotealways` is ignored if `dsv.quotes.policy` is set.

For example:

```
writer.dsv.quotealways=true
```

Changes: `. . .,1234,"Hello",10` to `. . .,"1234","Hello","10"`

8.2.6 LDV Specific Properties

LDV files have the following record format:

```
[RECLEN][METACOLS]{[FLAG][LEN][VALUE]}n
```

Where:

- `RECLEN` is the full record length in bytes
- `METACOLS` are all selected metadata columns
- `FLAG` can be (`M`)issing, (`P`)resent, or (`N`)ull
- `LEN` is the column values length (0 for missing and null)
- `VALUE` is the column value

For example:

```
01072007-01-10 10:20:31U302MY05TABLEP042000M00N00P04DAVE
```

8.2.6.1 *writer*.ldv.vals.missing.chars | *writer*.ldv.vals.missing.code

Specifies override values for missing indicators. Use ASCII values for `chars and` hexadecimal values for `code`. For example:

```
writer.ldv.vals.missing.chars=MI
```

8.2.6.2 *writer*.ldv.vals.present.chars | *writer*.ldv.vals.present.code

Specifies override values for present indicators. Use ASCII values for `chars and` hexadecimal values for `code`. For example:

```
writer.ldv.vals.present.chars=PR
```

8.2.6.3 *writer*.ldv.vals.null.chars | *writer*.ldv.vals.null.code

Specifies override values for null indicators. Use ASCII values for `chars and` hexadecimal values for `code`. For example:

```
writer.ldv.vals.null.chars=NL
```

8.2.6.4 *writer*.ldv.lengths.record.mode,*writer*. ldv.lengths.field.mode

Controls the output mode of record and field lengths. The value can be either `binary` or `ASCII`. The default is `binary`.

If `binary`, the number written to the file will be encoded in binary bytes. If `ASCII`, characters representing the decimal value of the length will be used. For example:

```
writer.ldv.lengths.record.mode=binary
writer.ldv.lengths.field.mode=binary
```

8.2.6.5 *writer*.ldv.lengths.record.length, *writer*.ldv.lengths.field.length

Specifies the record and field lengths as integer values. If the mode is `ASCII`, this represents the fixed number of decimal digits to use. If `binary,` it represents the number of bytes.

In ASCII mode the lengths can be any value, but the exit will stop if a length exceeds the maximum. In binary mode, the lengths can be 2,4, or 8 bytes, but record length must be greater than field length. For example:

```
# Lengths can be binary (2,4, or 8 bytes) or ASCII (any length)
writer.ldv.lengths.record.length=4
writer.ldv.lengths.field.length=2
```

8.2.7 Statistics and Reporting

There are two ways that statistics regarding the data written to data files can be obtained:

- As a report written to the Oracle GoldenGate report file
- As a separate summary file associated with a data file on rollover

These two mechanisms can be used together or separately.

The data that can be obtained includes, 1) the total records processed, broken down to inserts, updates, deletes; 2) records processed per table, also broken down; 3) total rate and rate per table; 4) delta for these since last report. Reporting can be time based, or synced to file rollover

This data can be written to the report file or as a summary file linked to a data file on rollover. The reporting format is fixed. The summary file contains the data in a delimited format, but related to the contents of a particular data file. This can be used by a data integration product to cross-check processing. It will have the same name as the data file, but a different extension.

8.2.7.1 *writer*.statistics.toreportfile

Controls whether or not statistics are output to the Oracle GoldenGate report file. For example:

```
writer.statistics.toreportfile=true
```

8.2.7.2 *writer*.statistics.period

Specifies the time period for statistics. The value can be either `timebased` or `onrollover`.

For example:

```
writer.statistics.period=onrollover
writer.statistics.period=timebased
```

If `timebased`, the time period should be set in `statistics.time`.

Note:

These values are valid only for outputting statistics to the report file. Statistics will be output to the summary file only on rollover.

8.2.7.3 *writer*.statistics.time

Specifies a time interval in seconds after which statistics will be reported.

For example:

```
writer.statistics.time=5
```

8.2.7.4 *writer*.statistics.tosummaryfile

Controls whether or not a summary file containing statistics for each data file will be created on rollover.

The following example creates the summary file.

```
writer.statistics.tosummaryfile=true
```

8.2.7.5 *writer*.statistics.summary.fileformat

Controls the content of the summary files and the order in which the content is written. Multiple comma separated ASCII values can be specified.

Valid values are:

- **schema** – schema or owner of the table that the statistics relate to
- **table** – table that the statistics relate to
- **schemaandtable** – schema and table in one column separated by a period '.'
- **gtotal** – total number of records output for the specified table since the user exit was started
- **gtotaldetail** – total number of inserts, updates and deletes separated by the delimiter since the user exit was started
- **gctimestamp** – minimum and maximum commit timestamp for the specified table since user exit was started
- **ctimestamp** – minimum and maximum commit timestamps for the specified table in the related data file.
- **total** – total number of records output for the specified table in the related data file
- **totaldetail** – total number of inserts, updates and deletes output for the specified table in the related data file
- **rate** – average rate of output of data for the specified table in the related data file in records per second
- **ratedetail** – average rate of inserts, updates and deletes for the specified table in the related data file in records per second

For example:

```
writer.statistics.summary.fileformat=
   schema,table,total,totaldetail,gctimestamp,ctimestamp
```

8.2.7.6 *writer*.statistics.overall

Controls whether or not an additional statistics row is written to the summary files. This row contains the overall (across all tables) statistics defined by the user using the `statistics.summary.fileformat` property.

The following example will write this row.

```
writer.statistics.overall=true
```

8.2.7.7 *writer*.statistics.summary.delimiter.chars/code, *writer*.statistics.summary.eol.chars/code

Specifies override values for the field delimiter and end of line delimiter for the summary files. Use ASCII values for `chars` and hexadecimal values for `code`. The default is a comma ',' delimiter and new line character. For example:

```
writer.statistics.summary.delimiter.chars=|
writer.statistics.summary.eol.code=0a0c
```

8.2.7.8 *writer*.statistics.summary.extension

Specifies the override extension to use for the statistics summary file output per data file. The default is `stats`.

The following example changes the extension from .`stats` to .`statistics`.

```
writer.statistics.summary.extension=.statistics
```

Part III

Capturing JMS Messages

This part of the book explains using the Oracle GoldenGate Adapter to capture Java Message Service (JMS) messages to be written to an Oracle GoldenGate trail.

Part IV contains the following chapters:

9

Configuring Message Capture

This chapter explains how to configure the VAM Extract to capture JMS messages.

This chapter includes the following sections:

9.1 Configuring the VAM Extract

To run the Java message capture application you need the following:

- Oracle GoldenGate for Java adapter
- Extract process
- Extract parameter file configured for message capture
- Description of the incoming data format, such as a source definitions file.

9.1.1 Adding the Extract

To add the message capture VAM to the Oracle GoldenGate installation, add an Extract and the trail that it will create using GGSCI commands:

```
ADD EXTRACT jmsvam, VAM
ADD EXTTRAIL dirdat/id, EXTRACT jmsvam, MEGABYTES 100
```

The process name (`jmsvam`) can be replaced with any process name that is no more than 8 characters. The trail identifier (`id`) can be any two characters.

Note:

Commands to position the Extract, such as `BEGIN` or `EXTRBA`, are not supported for message capture. The Extract will always resume by reading messages from the end of the message queue.

9.1.2 Configuring the Extract Parameters

The Extract parameter file contains the parameters needed to define and invoke the VAM. Sample Extract parameters for communicating with the VAM are shown in the table.

Parameter	Description
`EXTRACT jmsvam`	The name of the Extract process.
`VAM ggjava_vam.dll, PARAMS dirprm/jmsvam.properties`	Specifies the name of the VAM library and the location of the properties file. The VAM properties should be in the `dirprm` directory of the Oracle GoldenGate installation location.
`TRANLOGOPTIONS VAMCOMPATIBILITY 1`	Specifies the original (1) implementation of the VAM is to be used.
`TRANLOGOPTIONS GETMETADATAFROMVAM`	Specifies that metadata will be sent by the VAM.
`EXTTRAIL dirdat/id`	Specifies the identifier of the target trail Extract creates.
`TABLE OGG.*`	A list of tables to process. Wildcards may be used in the table name.

9.1.3 Configuring Message Capture

Message capture is configured by the properties in the VAM properties file. This file is identified by the `PARAMS` option of the Extract `VAM` parameter and used to determine logging characteristics, parser mappings and JMS connection settings.

9.2 Connecting and Retrieving the Messages

To process JMS messages you must configure the connection to the JMS interface, retrieve and parse the messages in a transaction, write each messages to a trail, commit the transaction, and remove its messages from the queue.

9.2.1 Connecting to JMS

Connectivity to JMS is through a generic JMS interface. Properties can be set to configure the following characteristics of the connection:

- Java class path for the JMS client
- Name of the JMS queue or topic source destination
- Java Naming and Directory Interface (JNDI) connection properties
  - Connection properties for Initial Context
  - Connection factory name
  - Destination name
- Security information
  - JNDI authentication credentials
  - JMS user name and password

The Extract process that is configured to work with the VAM (such as the `jmsvam` in the example) will connect to the message system. when it starts up.

Note:

The Extract may be included in the Manger's `AUTORESTART` list so it will automatically be restarted if there are connection problems during processing.

Currently the Oracle GoldenGate for Java message capture adapter supports only JMS text messages.

9.2.2 Retrieving Messages

The connection processing performs the following steps when asked for the next message:

- Start a local JMS transaction if one is not already started.
- Read a message from the message queue.
- If the read fails because no message exists, return an end-of-file message.
- Otherwise return the contents of the message.

9.2.3 Completing the Transaction

Once all of the messages that make up a transaction have been successfully retrieved, parsed, and written to the Oracle GoldenGate trail, the local JMS transaction is committed and the messages removed from the queue or topic. If there is an error the local transaction is rolled back leaving the messages in the JMS queue.

10

Parsing the Message

This chapter explains the types of parsers included with the Oracle GoldenGate Java Adapter and how each parser translates JMS text messages.

This chapter includes the following sections:

10.1 Parsing Overview

The role of the parser is to translate JMS text message data and header properties into an appropriate set of transactions and operations to pass into the VAM interface. To do this, the parser always must find certain data:

- Transaction identifier
- Sequence identifier
- Timestamp
- Table name
- Operation type
- Column data specific to a particular table name and operation type

Other data will be used if the configuration requires it:

- Transaction indicator
- Transaction name
- Transaction owner

The parser can obtain this data from JMS header properties, system generated values, static values, or in some parser-specific way. This depends on the nature of the piece of information.

10.1.1 Parser Types

The Oracle GoldenGate message capture adapter supports three types of parsers:

- Fixed – Messages contain data presented as fixed width fields in contiguous text.
- Delimited – Messages contain data delimited by field and end of record characters.
- XML – Messages contain XML data accessed through XPath expressions.

10.1.2 Source and Target Data Definitions

There are several ways source data definitions can be defined using a combination of properties and external files. The Oracle GoldenGate Gendef utility generates a standard source definitions file based on these data definitions and parser properties. The options vary based on parser type:

- Fixed – COBOL copybook, source definitions or user defined
- Delimited – source definitions or user defined
- XML – source definitions or user defined

There are several properties that configure how the selected parser gets data and how the source definitions are converted to target definitions.

10.1.3 Required Data

The following information is required for the parsers to translate the messages:

10.1.3.1 Transaction Identifier

The transaction identifier (`txid`) groups operations into transactions as they are written to the Oracle GoldenGate trail file. The Oracle GoldenGate message capture adapter supports only contiguous, non-interleaved transactions. The transaction identifier can be any unique value that increases for each transaction. A system generated value can generally be used.

10.1.3.2 Sequence Identifier

The sequence identifier (`seqid`) identifies each operation internally. This can be used during recovery processing to identify operations that have already been written to the Oracle GoldenGate trail. The sequence identifier can be any unique value that increases for each operation. The length should be fixed.

The JMS Message ID can be used as a sequence identifier if the message identifier for that provider increases and is unique. However, there are cases (e.g. using clustering, failed transactions) where JMS does not guarantee message order or when the ID may be unique but not be increasing. The system generated Sequence ID can be used, but it can cause duplicate messages under some recovery situations. The recommended approach is to have the JMS client that adds messages to the queue set the Message ID, a header property, or some data element to an application-generated unique value that is increasing.

10.1.3.3 Timestamp

The timestamp (`timestamp`) is used as the commit timestamp of operations within the Oracle GoldenGate trail. It should be increasing but this is not required, and it does not have to be unique between transactions or operations. It can be any date format that can be parsed.

10.1.3.4 Table Name

The table name is used to identify the logical table to which the column data belongs. The adapter requires a two part table name in the form `SCHEMA_NAME.TABLE_NAME`. This can either be defined separately (`schema and table`) or as a combination of schema and table (`schemaandtable`).

A single field may contain both schema and table name, they may be in separate fields, or the schema may be included in the software code so only the table name is required. How the schema and table names can be specified depends on the parser. In any case the two part logical table name is used to write records in the Oracle GoldenGate trail and to generate the source definitions file that describes the trail.

10.1.3.5 Operation Type

The operation type (`optype`) is used to determine whether an operation is an insert, update or delete when written to the Oracle GoldenGate trail. The operation type value for any specific operation is matched against the values defined for each operation type.

The data written to the Oracle GoldenGate trail for each operation type depends on the Extract configuration:

- Inserts
  - The after values of all columns are written to the trail.
- Updates
  - Default – The after values of keys are written. The after values of columns that have changed are written if the before values are present and can be compared. If before values are not present then all columns are written.
  - `NOCOMPRESSUPDATES` – The after values of all columns are written to the trail.
  - `GETUPDATEBEFORES` – The before and after values of columns that have changed are written to the trail if the before values are present and can be compared. If before values are not present only after values are written.
  - If both `NOCOMPRESSUPDATES` and `GETUPDATEBEFORES` are included, the before and after values of all columns are written to the trail if before values are present
- Deletes
  - Default – The before values of all keys are written to the trail.
  - `NOCOMPRESSDELETES` – The before values of all columns are written to the trail.

Primary key update operations may also be generated if the before values of keys are present and do not match the after values.

10.1.3.6 Column Data

All parsers retrieve column data from the message text and write it to the Oracle GoldenGate trail. In some cases the columns are read in index order as defined by the source definitions, in other cases they are accessed by name.

Depending on the configuration and original message text, both before and after or only after images of the column data may be available. For updates, the data for non-updated columns may or may not be available.

All column data is retrieved as text. It is converted internally into the correct data type for that column based on the source definitions. Any conversion problem will result in an error and the process will abend.

10.1.4 Optional Data

The following data may be included, but is not required.

10.1.4.1 Transaction Indicator

The relationship of transactions to messages can be:

- One transaction per message

  This is determined automatically by the scope of the message.

- Multiple transactions per message

  This is determined by the transaction indicator (`txind`). If there is no transaction indicator, the XML parser can create transactions based on a matching transaction rule.

- Multiple messages per transaction

  The transaction indicator (`txind`) is required to specify whether the operation is the beginning, middle, end or the whole transaction. The transaction indicator value for any specific operation is matched against the values defined for each transaction indicator type. A transaction is started if the indicator value is beginning or whole, continued if it is middle, and ended if it is end or whole.

10.1.4.2 Transaction Name

The transaction name (`txname`) is optional data that can be used to associate an arbitrary name to a transaction. This can be added to the trail as a token using a `GETENV` function.

10.1.4.3 Transaction Owner

The transaction owner (txowner) is optional data that can be used to associate an arbitrary user name to a transaction. This can be added to the trail as a token using a `GETENV` function, or used to exclude certain transactions from processing using the `EXCLUDEUSER` Extract parameter.

10.2 Fixed Width Parsing

Fixed width parsing is based on a data definition that defines the position and the length of each field. This is in the format of a Cobol copybook. A set of properties define rules for mapping the copybook to logical records in the Oracle GoldenGate trail and in the source definitions file.

The incoming data should consist of a standard format header followed by a data segment. Both should contain fixed width fields. The data is parsed based on the PIC definition in the copybook. It is written to the trail translated as explained in Header and Record Data Type Translation.

10.2.1 Header

The header must be defined by a copybook 01 level record that includes the following:

- A commit timestamp or a change time for the record
- A code to indicate the type of operation: insert, update, or delete
- The copybook record name to use when parsing the data segment

Any fields in the header record that are not mapped to Oracle GoldenGate header fields are output as columns.

The following example shows a copybook definition containing the required header values

Example 10-1 Specifying a Header

```
01 HEADER.
20 Hdr-Timestamp            PIC X(23)
20 Hdr-Source-DB-Function PIC X
20 Hdr-Source-DB-Rec-ID    PIC X(8)
```

For the above example, you must set the following properties:

```
fixed.header=HEADER
fixed.timestamp=Hdr-Timestamp
fixed.optype=Hdr-Source-DB-Function
fixed.table=Hdr-Source-DB-Rec-Id
```

The logical name table output in this case will be the value of `Hdr-Source-DB-Rec-Id`.

10.2.1.1 Specifying Compound Table Names

More than one field can be used for a table name. For example, you can define the logical schema name through a static property such as:

```
fixed.schema=MYSCHEMA
```

Then you can add a property that defines the data record as multiple fields from the copybook header definition.

Example 10-2 Specifying Compound Table Names

```
01  HEADER.
    20  Hdr-Source-DB              PIC X(8).
    20  Hdr-Source-DB-Rec-Id       PIC X(8).
    20  Hdr-Source-DB-Rec-Version  PIC 9(4).
    20  Hdr-Source-DB-Function     PIC X.
    20  Hdr-Timestamp              PIC X(22).
```

For the above example, you must set the following properties:

```
fixed.header=HEADER
fixed.table=Hdr-Source-DB-Rec-Id,Hdr-Source-DB-Rec-Version
fixed.schema=MYSCHEMA
```

The fields will be concatenated to result in logical schema and table names of the form:

```
MYSCHEMA.Hdr-Source-DB-Rec-Id+Hdr-Source-DB-Rec-Version
```

10.2.1.2 Specifying timestamp Formats

A timestamp is parsed using the default format `YYYY-MM-DD HH:MM:SS.FFF`, with `FFF` depending on the size of the field.

Specify different incoming formats by entering a comment before the datetime field as shown in the next example.

Example 10-3 Specifying timestamp formats

```
01  HEADER.
* DATEFORMAT YYYY-MM-DD-HH.MM.SS.FF
    20  Hdr-Timestamp       PIC X(23)
```

10.2.1.3 Specifying the Function

Use properties to map the standard Oracle GoldenGate operation types to the `optype` values. The following example specifies that the operation type is in the `Hdr-Source-DB-Function` field and that the value for insert is `A`, update is `U` and delete is `D`.

Example 10-4 Specifying the Function

```
fixed.optype=Hdr-Source-DB-Function
fixed.optype.insert=A
fixed.optype.update=U
fixed.optype.delete=D
```

10.2.2 Header and Record Data Type Translation

The data in the header and the record data are written to the trail based on the translated data type.

- A field definition preceded by a date format comment is translated to an Oracle GoldenGate datetime field of the specified size. If there is no date format comment, the field will be defined by its underlying data type.
- A `PIC X` field is translated to the `CHAR` data type of the indicated size.
- A `PIC 9` field is translated to a `NUMBER` data type with the defined precision and scale. Numbers that are signed or unsigned and those with or without decimals are supported.

The following examples show the translation for various `PIC` definitions.

Input	Output
`PIC XX`	`CHAR(2)`
`PIC X(16)`	`CHAR(16)`
`PIC 9(4)`	`NUMBER(4)`

Input	Output
* YYMMDD PIC 9(6)	DATE(10) YYYY-MM-DD
PIC 99.99	NUMBER(4,2)
PIC 9(5)V99	NUMBER(7,2)

In the example an input YYMMDD date of 100522 is translated to 2010-05-22. The number 1234567 with the specified format PIC 9(5)V99 is translated to a seven digit number with two decimal places, or 12345.67.

10.2.3 Key identification

A comment is used to identify key columns within the data record. The Gendef utility that generates the source definitions uses the comment to locate a key column.

In the following example Account has been marked as a key column for TABLE1.

```
01 TABLE1
* KEY
20  Account      PIC X(19)
20  PAN_Seq_Num PIC 9(3)
```

10.3 Delimited parsing

Delimited parsing is based a preexisting source definitions files and a set of properties. The properties specify the delimiters to use and other rules, such as whether there are column names and before values. The source definitions file determines the valid tables to be processed and the order and data type of the columns in the tables.

The format of the delimited message is:

{*METACOLS*}^{n}[,{*COLNAMES*}]^{m}[,{*COLBEFOREVALS*}]^{m},{*COLVALUES*}^{m}\n

Where:

- There can be *n* metadata columns each followed by a field delimiter such as the comma shown in the format statement.
- There can be *m* column values. Each of these are preceded by a field delimiter such as a comma.
- The column name and before value are optional.
- Each record is terminated by an end of line delimiter, such as \n.

10.3.1 Metadata Columns

The metadata columns correspond to the header and contain fields that have special meaning. Metadata columns should include the following information.

- **optype** contains values indicating if the record is an insert, update, or delete. The default values are `I`, `U`, and `D`.
- **timestamp** indicates type of value to use for the commit timestamp of the record. The format of the timestamp defaults to `YYYY-DD-MM HH:MM:SS.FFF`.
- **schemaandtable** is the full table name for the record in the format `SCHEMA.TABLE`.
- **schema** is the record's schema name.
- **table** is the record's table name.
- **txind** is a value that indicates whether the record is the beginning, middle, end or the only record in the transaction. The default values are 0, 1, 2, 3.
- **id** is the value used as the sequence number (RSN or CSN) of the record. The id of the first record (operation) in the transaction is used for the sequence number of the transaction.

10.3.2 Parsing Properties

Properties can be set to describe delimiters, values, and date and time formats.

10.3.2.1 Properties to Describe Delimiters

The following properties determine the parsing rules for delimiting the record.

- **fielddelim** specifies one or more ASCII or hexadecimal characters as the value for the field delimiter
- **recorddelim** specifies one or more ASCII or hexadecimal characters as the value for the record delimiter
- **quote** specifies one or more ASCII or hexadecimal characters to use for quoted values
- **nullindicator** specifies one or more ASCII or hexadecimal characters to use for `NULL` values

You can define escape characters for the delimiters so they will be replaced if the characters are found in the text. For example if a backslash and apostrophe (\') are specified, then the input "They used Mike\'s truck" is translated to "They used Mike's truck". Or if two quotes ("") are specified, "They call him ""Big Al""" is translated to "They call him "Big Al"".

Data values may be present in the record without quotes, but the system only removes escape characters within quoted values. A non-quoted string that matches a null indicator is treated as null.

10.3.2.2 Properties to Describe Values

The following properties provide more information:

- **hasbefores** indicates before values are present for each record
- **hasnames** indicates column names are present for each record
- **afterfirst** indicates column after values come before column before values

- **isgrouped** indicates all column names, before values and after values are grouped together in three blocks, rather than alternately per column

10.3.2.3 Properties to Describe Date and Time

The default format `YYYY-DD-MM HH:MM:SS.FFF` is used to parse dates. The user can use properties to override this on a global, table or column level. Examples of changing the format are shown below.

```
delim.dateformat.default=MM/DD/YYYY-HH:MM:SS
delim.dateformat.MY.TABLE=DD/MMM/YYYY
delim.dateformat.MY.TABLE.COL1=MMYYYY
```

10.3.3 Parsing Steps

The steps in delimited parsing are:

1. The parser first reads and validates the metadata columns for each record.
2. This provides the table name, which can then be used to look up column definitions for that table in the source definitions file.
3. If a definition cannot be found for a table, the processing will stop.
4. Otherwise the columns are parsed and output to the trail in the order and format defined by the source definitions.

10.4 XML Parsing

XML parsing is based on a preexisting source definitions file and a set of properties. The properties specify rules to determine XML elements and attributes that correspond to transactions, operations and columns. The source definitions file determines the valid tables to be processed and the ordering and data types of columns in those tables.

10.4.1 Styles of XML

The XML message is formatted in either dynamic or static XML. At runtime the contents of dynamic XML are data values that cannot be predetermined using a sample XML or XSD document. The contents of static XML that determine tables and column element or attribute names can be predetermined using those sample documents.

The following two examples contain the same data.

Example 10-5 An Example of Static XML

```
<NewMyTableEntries>
  <NewMyTableEntry>
    <CreateTime>2010-02-05:10:11:21</CreateTime>
    <KeyCol>keyval</KeyCol>
    <Col1>col1val</Col1>
  </NewMyTableEntry>
</NewMyTableEntries>
```

The `NewMyTableEntries` element marks the transaction boundaries. The `NewMyTableEntry` indicates an insert to `MY.TABLE`. The timestamp is present in an element text value, and the column names are indicated by element names.

You can define rules in the properties file to parse either of these two styles of XML through a set of XPath-like properties. The goal of the properties is to map the XML to a predefined source definitions file through XPath matches.

Example 10-6 An Example of Dynamic XML

```
<transaction id="1234" ts="2010-02-05:10:11:21">
  <operation table="MY.TABLE" optype="I">
    <column name="keycol" index="0">
      <aftervalue><![CDATA[keyval]]></aftervalue>
    </column>
    <column name="col1" index="1">
      <aftervalue><![CDATA[col1val]]></aftervalue>
    </column>
  </operation>
</transaction>
```

Every operation to every table has the same basic message structure consisting of transaction, operation and column elements. The table name, operation type, timestamp, column names, column values, etc. are obtained from attribute or element text values.

10.4.2 XML Parsing Rules

Independent of the style of XML, the parsing process needs to determine:

- Transaction boundaries
- Operation entries and metadata including:
  - Table name
  - Operation type
  - Timestamp
- Column entries and metadata including:
  - Either the column name or index; if both are specified the system will check to see if the column with the specified data has the specified name.
  - Column before or after values, sometimes both.

This is done through a set of interrelated rules. For each type of XML message that is to be processed you name a rule that will be used to obtain the required data. For each of these named rules you add properties to:

- Specify the rule as a transaction, operation, or column rule type. Rules of any type are required to have a specified name and type.
- Specify the XPath expression to match to see if the rule is active for the document being processed. This is optional; if not defined the parser will match the node of the parent rule or the whole document if this is the first rule.
- List detailed rules (`subrules`) that are to be processed in the order listed. Which `subrules` are valid is determined by the rule type. `Subrules` are optional.

In the following example the top-level rule is defined as `genericrule`. It is a `transaction` type rule. Its `subrules` are defined in `oprule` and they are of the type `operation`.

```
xmlparser.rules=genericrule
xmlparser.rules.genericrule.type=tx
xmlparser.rules.genericrule.subrules=oprule
xmlparser.rules.oprule.type=op
```

10.4.3 XPath Expressions

The XML parser supports a subset of XPath expressions necessary to match elements and extract data. An expression can be used to match a particular element or to extract data.

When doing data extraction most of the path is used to match. The tail of the expression is used for extraction.

10.4.3.1 Supported Constructs:

Supported Constructs	Description
`/e`	Use the absolute path from the root of the document to match `e`.
`./e or e`	Use the relative path from current node being processed to match `e`.
`../e`	Use a path based on the parent of the current node (can be repeated) to match `e`.
`//e`	Match `e` wherever it occurs in a document.
`*`	Match any element. Note: Partially wild-carded names are not supported.
`[n]`	Match the nth occurrence of an expression.
`[x=v]`	Match when x is equal to some value v where x can be: • `@att` – some attribute value • `text()` – some text value • `name()` – the element name • `position()` – the element position

10.4.3.2 Supported Expressions

Supported Expressions	Descriptions
Match root element	`/My/Element`
Match sub element to current node	`./Sub/Element`

Supported Expressions	Descriptions
Match nth element	`/My/*[n]`
Match nth Some element	`/My/Some[n]`
Match any text value	`/My/*[text() ='value']`
Match the text in Some element	`/My/Some[text() = 'value']`
Match any attribute	`/My/*[@att = 'value']`
Match the attribute in Some element	`/My/Some[@att = 'value']`

10.4.3.3 Obtaining Data Values

In addition to matching paths, the XPath expressions can also be used to obtain data values, either absolutely or relative to the current node being processed. Data value expressions can contain any of the path elements above, but must end with one of the value accessors listed below.

Value Accessors	Description
`@att`	Some attribute value.
`text()`	The text content (value) of an element.
`content()`	The full content of an element, including any child XML nodes.
`name()`	The name of an element.
`position()`	The position of an element in its parent.

Example 10-7 Examples of Extracting Data Values

To extract the relative element text value:

```
/My/Element/text()
```

To extract the absolute attribute value:

```
/My/Element/@att
```

To extract element text value with a match:

```
/My/Some[@att = 'value']/Sub/text()
```

Note:

Path accessors, such as ancestor/descendent/self, are not supported.

10.4.4 Other Value Expressions

The values extracted by the XML parser are either column values or properties of the transaction or operation, such as table or timestamp. These values are either obtained from XML using XPath or through properties of the JMS message, system values, or hard coded values. The XML parser properties specify which of these options are valid for obtaining the values for that property.

The following example specifies that `timestamp` can be an XPath expression, a JMS property, or the system generated timestamp.

```
{txrule}.timestamp={xpath-expression}|${jms-property}|*ts
```

The next example specifies that `table` can be an XPath expression, a JMS property, or hard coded value.

```
{oprule}.table={xpath-expression}|${jms-property}|"value"
```

The last example specifies that `name` can be a XPath expression or hard coded value.

```
{colrule}.timestamp={xpath-expression}|"value"
```

10.4.5 Transaction Rules

The rule that specifies the boundary for a transaction is at the highest level. Messages may contain a single transaction, multiple transactions, or a part of a transaction that spans messages. These are specified as follows:

- **single** - The transaction rule match is not defined.
- **multiple** - Each transaction rule match defines new transaction.
- **span** – No transaction rule is defined; instead a transaction indicator is specified in an operation rule.

For a transaction rule, the following properties of the rule may also be defined through XPath or other expressions:

- **timestamp** – The time at which the transaction occurred.
- **txid** – The identifier for the transaction.

Transaction rules can have multiple `subrules`, but each must be of type operation.

The following example specifies a transaction that is the whole message and includes a timestamp that comes from the JMS property.

Example 10-8 JMS Timestamp

```
singletxrule.timestamp=$JMSTimeStamp
```

The following example matches the root element transaction and obtains the timestamp from the `ts` attribute.

Example 10-9 ts Timestamp

```
dyntxrule.match=/Transaction
dyntxrule.timestamp=@ts
```

10.4.6 Operation Rules

An operation rule can either be a subrule of a transaction rule, or a highest level rule (if the transaction is a property of the operation).

In addition to the standard rule properties, an operation rule should also define the following through XPath or other expressions:

- **timestamp** – The timestamp of the operation. This is optional if the transaction rule is defined.
- **table** – The name of the table on which this is an operation. Use this with schema.
- **schema** – The name of schema for the table.
- **schemaandtable** – Both schema and table name together in the form `SCHEMA.TABLE`. This can be used in place of the individual table and schema properties.
- **optype** – Specifies whether this is an insert, update or delete operation based on `optype` values:
  - **optype.insertval** – The value indicating an insert. The default is `I`.
  - **optype.updateval** – The value indicating an update. The default is `U`.
  - **optype.deleteval** – The value indicating a delete. The default is `D`.
- **seqid** – The identifier for the operation. This will be the transaction identifier if `txid` has not already been defined at the transaction level.
- **txind** – Specifies whether this operation is the beginning of a transaction, in the middle or at the end; or if it is the whole operation. This property is optional and not valid if the operation rule is a subrule of a transaction rule.

Operation rules can have multiple subrules of type operation or column.

The following example dynamically obtains operation information from the `/Operation` element of a `/Transaction`.

Example 10-10 Operation

```
dynoprule.match=./Operation
dynoprule.schemaandtable=@table
dynoprule.optype=@type
```

The following example statically matches `/NewMyTableEntry` element to an insert operation on the `MY.TABLE` table.

Example 10-11 Operation example

```
statoprule.match=./NewMyTableEntry
statoprule.schemaandtable="MY.TABLE"
statoprule.optype="I"
statoprule.timestamp=./CreateTime/text()
```

10.4.7 Column Rules

A column rule must be a subrule of an operation rule. In addition to the standard rule properties, a column rule should also define the following through XPath or other expressions.

- **name** – The name of the column within the table definition.
- **index** – The index of the column within the table definition.

> **Note:**
>
> If only one of `name` and `index` is defined, the other will be determined.

- **before.value** – The before value of the column. This is required for deletes, but is optional for updates.
- **before.isnull** – Indicates whether the before value of the column is null.
- **before.ismissing** – Indicates whether the before value of the column is missing.
- **after.value** – The before value of the column. This is required for deletes, but is optional for updates.
- **after.isnull** – Indicates whether the before value of the column is null.
- **after.ismissing** – Indicates whether the before value of the column is missing.
- **value** – An expression to use for both `before.value` and `after.value` unless overridden by specific before or after values. Note that this does not support different before values for updates.
- **isnull** – An expression to use for both before.isnull and after.isnull unless overridden.
- **ismissing** – An expression to use for both before.ismissing and after.ismissing unless overridden.

The following example dynamically obtains column information from the `/Column` element of an `/Operation`

Example 10-12 Dynamic Extraction of Column Information

```
dyncolrule.match=./Column
dyncolrule.name=@name
dyncolrule.before.value=./beforevalue/text()
dyncolrule.after.value=./aftervalue/text()
```

The following example statically matches the `/KeyCol` and `/Col1` elements to columns in `MY.TABLE`.

Example 10-13 Static Matching of Elements to Columns

```
statkeycolrule.match=/KeyCol
statkeycolrule.name="keycol"
statkeycolrule.value=./text()
statcol1rule.match=/Col1
statcol1rule.name="col1"
statcol1rule.value=./text()
```

10.4.8 Overall Rules Example

The following example uses the XML samples shown earlier with appropriate rules to generate the same resulting operation on the `MY.TABLE` table.

Dynamic XML

```
<transaction id="1234"
    ts="2010-02-05:10:11:21">
  <operation table="MY.TABLE" optype="I">
    <column name="keycol" index="0">
      <aftervalue>
<![CDATA[keyval]]>
      </aftervalue>
    </column>
    <column name="col1" index="1">
      <aftervalue>
         <![CDATA[col1val]]>
      </aftervalue>
    </column>
  </operation>
</transaction>
```

Static XML

```
NewMyTableEntries>
  <NewMyTableEntry>
    <CreateTime>
       2010-02-05:10:11:21
    </CreateTime>
    <KeyCol>keyval</KeyCol>
    <Col1>col1val</Col1>
  </NewMyTableEntry>
</NewMyTableEntries>
```

Dynamic

```
dyntxrule.match=/Transaction
dyntxrule.timestamp=@ts
dyntxrule.subrules=dynoprule
dynoprule.match=./Operation
dynoprule.schemaandtable=@table
dynoprule.optype=@type
dynoprule.subrules=dyncolrule
dyncolrule.match=./Column
dyncolrule.name=@name


dyncolrule.before.value=./beforevalue/text()
dyncolrule.after.value=./aftervalue/text()
```

Static

```
stattxrule.match=/NewMyTableEntries
stattxrule.subrules= statoprule
statoprule.match=./NewMyTableEntry
statoprule.schemaandtable="MY.TABLE"
statoprule.optype="I"
statoprule.timestamp=./CreateTime/text()
statoprule.subrules= statkeycolrule,
statcol1rule
statkeycolrule.match=/KeyCol


statkeycolrule.name="keycol"
statkeycolrule.value=./text()
statcol1rule.match=/Col1
statcol1rule.name="col1"
statcol1rule.value=./text()
```

```
INSERT INTO MY.TABLE (KEYCOL, COL1)
VALUES ('keyval', 'col1val')
```

10.5 Source definitions Generation Utility

Oracle GoldenGate for Java includes a Gendef utility that generates an Oracle GoldenGate source definitions file from the properties defined in a properties file. It creates a normalized definition of tables based on the property settings and other parser-specific data definition values.

The syntax to run this utility is:

```
gendef –prop {property_file} [-out {output_file}
```

This defaults to sending the source definitions to standard out, but it can be directed to a file using the `–out` parameter. For example:

```
gendef –prop dirprm/jmsvam.properties -out dirdef/msgdefs.def
```

The output source definitions file can then be used in a pump or delivery process to interpret the trail data created through the VAM.

11

Message Capture Properties

This chapter explains the options available for configuration of the property file for the Oracle GoldenGate for Java VAM.

This chapter includes the following sections:

11.1 Logging and Connection Properties

The following properties control the connection to JMS and the log file names, error handling, and message output.

11.1.1 Logging Properties

Logging is controlled by the following properties.

11.1.1.1 gg.log

Specifies the type of logging that is to be used. The default implementation is the `JDK` option. This is the built-in Java logging called `java.util.logging` (`JUL`). The other logging options are `log4j` or `logback`. The syntax is:

```
gg.log={JDK|log4j|logback}
```

For example, you set the logging implementation to `log4j`, which is the preferred logging method, as follows:

```
gg.log=log4j
```

The log file is created in the report subdirectory of the installation. The default log file name includes the group name of the associated Extract and the file extension is `log`.

11.1.1.2 gg.log.level

Specifies the overall log level for all modules. The syntax is:

```
gg.log.level={ERROR|WARN|INFO|DEBUG}
```

The log levels are defined as follows:

- `ERROR` – Only write messages if errors occur
- `WARN` – Write error and warning messages
- `INFO` – Write error, warning and informational messages
- `DEBUG` – Write all messages, including debug ones.

The default logging level is `INFO`. The messages in this case will be produced on startup, shutdown and periodically during operation. If the level is switched to `DEBUG`, large volumes of messages may occur which could impact performance. For example, the following sets the global logging level to `INFO`:

```
# global logging level
gg.log.level=INFO
```

11.1.1.3 gg.log.file

Specifies the path to the log file. The syntax is:

```
gg.log.file=path_to_file
```

Where the *`path_to_file`* is the fully defined location of the log file. This allows a change to the name of the log, but you must include the Extract name if you have more than one Extract to avoid one overwriting the log of the other.

11.1.1.4 gg.log.classpath

Specifies the class path to the jars used to implement logging.

```
gg.log.classpath=path_to_jars
```

11.1.2 JMS Connection Properties

The JMS connection properties set up the connection, such as how to start up the JVM for JMS integration.

11.1.2.1 jvm.boot options

Specifies the class path and boot options that will be applied when the JVM starts up. The path needs colon (:) separators for UNIX/Linux and semicolons (;) for Windows.

The syntax is:

```
jvm.bootoptions=option[, option][. . .]
```

The *`options`* are the same as those passed to Java executed from the command line. They may include class path, system properties, and JVM memory options (such as maximum memory or initial memory) that are valid for the version of Java being used. Valid options may vary based on the JVM version and provider.

For example (all on a single line):

```
jvm.bootoptions= -Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=my-log4j.properties
```

The `log4j.configuration` property could be a fully qualified URL to a log4j properties file; by default this file is searched for in the class path. You may use your own log4j configuration, or one of the pre-configured log4j settings: `log4j.properties` (default level of logging), `debug-log4j.properties` (debug logging) or `trace-log4j.properties` (very verbose logging).

11.1.2.2 jms.report.output

Specifies where the JMS report is written. The syntax is:

```
jms.report.output={report|log|both}
```

Where:

- `report` sends the JMS report to the Oracle GoldenGate report file. This is the default.
- `log` will write to the Java log file (if one is configured)
- `both` will send to both locations.

11.1.2.3 jms.report.time

Specifies the frequency of report generation based on time.

```
jms.report.time=time_specification
```

The following examples write a report every 30 seconds, 45 minutes and eight hours.

```
jms.report.time=30sec
jms.report.time=45min
jms.report.time=8hr
```

11.1.2.4 jms.report.records

Specifies the frequency of report generation based on number of records. The syntax is:

```
jms.report.records=number
```

The following example writes a report every 1000 records.

```
jms.report.records=1000
```

11.1.2.5 jms.id

Specifies that a unique identifier with the indicated format is passed back from the JMS integration to the message capture VAM. This may be used by the VAM as a unique sequence ID for records.

```
jms.id={ogg|time|wmq|activemq|message_header|custom_java_class}
```

`Where:`

- `ogg` - returns the message header property `GG_ID` which is set by Oracle GoldenGate JMS delivery.
- `time` - uses a system timestamp as a starting point for the message ID
- `wmq` - reformats a WebSphere MQ Message ID for use with the VAM
- `activemq` - reformats an ActiveMQ Message ID for use with the VAM
- *`message_header`* - specifies the user customized JMS message header to be included, such as JMSMessageID, JMSCorrelationID, or JMSTimestamp.
- *`custom_java_class`* - specifies a custom Java class that creates a string to be used as an ID.

For example:

```
jms.id=time
jms.id=JMSMessageID
```

The ID returned must be unique, incrementing, and fixed-width. If there are duplicate numbers, the duplicates are skipped. If the message ID changes length, the Extract process will abend.

11.1.2.6 jms.destination

Specifies the queue or topic name to be looked up via JNDI.

```
jms.destination=jndi_name
```

For example:

```
jms.destination=sampleQ
```

11.1.2.7 jms.connectionFactory

Specifies the connection factory name to be looked up via JNDI.

```
jms.connectionFactory=jndi_name
```

For example

```
jms.connectionFactory=ConnectionFactory
```

11.1.2.8 jms.user, jms.password

Sets the user name and password of the JMS connection, as specified by the JMS provider.

```
jms.user=user_name
jms.password=password
```

This is not used for JNDI security. To set JNDI authentication, see the JNDI `java.naming.security` properties.

For example:

```
jms.user=myuser
jms.password=mypasswd
```

11.1.3 JNDI Properties

In addition to specific properties for the message capture VAM, the JMS integration also supports setting JNDI properties required for connection to an Initial Context to look up the connection factory and destination. The following properties must be set:

```
java.naming.provider.url=url
java.naming.factory.initial=java_class_name
```

If JNDI security is enabled, the following properties may be set:

```
java.naming.security.principal=user_name
java.naming.security.credentials=password_or_other_authenticator
```

For example:

```
java.naming.provider.url= t3://localhost:7001
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.security.principal=jndiuser
java.naming.security.credentials=jndipw
```

11.2 Parser Properties

Properties specify the formats of the message and the translation rules for each type of parser: fixed, delimited, or XML. Set the `parser.type` property to specify which parser to use. The remaining properties are parser specific.

11.2.1 Setting the Type of Parser

The following property sets the parser type.

11.2.1.1 parser.type

Specifies the parser to use.

```
parser.type={fixed|delim|xml}
```

`Where:`

- `fixed` invokes the fixed width parser
- `delim` invokes the delimited parser
- `xml` invokes the XML parser

For example:

```
parser.type=delim
```

11.2.2 Fixed Parser Properties

The following properties are required for the fixed parser.

11.2.2.1 fixed.schematype

Specifies the type of file used as metadata for message capture. The two valid options are `sourcedefs` and `copybook`.

```
fixed.schematype={sourcedefs|copybook}
```

For example:

```
fixed.schematype=copybook
```

The value of this property determines the other properties that must be set in order to successfully parse the incoming data.

11.2.2.2 fixed.sourcedefs

If the `fixed.schematype=sourcedefs`, this property specifies the location of the source definitions file that is to be used.

```
fixed.sourcedefs=file_location
```

For example:

```
fixed.sourcedefs=dirdef/hrdemo.def
```

11.2.2.3 fixed.copybook

If the `fixed.schematype=copybook`, this property specifies the location of the copybook file to be used by the message capture process.

```
fixed.copybook=file_location
```

For example:

```
fixed.copybook=test_copy_book.cpy
```

11.2.2.4 fixed.header

Specifies the name of the `sourcedefs` entry or copybook record that contains header information used to determine the data block structure:

```
fixed.header=record_name
```

For example:

```
fixed.header=HEADER
```

11.2.2.5 fixed.seqid

Specifies the name of the header field, JMS property, or system value that contains the `seqid` used to uniquely identify individual records. This value must be continually incrementing and the last character must be the least significant.

```
fixed.seqid={field_name|$jms_property|*seqid}
```

`Where:`

- `field_name` indicates the name of a header field containing the *`seqid`*
- `jms_property` uses the value of the specified JMS header property. A special value of this is `$jmsid` which uses the value returned by the mechanism chosen by the `jms.id` property
- `seqid` indicates a simple incrementing 64-bit integer generated by the system

For example:

```
fixed.seqid=$jmsid
```

11.2.2.6 fixed.timestamp

Specifies the name of the field, JMS property, or system value that contains the timestamp.

```
fixed.timestamp={field_name|$jms_property|*ts}
```

For example:

```
fixed.timestamp=TIMESTAMP
fixed.timestamp=$JMSTimeStamp
fixed.timestamp=*ts
```

11.2.2.7 fixed.timestamp.format

Specifies the format of the timestamp field.

```
fixed.timestamp.format=format
```

Where the format can include punctuation characters plus:

- YYYY – four digit year
- YY – two digit year

- M[M] – one or two digit month
- D[D] – one or two digit day
- HH – hours in twenty four hour notation
- MI – minutes
- SS – seconds
- Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
fixed.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

11.2.2.8 fixed.txid

Specifies the name of the field, JMS property, or system value that contains the txid used to uniquely identify transactions. This value must increment for each transaction.

```
fixed.txid={field_name|$jms_property|*txid}
```

For most cases using the system value of *txid is preferred.

For example:

```
fixed.txid=$JMSTxId
fixed.txid=*txid
```

11.2.2.9 fixed.txowner

Specifies the name of the field, JMS property, or static value that contains a user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
fixed.txowner={field_name|$jms_property|"value"}
```

For example:

```
fixed.txowner=$MessageOwner
fixed.txowner="jsmith"
```

11.2.2.10 fixed.txname

Specifies the name of the field, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
fixed.txname={field_name|$jms_property|"value"}
```

For example:

```
fixed.txname="fixedtx"
```

11.2.2.11 fixed.optype

Specifies the name of the field, or JMS property that contains the operation type, which is validated against the fixed.optype values specified in the next sections.

```
fixed.header.optype={field_name|$jms_property}
```

For example:

```
fixed.header.optype=FUNCTION
```

11.2.2.12 fixed.optype.insertval

This value identifies an insert operation. The default is `I`.

```
fixed.optype.insertval={value|\xhex_value}
```

For example:

```
fixed.optype.insertval=A
```

11.2.2.13 fixed.optype.updateval

This value identifies an update operation. The default is `U`.

```
fixed.optype.updateval={value|\xhex_value}
```

For example:

```
fixed.optype.updateval=M
```

11.2.2.14 fixed.optype.deleteval

This value identifies a delete operation.The default is `D`.

```
fixed.optype.deleteval={value|\xhex_value}
```

For example:

```
fixed.optype.deleteval=R
```

11.2.2.15 fixed.table

Specifies the name of the table. This enables the parser to find the data record definition needed to translate the non-header data portion.

```
fixed.table=field_name|$jms_property[, . . .]
```

More than one comma delimited field name may be used to determine the name of the table Each field name corresponds to a field in the header record defined by the `fixed.header` property or JMS property. The values of these fields are concatenated to identify the data record.

For example:

```
fixed.table=$JMSTableName
fixed.table=SOURCE_Db,SOURCE_Db_Rec_Version
```

11.2.2.16 fixed.schema

Specifies the static name of the schema when generating `SCHEMA.TABLE` table names.

```
fixed.schema="value"
```

For example:

```
fixed.schema="OGG"
```

11.2.2.17 fixed.txind

Specifies the name of the field or JMS property that contains a transaction indicator that is validated against the transaction indicator values. If this is not defined, all operations within a single message will be seen to have occurred within a whole

transaction. If defined, then it determines the beginning, middle and end of transactions. Transactions defined in this way can span messages. This is an optional property.

```
fixed.txind={field_name|$jms_property}
```

For example:

```
fixed.txind=$TX_IND
```

11.2.2.18 fixed.txind.beginval

This value identifies an operation as the beginning of a transaction. The defaults is B.

```
fixed.txind.beginval={value|\xhex_value}
```

For example:

```
fixed.txind.beginval=0
```

11.2.2.19 fixed.txind.middleval

This value identifies an operation as the middle of a transaction. The default is M.

```
fixed.txind.middleval={value|\xhex_value}
```

For example:

```
fixed.txind.middleval=1
```

11.2.2.20 fixed.txind.endval

This value identifies an operation as the end of a transaction. The default is *E*.

```
fixed.txind.endval={value|\xhex_value}
```

For example:

```
fixed.txind.endval=2
```

11.2.2.21 fixed.txind.wholeval

This value identifies an operation as a whole transaction. The default is *W*.

```
fixed.txind.wholeval={value|\xhex_value}
```

For example:

```
fixed.txind.wholeval=3
```

11.2.3 Delimited Parser Properties

The following properties are required for the delimited parser except where otherwise noted.

11.2.3.1 delim.sourcedefs

Specifies the location of the source definitions file to use.

```
delim.sourcedefs=file_location
```

For example:

```
delim.sourcedefs=dirdef/hrdemo.def
```

11.2.3.2 delim.header

Specifies the list of values that come before the data and assigns names to each.

```
delim.header=name[,name2][. . .]
```

The names must be unique. They can be referenced in other `delim` properties or wherever header fields can be used.

For example:

```
delim.header=optype, tablename, ts
delim.timestamp=ts
```

11.2.3.3 delim.seqid

Specifies the name of the header field, JMS property, or system value that contains the `seqid` used to uniquely identify individual records. This value must increment and the last character must be the least significant.

```
delim.seqid={field_name|$jms_property|*seqid}
```

`Where:`

- *`field_name`* indicates the name of a header field containing the *`seqid`*
- *`jms_property`* uses the value of the specified JMS header property, a special value of this is `$jmsid` which uses the value returned by the mechanism chosen by the `jms.id property`
- `seqid` indicates a simple continually incrementing 64-bit integer generated by the system

For example:

```
delim.seqid=$jmsid
```

11.2.3.4 delim.timestamp

Specifies the name of the JMS property, header field, or system value that contains the timestamp.

```
delim.timestamp={field_name|$jms_property|*ts}
```

For example:

```
delim.timestamp=TIMESTAMP
delim.timestamp=$JMSTimeStamp
delim.timestamp=*ts
```

11.2.3.5 delim.timestamp.format

Specifies the format of the timestamp field.

```
delim.timestamp.format=format
```

Where the *`format`* can include punctuation characters plus:

- YYYY – four digit year
- YY – two digit year

- `M[M]` – one or two digit month
- `D[D]` – one or two digit day
- `HH` – hours in twenty four hour notation
- `MI` – minutes
- `SS` – seconds
- `Fn` – n number of fractions

The default format is "`YYYY-MM-DD:HH:MI:SS.FFF`"

For example:

```
delim.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

11.2.3.6 delim.txid

Specifies the name of the JMS property, header field, or system value that contains the `txid` used to uniquely identify transactions. This value must increment for each transaction.

```
delim.txid={field_name|$jms_property|*txid}
```

For most cases using the system value of `*txid` is preferred.

For example:

```
delim.txid=$JMSTxId
delim.txid=*txid
```

11.2.3.7 delim.txowner

Specifies the name of the JMS property, header field, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
delim.txowner={field_name|$jms_property|"value"}
```

For example:

```
delim.txowner=$MessageOwner
delim.txowner="jsmith"
```

11.2.3.8 delim.txname

Specifies the name of the JMS property, header field, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
delim.txname={field_name|$jms_property|"value"}
```

For example:

```
delim.txname="fixedtx"
```

11.2.3.9 delim.optype

Specifies the name of the JMS property or header field that contains the operation type. This is compared to the values for `delim.optype.insertval`, `delim.optype.updateval` and `delim.optype.deleteval` to determine the operation.

```
delim.optype={field_name|$jms_property}
```

For example:

```
delim.optype=optype
```

11.2.3.10 delim.optype.insertval

This value identifies an insert operation. The default is I.

```
delim.optype.insertval={value|\xhex_value}
```

For example:

```
delim.optype.insertval=A
```

11.2.3.11 delim.optype.updateval

This value identifies an update operation. The default is U.

```
delim.optype.updateval={value|\xhex_value}
```

For example:

```
delim.optype.updateval=M
```

11.2.3.12 delim.optype.deleteval

This value identifies a delete operation. The default is D.

```
delim.optype.deleteval={value|\xhex_value}
```

For example:

```
delim.optype.deleteval=R
```

11.2.3.13 delim.schemaandtable

Specifies the name of the JMS property or header field that contains the schema and table name in the form SCHEMA.TABLE.

```
delim.schemaandtable={field_name|$jms_property}
```

For example:

```
delim.schemaandtable=$FullTableName
```

11.2.3.14 delim.schema

Specifies the name of the JMS property, header field, or hard-coded value that contains the schema name.

```
delim.schema={field_name|$jms_property|"value"}
```

For example:

```
delim.schema="OGG"
```

11.2.3.15 delim.table

Specifies the name of the JMS property or header field that contains the table name.

```
delim.table={field_name|$jms_property}
```

For example:

```
delim.table=TABLE_NAME
```

11.2.3.16 delim.txind

Specifies the name of the JMS property or header field that contains the transaction indicator to be validated against `beginval`, `middleval`, `endval` or `wholeval`. All operations within a single message will be seen as within one transaction if this property is not set. If it is set it determines the beginning, middle and end of transactions. Transactions defined in this way can span messages. This is an optional property.

```
delim.txind={field_name|$jms_property}
```

For example:

```
delim.txind=txind
```

11.2.3.17 delim.txind.beginval

The value that identifies an operation as the beginning of a transaction. The default is `B`.

```
delim.txind.beginval={value|\xhex_value}
```

For example:

```
delim.txind.beginval=0
```

11.2.3.18 delim.txind.middleval

The value that identifies an operation as the middle of a transaction. The default is `M`.

```
delim.txind.middleval={value|\xhex_value}
```

For example:

```
delim.txind.middleval=1
```

11.2.3.19 delim.txind.endval

The value that identifies an operation as the end of a transaction. The default is `E`.

```
delim.txind.endval={value|\xhex_value}
```

For example:

```
delim.txind.endval=2
```

11.2.3.20 delim.txind.wholeval

The value that identifies an operation as a whole transaction. The default is `W`.

```
delim.txind.wholeval={value|\xhex_value}
```

For example:

```
delim.txind.wholeval=3
```

11.2.3.21 delim.fielddelim

Specifies the delimiter value used to separate fields (columns) in the data. This value is defined through characters or hexadecimal values:

```
delim.fielddelim={value|\xhex_value}
```

For example:

```
delim.fielddelim=,
delim.fielddelim=\xc7
```

11.2.3.22 delim.linedelim

Specifies the delimiter value used to separate lines (records) in the data. This value is defined using characters or hexadecimal values.

```
delim.linedelim={value|\xhex_value}
```

For example:

```
delim.linedelim=||
delim.linedelim=\x0a
```

11.2.3.23 delim.quote

Specifies the value used to identify quoted data. This value is defined using characters or hexadecimal values.

```
delim.quote={value|\xhex_value}
```

For example:

```
delim.quote="
```

11.2.3.24 delim.nullindicator

Specifies the value used to identify `NULL` data. This value is defined using characters or hexadecimal values.

```
delim.nullindicator={value|\xhex_value}
```

For example:

```
delim.nullindicator=NULL
```

11.2.3.25 delim.fielddelim.escaped

Specifies the value that will replace the field delimiter when the field delimiter occurs in the input field. The syntax is:

```
delim.fielddelim.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.fielddelim=-
delim.fielddelim.escaped=$#$
```

If the data does not contain the hyphen delimiter within any of the field values:

```
one two three four
```

The resulting delimited data is:

```
one-two-three-four
```

If there are hyphen (-) delimiters within the field values:

```
one two three four-fifths two-fifths
```

The resulting delimited data is:

```
one-two-three-four$#$fifths-two$#$fifths
```

11.2.3.26 delim.linedelim.escaped

Specifies the value that will replace the line delimiter when the line delimiter occurs in the input data. The syntax is:

```
delim.linedelim.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.linedelim=\
delim.linedelim.escaped=%/%
```

If the input lines are:

```
These are the lines and they
do not contain the delimiter.
```

Because the lines do not contain the backslash (\) , the result is:

```
These are the lines and they\
do not contain the delimiter.\
```

However, if the input lines do contain the delimiter:

```
These are the lines\data values
and they do contain the delimiter.
```

So the results are:

```
These are the lines%/%data values\
and they do contain the delimiter.\
```

11.2.3.27 delim.quote.escaped

Specifies the value that will replace a quote delimiter when the quote delimiter occurs in the input data. The syntax is:

```
delim.quote.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.quote="
delim.quote.escaped="'"
```

If the input data does not contain the quote (") delimiter:

```
It was a very original play.
```

The result is:

```
"It was a very original play."
```

However, if the input data does contain the quote delimiter:

```
It was an "uber-original" play.
```

The result is:

```
"It was an "'"uber-original"'" play."
```

11.2.3.28 delim.nullindicator.escaped

Specifies the value that will replace a null indicator when a null indicator occurs in the input data. The syntax is:

```
delim.nullindicator.escaped={value|\xhex_value}
```

For example, given the following property settings:

```
delim.fielddelim=,
delim.nullindicator=NULL
delim.nullindicator.escaped=$NULL$
```

When the input data does not contain a `NULL` value or a `NULL` indicator:

```
1 2 3 4 5
```

The result is

```
1,2,3,4,5
```

When the input data contains a `NULL` value:

```
1 2 4 5
```

The result is

```
1,2,NULL,4,5
```

When the input data contains a `NULL` indicator:

```
1 2 NULL 4 5
```

The result is:

```
1,2,$NULL$,4,5
```

11.2.3.29 delim.hasbefores

Specifies whether before values are present in the data.

```
delim.hasbefores={true|false}
```

The default is `false`. The parser expects to find before and after values of columns for all records if `delim.hasbefores` is set to true. The before values are used for updates and deletes, the after values for updates and inserts. The `afterfirst` property specifies whether the before images are before the after images or after them. If `delim.hasbefores` is false, then no before values are expected.

For example:

```
delim.hasbefores=true
```

11.2.3.30 delim.hasnames

Specifies whether column names are present in the data.

```
delim.hasnames={true|false}
```

The default is false. If true, the parser expects to find column names for all records. The parser validates the column names against the expected column names. If false, no column names are expected.

For example:

```
delim.hasnames=true
```

11.2.3.31 delim.afterfirst

Specifies whether after values are positioned before or after the before values.

```
delim.afterfirst={true|false}
```

The default is false. If true, the parser expects to find the after values before the before values. If false, the after values are before the before values.

For example:

```
delim.afterfirst=true
```

11.2.3.32 delim.isgrouped

Specifies whether the column names and before and after images should be expected grouped together for all columns or interleaved for each column.

```
delim.isgrouped={true|false}
```

The default is false. If true, the parser expects find a group of column names (if `hasnames` is true), followed by a group of before values (if `hasbefores`), followed by a group of after values (the `afterfirst` setting will reverse the before and after value order). If false, the parser will expect to find a column name (if `hasnames`), before value (if `hasbefores`) and after value for each column.

For example:

```
delim.isgrouped=true
```

11.2.3.33 delim.dateformat | delim.dateformat.*table* | delim.dateform.table.column

Specifies the date format for column data. This is specified at a global level, table level or column level.The format used to parse the date is a subset of the formats used for `parser.timestamp.format`.

```
delim.dateformat=format
delim.dateformat.TABLE=format
delim.dateformat.TABLE.COLUMN=format
```

`Where:`

- *`format`* is the format defined for `parser.timestamp.format`.
- *`table`* is the fully qualified name of the table that is currently being processed.
- *`column`* is a column of the specified table.

For example:

```
delim.dateformat=YYYY-MM-DD HH:MI:SS
delim.dateformat.MY.TABLE=DD/MM/YY-HH.MI.SS
delim.dateformat.MY.TABLE.EXP_DATE=YYMM
```

11.2.4 XML Parser Properties

The following properties are used by the XML parser.

11.2.4.1 xml.sourcedefs

Specifies the location of the source definitions file.

```
xml.sourcedefs=file_location
```

For example:

```
xml.sourcedefs=dirdef/hrdemo.def
```

11.2.4.2 xml.rules

Specifies the list of XML rules for parsing a message and converting to transactions, operations and columns:

```
xml.rules=xml_rule_name[, . . .]
```

The specified XML rules are processed in the order listed. All rules matching a particular XML document may result in the creation of transactions, operations and columns. The specified XML rules should be transaction or operation type rules.

For example:

```
xml.rules=dyntxrule, statoprule
```

11.2.4.3 *rulename*.type

Specifies the type of XML rule.

```
rulename.type={tx|op|col}
```

Where:

- `tx` indicates a transaction rule
- `op` indicates an operation rule
- `col` indicates a column rule

For example:

```
dyntxrule.type=tx
statoprule.type=op
```

11.2.4.4 *rulename*.match

Specifies an XPath expression used to determine whether the rule is activated for a particular document or not.

```
rulename.match=xpath_expression
```

If the XPath expression returns any nodes from the document, the rule matches and further processing occurs. If it does not return any nodes, the rule is ignored for that document.

The following example activates the `dyntxrule` if the document has a root element of `Transaction`

```
dyntxrule.match=/Transaction
```

Where `statoprule` is a subrule of `stattxtule`, the following example activates the `statoprule` if the parent rule's matching nodes have child elements of `NewMyTableEntry.`

```
statoprule.match=./NewMyTableEntry
```

11.2.4.5 *rulename*.subrules

Specifies a list of rule names to check for matches if the parent rule is activated by its match.

rulename.subrules=*xml_rule_name*[, . . .]

The specified XML rules are processed in the order listed. All matching rules may result in the creation of transactions, operations and columns.

Valid sub-rules are determined by the parent type. Transaction rules can only have operation sub-rules. Operation rules can have operation or column sub-rules. Column rules cannot have sub-rules.

For example:

```
dyntxrule.subrules=dynoprule
statoprule.subrules=statkeycolrule, statcol1rule
```

11.2.4.6 *txrule*.timestamp

Controls the transaction timestamp by instructing the adapter to 1) use the transaction commit timestamp contained in a specified XPath expression or JMS property or 2) use the current system time. This is an optional property.

txrule.timestamp={*xpath_expression*|$*jms_property*|*ts}

The timestamp for the transaction may be overridden at the operation level, or may only be present at the operation level. Any XPath expression must end with a value accessor such as `@att` or `text()`.

For example:

```
dyntxrule.timestamp=@ts
```

11.2.4.7 *txrule*.timestamp.format

Specifies the format of the timestamp field.

txrule.timestamp.format=*format*

Where the format can include punctuation characters plus:

- `YYYY` – four digit year
- `YY` – two digit year
- `M[M]` – one or two digit month
- `D[D]` – one or two digit day
- `HH` – hours in twenty four hour notation
- `MI` – minutes
- `SS` – seconds
- `Fn` – n number of fractions

The default format is "`YYYY-MM-DD:HH:MI:SS.FFF`"

For example:

```
dyntxrule.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

11.2.4.8 *txrule*.seqid

Specifies the `seqid` for a particular transaction. This can be used when there are multiple transactions per message. Determines the XPath expression, JMS property, or system value that contains the transactions `seqid`. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
txrule.seqid={xpath_expression|$jms_property|*seqid}
```

For example:

```
dyntxrule.seqid=@seqid
```

11.2.4.9 *txrule*.txid

Specifies the XPath expression, JMS property, or system value that contains the `txid` used to unique identify transactions. This value must increment for each transaction.

```
txrule.txid={xpath_expression|$jms_property|*txid}
```

For most cases using the system value of `*txid` is preferred.

For example:

```
dyntxrule.txid=$JMSTxId
dyntxrule.txid=*txid
```

11.2.4.10 *txrule*.txowner

Specifies the XPath expression, JMS property, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing.

```
txrule.txowner={xpath_expression|$jms_property|"value"}
```

For example:

```
dyntxrule.txowner=$MessageOwner
dyntxrule.txowner="jsmith"
```

11.2.4.11 *txrule*.txname

Specifies the XPath expression, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
txrule.txname={xpath_expression|$jms_property|"value"}
```

For example:

```
dyntxrule.txname="fixedtx"
```

11.2.4.12 *oprule*.timestamp

Controls the operation timestamp by instructing the adapter to 1) use the transaction commit timestamp contained in a specified XPath expression or JMS property or 2) use the current system time. This is an optional property.

```
oprule.timestamp={xpath_expression|$jms_property|*ts}
```

The timestamp for the operation will override a timestamp at the transaction level.

Any XPath expression must end with a value accessor such as `@att` or `text()`.

For example:

```
statoprule.timestamp=./CreateTime/text()
```

11.2.4.13 *oprule*.timestamp.format

Specifies the format of the timestamp field.

```
oprule.timestamp.format=format
```

Where the *format* can include punctuation characters plus:

- YYYY – four digit year
- YY – two digit year
- M[M] – one or two digit month
- D[D] – one or two digit day
- HH – hours in twenty four hour notation
- MI – minutes
- SS – seconds
- Fn – n number of fractions

The default format is "YYYY-MM-DD:HH:MI:SS.FFF"

For example:

```
statoprule.timestamp.format=YYYY-MM-DD-HH.MI.SS
```

11.2.4.14 *oprule*.seqid

Specifies the seqid for a particular operation. Use the XPath expression, JMS property, or system value that contains the operation seqid. This overrides any seqid defined in parent transaction rules. Must be present if there is no parent transaction rule.

Any XPath expression must end with a value accessor such as @att or text().

```
oprule.seqid={xpath_expression|$jms_property|*seqid}
```

For example:

```
dynoprule.seqid=@seqid
```

11.2.4.15 *oprule*.txid

Specifies the XPath expression, JMS property, or system value that contains the txid used to uniquely identify transactions. This overrides any txid defined in parent transaction rules and is required if there is no parent transaction rule. The value must be incremented for each transaction.

```
oprule.txid={xpath_expression|$jms_property|*txid}
```

For most cases using the system value of *txid is preferred.

For example:

```
dynoprule.txid=$JMSTxId
dynoprule.txid=*txid
```

11.2.4.16 *oprule*.txowner

Specifies the XPath expression, JMS property, or static value that contains an arbitrary user name associated with a transaction. This value may be used to exclude certain transactions from processing. This is an optional property.

```
oprule.txowner={xpath_expression|$jms_property|"value"}
```

For example:

```
dynoprule.txowner=$MessageOwner
dynoprule.txowner="jsmith"
```

11.2.4.17 *oprule*.txname

Specifies the XPath expression, JMS property, or static value that contains an arbitrary name to be associated with a transaction. This is an optional property.

```
oprule.txname={xpath_expression|$jms_property|"value"}
```

For example:

```
dynoprule.txname="fixedtx"
```

11.2.4.18 *oprule*.schemandtable

Specifies the XPath expression JMS property or hard-coded value that contains the schema and table name in the form `SCHEMA.TABLE`. Any XPath expression must end with a value accessor such as `@att` or `text()`. The value is verified to ensure the table exists in the source definitions.

```
oprule.schemaandtable={xpath_expression|$jms_property|"value"}
```

For example:

```
statoprule.schemaandtable="MY.TABLE"
```

11.2.4.19 *oprule*.schema

Specifies the XPath expression, JMS property or hard-coded value that contains the schema name. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
oprule.schema={xpath_expression|$jms_property|"value"}
```

For example:

```
statoprule.schema=@schema
```

11.2.4.20 *oprule*.table

Specifies the XPath expression, JMS property or hard-coded value that contains the table name. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
oprule.table={xpath_expression|$jms_property|"value"}
```

For example:

```
statoprule.table=$TableName
```

11.2.4.21 *oprule*.optype

Specifies the XPath expression, JMS property or literal value that contains the `optype` to be validated against an optype `insertval`, etc. Any XPath expression must end with a value accessor such as `@att` or `text()`.

oprule.optype={*xpath_expression*|$*jms_property*|"*value*"}

For example:

```
dynoprule.optype=@type
statoprule.optype="I"
```

11.2.4.22 *oprule*.optype.insertval

Specifies the value that identifies an insert operation. The default is `I`.

oprule.optype.insertval={*value*|\x*hex_value*}

For example:

```
dynoprule.optype.insertval=A
```

11.2.4.23 *oprule*.optype.updateval

Specifies the value that identifies an update operation. The default is `U`.

oprule.optype.updateval={*value*|\x*hex_value*}

For example:

```
dynoprule.optype.updateval=M
```

11.2.4.24 *oprule*.optype.deleteval

Specifies the value that identifies a delete operation. The default is `D`.

oprule.optype.deleteval={*value*|\x*hex_value*}

For example:

```
dynoprule.optype.deleteval=R
```

11.2.4.25 *oprule*.txind

Specifies the XPath expression or JMS property that contains the transaction indicator to be validated against `beginval` or other value that identifies the position within the transaction. All operations within a single message are regarded as occurring within a whole transaction if this property is not defined. Specifies the begin, middle and end of transactions. Any XPath expression must end with a value accessor such as `@att` or `text()`. Transactions defined in this way can span messages. This is an optional property.

oprule.txind={*xpath_expression*|$*jms_property*}

For example:

```
dynoprule.txind=@txind
```

11.2.4.26 *oprule*.txind.beginval

Specifies the value that identifies an operation as the beginning of a transaction. The default is B.

```
oprule.txind.beginval={value|\xhex_value}
```

For example:

```
dynoprule.txind.beginval=0
```

11.2.4.27 *oprule*.txind.middleval

Specifies the value that identifies an operation as the middle of a transaction. The default is M.

```
oprule.txind.middleval={value|\xhex_value}
```

For example:

```
dynoprule.txind.middleval=1
```

11.2.4.28 *oprule*.txind.endval

Specifies the value that identifies an operation as the end of a transaction. The default is E.

```
oprule.txind.endval={value|\xhex_value}
```

For example:

```
dynoprule.txind.endval=2
```

11.2.4.29 *oprule*.txind.wholeval

Specifies the value that identifies an operation as a whole transaction. The default is W.

```
oprule.txind.wholeval={value|\xhex_value}
```

For example:

```
dynoprule.txind.wholeval=3
```

11.2.4.30 *colrule*.name

Specifies the XPath expression or hard-coded value that contains a column name. The column index must be specified if this is not and the column name will be resolved from that. If specified the column name will be verified against the source definitions file. Any XPath expression must end with a value accessor such as `@att` or `text()`.

```
colrule.name={xpath_expression|"value"}
```

For example:

```
dyncolrule.name=@name
statkeycolrule.name="keycol"
```

11.2.4.31 *colrule*.index

Specifies the XPath expression or hard-coded value that contains a column index. If not specified then the column name must be specified and the column index will be resolved from that. If specified the column index will be verified against the source

definitions file. Any XPath expression must end with a value accessor such as `@att` or `text()`.

colrule.index={*xpath_expression*|*"value"*}

For example:

```
dyncolrule.index=@index
statkeycolrule.index=1
```

11.2.4.32 *colrule*.value

Specifies the XPath expression or hard-coded value that contains a column value. Any XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression fails to return any data because a node or attribute does not exist, the column value will be deemed as null. To differentiate between null and missing values (for updates) the `isnull` and `ismissing` properties should be set. The value returned is used for delete before values, and update/insert after values.

colrule.value={*xpath_expression*|*"value"*}

For example:

```
statkeycolrule.value=./text()
```

11.2.4.33 *colrule*.isnull

Specifies the XPath expression used to discover if a column value is null. The XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression returns any value, the column value is null. This is an optional property.

colrule.isnull=*xpath_expression*

For example:

```
dyncolrule.isnull=@isnull
```

11.2.4.34 *colrule*.ismissing

Specifies the XPath expression used to discover if a column value is missing. The XPath expression must end with a value accessor such as `@att` or `text()`. If the XPath expression returns any value, then the column value is missing. This is an optional property.

colrule.ismissing=*xpath_expression*

For example:

```
dyncolrule.ismissing=./missing
```

11.2.4.35 *colrule*.before.value

Overrides *colrule*.value to specifically say how to obtain before values used for updates or deletes. This has the same format as *colrule*.value. This is an optional property.

For example:

```
dyncolrule.before.value=./beforevalue/text()
```

11.2.4.36 *colrule*.before.isnull

Overrides *colrule*.isnull to specifically say how to determine if a before value is null for updates or deletes. This has the same format as *colrule*.isnull. This is an optional property.

For example:

```
dyncolrule.before.isnull=./beforevalue/@isnull
```

11.2.4.37 *colrule*.before.ismissing

Overrides *colrule*.ismissing to specifically say how to determine if a before value is missing for updates or deletes. This has the same format as *colrule*.ismissing. This is an optional property.

For example:

```
dyncolrule.before.ismissing=./beforevalue/missing
```

11.2.4.38 *colrule*.after.value

Overrides *colrule*.value to specifically say how to obtain after values used for updates or deletes. This has the same format as *colrule*.value. This is an optional property.

For example:

```
dyncolrule.after.value=./aftervalue/text()
```

11.2.4.39 *colrule*.after.isnull

Overrides *colrule*.isnull to specifically say how to determine if an after value is null for updates or deletes. This has the same format as *colrule*.isnull. This is an optional property.

For example:

```
dyncolrule.after.isnull=./aftervalue/@isnull
```

11.2.4.40 *colrule*.after.ismissing

Overrides *colrule*.ismissing to specifically say how to determine if an after value is missing for updates or deletes. This has the same format as *colrule*.ismissing. This is an optional property.

For example:

```
dyncolrule.after.ismissing=./aftervalue/missing
```

Part IV

Delivering Java Messages

This part of the book contains information on using Oracle GoldenGate Adapters to process transaction information to create JMS messages for delivery to third party applications.

Part V contains the following chapters:

- Configuring Message Delivery
- Using the Java User Exit
- Configuring Event Handlers
- Message Delivery Properties
- Developing Custom Filters, Formatters, and Handlers

12

Configuring Message Delivery

This chapter explains how to configure the adapter for delivering messages. To do this, you must set up the properties in the user exit properties file, configure an Extract or Replicate process to run the user exit, and identify the built-in or custom event handlers you will use.

This chapter includes the following sections:

12.1 Configure the JRE in the User Exit Properties File

Modify the user exit properties file to point to the location of the Oracle GoldenGate for Java main jar (ggjava.jar) and set any additional JVM runtime boot options as required (these are passed directly to the JVM at startup):

```
jvm.bootoptions=-Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=log4j.properties -Xmx512m
```

Note the following options in particular:

- java.class.path can include any custom jars in addition to the core application (`ggjava.jar`). The current directory (.) is included by default in the class path. You can reference files relative to the Oracle GoldenGate install directory, to allow storing Java property files, Velocity templates and other class path resources in the `dirprm` subdirectory. It is also possible to append to the class path in the Java application properties file.
- The log4j.configuration option specifies a log4j properties file, found in the class path. There are pre-configured default log4j settings for basic logging (`log4j.properties`), debug logging (`debug-log4j.properties`), and detailed trace-level logging (`trace-log4j.properties`), found in the resources/classes directory.

Once the user exit properties file is correctly configured for your system, it usually remains unchanged. See User Exit Properties, for additional configuration options.

12.2 Configure Extract to Run the User Exit

The user exit Extract is configured as a data pump. The data pump consumes a local trail (for example `dirdat/aa`) and sends the data to the user exit. The user exit is responsible for processing all the data.

Following is an example of adding a data pump Extract:

```
ADD EXTRACT javaue, EXTTRAILSOURCE ./dirdat/aa
```

The process names and trail names used above can be replaced with any valid name. Process names must be 8 characters or less, trail names must to be two characters. In the user exit Extract parameter file (`javaue.prm`) specify the location of the user exit library.

Table 12-1 User Exit Extract Parameters

Parameter	Explanation
`EXTRACT javaue`	All Extract parameter files start with the Extract name
`SOURCEDEFS ./dirdef/tcust.def`	The Extract process requires metadata describing the trail data. This can come from a database or a source definitions file. This metadata defines the column names and data types in the trail being read (`./dirdat/aa`).
`SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")`	(Optional) An absolute or relative path (relative to the Extract executable) to the properties file for the C user exit library. The default value is *`extract_name`*`.properties` in the `dirprm` directory.
`SETENV (GGS_JAVAUSEREXIT_CONF = "dirprm/javaue.properties")`	(Optional) The Java properties.This example places the properties file in the `dirpm` directory.
`CUSEREXIT ggjava_ue.dll CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES`	The `CUSEREXIT` parameter includes the following: • The location of the user exit library. For UNIX, the library would be suffixed `.so` • `CUSEREXIT` - the callback function name that must be uppercase. • `PASSTHRU` - avoids the need for a dummy target trail. • `INCLUDEUPDATEBEFORES` - needed for transaction integrity.
`TABLE schema.*;`	The tables to pass to the User Exit; tables not included will be skipped. *No filtering* may be done in the user exit Extract; otherwise transaction markers will be missed. You can filter in the primary Extract, or use another, upstream data pump, or filter data directly in the Java application.

Note:

Using `PASSTHRU` disables the statistical reporting that allows report counts to be included in the processing report. To collect report count statistics and send them to the Extract from the user exit, use the property `gg.report.time`.

The two environment properties shown above are optional.

- `SETENV (GGS_USEREXIT_CONF = "dirprm/javaue.properties")`

This changes the default configuration file used for the User Exit shared library. The value given is either an absolute path, or a path relative to Extract (or Replicat). The example above uses a relative path to put this property file in the `dirprm` directory.

The default file used is *`extract_name`*`.properties`, located in the `dirprm` directory. So, if the Extract name is `pumpA`, then the `prm` file is `dirprm/pumpA.prm` and the properties file is `dirprm/pumpA.properties`.

- `SETENV (GGS_JAVAUSEREXIT_CONF = "dirprm/javaue.properties")`

  This changes the default properties file used for the Oracle GoldenGate for Java framework. The value found is a path to a file found in the class path or in a normal file system path.

  Both `GGS_USEREXIT_CONF` and `GGS_JAVAUSEREXIT_CONF` default to the same file; `dirprm/`*`extract_name`*`.properties`.

12.3 Configure the Java Handlers

The Oracle GoldenGate Java API has a property file used to configure active event handlers. To test the configuration, you may use the built-in file handler. Here are some example properties, followed by explanations of the properties (comment lines start with #):

```
# the list of active handlers
gg.handlerlist=myhandler
# set properties on 'myhandler'
gg.handler.myhandler.type=file
gg.handler.myhandler.format=tx2xml.vm
gg.handler.myhandler.file=output.xml
```

This property file declares the following:

- Active event handlers. In the example a single event handler is active, called `myhandler`. Multiple handlers may be specified, separated by commas. For example: `gg.handlerlist=myhandler, yourhandler`
- Configuration of the handlers. In the example `myhandler` is declared to be a `file` type of handler: `gg.handler.myhandler.type=file`

  Note:

  See the documentation for each type of handler (e.g. the JMS handler or the file writer handler) for the list of valid properties that can be set.

- The format of the output is defined by the Velocity template `tx2xml.vm`. You may specify your own custom template to define the message format; just specify the path to your template relative to the Java class path (this is discussed later).

This property file is actually a complete example that will write captured transactions to the output file output.xml. Other handler types can be specified using the key words: `jms_text` (or `jms`), `jms_map`, `singlefile` (a file that does not roll), and others. Custom handlers can be implemented, in which case the type would be the fully qualified name of the Java class for the handler.

13

Using the Java User Exit

This chapter describes how to start and restart the Oracle GoldenGate Adapter user exit that delivers messages. It assumes that the primary Extract has already generated a trail to be consume the user exit Replicat.

This chapter includes the following sections:

13.1 Starting the Application

Oracle GoldenGate release 12.2 introduced metadata in trail, which can be used by either the Extract or the Replicat process as the metadata definitions. Metadata in trail makes the separate source definitions file unnecessary as in previous releases. For the examples that follow, a simple `TCUSTMER` and `TCUSTORD` trail is used (matching the demo SQL provided with the Oracle GoldenGate software download), along with a source definitions file defining the data types used in the trail.

Note:

Replicat user exit must have access to the source definitions in order to run. Source metadata must be provided either from a static source definitions file or by using the metadata in trail feature where the metadata can be obtained from the trail. However, the Extract process does require metadata describing the trail data. Either the Extract must login to a database for metadata, or a source definitions file can be provided. In either case, the Extract cannot be in `PASSTHRU` mode when using a user exit.

To run the user exit, simply start the Replicat process from GGSCI:

```
GGSCI> START REPLICAT javaue
GGSCI> INFO REPLICAT javaue
```

The `INFO` command returnS information similar to the following:

```
REPLICAT JAVAUE Last Started 2011-08-25 18:41 Status RUNNING
Checkpoint Lag 00:00:00 (updated 00:00:00 ago)
Log Read Checkpoint File ./dirdat/bb000000
2011-09-24 12:52:58.000000 RBA 2702
```

If the Replicat process is running and the file handler is being used (as in the example above), then you should see the output file `output.xml` in the Oracle GoldenGate installation directory (the same directory as the Replicat executable).

If the process does not start or abends, see Checking for Errors.

13.2 Restarting the Application at the Beginning of a Trail

There are two checkpoints for an Replicat running the user exit: the user exit checkpoint and the Replicat checkpoint. Before rerunning the Replicat, you must reset both checkpoints:

1. Delete the user exit checkpoint file.

   In this example, the name of the Replicat group is `javaue`, so this will default to the checkpoint prefix.

   Windows: `cmd>d el JAVAUE.cpj`

   UNIX: `$ rm JAVAUE.cpj`

   Note:

   Do not modify checkpoints or delete the user exit checkpoint file on a production system.

2. Reset the Replicat to the beginning of the trail data:

   ```
   GGSCI> ALTER REPLICAT JAVAUE, EXTSEQNO 0, EXTRBA 0
   ```

3. Restart the Replicat:

   ```
   GGSCI> START JAVAUE
   GGSCI> INFO JAVAUE
   REPLICAT    JAVAUE    Last Started 2011-08-25 18:41   Status RUNNING
   Checkpoint Lag       00:00:00 (updated 00:00:00 ago)
   Log Read Checkpoint  File ./dirdat/ps000000
                        2011-09-24 12:52:58.000000  RBA 2702
   ```

   It may take a few seconds for the Replicat process status to report itself as running. Check the report file to see if it abended or is still in the process of starting:

   ```
   GGSCI> VIEW REPORT JAVAUE
   ```

14

Configuring Event Handlers

This chapter discusses types of event handlers explaining how to specify the event handler to use and what your options are. It explains how to format the output and what you can expect from the Oracle GoldenGate Report file.

This chapter includes the following sections:

14.1 Specifying Event Handlers

Processing transaction, operation and metadata events in Java works as follows:

- The Oracle GoldenGate Replicat reads local trail data and passes the transactions, operations and database metadata to the user exit. Metadata can come from a source definitions file or a metadata definition included in the trail.
- Events are fired by the Java framework, optionally filtered by custom Event Filters.
- Handlers (event listeners) process these events, and process the transactions, operations and metadata. Custom formatters may be applied for certain types of targets.

There are several existing handlers:

- A message handler to send to a JMS provider using either a MapMessage, or using a TextMessage with customized formatters.
- A specialized message handler to send JMS messages to Oracle Advanced Queuing (AQ).
- A file writer handler, for writing to a single file, or a rolling file.

Note:

The file writer handler is particularly useful as development utility, since the file writer can take the exact same formatter as the JMS TextMessage handler. Using the file writer provides a simple way to test and tune the formatters for JMS without actually sending the messages to JMS.

Event handlers can be configured using the main Java property file or they may optionally read in their own properties directly from yet another property file (depending on the handler implementation). Handler properties are set using the following syntax:

```
gg.handler.{name}.someproperty=somevalue
```

This will cause the property *someproperty* to be set to the value *somevalue* for the handler instance identified in the property file by *name*. This *name* is used in the property file to define active handlers and set their properties; it is user-defined.

Implementation note (for Java developers): Following the above example: when the handler is instantiated, the method `void` setSomeProperty(String value) will be called on the handler instance, passing in the String value *somevalue*. A JavaBean `PropertyEditor` may also be defined for the handler, in which case the string can be automatically converted to the appropriate type for the setter method. For example, in the Java application properties file, we may have the following:

```
# the list of active handlers: only two are active
gg.handlerlist=one, two
# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=com.mycompany.MyFormatter
gg.handler.one.file=output.xml
# properties for handler 'two'
gg.handler.two.type=jms_text
gg.handler.two.format=com.mycompany.MyFormatter
gg.handler.two.properties=jboss.properties
# set properties for handler 'foo'; this handler is ignored
gg.handler.foo.type=com.mycompany.MyHandler
gg.handler.foo.someproperty=somevalue
```

The type identifies the handler class; the other properties depend on the type of handler created. If a separate properties file is used to initialize the handler (such as the JMS handlers), the properties file is found in the class path. For example, if properties file is at: {*gg_install_dir*}`/dirprm/foo.properties`, then specify in the properties file as follows:
`gg.handler.`*name*`.properties=foo.properties`.

14.2 JMS Handler

The main Java property file identifies active handlers. The JMS handler may optionally use a separate property file for JMS-specific configuration. This allows more than one JMS handler to be configured to run at the same time.

There are examples included for several JMS providers (JBoss, TIBCO, Solace, ActiveMQ, WebLogic). For a specific JMS provider, you can choose the appropriate properties files as a starting point for your environment. Each JMS provider has slightly different settings, and your environment will have unique settings as well.

The installation directory for the Java jars (`ggjava`) contains the core application jars (`ggjava.jar`) and its dependencies in `resources/lib/*.jar`. The resources directory contains all dependencies and configuration, and is in the class path.

If the JMS client jars already exist somewhere on the system, they can be referenced directly and added to the class path without copying them.

The following types of JMS handlers can be specified:

- **jms** – sends text messages to a topic or queue. The messages may be formatted using Velocity templates or by writing a formatter in Java. The same formatters can be used for a `jms_text message` as for writing to files. (`jms_text` is a synonym for `jms`.)
- **aq** – sends text messages to Oracle Advanced Queuing (AQ). The `aq` handler is a `jms` handler configured for delivery to AQ. The messages can be formatted using Velocity temlates or a custom formatter.
- **jms_map** – sends a JMS MapMessage to a topic or queue. The `JMSType` of the message is set to the name of the table. The body of the message consists of the following metadata, followed by column name and column value pairs:
  - `GG_ID` – position of the record, uniquely identifies this operation
  - `GG_OPTYPE` – type of SQL (insert/update/delete),
  - `GG_TABLE` – table name on which the operation occurred
  - `GG_TIMESTAMP` – timestamp of the operation

14.3 File Handler

The file handler is often used to verify the message format when the actual target is JMS, and the message format is being developed using custom Java or Velocity templates. Here is a property file using a file handler:

```
# one file handler active, using Velocity template formatting
gg.handlerlist=myfile
gg.handler.myfile.type=file
gg.handler.myfile.rollover.size=5M
gg.handler.myfile.format=sample2xml.vm
gg.handler.myfile.file=output.xml
```

This example uses a single handler (though, a JMS handler and the file handler could be used at the same time), writing to a file called `output.xml`, using a Velocity template called `sample2xml.vm`. The template is found via the classpath.

14.4 Custom Handlers

For information on coding a custom handler, see Coding a Custom Handler in Java .

14.5 Formatting the Output

As previously described, the existing JMS and file output handlers can be configured through the properties file. Each handler has its own specific properties that can be set: for example, the output file can be set for the file handler, and the JMS destination can be set for the JMS handler. Both of these handlers may also specify a custom formatter. The same formatter may be used for both handlers. As an alternative to writing Java

code for custom formatting, a Velocity template may be specified. For further information, see Filtering Events .

14.6 Reporting

Summary statistics about the throughput and amount of data processed are generated when the Replicat process stops. Additionally, statistics can be written periodically either after a specified amount of time or after a specified number of records have been processed. If both time and number of records are specified, then the report is generated for whichever event happens first. These statistical summaries are written to the Oracle GoldenGate report file and the user exit log files.

15

Message Delivery Properties

This chapter explains the options available for configuration of the property files for user exit properties and Java application properties.

This chapter includes the following sections:

15.1 User Exit Properties

The following properties set the log files and the type of logging.

15.1.1 Logging Properties

Logging is controlled by the following properties.

15.1.1.1 gg.log

Specifies the type of logging that is to be used. The default implementation for the Oracle GoldenGate Adapters is the `JDK` option. This is the built-in Java logging called `java.util.logging` (`JUL`). The other logging options are `log4j` or `logback`.

For example, to set the type of logging to `log4j`:

```
gg.log=log4j
```

The log file is created in the report subdirectory of the installation. The default log file name includes the group name of the associated Extract and the file extension is `.log`.

15.1.1.2 gg.log.level

Specifies the overall log level for all modules. The syntax is:

```
gg.log.level={ERROR|WARN|INFO|DEBUG}
```

The log levels are defined as follows:

- `ERROR` – Only write messages if errors occur
- `WARN` – Write error and warning messages
- `INFO` – Write error, warning and informational messages
- `DEBUG` – Write all messages, including debug ones.

The default logging level is `INFO`. The messages in this case will be produced on startup, shutdown and periodically during operation. If the level is switched to `DEBUG`, large volumes of messages may occur which could impact performance. For example, the following sets the global logging level to `INFO`:

```
# global logging level
gg.log.level=INFO
```

15.1.1.3 gg.log.file

Specifies the path to the log file. The syntax is:

```
gg.log.file=path_to_file
```

Where the *path_to_file* is the fully defined location of the log file. This allows a change to the name of the log, but you must include the Extract name if you have more than one Extract to avoid one overwriting the log of the other.

15.1.1.4 gg.log.classpath

Specifies the class path to the jars used to implement logging.

```
gg.log.classpath=path_to_jars
```

15.1.2 General Properties

The following properties apply to all writer type user exits and are not specific to the user exit.

15.1.2.1 goldengate.userexit.writers

Specifies the name of the writer. This is always `jvm` and should not be modified.

For example:

```
goldengate.userexit.writers=jvm
```

All other properties in the file should be prefixed by the writer name, *jvm*.

15.1.2.2 goldengate.userexit.chkptprefix

Specifies a string value for the prefix added to the checkpoint file name. For example:

```
goldengate.userexit.chkptprefix=javaue_
```

15.1.2.3 goldengate.userexit.nochkpt

Disables or enables the user exit checkpoint file. The default is `false`, the checkpoint file is enabled. Set this property to `true` if transactions are supported and enabled on the target.

For exampe, if JMS is the target and JMS local transactions are enabled (the default), set `goldengate.userexit.nochkpt=true` to disable the user exit checkpoint file. If JMS transactions are disabled by setting `localTx=false` on the handler, the user exit checkpoint file should be enabled by setting `goldengate.userexit.nochkpt=false`.

```
goldengate.userexit.nochkpt={true|false}
```

15.1.2.4 goldengate.userexit.usetargetcols

Specifies whether or not mapping to target columns is allowed. The default is `false`, no target mapping.

```
goldengate.userexit.usetargetcols={true|false}
```

15.1.3 JVM boot Options

The following options configure the Java Runtime Environment. In particular, this is where the maximum memory the JVM can use is specified; if you see Java out-of-memory errors, edit these settings.

15.1.3.1 jvm.bootoptions

Specifies the classpath and boot options that will be applied when the user exit starts up the JVM. The path needs colon (:) separators for UNIX/Linux and semicolons (;) for Windows. This is where to specify various options for the JVM, such as heap size and class path; for example:

- **-Xms**: initial java heap size
- **-Xmx**: maximum java heap size
- `-`**Djava.class.path**: class path specifying location of at least the main application jar, `ggjava.jar`. Other jars, such as JMS provider jars, may also be specified here as well; alternatively, these may be specified in the Java application properties file.
- **-verbose:jni**: run in verbose mode (for JNI)

For example (all on a single line):

```
jvm.bootoptions= -Djava.class.path=ggjava/ggjava.jar
-Dlog4j.configuration=my-log4j.properties -Xmx512m
```

The `log4j.configuration` property can be a fully qualified URL to a log4j properties file; by default this file is searched for in the class path. You may use your own log4j configuration, or one of the preconfigured log4j settings: `log4j.properties` (default level of logging), `debug-log4j.properties` (debug logging) or `trace-log4j.properties` (very verbose logging).

15.1.4 Statistics and Reporting

The use of the user exit causes Extract to assume that the records handled by the exit are ignored. This causes the standard Oracle GoldenGate reporting to be incomplete. Oracle GoldenGate for Java adds its own reporting to handle this issue.

Statistics can be reported every `t` seconds or every `n` records - or if both are specified, whichever criteria is met first.

There are two sets of statistics recorded: those maintained by the User Exit shared library (on the C side) and those obtained from the Java libraries. The reports received from the Java side are formatted and returned by the individual handlers.

The User Exit statistics include the total number of operations, transactions and corresponding rates.

15.1.4.1 *jvm*.stats.display

Controls the output of statistics to the Oracle GoldenGate report file and to the user exit log files.

The following example outputs these statistics.

```
jvm.stats.display=true
```

15.1.4.2 *jvm*.stats.full

Controls the output of statistics from the Java side, in addition to the statistics from the C side.

Java side statistics are more detailed but also involve some additional overhead, so if statistics are reported often and a less detailed summary is adequate, it is recommended that `stats.full` property is set to `false`.

The following example will output Java statistics in addition to C.

```
jvm.stats.full=true
```

15.1.4.3 *jvm*.stats.time | *jvm*.stats.numrecs

Specifies a time interval, in seconds or a number of records, after which statistics will be reported. The default is to report statistics every hour or every 10000 records (which ever occurs first).

For example, to report ever 10 minutes or every 1000 records, specify:

```
jvm.stats.time=600
jvm.stats.numrecs=1000
```

The Java application statistics are handler-dependent:

- For the all handlers, there is at least the total elapsed time, processing time, number of operations, transactions;
- For the JMS handler, there is additionally the total number of bytes received and sent.
- The report can be customized using a template.

15.2 Java Application Properties

The following defines the properties which may be set in the Java application property file.

15.2.1 Properties for All Handlers

The following properties apply to all handlers.

15.2.1.1 gg.handlerlist

The handler list is a list of active handlers separated by commas. These values are used in the rest of the property file to configure the individual handlers. For example:

```
gg.handlerlist=name1, name2
gg.handler.name1.propertyA=value1
gg.handler.name1.propertyB=value2
gg.handler.name1.propertyC=value3
gg.handler.name2.propertyA=value1
gg.handler.name2.propertyB=value2
gg.handler.name2.propertyC=value3
```

Using the `handlerlist` property, you may include completely configured handlers in the property file and just disable them by removing them from the `handlerlist`.

15.2.1.2 gg.handler.*name*.type

This type of handler. This is either a predefined value for built-in handlers, or a fully qualified Java class name. The syntax is:

```
gg.handler.name.type={jms|jms_map|aq|singlefile|rollingfile|custom_java_class}
```

Where:

All but the last are pre-defined handlers:

- **jms** – Sends transactions, operations, and metadata as formatted messages to a JMS provider
- **aq** – Sends transactions, operations, and metadata as formatted messages to Oracle Advanced Queuing (AQ)
- **jms_map** – Sends JMS map messages
- **singlefile** – Writes to a single file on disk, but does not roll the file
- **rollingfile** – Writes transactions, operations, and metadata to a file on disk, rolling the file over after a certain size or after a certain amount of time
- *custom_java_class* – Any class that extends the Oracle GoldenGate for Java `AbstractHandler` class and can handle transaction, operation, or metadata events

15.2.2 Properties for Formatted Output

The following properties apply to all handlers capable of producing formatted output; this includes:

- The `jms_text handler` (but not the `jms_map handler`)
- The `aq` handler
- The `singlefile` and `rolling` handlers, for writing formatted output to files

15.2.2.1 gg.handler.*name*.format

Specifies the format used to transform operations and transactions into messages sent to JMS, to the Big Data target or to a file. The format is specified uniquely for each handler. The value may be:

- **Velocity template**
- **Java class name** (fully qualified - the class specified must be a type of formatter)
- **csv** for delimited values (such as comma separated values; the delimiter can be customized)
- **fixed** for fixed-length fields
- **Built-in formatter**, such as:
  - `xml` – demo XML format
  - `xml2` – internal XML format

For example, to specify a custom Java class:

```
gg.handlerlist=abc
gg.handler.abc.format=com.mycompany.MyFormat
```

Or, for a Velocity template:

```
gg.handlerlist=xyz
gg.handler.xyz.format=path/to/sample.vm
```

If using templates, the file is found relative to some directory or JAR that is in the classpath. By default, the Oracle GoldenGate installation directory is in the classpath, so the preceding template could be placed in the `dirprm` directory of the Oracle GoldenGate installation location.

The default format is to use the built-in XML formatter.

15.2.2.2 gg.handler.*name*.includeTables

Specifies a list of tables this handler will include. If the schema (or owner) of the table is specified, then only that schema matches the table name; otherwise, the table name matches any schema. A comma separated list of tables can be specified.

For example, to have the handler only process tables `foo.customer` and `bar.orders`:

```
gg.handler.myhandler.includeTables=foo.customer, bar.orders
```

Note:

In order to selectively process operations on a table by table basis, the handler must be processing in operation mode. If the handler is processing in transaction mode, then when a single transaction contains several operations spanning several tables, if any table matches the include list of tables, the transaction will be included.

15.2.2.3 gg.handler.*name*.excludeTables

Specifies a list of tables this handler will exclude. If the schema (or owner) of the table is specified, then only that schema matches the table name; otherwise, the table name matches any schema. A list of tables may be specified, comma-separated. For example, to have the handler process all operations on all tables except table `date_modified` in all schemas:

```
gg.handler.myhandler.excludeTables=date_modified
```

15.2.2.4 gg.handler.*name*.mode, gg.handler.*name*.format.mode

Specifies whether to output one operation per message (`op`) or one transaction per message (`tx`). The default is `op`. Use `gg.handler.name.format.mode` when you have a custom formatter.

Note:

This property must be set to one transaction per message (`tx`) if you are using group transaction properties. If it is set to one operation per message (`op`), `gg.handler.`*`name`*`.minGroupSize` and `gg.handler.`*`name`*`.maxGroupSize` will be ignored

15.2.3 Properties for CSV and Fixed Format Output

If the handler is set to use either `CSV` or `fixed` format output, the following properties may also be set.

15.2.3.1 gg.handler.*name*.format.delim

Specifies the delimiter to use between fields. Set this to no value to have no delimiter used. For example:

```
gg.handler.handler1.format.delim=,
```

15.2.3.2 gg.handler.*name*.format.quote

Specifies the quote character to be used if column values are quoted. For example:

```
gg.handler.handler1.format.quote='
```

15.2.3.3 gg.handler.*name*.format.metacols

Specifies the metadata column values to appear at the beginning of the record, before any column data. Specify any of the following, in the order they should appear:

- **position** – unique position indicator of records in a trail
- **opcode** – `I`, `U`, or `D` for insert, update, or delete records (see: `insertChar`, `updateChar`, `deleteChar`)
- **txind** – transaction indicator – such as `0`=begin, `1`=middle, `2`=end, `3`=whole tx (see `beginTxChar`, `middleTxChar`, `endTxChar`, `wholeTxChar`)
- **opcount** – position of a record in a transaction, starting from 0
- **schema** – schema/owner of the table for the record
- **tableonly** – just table (no schema/owner)
- **table** – full name of table, schema.table
- **timestamp** – commit timestamp of record

For example:

```
gg.handler.handler1.format.metacols=opcode, table, txind, position
```

15.2.3.4 gg.handler.*name*.format.missingColumnChar

Specifies a special column prefix for a column value that was not captured from the source database transaction log. The column value is not in trail and it is unknown if it has a value or is `NULL`

The character used to represent the missing state of the column value can be customized. For example:

```
gg.handler.handler1.format.missingColumnChar=M
```

By default, the missing column value is set to an empty string and does not show.

15.2.3.5 gg.handler.*name*.format.presentColumnChar

Specifies a special column prefix for a column value that exists in the trail and is not `NULL`.

The character used to represent the state of the column can be customized. For example:

```
gg.handler.handler1.format.presentColumnChar=P
```

By default, the present column value is set to an empty string and does not show.

15.2.3.6 gg.handler.*name*.format.nullColumnChar

Specifies a special column prefix for a column value that exists in the trail and is set to `NULL`.

The character used to represent the state of the column can be customized. For example:

```
gg.handler.handler1.format.nullColumnChar=N
```

By default, the null column value is set to an empty string and does not show.

15.2.3.7 gg.handler.*name*.format.beginTxChar

Specifies the header metadata character (see `metacols`) used to identify a record as the `begin` of a transaction. For example:

```
gg.handler.handler1.format.beginTxChar=B
```

15.2.3.8 gg.handler.*name*.format.middleTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as the `middle` of a transaction. For example:

```
gg.handler.handler1.format.middleTxChar=M
```

15.2.3.9 gg.handler.*name*.format.endTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as the `end` of a transaction. For example:

```
gg.handler.handler1.format.endTxChar=E
```

15.2.3.10 gg.handler.*name*.format.wholeTxChar

Specifies the header metadata characters (see `metacols`) used to identify a record as a complete transaction; referred to as a `whole` transaction. For example:

```
gg.handler.handler1.format.wholeTxChar=W
```

15.2.3.11 gg.handler.*name*.format.insertChar

Specifies the character to identify an insert operation. The default `I`.

For example, to use `INS` instead of `I` for insert operations:

```
gg.handler.handler1.format.insertChar=INS
```

15.2.3.12 gg.handler.*name*.format.updateChar

Specifies the character to identify an update operation. The default is `U`.

For example, to use `UPD` instead of `U` for update operations:

```
gg.handler.handler1.format.updateChar=UPD
```

15.2.3.13 gg.handler.*name*.format.deleteChar

Specifies the character to identify a delete operation. The default is `D`.

For example, to use `DEL` instead of `D` for delete operations:

```
gg.handler.handler1.format.deleteChar=DEL
```

15.2.3.14 gg.handler.*name*.format.endOfLine

Specifies the end-of-line character as:

- `EOL` - Native platform
- `CR` - Neutral (UNIX-style `\n`)
- `CRLF` - Windows (`\r\n`)

For example:

```
gg.handler.handler1.format.endOfLine=CR
```

15.2.3.15 gg.handler.*name*.format.justify

Specifies the left or right justification of fixed fields. For example:

```
gg.handler.handler1.format.justify=left
```

15.2.3.16 gg.handler.*name*.format.includeBefores

Controls whether before images should be included in the output. There must be before images in the trail. For example:

```
gg.handler.handler1.format.includeBefores=false
```

15.2.4 File Writer Properties

The following properties only apply to handlers that write their output to files: the `file` handler and the `singlefile` handler.

15.2.4.1 gg.handler.*name*.file

Specifies the name of the output file for the given handler. If the handler is a rolling file, this name is used to derive the rolled file names. The default file name is `output.xml`.

15.2.4.2 gg.handler.*name*.append

Controls whether the file should be appended to (`true`) or overwritten upon restart (`false`).

15.2.4.3 gg.handler.*name*.rolloverSize

If using the file handler, this specifies the size of the file before a rollover should be attempted. The file size will be at least this size, but will most likely be larger.

Operations and transactions are not broken across files. The size is specified in bytes, but a suffix may be given to identify MB or KB. For example:

```
gg.handler.myfile.rolloverSize=5MB
```

The default rollover size is `10 MB`.

15.2.5 JMS Handler Properties

The following properties apply to the JMS handlers. Some of these values may be defined in the Java application properties file using the name of the handler. Other properties may be placed into a separate JMS properties file, which is useful if using more than one JMS handler at a time. For example:

```
gg.handler.myjms.type=jms_text
gg.handler.myjms.format=xml
gg.handler.myjms.properties=weblogic.properties
```

Just as with Velocity templates and formatting property files, this additional JMS properties file is found in the classpath. The above properties file `weblogic.properties` would be found in {*gg_install_dir*}`/dirprm/weblogic.properties`, since the `dirprm` directory is included by default in the class path.

Settings that can be made in the Java application properties file will override the corresponding value set in the supplemental JMS properties file (`weblogic.properties` in the example above). In the following example, the destination property is specified in the Java application properties file. This allows the same default connection information for the two handlers `myjms1` and `myjms2`, but customizes the target destination queue.

```
gg.handlerlist=myjms1,myjms2
gg.handler.myjms1.type=jms_text
gg.handler.myjms1.destination=queue.sampleA
gg.handler.myjms1.format=sample.vm
gg.handler.myjms1.properties=tibco-default.properties
gg.handler.myjms2.type=jms_map
gg.handler.myjms2.destination=queue.sampleB
gg.handler.myjms2.properties=tibco-default.properties
```

To set a property, specify the handler name as a prefix; for example:

```
gg.handlerlist=sample,sample2
gg.handler.sample.type=jms_text
gg.handler.sample.format=my_template.vm
gg.handler.sample.destination=gg.myqueue
gg.handler.sample.queueortopic=queue
gg.handler.sample.connectionUrl=tcp://host:61616?jms.useAsyncSend=true
gg.handler.sample.useJndi=false
gg.handler.sample.connectionFactory=ConnectionFactory
gg.handler.sample.connectionFactoryClass=\

    org.apache.activemq.ActiveMQConnectionFactory

gg.handler.sample.connectionUrl=\
    tcp://localhost:61616?jms.useAsyncSend=true
gg.handler.sample.timeToLive=50000
```

15.2.5.1 Standard JMS Settings

The following outlines the JMS properties which may be set, and the accepted values. These apply for both JMS handler types: `jms_text` (`TextMessage`) and `jms_map` (`MapMessage`).

15.2.5.1.1 gg.handler.*name*.destination

The queue or topic to which the message is sent. This must be correctly configured on the JMS server. Typical values may be: `queue/A`, `queue.Test`, `example.MyTopic`, etc.

```
gg.handler.name.destination=queue_or_topic
```

15.2.5.1.2 gg.handler.*name*.user

(Optional) User name required to send messages to the JMS server.

```
gg.handler.name.user=user_name
```

15.2.5.1.3 gg.handler.*name*.password

(Optional) Password required to send messages to the JMS server

```
gg.handler.name.password=password
```

15.2.5.1.4 gg.handler.*name*.queueOrTopic

Whether the handler is sending to a queue (a single receiver) or a topic (publish / subscribe). This must be correctly configured in the JMS provider. This property is an alias of `gg.handler.name.destination`. The syntax is:

```
gg.handler.name.queueOrTopic={queue|topic}
```

Where:

- `queue` – a message is removed from the queue once it has been read. This is the default.
- `topic` – messages are published and may be delivered to multiple subscribers.

15.2.5.1.5 gg.handler.*name*.persistent

If the delivery mode is set to persistent or not. If the messages are to be persistent, the JMS provider must be configured to log the message to stable storage as part of the client's send operation. The syntax is:

```
gg.handler.name.persistent={true|false}
```

15.2.5.1.6 gg.handler.*name*.priority

JMS defines a 10 level priority value, with 0 as the lowest and 9 as the highest. Priority is set to 4 by default. The syntax is:

```
gg.handler.name.priority=integer
```

For example:

```
gg.handler.name.priority=5
```

15.2.5.1.7 gg.handler.*name*.timeToLive

The length of time in milliseconds from its dispatch time that a produced message should be retained by the message system. A value of zero specifies the time is unlimited. The default is zero. The syntax is:

```
gg.handler.name.timeToLive=milliseconds
```

For example:

```
gg.handler.name.timeToLive= 36000
```

15.2.5.1.8 gg.handler.*name*.connectionFactory

Name of the connection factory to lookup via JNDI. `ConnectionFactoryJNDIName` is an alias. The syntax is:

```
gg.handler.name.connectionFactory=JNDI_name
```

15.2.5.1.9 gg.handler.*name*.useJndi

If `gg.handler.name.usejndi` is `false`, then JNDI is not used to configure the JMS client. Instead, factories and connections are explicitly constructed. The syntax is:

```
gg.handler.name.useJndi={true|false}
```

15.2.5.1.10 gg.handler.*name*.connectionUrl

Connection URL is used only when not using JNDI to explicitly create the connection. The syntax is:

```
gg.handler.name.connectionUrl=url
```

15.2.5.1.11 gg.handler.*name*.connectionFactoryClass

The Connection Factory Class is used to access a factory only when not using JNDI. The value of this property is the Java class name to instantiate; constructing a factory object explicitly.

```
gg.handler.name.connectionFactoryClass=java_class_name
```

15.2.5.1.12 gg.handler.*name*.localTX

Specifies whether or not local transactions are used. The default is `true`, local transactions are used. The syntax is:

```
gg.handler.name.localTX={true|false}
```

15.2.5.1.13 gg.handlerlist.nop

Disables the sending of JMS messages to allow testing of message generation. This is a global property used only for testing. The events are still generated and handled and the message is constructed. The default is `false`; do not disable message send. The syntax is:

```
gg.handlerlist.nop={true|false}
```

15.2.5.2 Group Transaction Properties

These properties set limits for grouping transactions.

Note:

When you use group transaction properties, you must:

- Ensure that `gg.handler.`*`name`*`.mode` is set to one transaction per message (`tx`). Otherwise the group transaction properties will be ignored.
- Ensure that the `goldengate.userexit.nochkpt` property is set to `false`.
- Ignore the transaction indicator on the operations and not use it to determine transaction boundaries.
- Use only one named handler per installation.

15.2.5.2.1 gg.handler.*name*.minGroupSize

Specifies the minimum number of operations that must accumulate before the transaction will be sent.

The syntax is:

```
gg.handler.name.minGroupSize=number_ops
```

Where:

- *`number_ops`* specifies the minimum number of operations that must be accumulated before the transaction is sent.

The maximum value allowed is `integer.MAX.VALUE` or 2147483647. The minimum value is one.

If you use both properties, the value set for `gg.handler.`*`name`*`.minGroupSize` should be less than or equal to the value set for `gg.handler.`*`name`*`.maxGroupSize`.

The following example will test for a minimum of 50 operations before a send.

```
gg.handler.name.minGroupSize=50
```

15.2.5.2.2 gg.handler.*name*.maxGroupSize

Specifies the maximum number of operations that will rigger the transaction send.

The syntax is:

```
gg.handler.name.maxGroupSize=number_ops
```

Where:

- *`number_ops`* specifies the maximum number of operations that will be accumulated before the transaction is sent.

The maximum value allowed is `integer.MAX.VALUE` or 2147483647. The minimum value is one.

If you use both properties, the value set for `gg.handler.`*`name`*`.minGroupSize` should be less than or equal to the value set for `gg.handler.`*`name`*`.maxGroupSize`.

The following example will send when the maximum of 50 operations is reached.

```
gg.handler.name.maxGroupSize=50
```

15.2.6 JNDI Properties

These JNDI properties are required for connection to an Initial Context to look up the connection factory and initial destination.

```
java.naming.provider.url=url
java.naming.factory.initial=java-class-name
```

If JNDI security is enabled, the following properties may be set:

```
java.naming.security.principal=user-name
java.naming.security.credentials=password-or-other-authenticator
```

For example:

```
java.naming.provider.url= t3://localhost:7001
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.security.principal=jndiuser
java.naming.security.credentials=jndipw
```

15.2.7 General Properties

The following are general properties that are used for the user exit Java framework.

15.2.7.1 gg.classpath

Specifies a comma delimited list of additional paths to directories or jars to add to the class path. Optionally, the list can be delimited by semicolons for Windows systems or by colons for UNIX. For example:

```
gg.classpath=C:\Program Files\MyProgram\bin;C:\Program Files\ProgramB\app\bin;
```

15.2.7.2 gg.report.time

Specifies how often statistics are calculated and sent to Extract for the processing report. If Extract is configured to print a report, these statistics are included. The syntax is:

```
gg.report.time=report_interval{s|m|h}
```

Where:

- *report_interval* is an integer
- The valid time units are:
  - s - seconds
  - m - minutes
  - h - hours

If no value is entered, the default is to calculate and send every 24 hours.

16

Developing Custom Filters, Formatters, and Handlers

This chapter discusses writing Java code to implement an event filter, a custom formatter for a built-in handler, or a custom event handler. Specifying custom formatting through a Velocity template is also covered.

The Java package names are compliant with the Oracle standard. You must migrate the any previous release custom formatters, handlers, or filters to the new package names.

This chapter includes the following sections:

16.1 Filtering Events

By default, all transactions, operations and metadata events are passed to the `DataSourceListener` event handlers. An event filter can be implemented to filter the events sent to the handlers. The filter could select certain operations on certain tables containing certain column values, for example

Filters are additive: if more than one filter is set for a handler, then all filters must return true in order for the event to be passed to the handler.

You can configure filters using the Java application properties file:

```
# handler "foo" only receives certain events
gg.handler.one.type=jms
gg.handler.one.format=mytemplate.vm
gg.handler.one.filter=com.mycompany.MyFilter
```

To activate the filter, you write the filter and set it on the handler; no additional logic needs to be added to specific handlers.

You can write a custom filter by implementing the `oracle.goldengate.datasource.DsEventFilterinterface` filter, which contains filter contracts as in the following example:

```
package com.mycompany;

import oracle.goldengate.datasource.DsConfiguration;
import oracle.goldengate.datasource.DsEventFilter;
```

```
import oracle.goldengate.datasource.DsOperation;
import oracle.goldengate.datasource.DsTransaction;
import oracle.goldengate.datasource.meta.DsMetaData;
import oracle.goldengate.datasource.meta.TableMetaData;

public class MyFilter implements DsEventFilter {
  @Override public void init(DsConfiguration conf) {
  }

  @Override public void destroy() {
  }

  @Override public boolean doProcess(DsTransaction tx, DsMetaData metaData) {return
true// transactionBegin(), transactionCommit(), transactionRollback() will be
allowed.}

  @Override public boolean doProcess(DsOperation op, DsMetaData metaData) {return
true// operationAdded() will be allowed.}

  @Override public boolean doProcess(TableMetaData tbl,DsMetaData meta) {return
true// metaDataChanged() will be allowed.}
}
```

16.2 Custom Formatting

You can customize the output format of a built-in handler by:

- Writing a custom formatter in Java or
- Using a Velocity template

16.2.1 Using a Velocity Template

As an alternative to writing Java code for custom formatting, Velocity templates can be a good alternative to quickly prototype formatters. For example, the following template could be specified as the format of a JMS or file handler:

```
Transaction: numOps='$tx.size' ts='$tx.timestamp'
#for each( $op in $tx )
operation: $op.sqlType, on table "$op.tableName":
#for each( $col in $op )
$op.tableName, $col.meta.columnName = $col.value
#end
#end
```

If the template were named sample.vm, it could be placed in the classpath, for example:

gg_install_dir/dirprm/sample.vm

Note:

If using Velocity templates, the file name must end with the suffix .vm; otherwise the formatter is presumed to be a Java class.

Update the Java application properties file to use the template:

```
# set properties on 'one'
gg.handler.one.type=file
```

```
gg.handler.one.format=sample.vm
gg.handler.one.file=output.xml
```

When modifying templates, there is no need to recompile any Java source; simply save the template and re-run the Java application. When the application is run, the following output would be generated (assuming a table named `SCHEMA.SOMETABLE`, with columns `TESTCOLA` and `TESTCOLB`):

```
Transaction: numOps='3' ts='2008-12-31 12:34:56.000'
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 123
SCHEMA.SOMETABLE, TESTCOLB = value abc
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 456
SCHEMA.SOMETABLE, TESTCOLB = value def
operation: UPDATE, on table "SCHEMA.SOMETABLE":
SCHEMA.SOMETABLE, TESTCOLA = value 789
SCHEMA.SOMETABLE, TESTCOLB = value ghi
```

16.2.2 Coding a Custom Formatter in Java

The preceding examples show a JMS handler and a file output handler using the same formatter (`com.mycompany.MyFormatter`). The following is an example of how this formatter may be implemented.

Example 16-1 Custom Formatting Implementation

```
package com.mycompany.MyFormatter;
import oracle.goldengate.datasource.DsOperation;
import oracle.goldengate.datasource.DsTransaction;
import oracle.goldengate.datasource.format.DsFormatterAdapter;
import oracle.goldengate.datasource.meta.ColumnMetaData;
import oracle.goldengate.datasource.meta.DsMetaData;
import oracle.goldengate.datasource.meta.TableMetaData;
import java.io.PrintWriter;
public class MyFormatter extends DsFormatterAdapter {

        public MyFormatter() { }
        @Override
        public void formatTx(DsTransaction tx,

DsMetaData meta,
PrintWriter out)

        {

            out.print("Transaction: " );
            out.print("numOps=\'" + tx.getSize() + "\' " );
            out.println("ts=\'" + tx.getStartTxTimeAsString() + "\'");
            for(DsOperation op: tx.getOperations()) {
TableName currTable = op.getTableName();
TableMetaData tMeta = dbMeta.getTableMetaData(currTable);
String opType = op.getOperationType().toString();
String table = tMeta.getTableName().getFullName();
out.println(opType + " on table \"" + table + "\":" );
int colNum = 0;
for(DsColumn col: op.getColumns())
{

ColumnMetaData cMeta = tMeta.getColumnMetaData( colNum++ );
out.println(
```

```
cMeta.getColumnName() + " = " + col.getAfterValue() );
}

        }
        @Override
        public void formatOp(DsTransaction tx,

DsOperation op,
TableMetaData tMeta,
PrintWriter out)

        {

            // not used...

        }

}
```

The formatter defines methods for either formatting complete transactions (after they are committed) or individual operations (as they are received, before the commit). If the formatter is in operation mode, then `formatOp`(...) is called; otherwise, `formatTx`(...) is called at transaction commit.

To compile and use this custom formatter, include the Oracle GoldenGate for Java JARs in the classpath and place the compiled `.class` files in g*g_install_dir*`/dirprm`:

```
javac -d gg_install_dir/dirprm
-classpath ggjava/ggjava.jar MyFormatter.java
```

The resulting class files are located in `resources/classes` (in correct package structure):

```
gg_install_dir/dirprm/com/mycompany/MyFormatter.class
```

Alternatively, the custom classes can be put into a JAR; in this case, either include the JAR file in the JVM classpath using the user exit properties (using `java.class.path` in the j`vm.bootoptions` property), or by setting the Java application properties file to include your custom JAR:

```
# set properties on 'one'
gg.handler.one.type=file
gg.handler.one.format=com.mycompany.MyFormatter
gg.handler.one.file=output.xml
gg.classpath=/path/to/my.jar,/path/to/directory/of/jars/*
```

16.2.3 Coding a Custom Handler in Java

A custom handler can be implemented by extending `AbstractHandler` as in the following example:

```
import oracle.goldengate.datasource.*;
import static oracle.goldengate.datasource.GGDataSource.Status;
public class SampleHandler extends AbstractHandler {
        @Override
        public void init(DsConfiguration conf, DsMetaData metaData) {
            super.init(conf, metaData);
            // ... do additional config...
        }
        @Override
```

```
        public Status operationAdded(DsEvent e, DsTransaction tx, DsOperation op)
{ ... }
        @Override
        public Status transactionCommit(DsEvent e, DsTransaction tx) { ... }
        @Override
        public Status metaDataChanged(DsEvent e, DsMetaData meta) { .... }
        @Override
        public void destroy() { /* ... do cleanup ... */ }
        @Override
        public String reportStatus() { return "status report..."; }
        @Override
        public Status ddlOperation(OpType opType, ObjectType objectType, String
objectName, String ddlText) }
```

The method in `AbstractHandler` is not abstract rather it has a body. In the body it performs cached metadata invalidation by marking the metadata object as dirty. It also provides TRACE-level logging of DDL events when the `ddlOperation` method is specified. You can override this method in your custom handler implementations. You should always call the super method before any custom handling to ensure the functionality in `AbstractHandler` is executed

When a transaction is processed from the Extract, the order of calls into the handler is as follows:

1. Initialization:
   - First, the handler is constructed.
   - Next, all the "setters" are called on the instance with values from the property file.
   - Finally, the handler is initialized; the `init(...)` method is called before any transactions are received. It is important that the `init(...)` method call `super.init(...)` to properly initialize the base class.
2. Metadata is thenreceived. If the user exit is processing an operation on a table not yet seen during this run, a metadata event is fired, and the `metadataChanged(...)` method is called. Typically, there is no need to implement this method. The `DsMetaData` is automatically updated with new data source metadata as it is received.
3. A transaction is started. A transaction event is fired, causing the `transactionBegin(...)` method on the handler to be invoked (this is not shown). This is typically not used, since the transaction has zero operations at this point.
4. Operations are added to the transaction, one after another. This causes the `operationAdded(...)` method to be called on the handler for each operation added. The containing transaction is also passed into the method, along with the data source metadata that contains all processed table metadata. The transaction has not yet been committed, and could be aborted before the commit is received.

   Each operation contains the column values from the transaction (possibly just the changed values when Extract is processing with compressed updates.) The column values may contain both before and after values.

   For the `ddlOperation` method, the options are:

   - `opType` - Is an enumeration that identifies the DDL operation type that is occurring (`CREATE`, `ALTER`, and so on).

- `objectType` - Is an enumeration that identifies the type of the target of the DDL (`TABLE`, `VIEW`, and so on).
- `objectName` - Is the fully qualified source object name; typically a fully qualified table name.
- `ddlText` - Is the raw DDL text executed on the source relational database.

5. The transaction is committed. This causes the `transactionCommit(...)` method to be called.
6. Periodically, `reportStatus` may be called; it is also called at process shutdown. Typically, this displays the statistics from processing (the number of operations andtransactions processed and other details).

An example of a simple printer handler, which just prints out very basic event information for transactions, operations and metadata follows. The handler also has a property `myoutput` for setting the output file name; this can be set in the Java application properties file as follows:

```
gg.handlerlist=sample
# set properties on 'sample'
gg.handler.sample.type=sample.SampleHandler
gg.handler.sample.myoutput=out.txt
```

To use the custom handler,

1. Compile the class
2. Include the class in the application classpath,
3. Add the handler to the list of active handlers in the Java application properties file.

To compile the handler, include the Oracle GoldenGate for Java JARs in the classpath and place the compiled `.class` files in *`gg_install_dir`*`/javaue/resources/classes`:

```
javac -d gg_install_dir/dirprm
-classpath ggjava/ggjava.jar SampleHandler.java
```

The resulting class files would be located in `resources/classes`, in correct package structure, such as:

```
gg_install_dir/dirprm/sample/SampleHandler.class
```

Note:

For any Java application development beyond *hello world* examples, either Ant or Maven would be used to compile, test and package the application. The examples showing `javac` are for illustration purposes only.

Alternatively, custom classes can be put into a JAR and included in the classpath. Either include the custom JAR file(s) in the JVM classpath using the user exit properties (using `java.class.path` in the `jvm.bootoptions` property), or by setting the Java application properties file to include your custom JAR:

```
# set properties on 'one'
gg.handler.one.type=sample.SampleHandler
```

```
gg.handler.one.myoutput=out.txt
gg.classpath=/path/to/my.jar,/path/to/directory/of/jars/*
```

The classpath property can be set on any handler to include additional individual JARs, a directory (which would contain resources or extracted class files) or a whole directory of JARs. To include a whole directory of JARs, use the Java 6 style syntax:

```
c:/path/to/directory/* (or on UNIX: /path/to/directory/* )
```

Only the wildcard * can be specified; a file pattern cannot be used. This automatically matches all files in the directory ending with the `.jar` suffix. To include multiple JARs or multiple directories, you can use the system-specific path separator (on UNIX, the colon and on Windows the semicolon) or you can use platform-independent commas, as shown in the preceding example.

If the handler requires many properties to be set, just include the property in the parameter file, and your handler's corresponding "setter" will be called. For example:

```
gg.handler.one.type=com.mycompany.MyHandler
gg.handler.one.myOutput=out.txt
gg.handler.one.myCustomProperty=12345
```

The preceding example would invoke the following methods in the custom handler:

```
public void setMyOutput(String s) {

        // use the string...

} public void setMyCustomProperty(int j) {

        // use the int...

}
```

Any standard Java type may be used, such as int, long, String, boolean. For custom types, you may create a custom property editor to convert the String to your custom type.

16.2.4 Coding a Custom Formatter for Java Delivery

You can develop a custom formatter for use with Java Delivery. The following `ExampleFormatter.java` example demonstrates how to:

- access and output metadata,
- access and output the column data values of the operation, and
- ascertain if the operation is a insert, update, delete, or primary key update.

Example 16-2 Custom Formatter for Java Delivery

```
/*
 *
 * Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.
 *
 */
package oracle.goldengate.datasource.format;

//Logging imports
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import java.io.PrintWriter;
import oracle.goldengate.datasource.DsColumn;
import oracle.goldengate.datasource.DsConfiguration;
import oracle.goldengate.datasource.DsOperation;
import oracle.goldengate.datasource.DsToken;
import oracle.goldengate.datasource.DsTransaction;
import oracle.goldengate.datasource.TxOpMode;
import oracle.goldengate.datasource.meta.ColumnMetaData;
import oracle.goldengate.datasource.meta.DsMetaData;
import oracle.goldengate.datasource.meta.TableMetaData;
import oracle.goldengate.datasource.meta.TableName;
import oracle.goldengate.util.CDataUtil;

/**
 * This class provides an example for Oracle GoldenGate Java Adapter customers
 * to write their own custom formatter which can be plugged into the Java file
 * writer or Java JMS handler.
 * This example does NOT work with the new Big Data integrations such as HDFS, Kafka,
 * or Flume as formatter interface for those has been changed.
 * This formatter formats data in CSV format, by default.  The field and line
 * delimiters are configurable with the default as a comma and a line feed
 * respectively.
 * The goal is to provide an example custom formatter for customers wishing
 * to develop a custom formatter, to see how it works.  The functionality
 * can be extended and/or altered as needed to fulfill the specific needs of
 * the customer.
 * User configures the use of this formatter by setting the follow configuration
 * in the Oracle GoldenGate Java Adapter properties file.
 * gg.handler.name.format=oracle.goldengate.datasource.format.ExampleFormatter
 * @author Tom Campbell
 * @email thomas.campbell@oracle.com
 */
public class ExampleFormatter extends DsFormatterAdapter{
    private static final Logger
logger=LoggerFactory.getLogger(ExampleFormatter.class);

    //The field delimiter defaults to a comma
    private String fieldDelimiter = ",";
    //The line delimiter defaults to the line separator
    private String lineDelimiter = System.lineSeparator();

    /**
     * Default Constructor
     */
    public ExampleFormatter() {
        super(TxOpMode.op);
        //IMPLEMENTERS - Add constructor implementation code below
    }

    /**
     * Constructor with operation mode argument.
     * @param mode Mode of operation Transaction (tx) or Operation (op)
     */
    public ExampleFormatter(TxOpMode mode) {
        super(mode);
        //IMPLEMENTERS - Add constructor implementation code below
    }

    /**
     * Setter method to set the field delimiter.  User can configure
```

```
     * generic setters on the formatter than are configured in the Oracle
     * GoldenGate Java Adapter properties file as follows:
     * gg.handler.name.format.fieldDelimiter=somevalue
     * The user is free to generate any setter methods for the formatter.  The
     * configuration initialization code uses Java reflection to find and invoke
     * the corresponding setter method.  If the method is not found an exception
     * will be thrown and the Oracle GoldenGate process will ABEND.
     * @param fd The field delimiter value
     */
    void setFieldDelimiter(String fd){
        //The CDataUtil.unwrapCData allows the user to configure values that
        //are considered to be whitespace.  Examples are spaces, tabs, carriage
        //returns, line feeds, etc.  To configure white space use the CDATA[] wrapper
        //as follows.
        //gg.handler.name.format.fieldDelimiter=CDATA[somevalue]
        //The configuration value will be "somevalue".
        //The CDATA[] functionality is needed to preserve whitespace because the
        //default behavior of the GG config code is to trim whitespace.
        fieldDelimiter = CDataUtil.unwrapCData(fd);
    }

    /**
     * Setter method to set the line delimiter.  User can configure
     * generic setters on the formatter than are configured in the Oracle
     * GoldenGate Java Adapter properties file as follows:
     * gg.handler.name.format.lineDelimiter=somevalue
     * The user is free to generate any setter methods for the formatter.  The
     * configuration initialization code uses Java reflection to find and invoke
     * the corresponding setter method.  If the method is not found an exception
     * will be thrown and the Oracle GoldenGate process will ABEND.
     * @param fd The field delimiter value
     */
    void setLineDelimiter(String ld){
        //The CDataUtil.unwrapCData allows the user to configure values that
        //are considered to be whitespace.  Examples are spaces, tabs, carriage
        //returns, line feeds, etc.  To configure whitespace use the CDATA[] wrapper
        //as follows.
        //gg.handler.name.format.fieldDelimiter=CDATA[\n]
        //The configuration value will be a line feed.
        //The CDATA[] functionality is needed to preserve whitespace because the
        //default behavior of the GG config code is to trim whitespace.
        lineDelimiter = CDataUtil.unwrapCData(ld);
    }

    /**
     * Initialization method.  This is called once after all of the setter
     * methods are called to set the explicit configuration methods of the
     * formatter.
     * @param conf data source configuration info from properties file
     * @param meta metadata from data source, describing columns/tables, etc
     */
    @Override
    public void init(DsConfiguration conf, DsMetaData meta) {
        //Call the base class first.
        super.init(conf, meta);
        //IMPLEMENTERS - Add initialization code below
        logger.info("Initializing the example formatter.");
        logger.info("  The field delimiter is configured as [" + fieldDelimiter +
"].");
        logger.info("  The line delimiter is configured as [" + lineDelimiter +
"].");
```

```
    }

    /**
     * The format operation method.  This is where the bulk of the formatting
     * logic will reside.
     * @param tx The transaction object
     * @param op The operation object
     * @param tmeta The table metadata object
     * @param writer The print writer object to which results should be written
     */
    @Override
    public void formatOp(DsTransaction tx, DsOperation op, TableMetaData tmeta,
            PrintWriter writer) {
        logger.info("Method formatOp called...");
        //IMPLEMENTERS - Below is the call the example implementation
        //comment out and insert custom implementation here.
        internalFormatOperation(tx, op, tmeta, writer);
    }

    @Override
    public void formatTx(DsTransaction tx, DsMetaData dbmeta, PrintWriter writer) {
        logger.info("Method formatTx called...");
        //In transaction mode (i.e. gg.handler.name.mode=tx) the handler may
        //be configured to call this method to format data at the end of the
        //into a single print writer.  This is probably not the usual use case.
        //Using replicat transaction grouping (see GROUPTRANSOPS replicat
configuration)
        //the number of operations in a grouped transaction can be substantial.
        //Replicat GROUPTRANSOPS is set to 1000 by default meaning as many as
        //1000 source transactions are getting grouped into a single target
        //transaction.
        for (DsOperation op : tx) {
            TableName currTable = op.getTableName();
            TableMetaData tmeta = dbmeta.getTableMetaData(currTable);
            //Then call the internal formatting
            internalFormatOperation(tx, op, tmeta, writer);
        }
    }

    private void internalFormatOperation(DsTransaction tx, DsOperation op,
            TableMetaData tmeta, PrintWriter writer) {
        //Use a string builder for performance.
        StringBuilder sb = new StringBuilder();

        //IMPLEMENTERS - example code shown below outputs commonly used data.
        //Modify the code below as need for the specific implementation.

        //Format the table name
        formatTableName(op, sb);
        //Format the operation type.
        formatOperationType(op, sb);
        //Format the operation timestamp.
        formatOperationTimestamp(op, sb);
        //Format the position
        formatPosition(op, sb);
        //Add primary keys
        formatPrimaryKeys(tmeta, sb);
        //Format tokens
        formatGGTokens(op, sb);
        //Format the column data
        formatColumnData(op, tmeta, sb);
```

```
    //Transfer the message to the print writer.
    writer.print(sb.toString());
}

/**
 * This method formats the table name for output
 * @param op The operation object
 * @param sb The string builder object to append the data
 */
private void formatTableName(DsOperation op, StringBuilder sb){
    sb.append("TableName");
    sb.append(fieldDelimiter);
    //The original fully qualified table name in the original case
    sb.append(op.getTableName().getOriginalName());
    sb.append(fieldDelimiter);
}

/**
 * Method to format the operation type data.  Operation types are
 * insert, update, delete, and primary key update.
 * @param op The operation object
 * @param sb The string builder object to append the data
 */
private void formatOperationType(DsOperation op, StringBuilder sb){
            //Output the operation type
    sb.append("OperationType");
    sb.append(fieldDelimiter);
    if (op.getOperationType().isInsert()){
        sb.append("insert");
    }else if (op.getOperationType().isDelete()){
        sb.append("delete");
    }else if (op.getOperationType().isPkUpdate()){
        //This is a specialzied use case of update.  The isUpdate() call also
        //returns true for primary key updates.
        sb.append("primarykeyupdate");
    }else if (op.getOperationType().isUpdate()){
        sb.append("update");
    }
    sb.append(fieldDelimiter);
}

/**
 * Method to format the operation timestamp.  This is the transaction commit
 * time of the transaction that contains the operation.  All operations
 * within a transaction have the same operation timestamp.
 * @param op The operation object
 * @param sb The string builder object to append the data
 */
private void formatOperationTimestamp(DsOperation op, StringBuilder sb){
    sb.append("OperationTimestamp");
    sb.append(fieldDelimiter);
    sb.append(op.getTimestampAsString());
    sb.append(fieldDelimiter);
}

/**
 * Method to format the position.
 * The position is the concatentated trail file sequence number followed
 * by the RBA number (offset in the trail file).  The two together provide
 * traceability of the operation back to source trail file.
```

```
     * @param op The operation object
     * @param sb The string builder object to append the data.
     */
    private void formatPosition(DsOperation op, StringBuilder sb){
        sb.append("Position");
        sb.append(fieldDelimiter);
        sb.append(op.getPosition());
        sb.append(fieldDelimiter);
    }

    /**
     * Method to format the primary keys.
     * @param tmeta The table metadata object.
     * @param sb The string builder object to append the data.
     */
    private void formatPrimaryKeys(TableMetaData tmeta, StringBuilder sb){
        sb.append("PrimaryKeys");
        for(ColumnMetaData cmeta :tmeta.getColumnMetaData()){
            if (cmeta.isKeyCol()){
                sb.append(fieldDelimiter);
                sb.append(cmeta.getOriginalColumnName());
            }
        }
    }

    /**
     * Method to format the GoldenGate token key/value pairs from the source
     * trail file.
     * @param op The operation object.
     * @param sb The string builder object to append the data.
     */
    private void formatGGTokens(DsOperation op, StringBuilder sb){
        //If this is false there will not be any tokens
        if(op.getIncludeTokens()){
            sb.append(fieldDelimiter);
            sb.append("GGTokens");
            sb.append(fieldDelimiter);
            for(DsToken token : op.getTokens().values()){
                sb.append(token.getKey());
                sb.append(fieldDelimiter);
                sb.append(token.getValue());
                sb.append(fieldDelimiter);
            }
        }
    }

    /**
     * Method to output the column value data.  This data starts with the column
     * name, "before", the before image value, "after, and the after image value.
     * This before and after column values include special handling for missing
     * and null values.  Missing values are output as "MISSING".  Null values are
     * output as "NULL"
     * @param op The operation object.
     * @param tmeta The table metadata object
     * @param sb
     */
    private void formatColumnData(DsOperation op, TableMetaData tmeta, StringBuilder
sb){
        int cIndex = 0;
        sb.append(fieldDelimiter);
        sb.append("TableData");
```

```
        for(DsColumn col : op.getColumns()) {
            //Get the column metadata
            ColumnMetaData cmeta = tmeta.getColumnMetaData(cIndex++);
            //Output the column name
            sb.append(fieldDelimiter);
            sb.append(cmeta.getOriginalColumnName());
            sb.append(fieldDelimiter);

            //Output the before value
            //Insert operations have no before values
            //Delete operations generally only have column values for primary
            //keys and not the complete row.
            //Updates will have all column values if NOCOMPRESSUPDATES is configured
            //else updates will have only primary keys and the column values that
changed.
            sb.append("Before");
            sb.append(fieldDelimiter);
            if (col.getBefore() == null){
                //The before image value is missing for the column.
                sb.append("MISSING");
            }else if(col.getBefore().isValueNull()){
                sb.append("NULL");
            }else{
                sb.append(col.getBefore().getValue());
            }
            sb.append(fieldDelimiter);

            //Output the after value
            //Insert operations have after values
            //Delete operations have no after values
            //Update operations will have all column values if NOCOMPRESSUPDATES is
configured
            //else updates will have only primary keys and the column values that
changed.
            sb.append("After");
            sb.append(fieldDelimiter);
            if (col.getAfter() == null){
                //The before image value is missing for the column.
                sb.append("MISSING");
            }else if(col.getAfter().isValueNull()){
                sb.append("NULL");
            }else{
                sb.append(col.getAfter().getValue());
            }
        }
        //Add the line delimiter at the end
        sb.append(lineDelimiter);
    }

}
```

Note:

This example does *not* work with Oracle GoldenGate for Big Data handlers.

This type of custom formatter works well with the JMS and the Java File Writer handlers. Once processed, this outputs operation data in a CSV format. Operation metadata is output first, and then the column data including before and after change column image data. The field and line delimiters are configurable.

Tip:

You can obtain a copy of the maven project that you import to your Java IDE by contacting Oracle Support.

16.3 Additional Resources

There is Javadoc available for the Java API. The Javadoc has been intentionally reduced to a set of core packages, classes and interfaces in order to only distribute the relevant interfaces and classes useful for customizing and extension.

In each package, some classes have been intentionally omitted for clarity. The important classes are:

- `oracle.goldengate.datasource.DsTransaction`: represents a database transaction. A transaction contains zero or more operations.
- `oracle.goldengate.datasource.DsOperation`: represents a database operation (insert, update, delete). An operation contains zero or more column values representing the data-change event. Columns indexes are offset by zero in the Java API.
- `oracle.goldengate.datasource.DsColumn`: represents a column value. A column value is a composite of a before and an after value. A column value may be 'present' (having a value or be null) or 'missing' (is not included in the source trail).
  - `oracle.goldengate.datasource.DsColumnComposite` is the composite
  - `oracle.goldengate.datasource.DsColumnBeforeValue` is the column value before the operation (this is optional, and may not be included in the operation)
  - `oracle.goldengate.datasource.DsColumnAfterValue` is the value after the operation
- `oracle.goldengate.datasource.meta.DsMetaData`: represents all database metadata seen; initially, the object is empty. `DsMetaData` contains a hash map of zero or more instances of `TableMetaData`, using the `TableName` as a key.
- `oracle.goldengate.datasource.meta.TableMetaData`: represents all metadata for a single table; contains zero or more `ColumnMetaData`.
- `oracle.goldengate.datasource.meta.ColumnMetaData`: contains column names and data types, as defined in the database or in the Oracle GoldenGate source definitions file.

See the Javadoc for additional details.

Part V

Troubleshooting the Oracle GoldenGate Adapters

This part of the book provides information on troubleshooting problems with the Oracle GoldenGate Adapters.

Part I contains the following chapters:

- Troubleshooting the Flat File Adapter.
- Troubleshooting the Java Adapters.

17

Troubleshooting the Flat File Adapter

This chapter outlines steps you can take to solve problems with Oracle GoldenGate Adaptors for Flat Files. It lists the error checks to perform. If you do not succeed in identifying the problem, submit a support ticket or contact Oracle Support.

This chapter includes the following sections:

17.1 Checking Oracle GoldenGate

Before checking for specific issues related to the Oracle GoldenGate for Flat File, ensure that Oracle GoldenGate is configured correctly and any standard Oracle GoldenGate errors have been resolved. For further information, see the *Oracle GoldenGate Troubleshooting and Performance Tuning Guide*.

17.2 Checking the Configuration

Check the following:

- Is the shared library (.so or .dll) in the Extract parameter file correct? Is it specified in the path and accessible?
- Is the correct `SOURCEDEFS` file specified in the Extract parameter file? Is it in the specified path and accessible?
- Does the `SOURCEDEFS` file contain all the necessary tables?
- Is the `ffwriter.properties` user exit properties file in the Oracle GoldenGate install directory, or does it have the correct name and path specified in the `GG_USEREXIT_PROPFILE` environment variable?
- Do the output directories specified in the user exit properties file exist?
- Are file permissions correct to write to that directory?

17.3 Checking the Log File

Check the log file (`logname_yyyymmdd.log`). By default this file will be in the `dirrpt` subdirectory.

- Does the user exit properties file parse successfully? Are any invalid properties mentioned in the log file?
- Are any other errors or warnings in the log?

17.4 Contacting Oracle Support

If the problem is still not resolved:

- Set log.level=`DEBUG`
- Restart and save the log file

Before contacting Oracle Support, be prepared to send the log file, source trail file, source definitions file, user exit properties file, and Extract parameter file, together with any data files that have been written.

18

Troubleshooting the Java Adapters

This chapter outlines steps you can take to solve problems with Oracle GoldenGate Adaptors for Java. It lists the error checks you should perform. If you do not succeed in identifying the problem, submit a support ticket or contact Oracle Support.

This chapter includes the following sections:

18.1 Checking for Errors

There are two types of errors that can occur in the operation of Oracle GoldenGate for Java:

- The Replicat or Extract process running the Java Adapter user-exit abends
- The Extract process running VAM abends.
- The process runs successfully, but the data is incorrect or nonexistent

If the Extract process does not start or abends, check the error messages in order from the beginning of processing through to the end:

1. Check the Oracle GoldenGate event log for errors, and then view the Replicat or Extract report:

   ```
   GGSCI> VIEW GGSEVT
   GGSCI> VIEW REPORT {extract_or_replicat_name}
   ```

2. Check the applicable log file.

   For the user exit:

   - Look at the last messages reported in the log file for the user exit library. The file name is the log file prefix (`log.logname`) set in the property file and the current date.

     ```
     shell> more {log.logname}_{yyyymmdd}.log
     ```

     Note:

     This is only the log file for the shared library and not the Java application. It is only created for the Extract user-exit; it is not created for the Replicat user-exit.

3. If the user exit or VAM was able to launch the Java runtime, then a log4j log file will exist.

   The name of the log file is defined in your log4j.properties file. By default, the log file name is `ggjava-version-log4j.log`, where *version* is the version number of the jar file being used. For example:

   ```
   shell> more ggjava-*log4j.log
   ```

   To set a more detailed level of logging for the Java application, either:

   - Edit the current log4j properties file to log at a more verbose level or
   - Re-use one of the existing log4j configurations by editing properties file:

     ```
     jvm.bootoptions=-Djava.class.path=ggjava/ggjava.jar
     -Dlog4j.configuration=debug-log4j.properties -Xmx512m
     ```

   These pre-configured log4j property files are found in the class path, and are installed in:

   ```
   ./ggjava/resources/classes/*log4j.properties
   ```

4. If one of these log files does not reveal the source of the problem, run the Extract process directly from the shell (outside of GGSCI) so that `stderr` and `stdout` can more easily be monitored and environmental variables can be verified. For example:

   ```
   shell> EXTRACT PARAMFILE dirprm/javaue.prm
   ```

If the process runs successfully, but the data is incorrect or nonexistent, check for errors in any custom filter, formatter or handler you have written for the user exit.

To restart the user exit Extract from the beginning of a trail, see Restarting the Application at the Beginning of a Trail.

18.2 Recovering after an Abend

The Extract parameter `RECOVERYOPTIONS` defaults to `APPENDMODE` for release 10 and later trails. In append mode, Extract writes a recovery marker to the trail when it abends. When the Extract restarts and encounters the recovery marker, it requests a rollback of the incomplete transaction if local transactions are enabled. If local transactions are not enabled, a warning message is issued. Local transactions are enabled unless the property `gg.handler.{name}.localTX` is explicitly set to `false`.

18.3 Reporting Issues

If you have a support account for Oracle GoldenGate, submit a support ticket. Please include:

- Operating system and Java versions

  The version of the Java Runtime Environment can be displayed by:

  ```
  $ java -version
  ```

- Configuration files:

  – Parameter file for the Extract running the user exit

  - All properties files used, including any JMS or JNDI properties files
  - Velocity templates for the user exit

- Log files:

  In the Oracle GoldenGate install directory, all `.log` files: the Java log4j log files and the user exit or VAM log file.

Part VI

Appendix

The appendix provides information on targeted uses of the Oracle GoldenGate Adapters and lists samples that are available with the installation.

Part VI contains the following appendices:

- Adapter Examples

A

Adapter Examples

This appendix lists the examples that are included with the Oracle GoldenGate Adapter installation and explains examples for some use cases.

This appendix includes the following sections:

A.1 List of Included Examples

The following examples are located in the `AdaptersExamples` subdirectory of the installation location.

Flat File Writer

- Using the Oracle GoldenGate Flat File Adapter to convert Oracle GoldenGate trail data to text files.

Message Delivery

- Using the Oracle GoldenGate Java Adapter to send JMS messages with a custom message format.
- Using the Oracle GoldenGate Java Adapter to send JMS messages with custom message header properties.

Message Capture

- Using the Oracle GoldenGate Java Adapter to process JMS messages, creating an Oracle GoldenGate trail.

Java User Exit API

- Using the Oracle GoldenGate Java Adapter API to write a custom event handler.

A.2 Configuring Logging

This example explains how to configure logging for release 11.2.1 or later Oracle GoldenGate Adapters user exits. The first section configures a typical Extract pump, which triggers the logging defaults. The second section explains how to customize the logging implementation.

A.2.1 Example Oracle GoldenGate Java User Exit Defaults

The following Oracle GoldenGate Java user exit Extract example configuration triggers the logging defaults.

Extract Parameter File

```
EXTRACT jms1
SOURCEDEFS dirdef/aa.def
CUSEREXIT libggjava_ue.so CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES
GETUPDATEBEFORES
TABLE GG.*;
```

Properties file

The associated property file is named for the Extract group, `jms1.properties`. All JNI properties have default values and thus do not need to be specified, so this is a complete properties file.

```
gg.handlerlist=my_jms

gg.handler.my_jms.type=jms
gg.handler.my_jms.destination=dynamicQueues/testQ1
gg.handler.my_jms.format=xml2
gg.handler.my_jms.mode=op
gg.handler.my_jms.connectionFactory=ConnectionFactory

gg.java.naming.provider.url=tcp://localhost:61616

gg.java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory

gg.classpath=/opt/activemq/activemq-all.jar
```

The Resulting Log File

The log file will be created when you add and start the Extract in GGSCI, For example:

```
ggsci> ADD EXTRACT jms1, EXTRAILSOURCE dirdat/aa
ggsci> START MGR
ggsci> START EXTRACT jms1
```

The log file is written to the same directory as the report file. It is named for the Extract group. For Example:

```
$ ls -l dirrpt/
total 48
-rw-rw-rw- 1 1685   Apr 16 20:38   MGR.rpt
-rw-rw-rw- 1 1685   Apr 16 20:38   jms1.rpt
-rw-rw-rw- 1 21705  Apr 19 13:59   jms1_info_0.log.0
-rw-rw-rw- 1 0      Apr 19 13:58   jms1_info_0.log.0.lck
```

A.2.2 Customizing Logging

This example describes how to customize the logging for 11.2.1 and later Oracle GoldenGate user exit adapters by using one of two methods:

- Use Java adapter user exit properties

  ```
  gg.log={ jdk | logback | log4j }
  gg.log.level={ info | debug | trace }
  gg.log.classpath={ classpath for logging }
  ```

  If the log implementation property `gg.log` is not set, the `jdk` option defaults. This specifies that `java.util.logging` (`JUL`) is used. The log level defaults to `info`. To customize this, you can set the `gg.log` to either:

  - `log4j` - This automatically configures the class path to include the Log4j and appropriate `slf4j-log4j` binding.
  - `logback` - To use the `logback` option, the logback jars must be manually downloaded and copied into the install directory. The class path is still automatically configured as long as the jars are copied into the predefined location. See `ggjava/resources/lib/optional/logback/ReadMe-logback.txt` for more information.

- Use JVM options

  Instead of using default logging or setting logging properties, `jvm.bootoptions` can be used to define the logging. To do this, set `jvm.bootoptions` to include the system property that defines the configuration file by doing one of the following:

  - Specify a log4j configuration file:

    ```
    jvm.bootoptions=-Dlog4j.configuration=my-log4j.properties
    ```

    This implicitly sets `gg.log` to `log4j` as the type of logging implementation and appends `slf4j-log4j12` binding to the class path.

  - Specify a `java.util.logging` properties file or class:

    ```
    jvm.bootoptions=-Djava.util.logging.config.file=my-logging.properties
    ```

    This implicitly sets `gg.log=jdk`, which specifics `java.util.logging` (`JUL`). It appends `slf4j-jdk14` binding to the class path.

  - First, download and copy `logback-core-jar` and `logback-classic-jar` into `ggjava/resources/lib/optional/logback.` Then specify a logback configuration file:

    ```
    jvm.bootoptions=-Dlogback.configuationFile=my-logback.xml
    ```

    This implicitly sets `gg.log=logback` and appends `logback-classic` and `logback-core` to the class path.

  These are implicit settings of `gg.log` and `gg.log.classpath` that will be overridden by an explicit setting of either of these properties in the property file. The logging class path will also be overridden by setting the JVM class path to include specific jars, such as:

  ```
  jvm.bootoptions=...-Djava.class.path=mypath/my1.jar:mypath2/my2.jar...
  ```

  Note:

  Setting the JVM class path to includespecific jars may cause duplicate, possibly conflicting, impleentations in the class path.

Index

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

V

W

X