Oracle® GoldenGate for Flat File

Administration Guide

Version 3.0

October 2009

Administration Guide, version 3.0

Contents

CHAPTER 1
Introduction

This guide cover installing, configuring and running Oracle GoldenGate for Flat File.

Oracle GoldenGate Transactional Data Management

The core Oracle GoldenGate product is a Transactional Data Management (TDM) platform that:

- Captures transactional changes from a source database by reading the database transaction log
- Sends and queues these changes on local or remote disk, as a set of database-independent binary 'trail' files
- Optionally transforms the source data
- Applies the transactions in the trail to a target system: a database, JMS provider or other messaging system or custom application.

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

Integration Options

The transactional changes may be applied to targets other than a relational database: for example, ETL tools (DataStage, Ab Initio, Informatica), messaging systems (JMS, MQ Series) 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 for Flat File 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; for example: delimiter separated values, length delimited values, binary, etc. Near real-time feeds to these systems are accomplished by decreasing the time window for batch file rollover to minutes or even seconds.
- *Messaging systems*: transactions or operations can be published as messages (e.g. in XML) to JMS. The JMS provider is configurable; examples include: ActiveMQ, JBoss Messaging, Solace, 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 for Flat File provides filewriter libraries (implemented natively in C) that can be dynamically linked into the Oracle GoldenGate Extract process. No programming is required; they are customizable using a simple properties file.
- For Java integration using either JMS or the Java API, use the Integration Edition for JMS/Java.

This guide describes how to use Oracle GoldenGate for Flat File.

Oracle GoldenGate for Flat File

Oracle GoldenGate for Flat File is used to output transactional data captured by Oracle GoldenGate to rolling flat files to be consumed by a third party product.

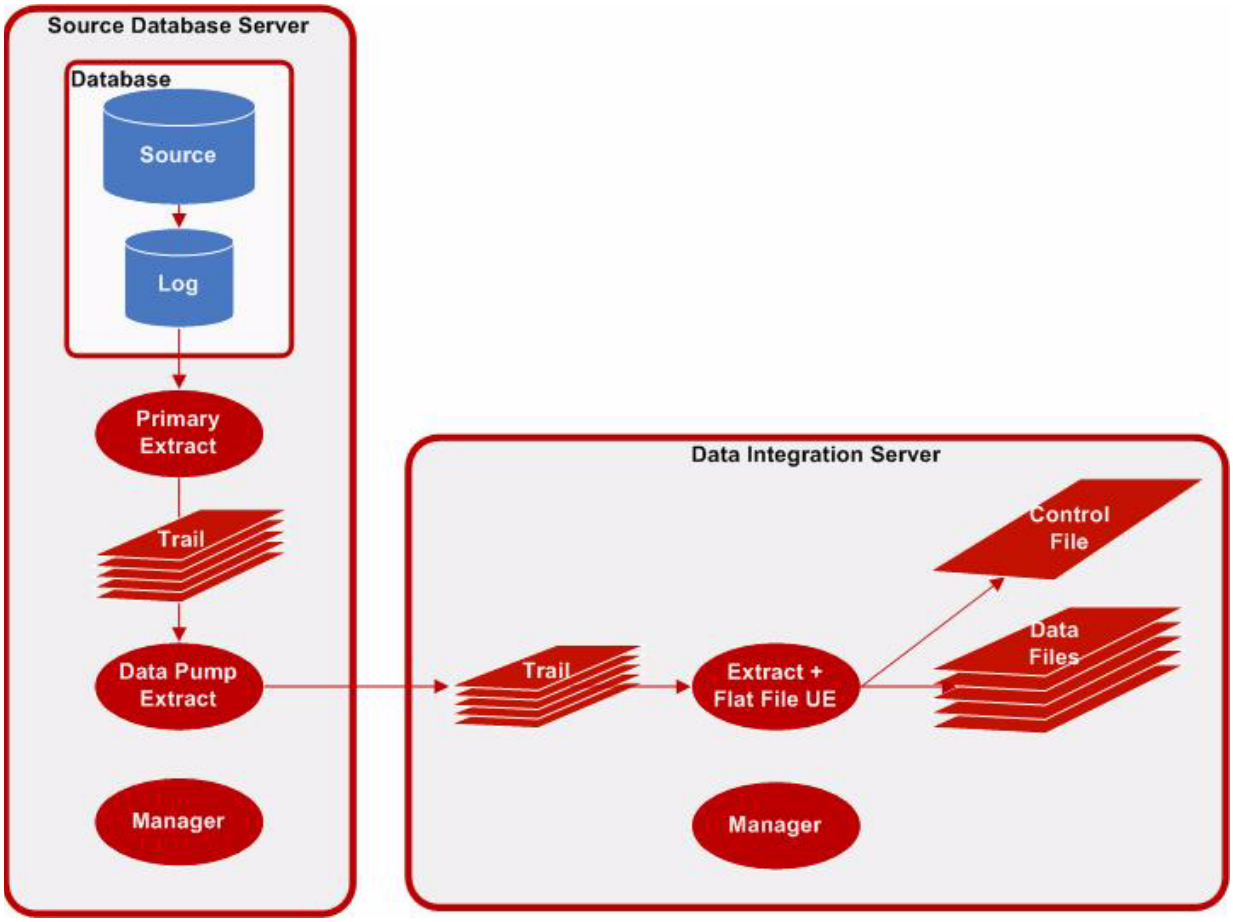

Figure 1 Oracle GoldenGate for Flat File

Oracle GoldenGate for Flat File is implemented as a user exit provided as a shared library (.so or .dll) that integrates into the Oracle GoldenGate Extract process.

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, or
- One file per table

The user exit can roll over based on time and/or size criteria and flushes files and maintains checkpoints whenever Oracle GoldenGate checkpoints to ensure recovery. In addition, it writes a control file containing list of rolled over files for synchronization with Data Integration Product and can also produce a summary file for use in auditing. Additional properties control formatting (delimiters, other values), directories, file extensions, meta columns (such as table name, file position, etc.) and data options.

Oracle GoldenGate Documentation

For information on installing and configuring the core Oracle GoldenGate software, see the Oracle GoldenGate documentation:

- *Oracle GoldenGate Installation and Setup Guides*: One for each database supported.
- *Oracle GoldenGate Administration Guide*: Introduces Oracle GoldenGate components and explains how to plan for, configure, and implement Oracle GoldenGate.
- *Oracle GoldenGate Reference Guide*: Provides detailed information about Oracle GoldenGate parameters, commands, and functions.
- *Oracle GoldenGate Troubleshooting and Performance Tuning Guide*: Provides suggestions for improving the performance of Oracle GoldenGate in different situations, and provides solutions to common problems.

These manuals are available for download from http://support.goldengate.com (support login ID and password required).

The Oracle GoldenGate website (http://www.goldengate.com) has white papers, web seminars and user conference information. The support site's Knowledge Base is an online resource containing the latest information on bug fixes and configuration notes.

CHAPTER 2

Installing, Configuring and Running the Oracle GoldenGate for Flat File

Installation

The Oracle GoldenGate for Flat File comes prebuilt for a particular platform.

The Oracle GoldenGate installation should be full installed and tested prior to installation of the user exit code. The user should have the necessary file permissions to write output data to the required output directories.

Oracle GoldenGate for Flat File is shipped:

- On Windows as a zip file
- On UNIX as a .tar.gz file

To install, unzip / gunzip; tar xvf the file in the Oracle GoldenGate install directory. The file contains:

- Shared library (flatfilewriter.dll on windows, flatfilewriter.so on UNIX)
- Sample user exit properties file (ffwriter.properties)
- Sample Extract parameter file (ffwriter.prm)

Configuration

Figure 2 shows a typical Oracle GoldenGate for Flat File configuration.

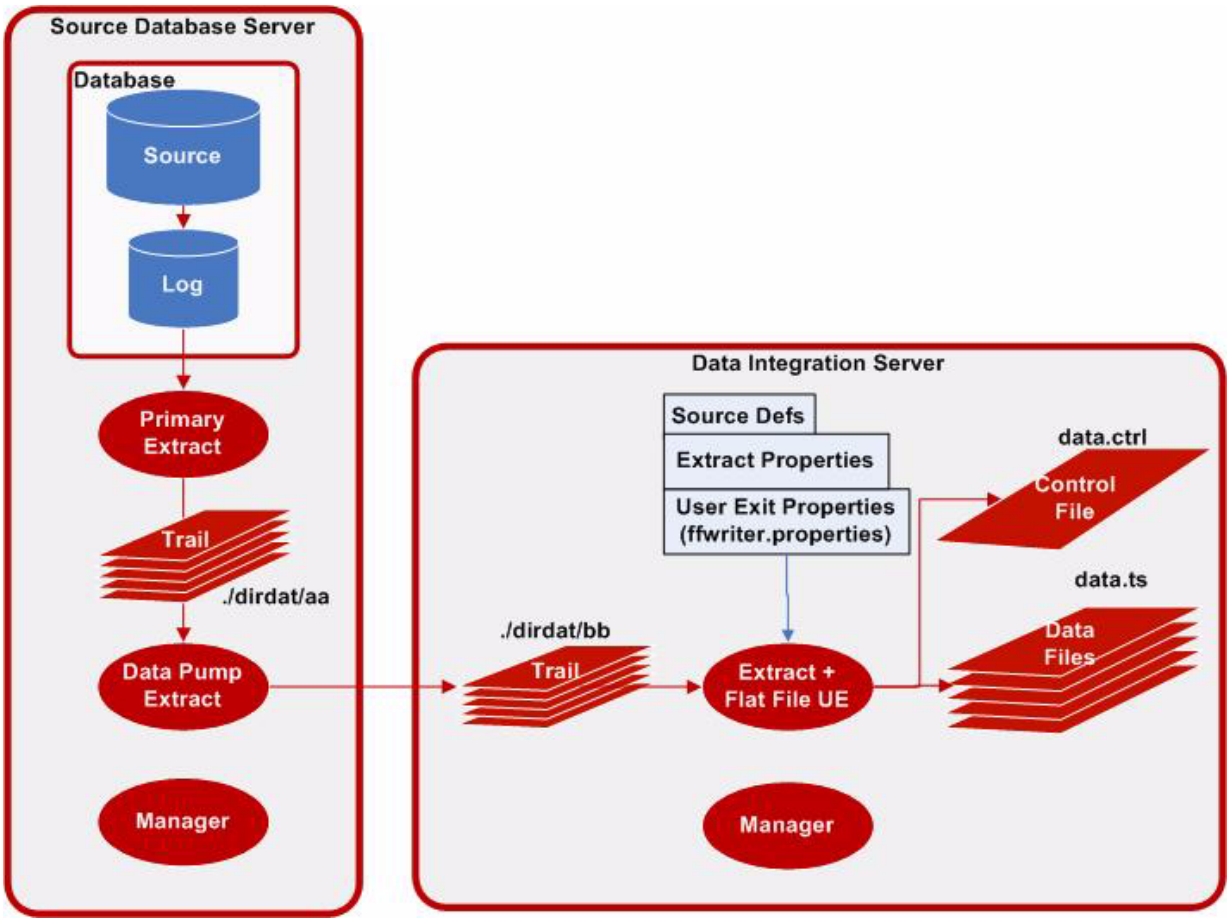

Figure 2 Typical Configuration

Source database system configuration:

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

Data integration system configuration:

```
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 need to be 8 characters or less, trail names need to be two characters.

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 name
`SOURCEDEFS .\dirdef\hr_ora.def`	A source defs file is required to determine trail contents

`CUSEREXIT .\flatfilewriter.dll`
`CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES`
`PARAMS <filename>`
	The location of the user exit .dll or .so library and function to call.
	`PASSTHRU` removes the need for a dummy 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.*;`	A list of tables to process

User Exit Properties

The user exit reads properties from a file named in the CUSEREXIT PARAMS (else defaults to ffwriter.properties).

The properties file contains full details of how the user exit should operate. For more information on individual properties see "User Exit Properties" on page 11.

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.

Operation

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
```

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

The data written by each writer can be in one or 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 on rollover of the file.

The output directory (for data files and control files separately), temporary extension, rolled extension, control extension and statistical summary extension are all configurable through properties. For output configuration details see “Output File Properties” on page 13.

Each data file 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_2007-08-03_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_2007-08-03_11:30:00_data.dsv
```

In addition to the basic data contents, additional *meta 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 meta columns see “Meta Columns” on page 17.

The contents of the data file depend on the mode, the input data, and the various properties determining which (if any) meta 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 User Exit Properties.

Working with Control Files

Control files contain information about which data files have been rolled over. If a data 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 output file will be created. If the writer is outputting to one file per table, a control file will also be created per table.

The generation of a control file, its output directory, prefix and extension are controlled by the properties defined in “Output File Properties” on page 13.

Each control file will contain a comma-delimited list of data files that have been rolled over since the control file was created, in the order that the files 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.

Working with Statistical Summaries

In addition to control files, summary statistics about the data production process can also be obtained. This statistically summary information can be written to the Oracle GoldenGate report file and/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 summary files, a summary file will be created for each rolled over file. This file contains information pertaining to that particular file in a delimited fashion. The extension of this file, the data to be output, the delimiter and line delimiter can all be controlled.

“Statistics and Reporting” on page 21 contains detailed property information about statistics and summary files.

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 Administration Guide*.

Handling Errors

There are three types of errors that could 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 goldengate.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" on page 24 contains more information on error handling.

User Exit Properties

All properties in the property file are of the form:

```
fully.qualified.name=value
```

The value may be a single string, integer, or boolean, or could be comma-delimited strings. Comments can be entered in to 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, which is useful in testing configurations without losing previous property settings.

Logging Properties

Logging is controlled by the following properties.

`goldengate.log.logname`

Takes any valid ASCII string as the prefix to the log file name. The log file produced has the current data appended to it in yyyymmdd format, together with the .log extension. For example:

```
# log file prefix
goldengate.log.logname=ffwriter
```

This would produce a log file of name ffwriter_20070803.log on August 3rd 2007. The log file will roll over each day independent of the starting / stopping of processes.

goldengate.log.level
Sets the overall log level of the logging module for all modules. 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, but would not impeded performance of the data path. If the level is switch to DEBUG, large volumes of messages may occur which could impede performance. For example:

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

Sets the global logging level to INFO.

goldengate.log.tostdout
goldengate.log.tofile
Boolean properties that determine if logged messages are written to stdout, and / or to the specified log file respectively. Most Oracle GoldenGate processes run as background processes, so stdout is generally not a useful option. For example:

```
# output params, to stdout and/or to file
goldengate.log.tostdout=false
goldengate.log.tofile=true
```

goldengate.log.modules
goldengate.log.level.[MODULENAME]
For advanced debugging use only, the log level of individual source modules that comprise the user exit can be overridden independently. The default levels should not be changed unless prompted to do so by a support engineer.

It is possible to increase the logging level to DEBUG on a per module basis to help troubleshoot issues.

Multiple Writer Properties

goldengate.flatfilewriter.writers
Mutiple ASCII values, enabling 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 equals or the commas. All other properties in the file should be prepended by one of the writer names.

`goldengate.userexit.chkptprefix`
ASCII value representing the prefix to be added to the checkpoint file name. It is indicated that when running multiple data pumps the checkpoint prefix be set to the name of the process. For example:

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

Output Format Properties

`mode`
ASCII value with two options, dsv and ldv corresponding to the two supported output formats:

- **DSV** – Delimiter Separated Values, e.g.
  POSITIONÇOPCODEÇTIMESTAMPÇCOLVALAÇCOLVALBÇETC

  NOTE The delimiter does not have to be a comma, so not just CSV.

- **LDV** – Length Delimited Values, e.g.
  0109TIMESTAMPI302MY05TABLEP042000P03ETC

  NOTE Lengths can be ASCII or binary, some meta columns can be fixed length (see "Meta Columns" on page 17) and this format will support unicode multi-byte data.

For example:

```
dsvwriter.mode=dsv
binarywriter.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.

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.

`files.onepertable`
Boolean value (true/false), defaults to true if not present. Determines whether all data is split over multiple files, one per table in the input data, or all data is written to one file. For example:

```
# output all data to one file, or to create one file per table
dsvwriter.files.onepertable=true
binarywriter.files.onepertable=false
```

`files.prefix`
ASCII value that specified prefix to be used for data files and control files. Only applies if the writer is not in one per table mode (files.onepertable=false). 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”. For example:

```
dsvwriter.files.prefix=test_
```

files.data.rootdir
files.data.ext
files.data.tmpext
All ASCII values. These properties determine the location and extension of all data files. Before rollover the files will have the tmpext extension, after rollover they will have the ext extension. The extension does not have to be just a .ext format, it can contain additional characters appended to the file name before the extension to differentiate the data output. The user 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
dsvwriter.files.data.rootdir=./out
# determine the extension for data files when rolled over
dsvwriter.files.data.ext=_data.dsv
# determine the extension for data files before rolling over
dsvwriter.files.data.tmpext=_data.dsv.temp
```

files.control.use
files.control.rootdir
files.control.ext
Use is a Boolean (true/false) value, defaulting 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
dsvwriter.files.control.use=true
# specify the extension to use for control files
dsvwriter.files.control.ext=_data.control
# directory in which to place control files, defaults to data directory
dsvwriter.files.control.rootdir=./out
```

NOTE Directory paths can use UNIX style ./ paths on Windows

files.formatstring
The filename format string 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 the difference being the following placeholders that can be added to the filename:

%s = schema

%t = table

%n = seqno (The format of the seqno can be specified as well, i.e. %05n which means a length of 5 digits will be displayed, padded with 0’s).

%d = timestamp

The seqno starts at 0 and will be incremented by 1 each time a file rolls over. The seqno is stored as a long int, as a result the maximum value is platform depended (for example on

a 64 bit machine the largest value would be 2^64-1).

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

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

File Rollover Properties

The following properties determine the policies for rolling over files.

`files.data.rollover.time`
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
dsvwriter.files.data.rollover.time=10
```

`files.data.rollover.size`
The maximum number of bytes that must be written to the file before the file is rolled over. For example:

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

`files.data.norecords.timeout`
The maximum number of seconds of elapsed time that must pass without any data written to a file before the file is rolled over. Defaults to 120 seconds or 2 minutes. For example:

```
# timeout to rollover in case no records for a period of time
dsvwriter.files.data.norecords.timeout=10
```

`files.rolloveronshutdown`
Boolean (true/false) value determining the policy for rollover 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:

```
# rollover non-empty and delete all empty files when Extract stops
binarywriter.files.rolloveronshutdown=true
```

> **NOTE** You can use time and/or size. If you use both, the first reached will cause a rollover. The timeout 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.

`csvwriter.writebuffer.size`
The write buffer chunk size. Use to reduce the number of system write calls. For example:

```
csvwriter.writebuffer.size=36863
```

Data Content Properties

The following properties determines which data is written to the data files. These

properties are independent of the format of the output data.

rawchars
Boolean (true/false) value, defaults to false. Determines whether character data is output as ASCII, or its original binary form is retained. 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)
dsvwriter.rawchars=false
binarywriter.rawchars=true
```

includebefores
Boolean (true/false) value, defaults to false. Determines whether, for update operations, both the before and after image of data is included in the output. 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
dsvwriter.includebefores=true
```

Producing …"VAL_B4_1","VAL_1","VAL_B4_2","VAL_2"…

includecolnames
Boolean (true/false) value, defaults to false. Determines whether column names are output before the column values. For example:

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

Producing …"COL_1","VAL_1","COL_2","VAL_2"…

omitvalues
Boolean (true/false) value, defaults to false. Determines whether column values are omitted in the output files. For example:

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

Producing …"COL_1","COL_2"…, if includecolnames is also set to true

diffsonly
Boolean (true/false) value, defaults to false. Determines whether all columns are output, or only those where the before image is different from the after image. 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)
dsvwriter.diffsonly=true
```

Producing …"VAL_1",,,"VAL_4",,,"VAL_7"…

omitplaceholders
Boolean (true /false) value, defaults to false. Determines whether delimiters / lengths are included in the output for missing columns. 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 diffsonly is true, values that are not different are said to be missing. For example:

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

Changing: …"VAL_1",,,"VAL_4",,,"VAL_7"… to …"VAL_1","VAL_4","VAL_7"…

Meta Columns

Meta 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. Valid meta columns are:

position - A unique position indicator of records in a trail

opcode - I, U or D for Insert, Update and Delete records

txind - Kind of record in a transaction (0 - begin, 1 - middle, 2 - end, 3 - whole)

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 param file

getenv - A GETENV value as documented in the *Oracle GoldenGate Reference Guide*; for example $GGHEADER.OPCODE

The ASCII values for opcode and txind can be overridden. For LDV metacols can be variable or fixed length. Position can be written in hex or decimal and any metacol can be the internal value or read from a column from the original data.

The following properties apply to meta columns.

metacols
Multiple ASCII values separated by commas. The values are the names of the meta columns to output in the order of output, as defined in the previous section. For example:

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

metacols.[metacol-name].fixedlen
Integer value determining the length of data to write for a meta column. This value is currently only used in LDV format. 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
dsvwriter.metacols.timestamp.fixedlen=23
```

Truncating: 2007-08-03 10:30:51.123456 to 2007-08-03 10:30:51.123

metacols.[metacol-name].column
ASCII value. Column name of data values to use instead of the internal value for a meta 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
dsvwriter.metacols.timestamp.column=MY_TIMESTAMP_COL
```

metacols.[token-name].novalue.chars/code
ASCII values for chars, hex values for code. These values represent characters or hex code to be used when the desired token value is not available. The default value is [NO VALUE]. For example:

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

metacols.[metacol-name].fixedjustify
ASCII value that can be set to "left" or "right". Meta Column value will be justified accordingly. By default all Meta Columns will be justified to the left. For example, to justify a token to the right:

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

metacols.[metacol-name].fixedpadchar.chars/code
ASCII values for chars, hex values for code. These values represent characters or hex code to be used in padding a meta column. The default character used for padding is a space " ". For example:

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

metacols.opcode.insert.chars/code
metacols.opcode.update.chars/code
metacols.opcode.delete.chars/code
ASCII values for chars, hex values for code. These values enable the default characters used to identify insert, update and delete operations to be overridden to other characters specified as characters or hex code. The default values are I for Insert, U for Update, D for delete. For example:

```
# op code values are overridden
dsvwriter.metacols.opcode.insert.chars=INS
dsvwriter.metacols.opcode.update.chars=UPD
dsvwriter.metacols.opcode.delete.chars=DEL
```

metacols.txind.begin.chars/code
metacols.txind.middle.chars/code
metacols.txind.end.chars/code
metacols.txind.whole.chars/code
ASCII values for chars, hex values for code. These values enable the default characters used to identify the beginning, middle and end of transactions, or if an operation is a whole transaction to be overridden to other characters specified as characters or hex code. The default values are 0 for Begin, 1 for Middle, 2 for End and 3 for Whole. For example:

```
# tx indicator values are overridden
dsvwriter.metacols.txind.begin.chars=B
dsvwriter.metacols.txind.middle.chars=M
dsvwriter.metacols.txind.end.chars=E
dsvwriter.metacols.txind.whole.chars=W
```

metacols.position.format

ASCII value of either dec or hex. Determines the output format of the position meta column. If hex 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)
dsvwriter.metacols.position.format=dec
```

Producing: `120000012345` for seqno 12, rba 12345

```
binarywriter.metacols.position.format=hex
```

Producing: `0000000c00003039` for seqno 12, rba 12345

DSV Specific Properties

DSV files have the following record format:

$\{[METACOL][FD]\}^{n}\{[COL][FD]\}^{m}[LD]$

Where:

METACOL is any defined meta column

COL is any data column

FD is the field delimiter

LD is the line delimiter

Column values may be quoted, e.g. `"2007-01-10 10:20:31","U","MY.TABLE", 2000,"DAVE"`

`dsv.nullindicator.chars/code`
ASCII values for chars, hex values for code. These values enable the indicator used for NULL column values to be overridden – the default is an empty string. For example:

```
# define the characters to use for NULL values in delimiter separated
files
dsvwriter.dsv.nullindicator.chars=NULL
dsvwriter.dsv.nullindicator.code=0a0a0a0a
```

`dsv.fielddelim.chars/code`
ASCII values for chars, hex values for code. These values enable the value used for the field delimiter to be overridden – the default is a comma (,). For example:

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

`dsv.linedelim.chars/code`
ASCII values for chars, hex values for code. These values enable the value used for the line delimiter to be overridden – the default is a newline appropriate to the operating system.

For example:

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

dsv.quote.chars/code
ASCII values for chars, hex values for code. These values enable the value used for the quote character to be overridden – the default is a double quote (“). For example:

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

dsv.nullindicator/fielddelim/linedelim/quotes.escaped.chars/code
ASCII values for chars, hex values for code. These values enable the escaped value for any of the above values to be set. If set, all values will be checked for the none escaped value and replaced with the escaped value when output. For example:

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

Changes: “some text” to ““some text““.

dsv.onecolperline
Boolean (true/false) value. Defaults to false. If true, forces each column value onto a new line. Each line will also contain the meta columns defined for this writer. For example:

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

Changes:`{metacols},val_1,val_2` to

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

dsv.quotealways
Boolean (true/false) value. Defaults to false. If true, forces each column to be surrounded by quotes, even if it is a numeric value. For example:

```
# Force column values, even if numeric, to be surrounded by quotes:
dsvwriter.dsv.quotealways=true
```

Changes: `…,1234,”Hello”,10` to

```
…,”1234”,”Hello”,”10”
```

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 meta 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
```

ldv.vals.missing.chars/code
ldv.vals.present.chars/code
ldv.vals.null.chars/code
ASCII values for chars, hex values for code. These values enable the values for the missing, present and null indicators to be overridden. For example:

```
binarywriter.ldv.vals.missing.chars=MI
binarywriter.ldv.vals.present.chars=PR
binarywriter.ldv.vals.null.chars=NL
```

ldv.lengths.record.mode
ldv.lengths.field.mode
ASCII values, can be either binary or ASCII, defaults to binary. These values determine the output mode of record and field lengths. 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:

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

ldv.lengths.record.length
ldv.lengths.field.length
Integer values representing the record and field lengths. 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 exits 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)
binarywriter.ldv.lengths.record.length=4
binarywriter.ldv.lengths.field.length=2
```

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, the total records processed, broken down to inserts, updates, deletes; records processed per table, also broken down; total rate / rate per table; 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 a delimited format with the above data, but related to the contents of a particular data file that can be used by a data integration product to cross-check processing. It will have the same name as the data

file, but different extension.

statistics.toreportfile
Boolean (true/false) value which, if true, will output statistics to the Oracle GoldenGate report file. For example:

```
binarywriter.statistics.tosummaryfile=true
```

statistics.period

> **NOTE** These values are valid only for outputting statistics to the report file. Statistics will be outputted to the summary file only on rollover.

ASCII value can be either timebased or onrollover. If timebased, the time period should be set in statistics.time. For example:

```
dsvwriter.statistics.period=onrollover
dsvwriter.statistics.period=timebased
```

statistics.time
A time interval in seconds after which statistics will be reported. For example:

```
binarywriter.statistics.time=5
```

statistics.tosummaryfile
Boolean (true/false) value which, if true, will create a summary file containing statistics for each data file on rollover. For example:

```
binarywriter.statistics.tosummaryfile=true
```

statistics.summary.fileformat
Multiple comma separated ASCII values. The values determine the content of the summary files, and the order in which the content is written. The fileformat can contain the following values:

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 – mix/max commit timestamp for the specified table since user exit was started

ctimestamp – min/max 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:

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

Boolean (true/false) value, which if true will result in an additional row being written to the summary files containing the overall (across all tables) statistics defined by the user via the statistics.summary.fileformat property. For example:

```
binarywriter.statistics.overall=true
```

statistics.summary.delimiter.chars/code
statistics.summary.eol.chars/code
ASCII values for chars, hex values for code. These values enable the values for the field delimiter and end of line delimiter to be overridden for the summary files. Defaults to a comma ',' delimiter and newline character. For example:

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

statistics.summary.extension
ASCII value, defaults to stats. Enables the extension to use for the statistics summary file output per data file to be changed. For example:

```
#binarywriter.statistics.summary.extension=.statistics
```

Troubleshooting

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*.

In addition, check the following:

1. Is the shared library (.so or dll) in the extract.prm parameter file correct, in the path specified, and accessible?
2. Is the sourcedefs file specified in the extract.prm parameter file correct, in the specified path, and accessible?
3. Does the sourcedefs file contain all the necessary tables?
4. 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?
5. Do the output directories specified in the user exit properties file exist?
6. Are file permissions correct to write to that directory?

Now check the user exit log file: logname_yyyymmdd.log

7. Does the user exit properties file parse successfully? Any invalid properties mentioned in the log file?
8. Any other errors or warnings in the log?

If the problem is still not resolved:

9. Set goldengate.log.level=DEBUG
10. Restart and save the log file

Before contacting Oracle Support, be prepared to send the log file, source trail file, sourcedefs file, user exit properties file and Extract parameter file, together with any data files that have been written.

CHAPTER 3
Usage Example: Netezza Integration

Netezza is a data warehouse appliance optimized for data scans through vast amounts of data. Netezza consists of proprietary database software running on proprietary hardware. No other products or utilities should run on the Netezza environment.

This chapter describes a test configuration that can load data changes (in all supported data types) into a Netezza database using Oracle GoldenGate for Flat File.

Loading Data into Netezza

Netezza provides a client-side data load utility, nzload, that loads data in batch. This is the bulk load method with the least impact on the Netezza system.

Because of Netezza's optimizations for query retrieval, concurrent DML may significantly slow down the system. Inserts are least intrusive to system performance. The recommended approach is to move data into staging tables and use Netezza's parallel SQL capabilities in batch mode to apply changes from the staging tables to the target tables. Figure 3 shows the incremental load into staging tables.

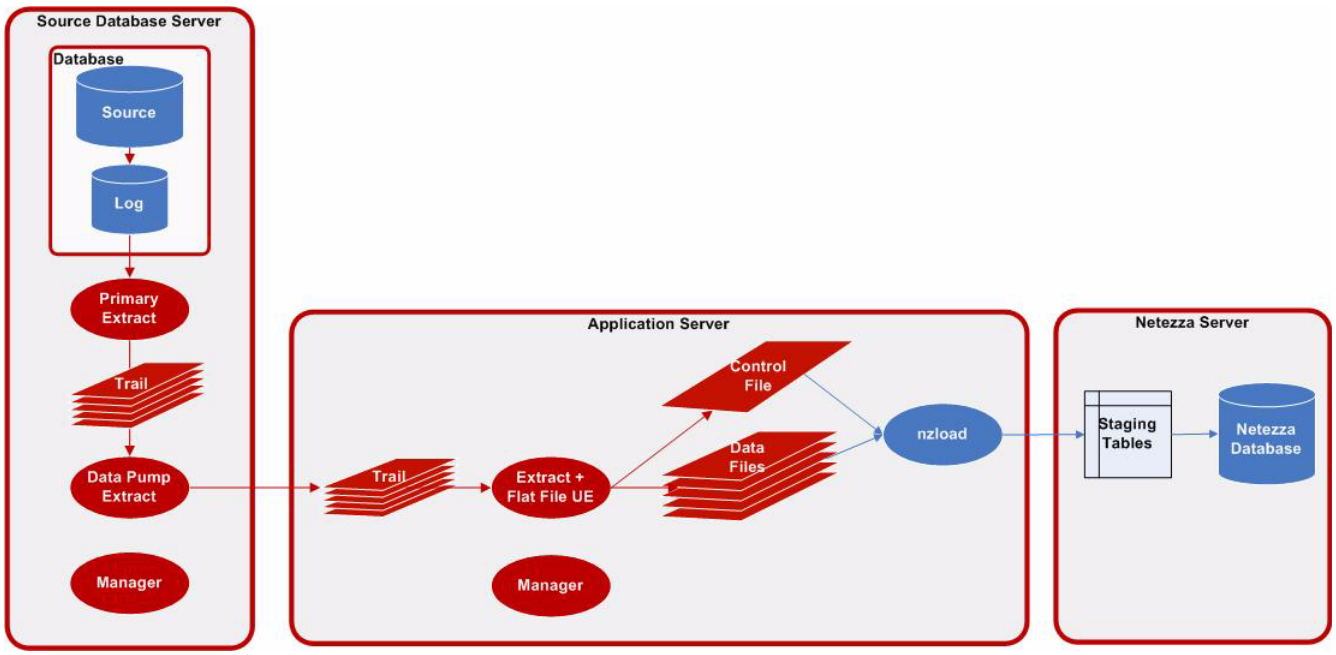

Figure 3 IncrementalLoad into a Netezza Database.

The example described below uses a single table in an Oracle database as the source and a single staging table in Netezza as the target. The example uses all supported data types so can be used as a template for any data loads targeting Netezza.

Connection to Netezza

As part of the initial setup process you connect to the Netezza server using a client application. Netezza supports ODBC using its own ODBC driver. This example assumes an initial connection from a Microsoft Windows environment, but other environments for which ODBC drivers are available should work similarly.

The installation process for the Netezza ODBC driver adds a System DSN entry in Windows. Once the driver has been installed you configure it to connect to the Netezza server. Figure 4 shows an example of the configuration.

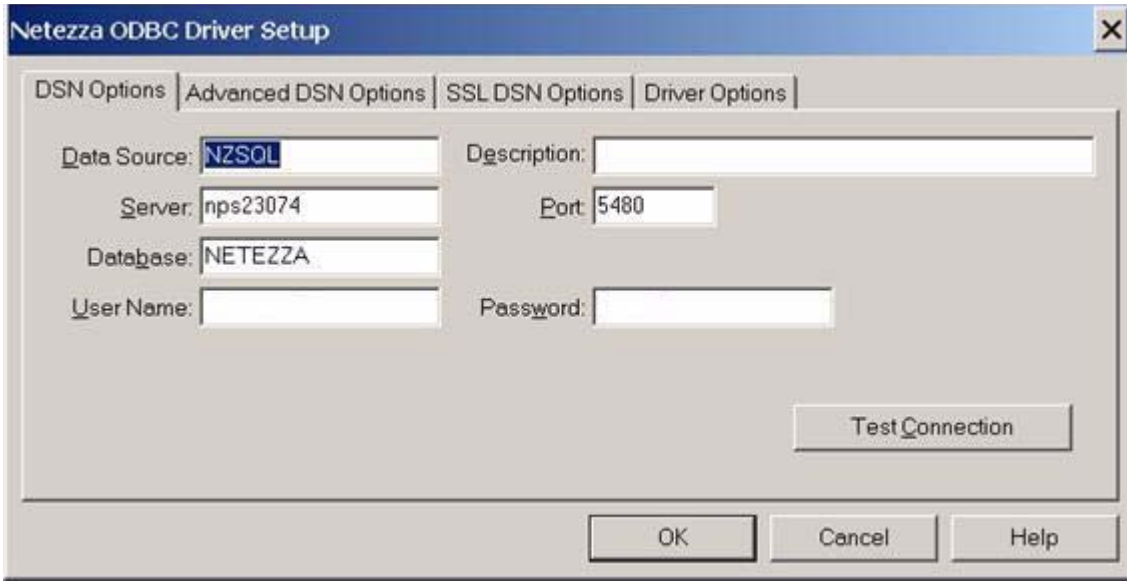

Figure 4 Netezza ODBC driver configuration

The sample configuration in Figure 4 uses NZSQL as the data source name for ODBC. You must use the same name for SOURCEDB/TARGETDB when configuring Oracle GoldenGate to connect to Netezza. NETEZZA is the database name on the Netezza system. Username and password are not stored with the DSN but are needed to test whether the connection works.

Supported Data Types

Netezza supports 19 different data types that can be categorized into 4 broad categories:

- Numeric
- Character
- Date/time
- Boolean

Depending on the data values one data type is stored more efficiently than another. Storing a column in one data type versus another may result in terabytes of storage costs/savings in a data warehouse environment with possibly billions of rows in a single table.

The following data types are supported:

- Numeric

  BYTEINT

  SMALLINT

  INTEGER

  BIGINT

  NUMERIC

  DECIMAL

  FLOAT

  REAL

  DOUBLE PRECISION

- Character

  CHAR

  VARCHAR

- Boolean

  BOOLEAN

The following datatypes may require workarounds:

- Date/time

  DATE: Oracle's DATE data type includes time. If in Netezza you define DATE and TIME and in Oracle there is just DATE then you can use 2 calls to the @DATE function to convert the date and time components into the Netezza columns.

  TIME: The Netezza TIME data type does not include decimal fractions down to microseconds. If you need this precision, map these columns to TIMESTAMP.

  TIMESTAMP: The default TIMESTAMP format written from Oracle has a colon between date and time portions. You must reformat the TIMESTAMP data without the colon using the @DATE function to enable direct load into the Netezza TIMESTAMP data type.

The following data types are currently not supported:

- Character

  The filewriter does not currently support multibyte character (NCHAR, NVARCHAR) data types when writing in ASCII format (which nzload requires as input).

- Date/time

  INTERVAL: nzload version 4.5 does not support the INTERVAL data type.

  TIME WITH TIME ZONE: Oracle GoldenGate Extract does not support time zone extraction from Oracle. TIME WITH TIME ZONE can probably supported by combining/reformatting one or more source columns.

Table Definitions

The following table is the staging table definition for Netezza.

```
create table nz_datatypes_stage
( c1_byteint byteint
, c2_smallint smallint
, c3_integer integer
, c4_bigint bigint
, c5_numeric_7_2 numeric(7,2)
, c6_numeric_9 numeric(9)
, c7_numeric numeric
, c8_decimal decimal
, c9_float float
, c10_real real
, c11_double_precision double precision
, c12_char_100 char(100)
, c13_varchar_100 varchar(100)
, c14_boolean boolean
, c15_date date
, c16_time time
, c17_timestamp timestamp
);
```

If you have to apply operations other than inserts to the target Netezza tables, then you have to include at least the operation type and commit timestamp in order to apply the changes to the target tables in the correct order.

DML on the staging tables will always be inserts only. You can process other DML operations upon applying the data changes from the staging tables to the actual target tables.

Below is the equivalent Oracle table that is used as the source table to load the Netezza staging table. Check constraints are used to ensure data values stay within the limits of

the Netezza data types.

```
create table nz_datatypes
( c1_byteint number
    , check (c1_byteint between -128 and 127)
    , check (trunc(c1_byteint)=c1_byteint)
, c2_smallint number
    , check (c2_smallint between -32768 and 32767)
    , check (trunc(c2_smallint)=c2_smallint)
, c3_integer number
    , check (c3_integer between -2147483648 and 2147483647)
    , check (trunc(c3_integer)=c3_integer)
, c4_bigint number
    , check (trunc(c4_bigint)=c4_bigint)
, c5_numeric_7_2 number(7,2)
, c6_numeric_9 number(9)
, c7_numeric number(18,0)
, c8_decimal number(18,0)
, c9_float number(15)
, c10_real number(6)
, c11_double_precision number(15)
, c12_char_100 char(100)
, c13_varchar_100 varchar2(100)
, c14_boolean varchar2(1)
    , check(c14_boolean in ('T','F'))
, c15_date date
    , check(c15_date = trunc(c15_date))
, c16_time timestamp
, c17_timestamp timestamp
);
```

In this example, the source and target schemas are the same, so we can use the target definitions as our source definitions. Use DEFGEN with Oracle GoldenGate to retrieve the table definitions from the Netezza server. The DEFGEN parameter file is shown below:

```
DEFSFILE dirdef\nzdatatp.def, PURGE
SOURCEDB NZSQL, USERID myusername, PASSWORD ********
TABLE <owner>.NZ_DATATYPES_STAGE;
```

Retrieve the definitions and copy the definition file to Oracle GoldenGate on the source system.

Notes:

1. If the source and target schemas are different, you must run DEFGEN against the source to get the source definitions.
2. NZSQL is the name of the SYSTEM DSN in Windows. If you used a different name, then you use it here.
3. Through the Netezza ODBC driver, all user tables always show up as owned by the user who created the database (rather than the user who created the table). This may be a problem in Netezza's ODBC driver because in the data dictionary table/view _v_table tables are owned by the user who created the table. If you cannot identify the

table owner you should use, run the following query against the Netezza database using the nzsql command-line utility on the Netezza server (nzsql is the Netezza equivalent of Oracle's SQL Plus).

```
select distinct table_schema
from _v_odbc_columns2
where table_cat = current_catalog
and table_name like '<table name>' ;
```

Use the outcome of this query for the table owner in the definitions file and in the Extract when defining the mapping.

Primary Extract

The first step in the extract and load process is to set up the primary Extract process. Below are sample Extract parameters. The Oracle table was created in a schema sh and the ORACLE_SID name is src.

```
EXTRACT nzdatatp
SETENV ORACLE_SID=src
USERID sh, PASSWORD ******
EXTTRAIL dirdat\nz
TARGETDEFS dirdef\nzdatatp.def


MAP sh.nz_datatypes, TARGET <owner>.NZ_DATATYPES_STAGE
,COLMAP (USEDEFAULTS
,c17_timestamp=@DATE ("YYYY-MM-DD HH:MI:SS.FFFFFF"
,"YYYY-MM-DD:HH:MI:SS.FFFFFF",c17_timestamp)
);
```

The @DATE function reformats the source timestamp column into a timestamp format that the nzload utility can read. The filewriter does not currently reformat dates as it reads them out of the trail.

For incremental updates login to GGSCI, add supplemental logging to the source table(s), add the Extract and the trail.

If the source application performs updates and deletes, then you have to consider how you want to handle these in Netezza. If you want to store a history of changes (to store a full history – slowly changing dimension type 2 – is not uncommon for a data warehouse application) then it may be easier to enable supplemental logging for all columns you want to capture rather than just the columns in the primary/unique key. That way down-stream processing from the staging tables into the actual target tables will be a lot simpler and likely perform better. If there are updates and you want to process updates in a particular way then you may require before updates. Add the `GETUPDATEBEFORES` parameter if you require the before image of updates.

If your source database is remote from the application server that is going to populate Netezza (as shown in Figure 3 on page 25), then configure one or more pumps to transfer the trails to the application server that will run Oracle GoldenGate to produce the flat files and run the nzload utility. This example does not describe the use of the pumps but is not different from a typical Oracle GoldenGate implementation.

Filewriter Configuration

The next step is to use the filewriter to read the trail files and output a character delimited file. Extract will call the filewriter to output trail files to separated files. Use the Oracle GoldenGate version for the source database you used (in this example, Oracle).

Filewriter Properties

The filewriter is controlled by a properties file. The most important properties for a production environment are:

```
# true/false flag to determine whether to output all data to one
# file, or to create one file per table
dsvwriter.files.onepertable=true

# number of seconds before rolling over
dsvwriter.files.data.rollover.time=100

# max file size in KB before rolling over
dsvwriter.files.data.rollover.size=100000

# additional timeout to rollover in case no records for a period
# of time
dsvwriter.files.data.norecords.timeout=10000
```

These parameters determine the way the files are created and with that how often incremental data can be loaded into the target database staging tables. You must create one file per table in order to enable direct data loads using the nzload utility.

The filewriter supports the creation of a control file for the data files. Once a data file is ready for processing (either because the data size limit was reached or because the timeout was reached) the temporary data file is renamed to its final name and the name of the file is written to the control file. A daemon can be used to monitor the control file and start a data load for the data files as file names are added to the control file.

Using the filewriter you can include metadata columns at the beginning of the record:

position – A unique position indicator of records in a trail

opcode – I, U or D for Insert, Update and Delete records

txind – Kind of record in a transaction (0 - begin, 1 - middle, 2 - end, 3 - whole)

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

Include at least `opcode` and `timestamp` if you have to process any operations other than inserts on the target tables in Netezza. The example described here does not use metadata columns.

Following is the filewriter properties that you would use for our example.

```
goldengate.log.logname=uelog
goldengate.log.level=INFO
goldengate.log.tostdout=false
goldengate.log.tofile=true
goldengate.log.modules=UEUTIL,LOGMALLOC,TXSTORE,UTILS,DSUSEREXIT,
FILEWRITER,CSVFILEWRITER,BINFILEWRITER
goldengate.flatfilewriter.writers=dsvwriter
goldengate.userexit.chkptprefix=shpump_
dsvwriter.mode=DSV
dsvwriter.rawchars=false
dsvwriter.includebefores=false
dsvwriter.includecolnames=false
dsvwriter.omitvalues=false
dsvwriter.diffsonly=false
dsvwriter.omitplaceholders=false
dsvwriter.files.onepertable=true
dsvwriter.files.data.rootdir=C:\datafiles
dsvwriter.files.data.ext=_data.dsv
dsvwriter.files.data.tmpext=_data.dsv.temp
dsvwriter.files.data.rollover.time=100
dsvwriter.files.data.rollover.size=100000
dsvwriter.files.data.norecords.timeout=10000
dsvwriter.files.control.use=true
dsvwriter.files.control.ext=_data.control
dsvwriter.files.control.rootdir=C:\datafiles\control
dsvwriter.dsv.nullindicator.chars=
dsvwriter.dsv.fielddelim.chars=;
dsvwriter.dsv.fielddelim.escaped.chars=
```

Filewriter Extract

The next step is to configure Extract to call filewriter. Here is an example of the Extract parameter file with a call to the filewriter dll on Windows:

```
EXTRACT wnzdtue
CUSEREXIT C:\ggora\flatfilewriter.dll CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES
SOURCEDEFS dirdef\nzdatatp.def
TABLE <owner>.NZ_DATATYPES_STAGE;
```

Extract writes to the flat files defined by the filewriter properties. The file name will be:

```
<table name>_<timestamp>_<suffix as defined in filewriter properties file>.dsv
```

For example: `<owner>.NZ_DATATYPES_STAGE_2008-05-14_11-37-01_data.dsv`

Using GGSCI, add the Extract, reading from an Extract trail.

Loading using nzload

Netezza's nzload utility is easy to use. The online help (nzload -h) is quite comprehensive. nzload can be called remotely (e.g. from the server hosting the source database) or run on

the Netezza database server.

To minimize the impact on the Netezza database, nzload utility should run on a separate application server with a fast connection to the Netezza database server (see Figure 3 on page 25). Trail files can be transferred to this application server using one or more data pumps. This application server can also run Oracle GoldenGate and the filewriter processing the trails.

When nzload is running remotely it does not require an ODBC connection. You can connect directly to the database using the host and database parameters referring to the database server that accepts the connections.

Following is an example of a call to nzload to load data from a data file:

```
nzload -host nps23074 -u myusername -pw ******* -db NETEZZA
-t NZ_DATATYPES_STAGE -df <owner>.NZ_DATATYPES_2008-05-13_11-35-50_data.dsv
-outputDir c:\TEMP -delim ";" -quotedValue DOUBLE -boolStyle T_F
```

The nzload utility supports basic error handling. By default, a single record failure will abort the transaction and no records will be applied. However if the row causing the failure was not at the beginning of the file, then space might have been allocated to insert the records into the target table. Use the nzreclaim utility to explicitly reclaim that space.

Index

O

P

R

S

T

W