Oracle® Fusion Middleware

Integrating Oracle GoldenGate for Big Data

Release 12.3.0.1

E79598-01

December 2016

ORACLE

Oracle Fusion Middleware Integrating Oracle GoldenGate for Big Data, Release 12.3.0.1

E79598-01

Contents

9 Using the Pluggable Formatters

10 Using the Metadata Provider

Preface

This book contains information about configuring and running Oracle GoldenGate for Big Data.

Audience

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

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 Information

The Oracle GoldenGate Product Documentation Libraries are found at

Oracle GoldenGate

Oracle GoldenGate Application Adapters

Oracle GoldenGate for Big Data

Oracle GoldenGate Director

Oracle GoldenGate Plug-in for EMCC

Oracle GoldenGate Monitor

Oracle GoldenGate for HP NonStop (Guardian)

Oracle GoldenGate Veridata

Oracle GoldenGate Studio

Additional Oracle GoldenGate information, including best practices, articles, and solutions, is found at:

Oracle GoldenGate A-Team Chronicles

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to GoldenGate for Big Data

This chapter provides an introduction to Oracle GoldenGate for Big Data concepts and features. It includes how to verify and set up the environment, use it with Replicat, logging data, and other configuration details. It contains the following sections:

1.1 Introduction

The Oracle GoldenGate for Big Data integrations run as pluggable functionality into the Oracle GoldenGate Java Delivery framework, also referred to as the Java Adapters framework. This functionality extends the Java Delivery functionality. Oracle recommends that you review the Java Delivery description in the *Administering Oracle GoldenGate Application Adapters*.

1.2 Understanding What is Supported

Oracle GoldenGate for Big Data supports specific configurations, the handlers are compatible with clearly defined software versions, and there are many support topics. This section provides all of the relevant support information.

Topics:

1.2.1 Verifying Certification and System Requirements

Make sure that you are installing your product on a supported hardware or software configuration. For more information, see the certification document for your release on the *Oracle Fusion Middleware Supported System Configurations* page.

Oracle has tested and verified the performance of your product on all certified systems and environments; whenever new certifications occur, they are added to the proper certification document right away. New certifications can occur at any time, and for this reason the certification documents are kept outside of the documentation libraries and are available on Oracle Technology Network.

1.2.2 Understanding Handler Compatibility

This section describes how each of the Oracle GoldenGate for Big Data Handlers are compatible with the various data collections including distributions, database releases, and drivers.

Topics:

1.2.2.1 HDFS Handler

The HDFS Handler is designed to work with the following versions :

Distribution	Version
Apache Hadoop	2.7.*x* 2.6.0 2.5.x 2.4.x 2.3.0 2.2.0 3.0.0-alpha 1
Hortonworks Data Platform (HDP)	HDP 2.5 (HDFS 2.7.3) HDP 2.4 (HDFS 2.7.1) HDP 2.3 (HDFS 2.7.1) HDP 2.2 (HDFS 2.6.0) HDP 2.1 (HDFS 2.4.0)
Cloudera Distribution Include Apache Hadoop (CDH)	CDH 5.8.*x* (HDFS 2.6.0) CDH 5.7.*x* (HDFS 2.6.0) CDH 5.6.*x* (HDFS 2.6.0) CDH 5.5.*x* (HDFS 2.6.0) CDH 5.4.*x* (HDFS 2.6.0) CDH 5.3.*x* (HDFS 2.5.0) CDH 5.2.*x* (HDFS 2.5.0) CDH 5.1.*x* (HDFS 2.3.0)

1.2.2.2 HBase Handler

Cloudera HBase 5.4.*x* and later did not fully adopt the Apache HBase 1.0.0 client interface so it is not fully in sync with the Apache HBase code line to provide reverse compatibility in that HBase client interface. This means that Cloudera HBase broke binary compatibility with the new HBase 1.0.0 interface resulting in `NoSuchMethodError` when integrating with the Oracle GoldenGate for Big Data HBase Handler. This can be solved one of the following two ways:

- Configure the HBase Handler to use the 0.98.x HBase interface by setting the HBase Handler configuration property, `hBase98Compatible`, to `true`.
- Alternatively, you can use the Apache HBase client libraries when connecting to CDH 5.4.*x* and later HBase.

The HBase Handler is designed to work with the following:

Distribution	Version
Apache HBase	0.98.*x* and 0.96.*x* when you set the `hBase98Compatible` property to true 1.2.*x*, 1.1.*x* and 1.0.*x*
Hortonworks Data Platform (HDP)	HDP 2.5 (HBase 1.1.2) HDP 2.4 (HBase 1.1.2) HDP 2.3 (HBase 1.1.1) HDP 2.2 (HBase 0.98.4) when you set the `hBase98Compatible` property to `true`.
Cloudera Distribution Including Apache Hadoop (CDH)	CDH 5.8.*x* (HBase 1.2.0) when you set the `hBase98Compatible` property to `true`. CDH 5.7.*x* (HBase 1.2.0) when you set the `hBase98Compatible` property to `true`. CDH 5.6.*x* (HBase 1.0.0) when you set the `hBase98Compatible` property to `true`. CDH 5.5.*x* (HBase 1.0.0) when you set the `hBase98Compatible` property to `true`. CDH 5.4.*x* (HBase 1.0.0) when you set the `hBase98Compatible` property to `true`. CDH 5.3.*x* (HBase 0.98.6) when you set the `hBase98Compatible` property to `true`. CDH 5.2.*x* (HBase 0.98.6) when you set the `hBase98Compatible` property to `true`. CDH 5.1.*x* (HBase 9.98.1) when you set the `hBase98Compatible` property to `true`.

1.2.2.3 Flume Handler

The Oracle GoldenGate for Big Data Flume Handler works with the Apache Flume versions 1.6.*x*, 1.5.*x* and 1.4.*x*. Compatibility with versions of Flume before 1.4.0 is not guaranteed.

The Flume Handler is compatible with the following versions:

Distribution	Version
Distribution: Apache Flume	Version: 1.7.*x*, 1.6.*x*, 1.5.*x*, 1.4.*x*
Hortonworks Data Platform (HDP)	HDP 2.5 (Flume 1.5.2) HDP 2.4 (Flume 1.5.2) HDP 2.3 (Flume 1.5.2) HDP 2.2 (Flume 1.5.2) HDP 2.1 (Flume 1.4.0)
Cloudera Distribution Including Apache Hadoop (CDH)	CDH 5.8.*x* (Flume 1.6.0) CDH 5.7.*x* (Flume 1.6.0) CDH 5.6.*x* (Flume 1.6.0) CDH 5.5.*x* (Flume 1.6.0) CDH 5.4.*x* (Flume 1.5.0) CDH 5.3.*x* (Flume 1.5.0) CDH 5.2.*x* (Flume 1.5.0) CDH 5.1.*x* (Flume 1.5.0)

1.2.2.4 Kafka Handler

The Kafka Handler is *not* compatible with Kafka version 8.2.2.2 and later.

The Kafka Handler is designed to work with the following:

Distribution	Version
Apache Kafka	0.9.0.*x* 0.10.0.0 0.10.0.1
Hortonworks Data Platform (HDP)	HDP 2.5 (Kafka 0.10.0) HDP 2.4 (Kafka 0.9.0)
Cloudera Distribution Including Apache Hadoop (CDH) does not currently include Kafka. Cloudera currently distributes Kafka separately as Cloudera Distribution of Apache Kafka	Cloudera Distribution of Apache Kafka 2.0.*x* (Kafka 0.9.0.0)
Confluent Platform	3.0.1 (Kafka 0.10.0.0) 2.0.0 (Kafka 0.9.0.0)

1.2.2.5 Cassandra Handler

The Cassandra Handler uses the Datastax 3.1.0 Java Driver for Apache Cassandra. This driver streams change data capture from a source trail file into the corresponding tables in the Cassandra database.

The HDFS Handler is designed to work with the following versions :

Distribution	Version
Apache Cassandra	1.2
	2.0
	2.1
	2.2
	3.0
Datastax Enterprise Cassandra	3.2
	4.0
	4.5
	4.6
	4.7
	4.8

1.2.2.6 MongoDB Handler

The MongoDB handler uses the native Java driver version 3.2.2. It is compatible with the following MongoDB versions:

- MongoDB 2.4
- MongoDB 2.6
- MongoDB 3.0
- MongoDB 3.2
- MongoDB 3.4

1.2.2.7 JBDC Handler

The JDBC handler internally uses generic JDBC API. Although it should be compliant with any JDBC complaint database driver we have certified the JDBC handler against the following targets:

- Oracle Database target using Oracle JDBC driver.
- MySQL Database target using MySQL JDBC driver.
- IBM Netezza target using Netezza JDBC driver.
- Amazon Redshift target using Redshift JDBC driver.

1.2.3 What are the Additional Support Considerations?

This section describes additional Oracle GoldenGate for Big Data Handlers additional support considerations.

Pluggable Formatters—Support
The handlers support the Pluggable Formatters as described in Using the Pluggable Formatters as follows:

- The HDFS Handler supports all of the pluggable handlers .

- Pluggable formatters are not applicable to the HBase Handler. Data is streamed to HBase using the proprietary HBase client interface.
- The Flume Handler supports all of the pluggable handlers described in Using the Pluggable Formatters .
- The Kafka Handler supports all of the pluggable handlers described in Using the Pluggable Formatters .
- The Cassandra , MongoDB, and JDBC Handlers do *not* use a pluggable formatter.

Avro Formatter—Improved Support for Binary Source Data

In previous releases, the Avro Formatter did not support the Avro bytes data type. Binary data was instead converted to Base64 and persisted in Avro messages as a field with a string data type. This required an additional conversion step to convert the data from Base64 back to binary.

The Avro Formatter now can identify binary source fields that will be mapped into an Avro bytes field and the original byte stream from the source trail file will be propagated to the corresponding Avro messages without conversion to Base64.

Avro Formatter—Generic Wrapper

The `schema_hash` field was changed to the `schema_fingerprint` field. The `schema_fingerprint` is a long and is generated using the `parsingFingerprint64(Schema s)` method on the `org.apache.avro.SchemaNormalization` class. This identifier provides better traceability from the Generic Wrapper Message back to the Avro schema that is used to generate the Avro payload message contained in the Generic Wrapper Message.

JSON Formatter—Row Modeled Data

The JSON formatter supports row modeled data in addition to operation modeled data.. Row modeled data includes the after image data for insert operations, the after image data for update operations, the before image data for delete operations, and special handling for primary key updates.

Java Delivery Using Extract

Java Delivery using Extract is *not* supported and was deprecated in this release. Support for Java Delivery is only supported using the Replicat process. Replicat provides better performance, better support for checkpointing, and better control of transaction grouping.

Kafka Handler—Versions

Support for Kafka versions 0.8.2.2, 0.8.2.1, and 0.8.2.0 was discontinued. This allowed the implementation of the flush call on the Kafka producer, which provides better support for flow control and checkpointing.

HDFS Handler—File Creation

A new feature was added to the HDFS Handler so that you can use Extract, Load, Transform (ELT). The new `gg.handler.name.openNextFileAtRoll=true` property was added to create new files immediately when the previous file is closed. The new file appears in the HDFS directory immediately after the previous file stream is closed.

This feature does not work when writing HDFS files in Avro Object Container File (OCF) format or sequence file format.

MongoDB Handler—Support

- The handler can only replicate unique rows from source table. If a source table has no primary key defined and has duplicate rows, replicating the duplicate rows to the MongoDB target results in a duplicate key error and the Replicat process abends.
- Missed updates and deletes are undetected so are ignored.
- Untested with sharded collections.
- Only supports date and time data types with millisecond precision. These values from a trail with microseconds or nanoseconds precision are truncated to millisecond precision.
- The `datetime` data type with `timezone` in the trail is not supported.
- A maximum BSON document size of 16 MB. If the trail record size exceeds this limit, the handler cannot replicate the record.
- No DDL propagation.
- No truncate operation.

JDBC Handler—Support

- The JDBC handler uses the generic JDBC API, which means any target database with a JDBC driver implementation should be able to use this handler. There are a myriad of different databases that support the JDBC API and Oracle cannot certify the JDBC Handler for all targets. Oracle has certified the JDBC Handler for the following RDBMS targets:

  Oracle
  MySQL
  Netezza
  Redshift

- The handler supports Replicat using the `REPERROR` and `HANDLECOLLISIONS` parameters, see *Reference for Oracle GoldenGate for Windows and UNIX*.
- The database metadata retrieved through the Redshift JDBC driver has known constraints, see *Release Notes for Oracle GoldenGate for Big Data*.

  Redshift target table names in the Replicat parameter file must be in lower case and double quoted. For example:

  ```
  MAP SourceSchema.SourceTable, target “public”.”targetable”;
  ```

- DDL operations are ignored by default and are logged with a `WARN` level.
- Coordinated Replicat is a multithreaded process that applies transactions in parallel instead of serially. Each thread handles all of the filtering, mapping, conversion, SQL construction, and error handling for its assigned workload. A coordinator thread coordinates transactions across threads to account for dependencies. It ensures that DML is applied in a synchronized manner preventing certain DMLs from occurring on the same object at the same time due to row locking, block locking, or table locking issues based on database specific rules. If there are database locking issue, then Coordinated Replicat performance

can be extremely slow or pauses, see *Administering Oracle GoldenGate for Windows and UNIX*

1.3 Setting Up Oracle GoldenGate for Big Data

This section contains the various tasks that you need to preform to set up Oracle GoldenGate for Big Data integrations with Big Data targets.

Topics:

1.3.1 Java Environment Setup

The Oracle GoldenGate for Big Data integrations create an instance of the Java virtual machine at runtime. Oracle GoldenGate for Big Data requires that you install Oracle Java 8 JRE at a minimum.

Oracle recommends that you set the `JAVA_HOME` environment variable to point to Java 8 installation directory. Additionally, the Java Delivery process needs to load the `libjvm.so` (`libjvm.dll` on Windows) and `libjsig.so` (`libjsig.dll` on Windows) Java shared libraries. These libraries are installed as part of the JRE. The location of these shared libraries need to be resolved and the appropriate environmental variable set to resolve the dynamic libraries needs to be set so the libraries can be loaded at runtime (that is, `LD_LIBRARY_PATH`, `PATH`, or `LIBPATH`).

1.3.2 Properties Files

There are two Oracle GoldenGate properties files required to run the Oracle GoldenGate Java Deliver user exit (alternatively called the Oracle GoldenGate Java Adapter). It is the Oracle GoldenGate Java Delivery that hosts Java integrations including the Big Data integrations. A Replicat properties file is required in order to run either process. The required naming convention for the Replicat file name is the *`process_name`*`.prm`. The exit syntax in the Replicat properties file provides the name and location of the Java Adapter properties file. It is the Java Adapter properties file that contains the configuration properties for the Java adapter include GoldenGate for Big Data integrations. The Replicat and Java Adapters properties files are required to run Oracle GoldenGate for Big Data integrations.

Alternatively the Java Adapters properties can be resolved using the default syntax, *`process_name`*`.properties`. It you use the default naming for the Java Adapter properties file then the name of the Java Adapter properties file can be omitted from the Replicat properties file.

Samples of the properties files for Oracle GoldenGate for Big Data integrations can be found in the subdirectories of the following directory:

`GoldenGate_install_dir``/AdapterExamples/big-data`

1.3.3 Transaction Grouping

The principal way to improve performance in Oracle GoldenGate for Big Data integrations is usingtransaction grouping. In transaction grouping, the operations of multiple transactions are grouped together in a single larger transaction. The application of a larger grouped transaction is typically much more efficient than the

application of individual smaller transactions. Transaction grouping is possible with the Replicat process discussed in Running with Replicat.

1.4 Configuring GoldenGate for Big Data

This section describes how to configure GoldenGate for Big Data Handlers.

Topics:

1.4.1 Running with Replicat

This section explains how to run the Java Adapter with the Oracle GoldenGate Replicat process. It includes the following sections:

1.4.1.1 Configuring Replicat

The following is an example of how you can configure a Replicat process properties file for use with the 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 these Replicat configuration entries:

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

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

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

`GROUPTRANSOPS 1000` - Groups 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` - Sets the tables to exclude.

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

1.4.1.2 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
```

1.4.1.3 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`.

1.4.1.4 Replicat Checkpointing

In addition to the Replicat checkpoint file `,.cpr`, an additional checkpoint file, `dirchk/`*`group`*`.cpj`, is created that contains information similar to `CHECKPOINTTABLE` in Replicat for the database.

1.4.1.5 Unsupported Replicat Features

The following Replicat features are not supported in this release:

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

1.4.1.6 Mapping Functionality

The Oracle GoldenGate Replicat process supports mapping functionality to custom target schemas. 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. For more information about the Replicat mapping syntax in the Replication configuration file, see *Administering Oracle GoldenGate for Windows and UNIX*.

1.4.2 Logging

Logging is essential to troubleshooting Oracle GoldenGate for Big Data integrations with Big Data targets. This section covers how Oracle GoldenGate for Big Data integration log and the best practices for logging. It includes the following sections:

Replicat Process Logging

Java Layer Logging

1.4.2.1 Replicat Process Logging

Oracle GoldenGate for Big Data integrations leverage the Java Delivery functionality described in the *Administering Oracle GoldenGate Application Adapters*. In this setup, either a Oracle GoldenGate Replicat process loads a user exit shared library. This shared library then loads a Java virtual machine to thereby interface with targets providing a Java interface. So the flow of data is as follows:

Replicat Process —>User Exit—> Java Layer

It is important that all layers log correctly so that users can review the logs to troubleshoot new installations and integrations. Additionally, if you have a problem that requires contacting Oracle Support, the log files are a key piece of information to be provided to Oracle Support so that the problem can be efficiently resolved.

A running Replicat process creates or appends log files into the *GoldenGate_Home*`/dirrpt` directory that adheres to the following naming convention: *process_name*`.rpt`. If a problem is encountered when deploying a new Oracle GoldenGate process, this is likely the first log file to examine for problems. The Java layer is critical for integrations with Big Data applications.

1.4.2.2 Java Layer Logging

The Oracle GoldenGate for Big Data product provides flexibility for logging from the Java layer. The recommended best practice is to use Log4j logging to log from the Java layer. Enabling simple Log4j logging requires the setting of two configuration values in the Java Adapters configuration file.

```
gg.log=log4j
gg.log.level=INFO
```

These `gg.log` settings will result in a Log4j file to be created in the *GoldenGate_Home*`/dirrpt` directory that adheres to this naming convention, *process_name_log level*`_log4j.log`. The supported Log4j log levels are in the following list in order of increasing logging granularity.

- `OFF`
- `FATAL`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

Selection of a logging level will include all of the coarser logging levels as well (that is, selection of `WARN` means that log messages of `FATAL`, `ERROR` and `WARN` will be written to the log file). The Log4j logging can additionally be controlled by separate Log4j properties files. These separate Log4j properties files can be enabled by editing the `bootoptions` property in the Java Adapter Properties file. These three example Log4j properties files are included with the installation and are included in the classpath:

```
log4j-default.properties
log4j-debug.properites
log4j-trace.properties
```

You can modify the `bootoptions`in any of the files as follows:

```
javawriter.bootoptions=-Xmx512m -Xms64m -
Djava.class.path=.:ggjava/ggjava.jar -
Dlog4j.configuration=samplelog4j.properties
```

You can use your own customized Log4j properties file to control logging. The customized Log4j properties file must be available in the Java classpath so that it can be located and loaded by the JVM. The contents of a sample custom Log4j properties file is the following:

```
# Root logger option
log4j.rootLogger=INFO, file

# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender

log4j.appender.file.File=sample.log
log4j.appender.file.MaxFileSize=1GB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L -
%m%n
```

There are two important requirements when you use a custom Log4j properties file. First, the path to the custom Log4j properties file must be included in the `javawriter.bootoptions` property. Logging initializes immediately when the JVM is initialized while the contents of the `gg.classpath` property is actually appended to the `classloader` after the logging is initialized. Second, the `classpath` to correctly load a properties file must be the directory containing the properties file without wildcards appended.

1.4.3 Metadata Change Events

The Oracle GoldenGate for Big Data Handlers and Formatters provide functionality to take action when a metadata change event is encountered. The ability to take action in the case of metadata change events depends on the metadata change events being available in the source trail file. Oracle GoldenGate supports metadata in trail and the propagation of DDL data from a source Oracle Database. If the source trail file does not have metadata in trail and DDL data (metadata change events) then it is not possible for Oracle GoldenGate for Big Data to provide and metadata change event handling.

1.4.4 Configuration Property `CDATA[]` Wrapping

The GoldenGate for Big Data Handlers and Formatters support the configuration of many parameters in the Java properties file, the value of which may be interpreted as white space. The configuration handling of the Java Adapter trims white space from configuration values from the Java configuration file. This behavior of trimming whitespace may be desirable for some configuration values and undesirable for other configuration values. Alternatively, you can wrap white space values inside of special syntax to preserve the whites pace for selected configuration variables. GoldenGate for Big Data borrows the XML syntax of `CDATA[]` to preserve white space. Values that would be considered to be white space can be wrapped inside of `CDATA[]`.

The following is an example attempting to set a new-line delimiter for the Delimited Text Formatter:

```
gg.handler.{name}.format.lineDelimiter=\n
```

This configuration will not be successful. The new-line character is interpreted as white space and will be trimmed from the configuration value. Therefore the `gg.handler` setting effectively results in the line delimiter being set to an empty string.

In order to preserve the configuration of the new-line character simply wrap the character in the `CDATA[]` wrapper as follows:

`gg.handler.{name}.format.lineDelimiter=CDATA[\n]`

Configuring the property with the `CDATA[]` wrapping preserves the white space and the line delimiter will then be a new-line character.

1.4.5 Using Regular Expression Search and Replace

You can perform more powerful search and replace operations of both schema data (catalog names, schema names, table names, and column names) and column value data, which are separately configured. Regular expressions (regex) are characters that customize a search string through pattern matching. You can match a string against a pattern or extract parts of the match. Oracle GoldenGate for Big Data uses the standard Oracle Java regular expressions package, `java.util.regex`. For more information, see "Regular Expressions" in the Base Definitions volume at **The Single UNIX Specification, Version 4**.

This section includes the following:

Using Schema Data Replace

Using Content Data Replace

1.4.5.1 Using Schema Data Replace

You can replace schema data using the `gg.schemareplaceregex` and `gg.schemareplacestring` properties. Use `gg.schemareplaceregex` to set a regular expression, and then use it to search catalog names, schema names, table names, and column names for corresponding matches. Matches are then replaced with the content of the `gg.schemareplacestring` value. The default value of `gg.schemareplacestring` is an empty string or `""`.

For example, some system table names start with a dollar sign like `$mytable`. You may want to replicate these tables even though most Big Data targets do not allow dollar signs in table names. To remove the dollar sign, you could configure the following replace strings:

```
gg.schemareplaceregex=[$]
gg.schemareplacestring=
```

The resulting example of searched and replaced table name is `mytable`. These properties also support `CDATA[]` wrapping to preserve whitespace in the value of configuration values. So the equivalent of the preceding example using `CDATA[]` wrapping use is:

```
gg.schemareplaceregex=CDATA[[$]]
gg.schemareplacestring=CDATA[]
```

The schema search and replace functionality supports using multiple search regular expressions and replacements strings using the following configuration syntax:

```
gg.schemareplaceregex=some_regex
gg.schemareplacestring=some_value
gg.schemareplaceregex1=some_regex
```

```
gg.schemareplacestring1=some_value
gg.schemareplaceregex2=some_regex
gg.schemareplacestring2=some_value
```

1.4.5.2 Using Content Data Replace

You can replace content data using the `gg.contentreplaceregex` and `gg.contentreplacestring` properties to search the column values using the configured regular expression and replace matches with the replacement string. For example, this is useful to replace line feed characters in column values. If the delimited text formatter is used then line feeds occurring in the data will be incorrectly interpreted as line delimiters by analytic tools.

You can configure *n* number of content replacement regex search values. The regex search and replacements are done in the order of configuration. Configured values must follow a given order as follows:

```
gg.conentreplaceregex=some_regex
gg.conentreplacestring=some_value
gg.conentreplaceregex1=some_regex
gg.conentreplacestring1=some_value
gg.conentreplaceregex2=some_regex
gg.conentreplacestring2=some_value
```

Configuring a subscript of 3 without a subscript of 2 would cause the subscript 3 configuration to be ignored.

NOT_SUPPORTED:

Regular express searches and replacements require computer processing and can reduce the performance of the Oracle GoldenGate for Big Data process.

To replace line feeds with a blank character you could use the following property configurations:

```
gg.contentreplaceregex=[\n]
gg.contentreplacestring=CDATA[ ]
```

This changes the column value from:

```
this is
me
```

to :

```
this is me
```

Both values support `CDATA` wrapping. The second value must be wrapped in a `CDATA[]` wrapper because a single blank space will be interpreted as whitespace and trimmed by the Oracle GoldenGate for Big Data configuration layer. In addition, you can configure multiple search a replace strings. For example, you may also want to trim leading and trailing white space out of column values in addition to trimming line feeds from:

```
^\\s+|\\s+$

gg.contentreplaceregex1=^\\s+|\\s+$
gg.contentreplacestring1=CDATA[]
```

1.4.6 Scaling Oracle GoldenGate for Big Data Delivery

Oracle GoldenGate for Big Data supports breaking down the source trail files into either multiple Replicat processes or by using Coordinated Delivery to instantiate multiple Java Adapter instances inside a single Replicat process to improve throughput.. This allows you to scale Oracle GoldenGate for Big Data delivery.

There are some cases where the throughput to Oracle GoldenGate for Big Data integration targets is not sufficient to meet your service level agreements even after you have tuned your Handler for maximum performance. When this occurs, you can configure parallel processing and delivery to your targets using one of the following methods:

- Multiple Replicat processes can be configured to read data from the same source trail files. Each of these Replicat processes are configured to process a subset of the data in the source trail files so that all of the processes collectively process the source trail files in their entirety. There is no coordination between the separate Replicat processes using this solution.
- Oracle GoldenGate Coordinated Delivery can be used to parallelize processing the data from the source trail files within a single Replicat process. This solution involves breaking the trail files down into logical subsets for which each configured subset is processed by a different delivery thread. For more information about Coordinated Delivery, see https://blogs.oracle.com/dataintegration/entry/goldengate_12c_coordinated_replicat.

With either method, you can split the data into parallel processing for improved throughput. Oracle recommends breaking the data down in one of the following two ways:

- Splitting Source Data By Source Table –Data is divided into subsections by source table. For example, Replicat process 1 might handle source tables table1 and table2, while Replicat process 2 might handle data for source tables table3 and table2. Data is split for source table and the individual table data is not subdivided.
- Splitting Source Table Data into Sub Streams – Data from source tables is split. For example, Replicat process 1 might handle half of the range of data from source table1, while Replicat process 2 might handler the other half of the data from source table1.

Additional limitations:

- Parallel apply is *not* supported.
- The `BATCHSQL` parameter not supported.

Example 1-1 Scaling Support for the Oracle GoldenGate for Big Data Handlers

Handler Name	Splitting Source Data By Source Table	Splitting Source Table Data into Sub Streams
HDFS	Supported	Not supported

Handler Name	Splitting Source Data By Source Table	Splitting Source Table Data into Sub Streams
HBase	Supported when all required HBase namespaces are pre-created in HBase.	Supported when: • All required HBase namespaces are pre-created in HBase. • All required HBase target tables are pre-created in HBase. Schema evolution is not an issue because HBase tables have no schema definitions so a source metadata change does not require any schema change in HBase. • The source data does not contain any truncate operations.
Kafka	Supported	Supported for formats that support schema propagation, such as Avro this is less desirable.
Flume	Supported	Supported for formats that support schema propagation, such as Avro this is less desirable.
Cassandra	Supported	Supported when: • Required target tables in Cassandra are pre-created. • Metadata change events do not occur.
MongoDB	Supported	Supported
JDBC	Supported	Supported

1.4.7 Using Identities in Oracle GoldenGate Credential Store

The Oracle GoldenGate credential store manages user IDs and their encrypted passwords (together known as credentials) that are used by Oracle GoldenGate processes to interact with the local database. The credential store eliminates the need to specify user names and clear-text passwords in the Oracle GoldenGate parameter files. An optional alias can be used in the parameter file instead of the user ID to map to a userid and password pair in the credential store. The credential store is implemented as an auto login wallet within the Oracle Credential Store Framework (CSF). The use of an LDAP directory is not supported for the Oracle GoldenGate credential store. The auto login wallet supports automated restarts of Oracle GoldenGate processes without requiring human intervention to supply the necessary passwords.

In Oracle GoldenGate for Big Data, you specify the alias and domain in the property file not the actual user ID or password. User credentials are maintained in secure wallet storage.

This section includes the following:

1.4.7.1 Creating a Credential Store

You can create a credential store for your Big Data environment.

Run the GGSCI `ADD CREDENTIALSTORE` command to create a file called `cwallet.sso` in the `dircrd/` subdirectory of your Oracle GoldenGate installation directory (the default).

You can the location of the credential store (`cwallet.sso` file by specifying the desired location with the `CREDENTIALSTORELOCATION` parameter in the `GLOBALS` file.

For more information about credential store commands, see *Reference for Oracle GoldenGate for Windows and UNIX*.

Note:

Only one credential store can be used for each Oracle GoldenGate instance.

1.4.7.2 Adding Users to a Credential Store

After you create a credential store for your Big Data environment, you can added users to the store.

Run the GGSCI `ALTER CREDENTIALSTORE ADD USER` *`userid`* `PASSWORD` *`password`* `[ALIAS` *`alias`*`] [DOMAIN` *`domain`*`]` command to create each user, where:

- *`userid`* is the user name. Only one instance of a user name can exist in the credential store unless the `ALIAS` or `DOMAIN` option is used.
- *`password`* is the user's password. The password is echoed (not obfuscated) when this option is used. If this option is omitted, the command prompts for the password, which is obfuscated as it is typed (recommended because it is more secure).
- *`alias`* is an alias for the user name. The alias substitutes for the credential in parameters and commands where a login credential is required. If the `ALIAS` option is omitted, the alias defaults to the user name.

For example:

```
ALTER CREDENTIALSTORE ADD USER scott PASSWORD tiger ALIAS scsm2 domain ggadapters
```

For more information about credential store commands, see *Reference for Oracle GoldenGate for Windows and UNIX*.

1.4.7.3 Configuring Properties to Access the Credential Store

The Oracle GoldenGate Java Adapter properties file requires specific syntax to resolve user name and password entries in the Credential Store at runtime. For resolving a user name the syntax is the following:

```
ORACLEWALLETUSERNAME alias domain_name
```

For resolving a password the syntax required is the following:

```
ORACLEWALLETPASSWORD alias domain_name
```

The following example illustrate how to configure a Credential Store entry with an alias of `myalias` and a domain of `mydomain`.

Note:

With HDFS Hive JDBC the user name and password is encrypted.

```
gg.handler.hdfs.hiveJdbcUsername=ORACLEWALLETUSERNAME[myalias mydomain]
gg.handler.hdfs.hiveJdbcPassword=ORACLEWALLETPASSWORD[myalias mydomain]
```

Although the Credential Store is intended to store user name and password pair type credentials, you can apply this functionality more generically. Consider the user name and password entries as accessible values in the Credential Store. Any configuration property resolved in the Java Adapter layer (not accessed in the C user exit layer) can be resolved from the Credential Store. This allows you more flexibility to be creative in how you protect sensitive configuration entries.

2

Using the HDFS Handler

This chapter explains the HDFS Handler, which is designed to stream change capture data into the Hadoop Distributed File System (HDFS).

Topics:

2.1 Overview

The HDFS is the primary application for Big Data. Hadoop is typically installed on multiple machines that work together as a Hadoop cluster. Hadoop allows you to store very large amounts of data in the cluster that is horizontally scaled across the machines in the cluster. You can then perform analytics on that data using a variety of Big Data applications.

2.2 Writing into HDFS in SequenceFile Format

The HDFS SequenceFile is a flat file consisting of binary key and value pairs. You can enable writing data in SequenceFile format by setting the `gg.handler.`*`name`*`.format` property to `sequencefile`. The `key` part of the record is set to null and the actual data is set in the `value` part.

For information about Hadoop SequenceFile, see https://wiki.apache.org/hadoop/SequenceFile.

2.2.1 Integrating with Hive

Oracle GoldenGate for Big Data release does not include a Hive Handler because the HDFS Handler provides all of the necessary Hive functionality .

You can create a Hive integration to create tables and update table definitions in the case of DDL events, which is limited to only data formatted as Avro Object Container File format. For more information, see Writing in HDFS in Avro Object Container File Format and HDFS Handler Configuration.

DDL to create Hive tables should include `STORED as sequencefile` for Hive to consume Sequence Files. Following is a sample create table script:

```
CREATE EXTERNAL TABLE table_name (
  col1 string,
  ...
  ...
  col2 string)
ROW FORMAT DELIMITED
STORED as sequencefile
LOCATION '/path/to/hdfs/file';
```

Note:

If files are intended to be consumed by Hive, then the `gg.handler.name.partitionByTable` property should be set to `true`.

2.2.2 Understanding the Data Format

The data written in the `value` part of each record and is in delimited text format. All of the options described in the Delimited Text Formatter section are applicable to HDFS SequenceFile when writing data to it.

For example:

```
gg.handler.name.format=sequencefile
gg.handler.name.format.includeColumnNames=true
gg.handler.name.format.includeOpType=true
gg.handler.name.format.includeCurrentTimestamp=true
gg.handler.name.format.updateOpKey=U
```

2.2.3 Setting Up and Running the HDFS Handler

To run the HDFS Handler, a Hadoop single instance or Hadoop cluster must be installed, running, and network accessible from the machine running the HDFS Handler. Apache Hadoop is open source and available for download at `http://hadoop.apache.org/`. Follow the Getting Started links for information on how to install a single-node cluster (also called pseudo-distributed operation mode) or a clustered setup (also called fully-distributed operation mode).

Instructions for configuring the HDFS Handler components and running the handler are described in the following sections.

Classpath Configuration

HDFS Handler Configuration

Sample Configuration

Performance Considerations

Security

2.2.3.1 Classpath Configuration

Two things must be configured in the `gg.classpath` configuration variable in order for the HDFS Handler to connect to HDFS and run. The first thing is the `HDFS core-site.xml` file and the second are the HDFS client jars. The HDFS client jars must match the version of HDFS that the HDFS Handler is connecting. For a listing of the required client JAR files by release, see HDFS Handler Client Dependencies.

The default location of the `core-site.xml` file is the follow:

Hadoop_Home`/etc/hadoop`

The default location of the HDFS client jars are the following directories:

Hadoop_Home`/share/hadoop/common/lib/*`

Hadoop_Home`/share/hadoop/common/*`

Hadoop_Home`/share/hadoop/hdfs/lib/*`

Hadoop_Home`/share/hadoop/hdfs/*`

The `gg.classpath` must be configured exactly as shown. Pathing to the `core-site.xml` should simply contain the path to the directory containing the `core-site.xml`file with no wild card appended. The inclusion of the * wildcard in the path to the `core-site.xml` file will cause it not to be picked up. Conversely, pathing to the dependency jars should include the * wildcard character in order to include all of the jar files in that directory in the associated classpath. Do *not* use `*.jar`.

An example of a correctly configured `gg.classpath` variable is the following:

```
gg.classpath=/ggwork/hadoop/hadoop-2.6.0/etc/hadoop:/ggwork/hadoop/hadoop-2.6.0/
share/hadoop/common/lib/*:/ggwork/hadoop/hadoop-2.6.0/share/hadoop/common/*:/ggwork/
hadoop/hadoop-2.6.0/share/hadoop/hdfs/*:/ggwork/hadoop/hadoop-2.6.0/share/hadoop/
hdfs/lib/*
```

The HDFS configuration file `hdfs-site.xml` is also required to be in the classpath if Kerberos security is enabled. The `hdfs-site.xml` file is by default located in the *Hadoop_Home*`/etc/hadoop` directory. Either or both files can be copied to another machine if the HDFS Handler is not collocated with Hadoop.

2.2.3.2 HDFS Handler Configuration

The following are the configurable values for the HDFSHandler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 2-1 HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handlerlist`	Required	Any string	None	Provides a name for the HDFS Handler. The HDFS Handler name then becomes part of the property names listed in this table.
`gg.handler.`*`name`*`.type=hdfs`	Required	`-`	`-`	Selects the HDFS Handler for streaming change data capture into HDFS.
`gg.handler.`*`name`*`.mode`	Optional	`tx` | `op`	`op`	Selects operation (`op`) mode or transaction (`tx`) mode for the handler. In almost all scenarios, transaction mode results in better performance.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.maxFileSize`	Optional	Default unit of measure is bytes. You can stipulate `k`, `m`, or `g` to signify kilobytes, megabytes, or gigabytes respectively. Examples of legal values include `10000`, `10k`, `100m`, `1.1g`.	`1g`	Selects the maximum file size of created HDFS files.
`gg.handler.name.rootFilePath`	Optional	Any path name legal in HDFS.	`/ogg`	The HDFS Handler will create subdirectories and files under this directory in HDFS to store the data streaming into HDFS.
`gg.handler.name.fileRollInterval`	Optional	The default unit of measure is milliseconds. You can stipulate `ms`, `s`, `m`, `h` to signify milliseconds, seconds, minutes, or hours respectively. Examples of legal values include `10000`, 10000ms, `10s`, `10m`, or `1.5h`. Values of `0` or less indicate that file rolling on time is turned off.	File rolling on time is off.	The timer starts when an HDFS file is created. If the file is still open when the interval elapses then the file will be closed. A new file will not be immediately opened. New HDFS files are created on a just in time basis.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.inactivityRollInterval`	Optional	The default unit of measure is milliseconds. You can stipulate `ms`, `s`, `m`, `h` to signify milliseconds, seconds, minutes, or hours respectively. Examples of legal values include `10000`, `10000ms`, `10s`, `10.5m`, or `1h`. Values of `0` or less indicate that file inactivity rolling on time is turned off.	File inactivity rolling on time is off.	The timer starts from the latest write to an HDFS file. New writes to an HDFS file restart the counter. If the file is still open when the counter elapses the HDFS file will be closed. A new file will not be immediately opened. New HDFS files are created on a just in time basis.
`gg.handler.name.fileSuffix`	Optional	Any string conforming to HDFS file name restrictions.	`.txt`	This is a suffix that is added on to the end of the HDFS file names. File names typically follow the format, `{fully qualified table name}{current time stamp}{suffix}`.
`gg.handler.name.partitionByTable`	Optional	`true` | `false`	`true` (data is partitioned by table)	Determines if data written into HDFS should be partitioned by table. If set to `true`, then data for different tables are written to different HDFS files. If se to `false`, then data from different tables is interlaced in the same HDFS file. Must be set to `true` to use the Avro Object Container File Formatter. Set to `false` results in a configuration exception at initialization.
`gg.handler.name.rollOnMetadataChange`	Optional	`true` | `false`	`true` (HDFS files are rolled on a metadata change event)	Determines if HDFS files should be rolled in the case of a metadata change. True means the HDFS file is rolled, false means the HDFS file is not rolled. Must be set to `true` to use the Avro Object Container File Formatter. Set to `false` results in a configuration exception at initialization.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format`	Optional	`delimitedtext` | `json` | `json_ROW` | `xml` | `avro_row` | `avro_op` | `avro_row_ocf` | `avro_op_ocf` | `sequencefile`	`delimitedtext`	Selects the formatter for the HDFS Handler for how output data will be formatted • `delimitedtext` - Delimited text • `json` - JSON • `json_row` - JSON output modeling row data • `xml` - XML • `avro_row` - Avro in row compact format • `avro_op` - Avro in operation more verbose format. • `avro_row_ocf` - Avro in the row compact format written into HDFS in the Avro Object Container File (OCF) format. • `avro_op_ocf` - Avro in the more verbose format written into HDFS in the Avro Object Container File format. • `sequencefile` - Delimited text written in sequence into HDFS is sequence file format.
`gg.handler.name.includeTokens`	Optional	`true` | `false`	`false`	Set to `true` to include the tokens field and tokens key/values in the output, `false` to suppress tokens output.
`gg.handler.name.partitioner.fully_qualified_table_ name` Equals one or more column names separated by commas.	Optional	Fully qualified table name and column names must exist.	-	This partitions the data into subdirectories in HDFS in the following format, `par_{column name}={column value}`
`gg.handler.name.authType`	Optional	`kerberos`	`none`	Setting this property to `kerberos` enables Kerberos authentication.
`gg.handler.name.kerberosKeytabFile`	Optional (Required if `authType=Kerberos`)	Relative or absolute path to a Kerberos `keytab` file.	-	The `keytab` file allows the HDFS Handler to access a password to perform a `kinit` operation for Kerberos security.
`gg.handler.name.kerberosPrincipal`	Optional (Required if `authType=Kerberos`)	A legal Kerberos principal name like `user/FQDN@MY.REALM`.	-	The Kerberos principal name for Kerberos authentication.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.schemaFilePath`	Optional		`null`	Set to a legal path in HDFS so that schemas (if available) are written in that HDFS directory. Schemas are currently only available for Avro and JSON formatters. In the case of a metadata change event, the schema will be overwritten to reflect the schema change.
`gg.handler.name.compressionType` Applicable to Sequence File Format only.	Optional	`block | none | record`	`none`	Hadoop Sequence File Compression Type. applicable only if `gg.handler.name.format` is set to `sequencefile`
`gg.handler.name.compressionCodec` Applicable to Sequence File and writing to HDFS is Avro OCF formats only.	Optional	`org.apache.hadoop.io.compress.DefaultCodec | org.apache.hadoop.io.compress.BZip2Codec | org.apache.hadoop.io.compress.SnappyCodec | org.apache.hadoop.io.compress.GzipCodec`	`org.apache.hadoop.io.compress.DefaultCodec`	Hadoop Sequence File Compression Codec. applicable only if `gg.handler.name.format` is set to `sequencefile`
	Optional	`null | snappy | bzip2 | xz | deflate`	`null`	Avro OCF Formatter Compression Code. This configuration controls the selection of the compression library to be used for Avro OCF files generated. Snappy includes native binaries in the Snappy JAR file and performs a Java-native traversal when performing compression or decompression. Use of Snappy may introduce runtime issue and platform porting issues that you may not experience when working with Java. You may need to perform additional testing to ensure Snappy works on all of your required platforms. Snappy is an open source library so Oracle cannot guarantee its ability to operate on all of your required platforms.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.hiveJdbcUrl`	Optional	A legal URL for connecting to Hive using the Hive JDBC interface.	`null` (Hive integration disabled)	Only applicable to the Avro OCF Formatter. This configuration value provides a JDBC URL for connectivity to Hive through the Hive JDBC interface. Use of this property requires that you include the Hive JDBC library in the `gg.classpath`. Hive JDBC connectivity can be secured through basic credentials, SSL/TLS, or Kerberos. Configuration properties are provided for the user name and password for basic credentials. See the Hive documentation for how to generate a Hive JDBC URL for SSL/TLS. See the Hive documentation for how to generate a Hive JDBC URL for Kerberos. (If Kerberos is used for Hive JDBC security, it must be enabled for HDFS connectivity. Then the Hive JDBC connection can piggyback on the HDFS Kerberos functionality by using the same Kerberos principal.)
`gg.handler.name.hiveJdbcUsername`	Optional	A legal user name if the Hive JDBC connection is secured through credentials.	Java call result from `System.getProperty(user.name)`	Only applicable to the Avro Object Container File OCF Formatter. This property is only relevant if the `hiveJdbcUrl`property is set. It may be required in your environment when the Hive JDBC connection is secured through credentials. Hive requires that Hive DDL operations be associated with a user. If you do not set the value, it defaults to the result of the Java call `System.getProperty(user.name)`
`gg.handler.name.hiveJdbcPassword`	Optional	A legal password if the Hive JDBC connection requires a password.	None	Only applicable to the Avro OCF Formatter. This property is only relevant if the `hiveJdbcUrl` property is set. It may be required in your environment when the Hive JDBC connection is secured through credentials. This is required if Hive is configured to required passwords for the JDBC connection.
`gg.handler.name.hiveJdbcDriver`	Optional	The fully qualified Hive JDBC driver class name.	`org.apache.hive.jdbc.HiveDriver`	Only applicable to the Avro OCF Formatter. This property is only relevant if the `hiveJdbcUrl` property is set. The default is the Hive Hadoop2 JDBC driver name. Typically, this property does not require configuration and is provided for use when Apache Hive introduces a new JDBC driver class.

Table 2-1 (Cont.) HDFS Handler Configuration Properties

Property	Optional / Required	Legal Values	Default	Explanation
`gg.handler.hdfs.openNextFileAtRoll`	Optional		`false`	Only applicable to the HDFS Handler that is *not* writing Avro OCF or Sequence file to support extract, load, transform (ELT) situations. When set to `true`, this property creates a new file immediately on the occurrence of a file roll. File rolls can be triggered by any one of the following • Metadata change • File roll interval elapsed • Inactivity interval elapsed Data files are being loaded into HDFS and a monitor program is monitoring the write directories waiting to consume the data. The monitoring programs use the appearance of a new file as a trigger so that the previous file can be consumed by the consuming application.

2.2.3.3 Sample Configuration

The following is sample configuration for the HDFS Handler from the Java Adapter properties file:

```
gg.handlerlist=hdfs
gg.handler.hdfs.type=hdfs
gg.handler.hdfs.mode=tx
gg.handler.hdfs.includeTokens=false
gg.handler.hdfs.maxFileSize=1g
gg.handler.hdfs.rootFilePath=/ogg
gg.handler.hdfs.fileRollInterval=0
gg.handler.hdfs.inactivityRollInterval=0
gg.handler.hdfs.fileSuffix=.txt
gg.handler.hdfs.partitionByTable=true
gg.handler.hdfs.rollOnMetadataChange=true
gg.handler.hdfs.authType=none
gg.handler.hdfs.format=delimitedtext
```

2.2.3.4 Performance Considerations

The HDFS Handler calls the HDFS flush method on the HDFS write stream to flush data to the HDFS data nodes at the end of each transaction in order to maintain write durability. This is an expensive call and performance can be adversely affected especially in the case of transactions of one or few operations that results in numerous HDFS flush calls.

Performance of the HDFS Handler can be greatly improved by batching multiple small transactions into a single larger transaction. If you have requirements for high performance, you should configure batching functionality for Replicat process. For more information, see Replicat Grouping.

The HDFS client libraries spawn threads for every HDFS file stream opened by the HDFS Handler. The result is that the number threads executing in the JMV grows proportionally to the number HDFS file streams that are open. Performance of the

HDFS Handler can degrade as more HDFS file streams are opened. Configuring the HDFS Handler to write to many HDFS files due to many source replication tables or extensive use of partitioning can result in degraded performance. If your use case requires writing to many tables, then Oracle recommends that you enable the roll on time or roll on inactivity features to close HDFS file streams. Closing an HDFS file stream causes the HDFS client threads to terminate and the associated resources can be reclaimed by the JVM.

2.2.3.5 Security

The HDFS cluster can be secured using Kerberos authentication. The HDFS Handler can connect to Kerberos secured cluster. The HDFS `core-site.xml` should be in the handlers classpath with the `hadoop.security.authentication` property set to `kerberos` and `hadoop.security.authorization` property set to `true`. Additionally, you must set the following properties in the HDFS Handler Java configuration file:

```
gg.handler.name.authType=kerberos
gg.handler.name.keberosPrincipalName=legal Kerberos principal name
gg.handler.name.kerberosKeytabFile=path to a keytab file that contains the password
for the Kerberos principal so that the HDFS Handler can programmatically perform the
Kerberos kinit operations to obtain a Kerberos ticket
```

See the HDFS documentation to understand how to secure a Hadoop cluster using Kerberos.

2.3 Writing in HDFS in Avro Object Container File Format

The HDFS Handler includes specialized functionality to write to HDFS in Avro Object Container File (OCF) format. This Avro OCF is part of the Avro specification and is detailed in the Avro Documentation at

https://avro.apache.org/docs/current/spec.html#Object+Container+Files

Avro OCF format may be a good choice for you because it

- integrates with Apache Hive (raw Avro written to HDFS is not supported by Hive)
- and provides good support for schema evolution.

Configure the following to enable writing to HDFS in Avro OCF format:

To write row data to HDFS in Avro OCF format configure the `gg.handler.name.format=avro_row_ocf` property.

To write operation data to HDFS is Avro OCF format configure the `gg.handler.name.format=avro_op_ocf` property.

The HDFS and Avro OCF integration includes optional functionality to create the corresponding tables in Hive and update the schema for metadata change events. The configuration section provides information on the properties to enable integration with Hive. The Oracle GoldenGate Hive integration accesses Hive using the JDBC interface so the Hive JDBC server *must* be running to enable this integration.

2.4 Metadata Change Events

Metadata change events are now handled in the HDFS Handler. The default behavior of the HDFS Handler is to roll the current relevant file in the event of a metadata change event. This behavior allows for the results of metadata changes to at least be

separated into different files. File rolling on metadata change is configurable and can be turned off.

To support metadata change events the process capturing changes in the source database must support both DDL changes and metadata in trail. Oracle GoldenGate does *not* support DDL replication for all database implementations, see the Oracle GoldenGate installation and configuration guide for the appropriate database to understand if DDL replication is supported.

2.5 Partitioning

The HDFS Handler supports partitioning of table data by one or more column values. The configuration syntax to enable partitioning is the following:

```
gg.handler.name.partitioner.fully qualified table name=one mor more column names
separated by commas
```

Consider the following example:

```
gg.handler.hdfs.partitioner.dbo.orders=sales_region
```

This example can result in the following breakdown of files in HDFS:

```
/ogg/dbo.orders/par_sales_region=west/data files
/ogg/dbo.orders/par_sales_region=east/data files
/ogg/dbo.orders/par_sales_region=north/data files
/ogg/dbo.orders/par_sales_region=south/data files
```

You should exercise care when choosing columns for partitioning. The key is to choose columns that contain only a few (10 or less) possible values and those values are also meaningful for the grouping and analysis of the data. An example of a good partitioning column is sales regions. An example of a poor partitioning column is customer date of birth. Configuring partitioning on a column that has many possible values can be problematic. A poor choice can result in hundreds of HDFS file streams being opened and performance can degrade for the reasons discussed in Performance Considerations. Additionally, poor partitioning can result in problems while performing analysis on the data. Apache Hive requires that all `where` clauses specify partition criteria if the Hive data is partitioned.

2.6 Additional Considerations

The most common problems encountered are Java classpath issues. The Oracle HDFS Handler requires certain HDFS client libraries to be resolved in its classpath as a prerequisite for streaming data to HDFS.

For a listing of the required client JAR files by version, see HDFS Handler Client Dependencies. The HDFS client jars *do not* ship with the Oracle GoldenGate for Big Data product. The HDFS Handler supports multiple versions of HDFS and it is required that the HDFS client jars be the same version as the HDFS version to which the HDFS Handler is connecting. The HDFS client jars are open source and freely available to download from sites such as the Apache Hadoop site or the maven central repository.

In order to establish connectivity to HDFS, the HDFS `core-site.xml` file needs to be in the classpath of the HDFS Handler. If the `core-site.xml` file is not in the classpath the HDFS client code defaults to a mode that attempts to write to the local file system. Writing to the local file system instead of HDFS can in fact be an

advantageous for troubleshooting, building a point of contact (POC), or as a step in the process of building an HDFS integration.

Another common concern is that data streamed to HDFS using the HDFS Handler is often not immediately available to Big Data analytic tools such as Hive. This behavior commonly occurs when the HDFS Handler is in possession of an open write stream to an HDFS file. HDFS writes in blocks of 128MB by default. HDFS blocks under construction are not always visible to analytic tools. Additionally, inconsistencies between file sizes when using the `-ls`, `-cat`, and `-get` commands in the HDFS shell are commonly seen. This is an anomaly of HDFS streaming and is discussed in the HDFS specification. This anomaly of HDFS leads to a potential 128MB per file blind spot in analytic data. This may not be an issue if you have a steady stream of Replication data and do not require low levels of latency for analytic data from HDFS. However, this may be a problem in some use cases because closing the HDFS write stream causes the block writing to finalize. Data is immediately visible to analytic tools and file sizing metrics become consistent again. So the new file rolling feature in the HDFS Handler can be used to close HDFS writes streams thus making all data visible.

Important:

The file rolling solution may present its own potential problems. Extensive use of file rolling can result in lots of small files in HDFS. Lots of small files in HDFS can be its own problem resulting in performance issues in analytic tools.

You may also notice the HDFS inconsistency problem in the following scenarios.

- The HDFS Handler process crashes.
- A forced shutdown is called on the HDFS Handler process.
- A network outage or some other issue causes the HDFS Handler process to abend.

In each of these scenarios, it is possible for the HDFS Handler to end without explicitly closing the HDFS write stream and finalizing the writing block. HDFS in its internal process ultimately recognizes that the write stream has been broken so HDFS finalizes the write block. In this scenario, you may experience a short term delay before the HDFS process finalizes the write block.

2.7 Best Practices

It is considered a Big Data best practice for the HDFS cluster to operate on dedicated servers called cluster nodes. Edge nodes are server machines that host the applications to stream data to and retrieve data from the HDFS cluster nodes. This physical architecture delineation between the HDFS cluster nodes and the edge nodes provides a number of benefits including the following:

- The HDFS cluster nodes are not competing for resources with the applications interfacing with the cluster.
- HDFS cluster nodes and edge nodes likely have different requirements. This physical topology allows the appropriate hardware to be tailored to the specific need.

It is a best practice for the HDFS Handler to be installed and running on an edge node and streaming data to the HDFS cluster using network connection. The HDFS Handler

can run on any machine that has network visibility to the HDFS cluster. The installation of the HDFS Handler on an edge node requires that the `core-site.xml` files and the dependency jars be copied to the edge node so that the HDFS Handler can access them. The HDFS Handler can also run collocated on a HDFS cluster node if required.

2.8 Troubleshooting the HDFS Handler

Troubleshooting of the HDFS Handler begins with the contents for the Java `log4j` file. Follow the directions in the Java Logging Configuration to configured the runtime to correctly generate the Java `log4j` log file. Review the following topics for additional help:

Java Classpath

HDFS Connection Properties

Handler and Formatter Configuration

2.8.1 Java Classpath

As previously stated, issues with the Java classpath are one of the most common problems. The usual indication of a Java classpath problem is a `ClassNotFoundException` in the Java `log4j` log file. The Java `log4j` log file can be used to troubleshoot this issue. Setting the log level to `DEBUG` allows for logging of each of the jars referenced in the `gg.classpath` object to be logged to the log file. In this way, you can ensure that all of the required dependency jars are resolved by enabling `DEBUG` level logging and search the log file for messages as in the following:

```
2015-09-21 10:05:10 DEBUG ConfigClassPath:74 - ...adding to classpath: url="file:/
ggwork/hadoop/hadoop-2.6.0/share/hadoop/common/lib/guava-11.0.2.jar
```

2.8.2 HDFS Connection Properties

The contents of the HDFS `core-site.xml` file (including default settings) are output to the Java `log4j` log file when the logging level is set to `DEBUG` or `TRACE`. This will show the connection properties to HDFS. Search for the following in the Java `log4j` log file:

```
2015-09-21 10:05:11 DEBUG HDFSConfiguration:58 - Begin - HDFS configuration object
contents for connection troubleshooting.
```

If the `fs.defaultFS` property is set as follows (pointing at the local file system) then the `core-site.xml` file is not properly set in the `gg.classpath` property.

```
  Key: [fs.defaultFS] Value: [file:///].
```

This shows to the `fs.defaultFS` property properly pointed at and HDFS host and port.

```
Key: [fs.defaultFS] Value: [hdfs://hdfshost:9000].
```

2.8.3 Handler and Formatter Configuration

The Java `log4j` log file contains information on the configuration state of the HDFS Handler and the selected formatter. This information is output at the `INFO` log level. Sample output is as follows:

```
2015-09-21 10:05:11 INFO  AvroRowFormatter:156 - **** Begin Avro Row Formatter -
 Configuration Summary ****
  Operation types are always included in the Avro formatter output.
    The key for insert operations is [I].
    The key for update operations is [U].
    The key for delete operations is [D].
    The key for truncate operations is [T].
  Column type mapping has been configured to map source column types to an
 appropriate corresponding Avro type.
  Created Avro schemas will be output to the directory [./dirdef].
  Created Avro schemas will be encoded using the [UTF-8] character set.
  In the event of a primary key update, the Avro Formatter will ABEND.
  Avro row messages will not be wrapped inside a generic Avro message.
  No delimiter will be inserted after each generated Avro message.
**** End Avro Row Formatter - Configuration Summary ****

2015-09-21 10:05:11 INFO  HDFSHandler:207 - **** Begin HDFS Handler -
 Configuration Summary ****
  Mode of operation is set to tx.
  Data streamed to HDFS will be partitioned by table.
  Tokens will be included in the output.
  The HDFS root directory for writing is set to [/ogg].
  The maximum HDFS file size has been set to 1073741824 bytes.
  Rolling of HDFS files based on time is configured as off.
  Rolling of HDFS files based on write inactivity is configured as off.
  Rolling of HDFS files in the case of a metadata change event is enabled.
  HDFS partitioning information:
    The HDFS partitioning object contains no partitioning information.
HDFS Handler Authentication type has been configured to use [none]
**** End HDFS Handler - Configuration Summary ****
```

3

Using the HBase Handler

The HBase Handler allows you to populate HBase tables from existing Oracle GoldenGate supported sources.

Topics:

3.1 Overview

HBase is an open source Big Data application that emulates much of the functionality of a relational database management system (RDBMS). Hadoop is specifically designed to store large amounts of unstructured data. Conversely, data stored in databases and being replicated through Oracle GoldenGate is highly structured. HBase provides a method of maintaining the important structure of data, while taking advantage of the horizontal scaling that is offered by the Hadoop Distributed File System (HDFS).

3.2 Detailed Functionality

The HBase Handler takes operations from the source trail file and creates corresponding tables in HBase, and then loads change capture data into those tables.

HBase Table Names

Table names created in an HBase map to the corresponding table name of the operation from the source trail file. It is case-sensitive.

HBase Table Namespace

For two part table names (schema name and table name), the schema name maps to the HBase table namespace. For a three part table name like `Catalog.Schema.MyTable`, the create HBase namespace would be `Catalog_Schema`. HBase table namespaces are case sensitive. A `NULL` schema name is supported and maps to the default HBase namespace.

HBase Row Key

HBase has a similar concept of the database primary keys called the HBase row key. The HBase row key is the unique identifier for a table row. HBase only supports a single row key per row and it cannot be empty or `NULL`. The HBase Handler maps the primary key value into the HBase row key value. If the source table has multiple primary keys, then the primary key values are concatenated, separated by a pipe delimiter (`|`).You can configure the HBase row key delimiter.

The source table *must* have at least one primary key column. Replication of a table without a primary key causes the HBase Handler to abend.

HBase Column Family

HBase has the concept of a column family. A column family is a grouping mechanism for column data. Only a single column family is supported. Every HBase column must belong to a single column family. The HBase Handler provides a single column family per table that defaults to `cf`. The column family name is configurable by you. However, once a table is created with a specific column family name, reconfiguration of the column family name in the HBase example without first modify or dropping the table results in an abend of the Oracle GoldenGate Replicat processes.

3.3 Setting Up and Running the HBase Handler

HBase must be up and running either collocated with the HBase Handler process or on a machine that is network connectable from the machine hosting the HBase Handler process. Additionally the underlying HDFS single instance or clustered instance serving as the repository for HBase data must be up and running.

Instructions for configuring the HBase Handler components and running the handler are described in the following sections.

3.3.1 Classpath Configuration

You must include two things in the `gg.classpath` configuration variable in order for the HBase Handler to connect to HBase and stream data. The first is the `hbase-site.xml` file and the second are the HBase client jars. The HBase client jars must match the version of HBase to which the HBase Handler is connecting. The HBase client jars are *not* shipped with the Oracle GoldenGate for Big Data product.

HBase Handler Client Dependencies includes the listing of required HBase client jars by version.

The default location of the `hbase-site.xml` file is *`HBase_Home`*`/conf`.

The default location of the HBase client JARs is *`HBase_Home`*`/lib/*`.

If the HBase Handler is running on Windows, follow the Windows classpathing syntax.

The `gg.classpath` must be configured exactly as described. Pathing to the `hbase-site.xml` should simply contain the path with no wild card appended. The inclusion of the * wildcard in the path to the `hbase-site.xml` file will cause it not to be accessible. Conversely, pathing to the dependency jars should include the * wild card character in order to include all of the jar files in that directory in the associated classpath. Do not use `*.jar`. An example of a correctly configured `gg.classpath` variable is the following:

```
gg.classpath=/var/lib/hbase/lib/*:/var/lib/hbase/conf
```

3.3.2 HBase Handler Configuration

The following are the configurable values for the HBase Handler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 3-1 HBase Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handlerlist`	Required	Any string	None	Provides a name for the HBase Handler. The HBase Handler name is then becomes part of the property names listed in this table.
`gg.handler.`*`name`*`.type=hbase`	Required	-	-	Selects the HBase Handler for streaming change data capture into HBase
`gg.handler.`*`name`*`.hBaseColumnFamilyName`	Optional	Any String legal for an HBase column family name	`cf`	Column family is a grouping mechanism for columns in HBase. The HBase Handler only supports a single column family in the 12.2 release.
`gg.handler.`*`name`*`.includeTokens`	Optional	`true` | `false`	`false`	True indicates that token values will be included in the output to HBase. False means token values will be not be included.
`gg.handler.`*`name`*`.keyValueDelimiter`	Optional	Any string	`=`	Provides a delimiter between key values in a map. For example, `key=value,key1=value1,key2=value2`. Tokens are mapped values. Configuration value supports `CDATA[]` wrapping.
`gg.handler.`*`name`*`.keyValuePairDelimiter`	Optional	Any string	`,`	Provides a delimiter between key value pairs in a map. For example, `key=value,key1=value1,key2=value2key=value,key1=value1,key2=value2`. Tokens are mapped values. Configuration value supports `CDATA[]` wrapping.

Table 3-1 (Cont.) HBase Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.encoding`	Optional	Any encoding name or alias supported by Java.[1] For a list of supported options, visit the Oracle Java Documentation website at `https://docs.oracle.com/javase/8/docs/technotes/guides/intl/encoding.doc.html`	The native system encoding of the machine hosting the Oracle GoldenGate process	Determines the encoding of values written the HBase. HBase values are written as bytes.
`gg.handler.name.pkUpdateHandling`	Optional	`abend` | `update` | `delete-insert`	`abend`	Provides configuration for how the HBase Handler should handle update operations that change a primary key. Primary key operations can be problematic for the HBase Handler and require special consideration by you. • `abend` - indicates the process will abend • `update` - indicates the process will treat this as a normal update • `delete-insert` - indicates the process will treat this as a delete and an insert. The full before image is required for this feature to work properly. This can be achieved by using full supplemental logging in Oracle Database. Without full before and after row images the insert data will be incomplete.

Table 3-1 (Cont.) HBase Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.nullValueRepresentation`	Optional	Any string	`NULL`	Allows you to configure what will be sent to HBase in the case of a NULL column value. The default is `NULL`. Configuration value supports `CDATA[]` wrapping.
`gg.handler.name.authType`	Optional	`kerberos`	None	Setting this property to `kerberos` enables Kerberos authentication.
`gg.handler.name.kerberosKeytabFile`	Optional (Required if `authType=kerberos`)	Relative or absolute path to a Kerberos `keytab` file	-	The `keytab` file allows the HDFS Handler to access a password to perform a `kinit` operation for Kerberos security.
`gg.handler.name.kerberosPrincipal`	Optional (Required if `authType=kerberos`)	A legal Kerberos principal name (for example, `user/FQDN@MY.REALM`)	-	The Kerberos principal name for Kerberos authentication.
`gg.handler.name.hBase98Compatible`	Optional	`true` | `false`	`false`	Set this configuration property to `true` to enable integration with the HBase 0.98.*x* and 0.96.*x* releases. You can use this to solve compatibility problems with Cloudera CDH 5.7.*x*, 5.6.*x*, 5.5.*x* and 5.4.*x*. For more information, see HBase Handler
`gg.handler.name.rowkeyDelimiter`	Optional	Any string	|	Configures the delimiter between primary key values from the source table when generating the HBase `rowkey`. This property supports `CDATA[]` wrapping of the value to preserve whitespace if the user wishes to delimit incoming primary key values with a character or characters determined to be whitespace.

[1] For more Java information, see *Java Internalization Support* at `https://docs.oracle.com/javase/8/docs/technotes/guides/intl/`.

3.3.3 Sample Configuration

The following is a sample configuration for the HBase Handler from the Java Adapter properties file:

```
gg.handlerlist=hbase
gg.handler.hbase.type=hbase
gg.handler.hbase.mode=tx
gg.handler.hbase.hBaseColumnFamilyName=cf
gg.handler.hbase.includeTokens=true
gg.handler.hbase.keyValueDelimiter=CDATA[=]
gg.handler.hbase.keyValuePairDelimiter=CDATA[,]
gg.handler.hbase.encoding=UTF-8
gg.handler.hbase.pkUpdateHandling=abend
gg.handler.hbase.nullValueRepresentation=CDATA[NULL]
gg.handler.hbase.authType=none
```

3.3.4 Performance Considerations

At each transaction commit, the HBase Handler performs a flush call to flush any buffered data to the HBase region server. This must be done to maintain write durability. Flushing to the HBase region server is an expensive call and performance can be greatly improved by using the Replicat `GROUPTRANSOPS` parameter to group multiple smaller transactions in the source trail file into a larger single transaction applied to HBase. You can use Replicat base-batching by adding the configuration syntax in the Replicat configuration file.

Operations from multiple transactions are grouped together into a larger transaction, and it is only at the end of the grouped transaction that transaction commit is executed.

3.3.5 Security

HBase connectivity can be secured using Kerberos authentication. Follow the associated documentation for the HBase release to secure the HBase cluster. The HBase Handler can connect to Kerberos secured cluster. The HBase `hbase-site.xml` should be in handlers classpath with the `hbase.security.authentication` property set to `kerberos` and `hbase.security.authorization` property set to `true`.

Additionally, you must set the following properties in the HBase Handler Java configuration file:

```
gg.handler.{name}.authType=kerberos
gg.handler.{name}.keberosPrincipalName={legal Kerberos principal name}
gg.handler.{name}.kerberosKeytabFile={path to a keytab file that contains the
password for the Kerberos principal so that the Oracle GoldenGate HDFS handler can
programmatically perform the Kerberos kinit operations to obtain a Kerberos ticket}.
```

3.4 Metadata Change Events

Oracle GoldenGate 12.2 includes metadata in trail and can handle metadata change events at runtime. The HBase Handler can handle metadata change events at runtime as well. One of the most common scenarios is the addition of a new column. The result in HBase will be that the new column and its associated data will begin being streamed to HBase after the metadata change event.

It is important to understand that in order to enable metadata change events the entire Replication chain must be upgraded to Oracle GoldenGate 12.2. The 12.2 HBase Handler can work with trail files produced by Oracle GoldenGate 12.1 and greater. However, these trail files do not include metadata in trail and therefore metadata change events cannot be handled at runtime.

3.5 Additional Considerations

HBase has been experiencing changes to the client interface in the last few releases. HBase 1.0.0 introduced a new recommended client interface and the 12.2 HBase Handler has moved to the new interface to keep abreast of the most current changes. However, this does create a backward compatibility issue. The HBase Handler is not compatible with HBase versions older than 1.0.0. If an Oracle GoldenGate integration is required with 0.99.*x* or older version of HBase, this can be accomplished using the 12.1.2.1.*x* HBase Handler. Contact Oracle Support to obtain a ZIP file of the 12.1.2.1.*x* HBase Handler.

Common errors on the initial setup of the HBase Handler are classpath issues. The typical indicator is occurrences of the `ClassNotFoundException` in the Java `log4j` log file. The HBase client JARS do *not* ship with the Oracle GoldenGate for Big Data product. You must resolve the required HBase client JARS. HBase Handler Client Dependencies includes the listing of HBase client JARS for each supported version. Either the `hbase-site.xml` or one or more of the required client JARS are not included in the classpath. For instructions on configuring the classpath of the HBase Handler, see Classpath Configuration.

3.6 Troubleshooting the HBase Handler

Troubleshooting of the HBase Handler begins with the contents for the Java `log4j` file. Follow the directions in the Java Logging Configuration to configure the runtime to correctly generate the Java `log4j` log file. Review the following topics for additional help:

Java Classpath

HBase Connection Properties

Logging of Handler Configuration

3.6.1 Java Classpath

Issues with the Java classpath are one of the most common problems. An indication of a classpath problem is a `ClassNotFoundException` in the Java `log4j` log file. The Java `log4j` log file can be used to troubleshoot this issue. Setting the log level to `DEBUG` allows for logging of each of the jars referenced in the `gg.classpath` object to be logged to the log file. You can make sure that all of the required dependency jars are resolved by enabling `DEBUG` level logging, and then search the log file for messages like the following:

```
2015-09-29 13:04:26 DEBUG ConfigClassPath:74 -  ...adding to classpath:
 url="file:/ggwork/hbase/hbase-1.0.1.1/lib/hbase-server-1.0.1.1.jar"
```

3.6.2 HBase Connection Properties

The contents of the HDFS `hbase-site.xml` file (including default settings) are output to the Java `log4j` log file when the logging level is set to `DEBUG` or `TRACE`. It shows the connection properties to HBase. Search for the following in the Java `log4j` log file.

```
2015-09-29 13:04:27 DEBUG HBaseWriter:449 - Begin - HBase configuration object
contents for connection troubleshooting.
Key: [hbase.auth.token.max.lifetime] Value: [604800000].
```

A common error is for the `hbase-site.xml` file to be either not included in the classpath or a pathing error to the `hbase-site.xml`. In this case the HBase Handler will not be able to establish a connection to HBase and the Oracle GoldenGate process will abend. The following error will be reported in the Java `log4j` log.

```
2015-09-29 12:49:29 ERROR HBaseHandler:207 - Failed to initialize the HBase handler.
org.apache.hadoop.hbase.ZooKeeperConnectionException: Can't connect to ZooKeeper
```

Verify that the classpath correctly includes the `hbase-site.xml` file and that HBase is running.

3.6.3 Logging of Handler Configuration

The Java `log4j` log file contains information on the configuration state of the HBase Handler. This information is output at the `INFO` log level. Sample output is as follows:

```
2015-09-29 12:45:53 INFO HBaseHandler:194 - **** Begin HBase Handler - Configuration
Summary ****
  Mode of operation is set to tx.
  HBase data will be encoded using the native system encoding.
  In the event of a primary key update, the HBase Handler will ABEND.
  HBase column data will use the column family name [cf].
  The HBase Handler will not include tokens in the HBase data.
  The HBase Handler has been configured to use [=] as the delimiter between keys and
values.
  The HBase Handler has been configured to use [,] as the delimiter between key
values pairs.
  The HBase Handler has been configured to output [NULL] for null values.
Hbase Handler Authentication type has been configured to use [none]
```

4

Using the Flume Handler

This chapter explains the Flume Handler and includes examples so that you can understand this functionality.

Topics:

4.1 Overview

The Flume Handler is designed to stream change capture data from a Oracle GoldenGate trail to a Flume source. Apache Flume is an open source application for which the primary purpose is streaming data into Big Data applications. The Flume architecture contains three main components, sources, channels, and sinks that collectively make a pipeline for data. A Flume source publishes the data to a Flume channel. A Flume sink retrieves the data out of a Flume channel and streams the data to different targets. A Flume Agent is a container process that owns and manages a source, channel and sink. A single Flume installation can host many agent processes. The Flume Handler can stream data from a trail file to Avro or Thrift RPC Flume sources.

4.2 Setting Up and Running the Flume Handler

To run the Flume Handler, a Flume Agent configured with an Avro or Thrift Flume source must be up and running. Oracle GoldenGate can be collocated with Flume or located on a different machine. If located on a different machine the host and port of the Flume source must be reachable with a network connection. For instructions on how to configure and start a Flume Agent process, see the *Flume User Guide* at

```
https://flume.apache.org/releases/content/1.6.0/
FlumeUserGuide.pdf
```

Instructions for configuring the Flume Handler components and running the handler are described in the following sections.

4.2.1 Classpath Configuration

You must configure two things in the `gg.classpathconfiguration` variable for the Flume Handler to connect to the Flume source and run. The Flume Agent configuration file and the Flume client JARS. The Flume Handler uses the contents of the Flume Agent configuration file to resolve the host, port, and source type for the connection to Flume source. The Flume client libraries do *not* ship with Oracle GoldenGate for Big Data. The Flume client library versions must match the version of Flume to which the Flume Handler is connecting. For a listing for the required Flume client JAR files by version, see Flume Handler Client Dependencies.

The Oracle GoldenGate property, `gg.classpath`, must be set to include the following default locations:

- The default location of the `core-site.xml` file is *`Flume_Home`*`/conf`.
- The default location of the Flume client JARS is *`Flume_Home`*`/lib/*`.

The `gg.classpath` must be configured exactly as shown in the preceding example. Pathing to the Flume Agent configuration file should simply contain the path with no wild card appended. The inclusion of the `*wildcard` in the path to the Flume Agent configuration file will cause it not to be accessible. Conversely, pathing to the dependency jars should include the `*` wildcard character in order to include all of the JAR files in that directory in the associated classpath. Do not use `*.jar`. An example of a correctly configured `gg.classpath` variable is the following:

```
gg.classpath=dirprm/:/var/lib/flume/lib/*
```

If the Flume Handler and Flume are not collocated, then the Flume Agent configuration file and the Flume client libraries must be copied to the machine hosting the Flume Handler process.

4.2.2 Flume Handler Configuration

The following are the configurable values for the Flume Handler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Property Name	Property Value	Mandatory	Description
`gg.handlerlist`	`flumehandler` (choice of any name)	Yes	List of handlers. Only one is allowed with grouping properties `ON`.
`gg.handler.flumehandler.type`	`flume`	Yes	Type of handler to use.

Property Name	Property Value	Mandatory	Description
`gg.handler.flumehandler.format`	Formatter class or short code	No. Defaults to `delimitedtext`	The Formatter to be used. Can be one of the following: • `avro_row` • `avro_op` • `delimitedtext` • `xml` • `json` • `json_row` Alternatively, it is possible to write a custom formatter and include the fully qualified class name here.
`gg.handler.flumehandler.RpcClientPropertiesFile`	Any choice of filename	No. Defaults to `default-flume-rpc.properties`	Either the default `default-flume-rpc.properties` or a specified custom RPC client properties file should exist in the classpath.
`gg.handler.flumehandler.mode`	`op|tx`	No. Defaults to `op`	Operation mode or Transaction Mode. Java Adapter grouping options can be used only in tx mode.
`gg.handler.flumehandler.EventHeaderClass`	A custom implementation fully qualified class name	No. Defaults to `DefaultFlumeEventHeader`	Class to be used which defines what headers properties are to be added to a flume event.
`gg.handler.flumehandler.EventMapsTo`	`op|tx`	No. Defaults to `op`	Defines whether each flume event would represent an operation or a transaction. If `handler mode = op`, `EventMapsTo` will always be `op`.
`gg.handler.flumehandler.PropagateSchema`	`true|false`	No. Defaults to `false`	When set to `true`, the Flume handler will begin to publish schema events.
`gg.handler.flumehandler.includeTokens`	`true|false`	No. Defaults to `false`	When set to `true`, includes token data from the source trail files in the output. When set to `false` to excludes the token data from the source trail files in the output.

4.2.3 Sample Configuration

```
gg.handlerlist = flumehandler
gg.handler.flumehandler.type = flume
gg.handler.flumehandler.RpcClientPropertiesFile=custom-flume-rpc.properties
gg.handler.flumehandler.format =avro_op
gg.handler.flumehandler.mode =tx
gg.handler.flumehandler.EventMapsTo=tx
gg.handler.flumehandler.PropagateSchema =true
gg.handler.flumehandler.includeTokens=false
```

4.3 Data Mapping of Operations to Flume Events

This section explains how operation data from the Oracle GoldenGate trail file is mapped by the Flume Handler into Flume Events based on different configurations. A Flume Event is a unit of data that flows through a Flume agent. The Event flows from source to channel to sink and is represented by an implementation of the Event interface. An Event carries a payload (byte array) that is accompanied by an optional set of headers (string attributes).

The following topics are included:

4.3.1 Operation Mode

The configuration for the Flume Handler is the following in the Oracle GoldenGate Java configuration file.

```
gg.handler.{name}.mode=op
```

The data for each individual operation from Oracle GoldenGate trail file maps into a single Flume Event. Each event is immediately flushed to Flume. Each Flume Event will have the following headers.

- `TABLE_NAME:` The table name for the operation.
- `SCHEMA_NAME`: The catalog name (if available) and the schema name of the operation.
- `SCHEMA_HASH`: The hash code of the Avro schema. (Only applicable for Avro Row and Avro Operation formatters.)

4.3.2 Transaction Mode and `EventMapsTo` Operation

The configuration for the Flume Handler is the following in the Oracle GoldenGate Java configuration file.

```
gg.handler.flume_handler_name.mode=tx
gg.handler.flume_handler_name.EventMapsTo=op
```

The data for each individual operation from Oracle GoldenGate trail file maps into a single Flume Event. Events are flushed to Flume at transaction commit. Each Flume Event will have the following headers.

- `TABLE_NAME`: The table name for the operation.
- `SCHEMA_NAME`: The catalog name (if available) and the schema name of the operation.
- `SCHEMA_HASH`: The hash code of the Avro schema. (Only applicable for Avro Row and Avro Operation formatters.)

It is suggested to use this mode when formatting data as Avro or delimited text. It is important to understand that configuring Replicat batching functionality increases the number of operations processed in a transaction.

4.3.3 Transaction Mode and `EventMapsTo` Transaction

The configuration for the Flume Handler is the following in the Oracle GoldenGate Java configuration file.

```
gg.handler.flume_handler_name.mode=tx
gg.handler.flume_handler_name.EventMapsTo=tx
```

The data for all operations for a transaction from the source trail file are concatenated and mapped into a single Flume Event. The event is flushed at transaction commit. Each Flume Event has the following headers.

- `GG_TRANID`: The transaction ID of the transaction
- `OP_COUNT`: The number of operations contained in this Flume payload event

It is suggested to use this mode only when using self describing formats such as JSON or XML. In is important to understand that configuring Replicat batching functionality increases the number of operations processed in a transaction.

4.4 Performance Considerations

- Replicat-based grouping is recommended to be used to improve performance.
- Transaction mode with `gg.handler.flume_handler_name.EventMapsTo=tx` setting is recommended for best performance.
- The maximum heap size of the Flume Handler may affect performance. Too little heap may result in frequent garbage collections by the JVM. Increasing the maximum heap size of the JVM in the Oracle GoldenGate Java properties file may improve performance.

4.5 Metadata Change Events

The Flume Handler is adaptive to metadata change events. To handle metadata change events, the source trail files must have metadata in the trail file. However, this functionality depends on the source replicated database and the upstream Oracle GoldenGate Capture process to capture and replicate DDL events. This feature is not available for all database implementations in Oracle GoldenGate, see the Oracle GoldenGate installation and configuration guide for the appropriate database to understand if DDL replication is supported.

Whenever a metadata change occurs at the source, the flume handler will notify the associated formatter of the metadata change event. Any cached schema that the formatter is holding for that table will be deleted. The next time the associated formatter encounters an operation for that table the schema will be regenerated.

4.6 Example Flume Source Configuration

This section contains the following sample Flume source configurations:

Avro Flume Source

Thrift Flume Source

4.6.1 Avro Flume Source

The following is sample configuration for an Avro Flume source from the Flume Agent configuration file:

```
client.type = default
hosts = h1
hosts.h1 = host_ip:host_port
batch-size = 100
connect-timeout = 20000
request-timeout = 20000
```

4.6.2 Thrift Flume Source

The following is sample configuration for an Avro Flume source from the Flume Agent configuration file:

```
client.type = thrift
hosts = h1
hosts.h1 = host_ip:host_port
```

4.7 Advanced Features

This section contains the following advanced features of the Flume Handler that you may choose to implement:

Schema Propagation

Security

Fail Over Functionality

Load Balancing Functionality

4.7.1 Schema Propagation

The Flume Handler can propagate schemas to Flume. This is currently only supported for the Avro Row and Operation formatters. To enable this feature set the following property:

```
gg.handler.name.propagateSchema=true
```

The Avro Row or Operation Formatters generate Avro schemas on a just in time basis. Avro schemas are generated the first time an operation for a table is encountered. A metadata change event results in the schema reference being for a table being cleared and thereby a new schema is generated the next time an operation is encountered for that table.

When schema propagation is enabled the Flume Handler will propagate schemas an Avro Event when they are encountered.

Default Flume Schema Event headers for Avro include the following information:

- `SCHEMA_EVENT`: true
- `GENERIC_WRAPPER`: true or false
- `TABLE_NAME`: The table name as seen in the trail
- `SCHEMA_NAME`: The catalog name (if available) and the schema name

- SCHEMA_HASH: The hash code of the Avro schema

4.7.2 Security

Kerberos authentication for the Oracle GoldenGate for Big Data Flume Handler connection to the Flume source is possible. This feature is only supported in Flume 1.6.0 and later using the Thrift Flume source. It is enabled by changing the configuration of the Flume source in the Flume Agent configuration file.

Following is an example of the Flume source configuration from the Flume Agent configuration file that shows how to enable Kerberos authentication. The Kerberos principal name of the client and the server must be provided. The path to a Kerberos `keytab` file must be provided so that the password of the client principal can be resolved at runtime. For information on how to administrate Kerberos, Kerberos principals and their associated passwords, and the creation of a Kerberos `keytab` file, see the Kerberos documentation.

```
client.type = thrift
hosts = h1
hosts.h1 =host_ip:host_port
kerberos=true
client-principal=flumeclient/client.example.org@EXAMPLE.ORG
client-keytab=/tmp/flumeclient.keytab
server-principal=flume/server.example.org@EXAMPLE.ORG
```

4.7.3 Fail Over Functionality

It is possible to configure the Flume Handler so that it will fail over in the event that the primary Flume source becomes unavailable. This feature is currently only supported in Flume 1.6.0 and later using the Avro Flume source. It is enabled with Flume source configuration in the Flume Agent configuration file. The following is sample configuration for enabling fail over functionality:

```
client.type=default_failover
hosts=h1 h2 h3
hosts.h1=host_ip1:host_port1
hosts.h2=host_ip2:host_port2
hosts.h3=host_ip3:host_port3
max-attempts = 3
batch-size = 100
connect-timeout = 20000
request-timeout = 20000
```

4.7.4 Load Balancing Functionality

You can configure the Flume Handler so that produced Flume events are load balanced across multiple Flume sources. It is currently only supported in Flume 1.6.0 and later using the Avro Flume source. This feature is enabled with Flume source configuration in the Flume Agent configuration file. The following is sample configuration for enabling load balancing functionality:

```
client.type = default_loadbalance
hosts = h1 h2 h3
hosts.h1 = host_ip1:host_port1
hosts.h2 = host_ip2:host_port2
hosts.h3 = host_ip3:host_port3
backoff = false
maxBackoff = 0
host-selector = round_robin
batch-size = 100
```

```
connect-timeout = 20000
request-timeout = 20000
```

4.8 Troubleshooting the Flume Handler

This section contains information to help you troubleshoot various issues and includes the following topics:

4.8.1 Java Classpath

Issues with the Java classpath are one of the most common problems. The indication of a classpath problem is a `ClassNotFoundException` in the Oracle GoldenGate Java `log4j` log file. The Java `log4j` log file can be used to troubleshoot this issue. Setting the log level to `DEBUG` allows for logging of each of the jars referenced in the `gg.classpath` object to be logged to the log file. This way, you can make sure that all of the required dependency JARs are resolved, see Classpath Configuration.

4.8.2 Flume Flow Control Issues

The Flume Handler may write to the Flume source faster than the Flume sink can dispatch messages in some situations. When this happens, the Flume Handler will work for a while, but once Flume can no longer accept messages it will abend. The cause logged in the Oracle GoldenGate Java log file will likely be an `EventDeliveryException` indicating the Flume Handler was unable to send an event. Check the Flume log to for the exact cause of the problem. You may be able to reconfigure the Flume channel to increase capacity or increase the configuration for Java heap if the Flume Agent is experiencing an `OutOfMemoryException`. This may not entirely solve the problem. If the Flume Handler can push data to the Flume source faster than messages are dispatched by the Flume sink, then any change may simply extend the period the Flume Handler can run before failing.

4.8.3 Flume Agent Configuration File Not Found

The Flume Handler will abend at start up if the Flume Agent configuration file is not in the classpath. The result is generally a `ConfigException` listing the issue as an error loading the Flume producer properties. Check the `gg.handler.`*`name`*`.RpcClientProperites` configuration file to ensure that the naming of the Flume Agent properties file is correct. Check the GoldenGate `gg.classpath` properties to ensure that the classpath contains the directory containing the Flume Agent properties file. Also, check the classpath to ensure that the path to the Flume Agent properties file does not end with a wildcard * character.

4.8.4 Flume Connection Exception

The Flume Handler will abend at start up if it is unable to make a connection to the Flume source. The root cause of this problem will likely be reported as an

`IOExeption` in the Oracle GoldenGate Java `log4j` file indicating a problem connecting to Flume at a given host and port. Check the following:

- That the Flume Agent process is running and
- the Flume agent configuration file that the Flume Handler is accessing contains the correct host and port.

4.8.5 Other Failures

Review the contents of the Oracle GoldenGate Java `log4j` file to identify any other issues for correction.

5

Using the Kafka Handler

The Oracle GoldenGate for Big Data Kafka Handler is designed to stream change capture data from a Oracle GoldenGate trail to a Kafka topic. Additionally, the Kafka Handler provides optional functionality to publish the associated schemas for messages to a separate schema topic. Schema publication is currently only supported for Avro schemas because of the direct dependency of Avro messages upon an Avro schema.

Topics:

5.1 Overview

Apache Kafka is an open source, distributed, partitioned, and replicated messaging service. Kafka and its associated documentation are available at http://kafka.apache.org/.

Kafka can be run as a single instance or as a cluster on multiple servers. Each Kafka server instance is called a broker. A Kafka topic is a category or feed name to which messages are published by the producers and retrieved by consumers.

The Kafka Handler implements a Kafka producer that writes serialized change data capture from multiple source tables to either a single configured topic or separating source operations to different Kafka topics in Kafka when the topic name corresponds to the fully-qualified source table name.

5.2 Detailed Functionality

Transaction Versus Operation Mode

The Kafka Handler sends instances of the Kafka `ProducerRecord` class to the Kafka producer API which in turn publishes the `ProducerRecord` to a Kafka topic. The

Kafka `ProducerRecord` effectively is the implementation of a Kafka message. The `ProducerRecord` has two components, a key and a value. Both the key and value are represented as byte arrays by the Kafka Handler. This section describes how the Kafka Handler publishes data.

Transaction Mode

Transaction mode is indicated by the following configuration of the Kafka Handler:

```
gg.handler.name.Mode=tx
```

In Transaction Mode the serialized data for every operation in a transaction from the source Oracle GoldenGate trail files is concatenated. The contents of the concatenated operation data is the value of the Kafka `ProducerRecord` object. The key of the Kafka `ProducerRecord` object is NULL. The result is that Kafka messages comprise the data from 1 to *N* operations, where *N* is the number of operations in the transaction. With grouped transactions, all of the data for all of the operations for a grouped transaction are concatenated into a single Kafka message. The result can be very large Kafka messages containing data for a large number of operations.

Operation Mode

Operation mode is indicated by the following configuration of the Kafka Handler:

```
gg.handler.name.Mode=op
```

In Operation Mode the serialized data for each operation is placed into an individual `ProducerRecord` object as the value. The `ProducerRecord` key is the fully qualified table name of the source operation. The `ProducerRecord` is immediately sent using the Kafka Producer API. This means there is a 1 to 1 relationship between the incoming operations and the number of Kafka messages produced.

Blocking Versus Non-Blocking Mode

The Kafka Handler can send messages to Kafka in either blocking mode (synchronous) or non-blocking mode (asynchronous).

Blocking Mode

Blocking mode is set by the following configuration property of the Kafka Handler:

```
gg.handler.name.BlockingSend=true
```

Messages are delivered to Kafka on a synchronous basis. The Kafka Handler does not send the next message until the current message has been written to the intended topic and an acknowledgement has been received. Blocking mode provides the best guarantee of message delivery though the cost is reduced performance.

You must *never* set the Kafka Producer `linger.ms` variable when in blocking mode as this causes the Kafka producer to wait for the entire timeout period before sending the message to the Kafka broker. When this happens, the Kafka Handler is waiting for acknowledgement that the message has been sent while at the same time the Kafka Producer is buffering messages to be sent to the Kafka brokers.

Non-Blocking Mode

Non-blocking mode is set by the following configuration property of the Kafka Handler:

```
gg.handler.name.BlockingSend=false
```

Message are delivered to Kafka on an asynchronous basis. Kafka messages are published one after the other without waiting for acknowledgements. The Kafka Producer client may buffer incoming messages in order to increase throughput.

On each transaction commit, the Kafka producer flush call is invoked to ensure all outstanding messages are transferred to the Kafka cluster. This allows the Kafka Handler to safely checkpoint ensuring zero data loss. Invocation of the Kafka producer flush call is not affected by the `linger.ms` duration. This allows the Kafka Handler to safely checkpoint ensuring zero data loss.

You can control when the Kafka Producer flushes data to the Kafka Broker by a number of configurable properties in the Kafka producer configuration file. In order to enable batch sending of messages by the Kafka Producer both the `batch.size` and `linger.ms` Kafka Producer properties must be set in the Kafka producer configuration file. The `batch.size` controls the maximum number of bytes to buffer before a send to Kafka while the `linger.ms` variable controls the maximum milliseconds to wait before sending data. Data is sent to Kafka once the `batch.size` is reached or the `linger.ms` period expires, whichever comes first. Setting the `batch.size` variable only causes messages to be sent immediately to Kafka.

Publishing to Multiple Topics

The Kafka Handler allows operation data from the source trail file to be published to separate topics based on the corresponding table name of the operation data. It allows the sorting of operation data from the source trail file by the source table name. It is enabled by setting the following configuration property in the Java Adapter properties file as follows:

```
gg.handler.kafka.topicPartitioning=table
gg.handler.kafka.mode=op
```

The mode *must* be set to `op` and the Kafka topic name used is the fully qualified table name of the source operation.

You can publish to multiple topics using the Kafka Handler. For example, you could publish one topic per table by setting `gg.handler.`*`name`*`.topicPartitioning` property to table.

The topics are automatically created and with the topic name equal to the fully-qualified table name.

Kafka Broker Settings

To enable the automatic creation of topics, set the `auto.create.topics.enable` property to `true` in the Kafka Broker Configuration. The default value for this property is true.

If the `auto.create.topics.enable` property is set to `false` in Kafka Broker configuration, then all the required topics should be created manually before starting the Replicat process.

Schema Propagation

The schema data for all tables is delivered to the schema topic configured with the `schemaTopicName` property. For more information , see Schema Propagation.

Note:

Multiple topics are supported in the *op* mode only. For example, when `gg.handler.kafkahandler.topicPartitioning` is set to table then `gg.handler.kafkahandler.mode` should be set to `op`.

5.3 Setting Up and Running the Kafka Handler

You must install and correctly configure Kafka either as a single node or a clustered instance. Information on how to install and configure Apache Kafka is available at:

http://kafka.apache.org/documentation.html

If you are using a Kafka distribution other than Apache Kafka, then consult the documentation for your specific Kafka distribution for installation and configuration instructions.

Zookeeper, a prerequisite component for Kafka and Kafka broker (or brokers), must be up and running.

Oracle recommends and considers it best practice that the data topic and the schema topic (if applicable) are preconfigured on the running Kafka brokers. You can create Kafka topics dynamically; though this relies on the Kafka brokers being configured to allow dynamic topics.

If the Kafka broker is not collocated with the Kafka Handler process, then the remote host port must be reachable from the machine running the Kafka Handler.

Instructions for configuring the Kafka Handler components and running the handler are described in the following sections.

Classpath Configuration

Kafka Handler Configuration

Java Adapter Properties File

Kafka Producer Configuration File

5.3.1 Classpath Configuration

Two things must be configured in the `gg.classpath` configuration variable so that the Kafka Handler can to connect to Kafka and run. The required items are the Kafka Producer properties file and the Kafka client JARs. The Kafka client JARs must match the version of Kafka that the Kafka Handler is connecting to. For a listing of the required client JAR files by version, see Kafka Handler Client Dependencies.

The recommending storage location for the Kafka Producer properties file is the Oracle GoldenGate `dirprm` directory.

The default location of the Kafka client JARs is *`Kafka_Home`*`/libs/*.`

The `gg.classpath` must be configured precisely. Pathing to the Kafka Producer Properties file should simply contain the path with no wildcard appended. The inclusion of the `*` wildcard in the path to the Kafka Producer Properties file will cause it not to be picked up. Conversely, pathing to the dependency JARs should include the `*` wild card character in order to include all of the JAR files in that directory in the associated classpath. Do *not* use `*.jar`. The following is an example of the correctly configured classpath:

```
gg.classpath={kafka install dir}/libs/*
```

5.3.2 Kafka Handler Configuration

The following are the configurable values for the Kafka Handler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 5-1 Configuration Properties for Kafka Handler

Property Name	Property Value	Mandatory	Description
`gg.handlerlist`	*`name`* (choice of any name)	Yes	List of handlers to be used.
`gg.handler.`*`name`*`.Type`	`kafka`	Yes	Type of handler to use. For example, Kafka, Flume, HDFS.
`gg.handler.`*`name`*`.KafkaProducerConfigFile`	Any custom file name	No. Defaults to `kafka-producer-default.properties`	Filename in classpath that holds Apache Kafka properties to configure the Apache Kafka producer.
`gg.handler.`*`name`*`.TopicName`	TopicName	Yes	Name of the Kafka topic where payload records will be sent.
`gg.handler.`*`name`*`.Format`	Formatter class or short code	No. Defaults to `delimitedtext`.	Formatter to use to format payload. Can be one of `xml`, `delimitedtext`, `json`, `json_row`, `avro_row`, `avro_op`
`gg.handler.`*`name`*`.SchemaTopicName`	Name of the schema topic	Yes, when schema delivery is required.	Topic name where schema data will be delivered. If this property is not set, schema will not be propagated. Schemas will be propagated only for Avro formatters.
`gg.handler.`*`name`*`.SchemaPrClassName`	Fully qualified class name of a custom class that implements Oracle GoldenGate for Big Data Kafka Handler's `CreateProducerRecord` Java Interface	No. Defaults to provided implementation class: `oracle.goldengate.handler.kafka`.Default `ProducerRecord`	Schema is also propagated as a `ProducerRecord`. The default key here is the fully qualified table name. If this needs to be changed for schema records, the custom implementation of the `CreateProducerRecord` interface needs to be created and this property needs to be set to point to the fully qualified name of the new class.

Table 5-1 (Cont.) Configuration Properties for Kafka Handler

Property Name	Property Value	Mandatory	Description
`gg.handler.name.BlockingSend`	`true` | `false`	No. Defaults to `false`.	If this property is set to true, then delivery to Kafka is made to work in a completely synchronous model. The next payload will be sent only after the current payload has been written out to the intended topic and an acknowledgement has been received. In transaction mode, this provides exactly once semantics. If this property is set to false, then delivery to Kafka is made to work in an asynchronous model. Payloads are sent one after the other without waiting for acknowledgements. Kafka internal queues may buffer contents to increase throughput. Checkpoints are made only when acknowledgements are received from Kafka brokers using Java Callbacks.
`gg.handler.name.ProducerRecordClass`	Fully qualified class name of a custom class that implements Oracle GoldenGate for Big Data Kafka Handler's `CreateProducerRecord` Java Interface	No. Defaults to out-of-box provided implementation class:`oracle.goldengate.handler.kafka.DefaultProducerRecord`	The unit of data in Kafka - a `ProducerRecord` holds the key field with the value representing the payload. This key is used for partitioning a Kafka Producer record that holds change capture data. By default, the fully qualified table name is used to partition the records. In order to change this key or behavior, the `CreateProducerRecord` Kafka Handler Interface needs to be implemented and this property needs to be set to point to the fully qualified name of the custom `ProducerRecord` class.
`gg.handler.name.mode`	`tx/op`	No. Defaults to `tx`.	With Kafka Handler operation mode, each change capture data record (Insert, Update, Delete etc) payload will be represented as a Kafka Producer Record and will be flushed one at a time. With Kafka Handler in transaction mode, all operations within a source transaction will be represented by as a single Kafka Producer record. This combined byte payload will be flushed on a transaction Commit event.
`gg.handler.name.topicPartitioning`	`none | table`	None	Controls whether data published into Kafka should be partitioned by table. Set to table, the data for different tables are written to different Kafka topics. Set to none, the data from different tables are interlaced in the same topic as configured in `topicName`property.

5.3.3 Java Adapter Properties File

A sample configuration for the Kafka Handler from the Adapter properties file is:

```
gg.handlerlist = kafkahandler
gg.handler.kafkahandler.Type = kafka
gg.handler.kafkahandler.KafkaProducerConfigFile = custom_kafka_producer.properties
gg.handler.kafkahandler.TopicName = oggtopic
gg.handler.kafkahandler.Format = avro_op
gg.handler.kafkahandler.SchemaTopicName = oggSchemaTopic
gg.handler.kafkahandler.ProducerRecordClass = com.company.kafka.CustomProducerRecord
gg.handler.kafkahandler.SchemaPrClassName = com.company.kafkaProdRec.SchemaRecord
gg.handler.kafkahandler.Mode = tx
gg.handler.kafkahandler.BlockingSend = true
```

A sample Replicat configuration and a Java Adapter Properties file for a Kafka integration can be found at the following directory:

GoldenGate_install_directory/AdapterExamples/big-data/kafka

5.3.4 Kafka Producer Configuration File

The Kafka Handler must access a Kafka producer configuration file in order publish messages to Kafka. The file name of the Kafka producer configuration file is controlled by the following configuration in the Kafka Handler properties.

gg.handler.kafkahandler.KafkaProducerConfigFile=*custom_kafka_producer*.properties

The Kafka Handler will attempt to locate and load the Kafka producer configuration file using the Java classpath. Therefore the Java classpath must include the directory containing the Kafka Producer Configuration File.

The Kafka producer configuration file contains Kafka proprietary properties. The Kafka documentation provides configuration information for the 0.8.2.0 Kafka producer interface properties. The Kafka Handler used these properties to resolve the host and port of the Kafka brokers and properties in the Kafka producer configuration file control the behavior of the interaction between the Kafka producer client and the Kafka brokers.

A sample of configuration file for the Kafka producer is as follows:

```
bootstrap.servers=localhost:9092
acks = 1
compression.type = gzip
reconnect.backoff.ms = 1000

value.serializer = org.apache.kafka.common.serialization.ByteArraySerializer
key.serializer = org.apache.kafka.common.serialization.ByteArraySerializer
# 100KB per partition
batch.size = 102400
linger.ms = 10000
max.request.size = 5024000
send.buffer.bytes = 5024000
```

5.4 Schema Propagation

The Kafka Handler provides the ability to publish schemas to a schema topic. Currently the Avro Row and Operation formatters are the only formatters that are

enabled for schema publishing. If the Kafka Handler `schemaTopicName` property is set, then the schema is published for the following events:

- The Avro schema for a specific table will be published the first time an operation for that table is encountered.
- If the Kafka Handler receives a metadata change event, the schema is flushed. The regenerated Avro schema for a specific table is published the next time an operation for that table is encountered.
- If the Avro wrapping functionality is enabled, then the generic wrapper Avro schema is published the first time any operation is encountered. The generic wrapper Avro schema functionality can be enabled in the Avro formatter configuration, see Avro Row Formatter and Avro Operation Formatter.

The Kafka `ProducerRecord` value is the schema and the key will be the fully qualified table name.

Avro over Kafka can be problematic because of the direct dependency of Avro messages on an Avro schema. Avro messages are binary so are not human readable. To deserialize an Avro message, the receiver must first have the correct Avro schema. Since each table from the source database results in a separate Avro schema, this can be problematic. The receiver of a Kafka message cannot determine which Avro schema to use to deserialize individual messages when the source Oracle GoldenGate trail file includes operations from multiple tables. To solve this problem, you can wrap the specialized Avro messages in a generic Avro message wrapper. This generic Avro wrapper provides the fully-qualified table name, the hashcode of the schema string, and the wrapped Avro message. The receiver can use the fully-qualified table name and the hashcode of the schema string to resolve the associated schema of the wrapped message, and then use that schema to deserialize the wrapped message.

5.5 Performance Considerations

Oracle recommends that you do *not* to use the `linger.ms` setting in the Kafka producer `config` file when `gg.handler.`*`name`*`.BlockingSend=true`. This causes each send to block for at least `linger.ms` leading to major performance issues because the Kafka Handler configuration and the Kafka Producer configuration are in conflict with each other. This configuration results a temporary deadlock scenario where the Kafka Handler is waiting for send acknowledgement while the Kafka producer is waiting for more messages before sending. The deadlock resolves once the `linger.ms` period has expired. This behavior repeats for every message sent.

For the best performance, Oracle recommends that you set the Kafka Handler to operate in operation mode using non-blocking (asynchronous) calls to the Kafka producer by using the following configuration in your Java Adapter properties file:

```
gg.handler.name.mode = op
gg.handler.name.BlockingSend = false
```

Additionally the recommendation is to set the `batch.size` and linger.ms values in the Kafka Producer properties file. The values to set the `batch.size` and `linger.ms` values are highly dependent upon the use case scenario. Typically, higher values results in better throughput but latency is increased. Smaller values in these properties reduces latency though overall throughput decreases. If you have a high volume of input data from the source trial files, then set the `batch.size` and `linger.ms` size to as high as possible.

Use of the `Replicat` variable `GROUPTRANSOPS` also improves performance. The recommended setting for that is `10000`.

If you need to have the serialized operations from the source trail file delivered in individual Kafka messages, then the Kafka Handler must be set to operation mode.

```
gg.handler.name.mode = op
```

The result is many more Kafka messages and performance is adversely affected.

5.6 Security

Kafka version 0.9.0.0 introduced security through SSL/TLS or Kerberos. The Kafka Handler can be secured using SSL/TLS or Kerberos. The Kafka producer client libraries provide an abstraction of security functionality from the integrations utilizing those libraries. The Kafka Handler is effectively abstracted from security functionality. Enabling security requires setting up security for the Kafka cluster, connecting machines, and then configuring the Kafka producer properties file, that the Kafka Handler uses for processing, with the required security properties. For detailed instructions about securing the Kafka cluster, see the Kafka documentation at

http://kafka.apache.org/documentation.html#security_configclients

5.7 Metadata Change Events

Metadata change events are now handled in the Kafka Handler. This is only relevant if you have configured a schema topic and the formatter used supports schema propagation (currently Avro row and Avro Operation formatters). The next time an operation is encountered for a table for which the schema has changed, the updated schema is published to the schema topic.

To support metadata change events, the Oracle GoldenGate process capturing changes in the source database must support the Oracle GoldenGate metadata in trail feature, which was introduced in Oracle GoldenGate 12*c* (12.2).

5.8 Snappy Considerations

The Kafka Producer Configuration file supports the use of compression. One of the configurable options is Snappy, which is an open source compression and decompression (`codec`) library that tends to provide better performance than other `codec` libraries. The Snappy JAR does not run on all platforms. Snappy seems to work on Linux systems though may or may not work on other UNIX and Windows implementations. If you want to use Snappy compression, they you should test Snappy on all required systems before implementing compression using Snappy. If Snappy does not port to all required systems, then Oracle recommends using an alternate `codec` library.

5.9 Troubleshooting

This section details troubleshooting options. Review the following topics for additional help:

Verify the Kafka Setup

Classpath Issues

Invalid Kafka Version

5.9.1 Verify the Kafka Setup

You can use the command line Kafka producer to write dummy data to a Kafka topic and a Kafka consumer can be used to read this data from the Kafka topic. Use this to verify the set up and read write permissions to Kafka topics on disk. For further details, refer to the online Kafka documentation at

`http://kafka.apache.org/documentation.html#quickstart`

5.9.2 Classpath Issues

One of the most common problems is Java classpath problems. Typically this is a `ClassNotFoundException` problem in the `log4j` log file though may be an error resolving the classpath if there is a typographic error in the `gg.classpath` variable. The Kafka client libraries do *not* ship with the Oracle GoldenGate for Big Data product. The requirement is on you to obtain the correct version of the Kafka client libraries and to properly configure the `gg.classpath` property in the Java Adapter Properties file to correctly resolve the Java the Kafka client libraries as described in Classpath Configuration.

5.9.3 Invalid Kafka Version

The Kafka Handler does *not* support Kafka versions 0.8.2.2 and older. The typical outcome when running with an unsupported version of Kafka is a runtime Java exception, `java.lang.NoSuchMethodError`, indicating that the `org.apache.kafka.clients.producer.KafkaProducer.flush()` method cannot be found. If this error is encountered, you must migrate to Kafka version 0.9.0.0 or later.

5.9.4 Kafka Producer Properties File Not Found

Typically, this problem is in the following exception.

```
ERROR 2015-11-11 11:49:08,482 [main] Error loading the kafka producer properties
```

The `gg.handler.kafkahandler.KafkaProducerConfigFile` configuration variable should be verified that the Kafka Producer Configuration file name is set correctly. Check the `gg.classpath` variable to verify that the classpath includes the path to the Kafka Producer properties file and that the path to the properties file does not contain a `*` wildcard at the end.

5.9.5 Kafka Connection Problem

This problem occurs when the Kafka Handler is unable to connect to Kafka with the following warnings:

```
WARN 2015-11-11 11:25:50,784 [kafka-producer-network-thread | producer-1] WARN
(Selector.java:276) - Error in I/O with localhost/127.0.0.1
java.net.ConnectException: Connection refused
```

The connection retry interval expires and the Kafka Handler process abends. Ensure that the Kafka Brokers is running and that the host and port provided in the Kafka Producer Properties file is correct. Network shell commands (such as, `netstat -l`)

can be used on the machine hosting the Kafka broker to verify that Kafka is listening on the expected port.

6

Using the Cassandra Handler

This chapter explains the Cassandra Handler and includes examples so that you can understand this functionality.

Topics:

6.1 Overview

The Cassandra Handler provides the interface to Apache Cassandra databases.

Apache Cassandra is a NoSQL Database Management System designed to store large amounts of data. A Cassandra cluster configuration provides horizontal scaling and replication of data across multiple machines. It can provide high-availability and eliminate a single point of failure by replicating data to multiple nodes within a Cassandra cluster. Apache Cassandra is open-source and designed to run on low cost commodity hardware.

Cassandra relaxes the axioms of traditional Relational Database Management Systems regarding atomicity, consistency, isolation, and durability. When considering implementing Cassandra it is important to understand its differences from a traditional RDBMS and how those differences affect your specific use case.

Cassandra provides eventual consistency. Under the eventual consistency model, accessing the state of data for a specific row will eventually return the latest state of the data for that row as defined by the most recent change. However, there may be a latency period between the creation and modification of the state of a row and what is returned when the state of that row is queried. The promise of eventual consistency is that the latency period is predictable based on your Cassandra configuration and the level of work load that your Cassandra cluster is currently under. See the Apache Cassandra website for more information:

http://cassandra.apache.org/

The Cassandra Handler provides some control over consistency with the configuration of the `gg.handler.name.consistencyLevel` property in the Java Adapter properties file.

6.2 Detailed Functionality

6.2.1 Cassandra Data Types

Cassandra provides a number of column data types and most of these data types are supported by the Cassandra Handler. A data type conversion from the column value in the source trail file to the corresponding Java type representing the Cassandra column type in the Cassandra Handler is required. This data conversion process does introduce the risk of a runtime conversion error. A poorly mapped field (like `varchar` as the source containing alpha numeric data to a Cassandra `int`) may cause a runtime error and cause the Cassandra Handler to abend. The following is a link to the Cassandra Java type mappings.

https://github.com/datastax/java-driver/tree/3.x/manual#cql-to-java-type-mapping

The following Cassandra column data types are not supported:

- `list`
- `map`
- `set`
- `tuple`

Certain use cases may exist where the data may require specialized processing to convert it to the corresponding Java type for intake into Cassandra. If this is the case, you have these options:

1. You may be able to use the general regular expression search and replace functionality to get the source column value data formatted into a way that can be then converted into the Java data type for use in Cassandra.
2. You could implement or extend the default data type conversion logic to override it with your custom logic for your use case. If this is required, contact Oracle Support for guidance.

6.2.2 Catalog, Schema, Table, and Column Name Mapping

Traditional RDBMSs separate structured data into tables. Related tables are included in higher-level collections called databases. Cassandra contains both of these concepts.

Tables in an RDBMS are also tables in Cassandra while databases in an RDBMS are keyspaces in Cassandra.

It is important to understand how data maps from the metadata definition in the source trail file are mapped to the corresponding keyspace and table in Cassandra. Source tables are generally either two-part names defined as `schema.table`or three-part names defined as `catalog.schema.table`. The following table explains how catalog, schema, and table names map into Cassandra. Note that unless special syntax is used, Cassandra converts all keyspace, table names, and column names to lowercase.

Table Name in Source Trail File	Cassandra Keyspace Name	Cassandra Table Name
`QASOURCE.TCUSTMER`	`qasource`	`tcustmer`
`dbo.mytable`	`dbo`	`mytable`
`GG.QASOURCE.TCUSTORD`	`gg_qasource`	`tcustord`

6.2.3 DDL Functionality

6.2.3.1 Keyspaces

The Cassandra Handler does *not* automatically create keyspaces in Cassandra. Keyspaces in Cassandra define a replication factor, the replication strategy, and topology. The Cassandra Handler does not possess enough information to create the keyspaces so you must manually create keyspaces.

You can create keyspaces in Cassandra using the `CREATE KEYSPACE` command from the Cassandra shell.

6.2.3.2 Tables

The Cassandra Handler can automatically create tables in Cassandra if you configure it to do so. The source table definition may be a poor source of information to create tables in Cassandra. Primary keys in Cassandra are divided into:

- **Partitioning keys** that define how data for a table is separated into partitions in Cassandra.
- **Clustering keys** that define the order of items within a partition.

The default mapping for automated table creation is that the first primary key is the partition key and any additional primary keys are mapped as clustering keys.

Automated table creation by the Cassandra Handler may be fine for proof of concept though may result in data definitions that do not scale well. Creation of tables in

Cassandra with poorly constructed primary keys may result in reduced performance for ingest and retrieval as the volume of data stored in Cassandra increases. Oracle recommends that you analyze the metadata of your replicated tables then strategically manually create the corresponding tables in Cassandra that are properly partitioned and clustered that can scale well.

Primary key definitions for tables in Cassandra are immutable once created. Changing a Cassandra table primary key definition requires the following manual steps:

1. Create a staging table.
2. Populate the data in the staging table from original table.
3. Drop the original table.
4. Recreate the original table with the modified primary key definitions.
5. Populate the data in the original table from the staging table.
6. Drop the staging table.

6.2.3.3 Add Column Functionality

You can configure the Cassandra Handler to add columns that exist in the source trail file table definition though are missing in the Cassandra table definition. The Cassandra Handler can accommodate metadata change events of adding a column. A reconciliation process occurs that reconciles the source table definition to the Cassandra table definition. When configured to add columns, any columns found in the source table definition that do not exist in the Cassandra table definition are added. The reconciliation process for a table occurs after application start up the first time an operation for the table is encountered. The reconciliation process reoccurs after a metadata change event on a source table, when the first operation for the source table is encountered after the change event.

6.2.3.4 Drop Column Functionality

You can configure the Cassandra Handler to drop columns that do not exist in the source trail file definition though exist in the Cassandra table definition. The Cassandra Handler can accommodate metadata change events of dropping a column. A reconciliation process occurs that reconciles the source table definition to the Cassandra table definition. When configured to drop columns any columns found in the Cassandra table definition that are not in the source table definition are dropped.

Caution:

Dropping a column is potentially dangerous because it is permanently removing data from a Cassandra table. You should carefully consider your use case before configuring this mode.

Note:

Primary key columns cannot be dropped. Attempting to do so results in an abend.

Note:

Column name changes are not well-handled because there is no actual DDL processing. A column name change event on the source database appears to the Cassandra Handler like dropping an existing column and adding a new column.

6.2.4 Operation Processing

The Cassandra Handler pushes operations to Cassandra using either the asynchronous or synchronous API. In asynchronous mode, operations are flushed at transaction commit (grouped transaction commit using `GROUPTRANSOPS`) to ensure write durability. The Cassandra Handler does not interface with Cassandra in a transactional way.

Insert, update, and delete operations are processed differently in Cassandra than a traditional RDBMS. The following explains how insert, update, and delete operations are interpreted by Cassandra:

- Inserts – If the row does not already exist in Cassandra, then an insert operation is processed as an insert. If the row already exists in Cassandra, then an insert operation is processed as an update.
- Updates – If a row does not exist in Cassandra, then an update operation is processed as an insert. If the row already exists in Cassandra, then an update operation is processed as insert.
- Delete – If the row does not exist in Cassandra, then a delete operation has no effect. If the row exists in Cassandra, then a delete operation is processed as a delete.

The state of the data in Cassandra is eventually idempotent. You can replay the source trail files or replay sections of the trail files. Ultimately, the state of the Cassandra database should be the same regardless of the number of times the trail data was written into Cassandra.

6.2.5 Compressed Updates vs. Full Image Updates

Oracle GoldenGate allows you to control the data that is propagated to the source trail file in the event of an update. The data for an update in the source trail file is either a compressed or a full image of the update and the column information is provided as follows:

Compressed
For the primary keys and the columns for which the value changed. Data for columns that did not change is not provided in the trail file.

Full Image
For all columns including primary keys, columns for which the value changed, and columns for which the value did not change.

The amount of available information on an update is important to the Cassandra Handler. If the source trail file contains full images of the change data then the Cassandra Handler can use prepared statements to perform row updates in Cassandra. Full images also allow the Cassandra Handler to perform primary key updates for a row in Cassandra. In Cassandra, primary keys are immutable so an

update that changes a primary key must be treated as a delete and an insert. Conversely, compressed updates means that prepared statements cannot be used for Cassandra row updates. Simple statements identifying the changing values and primary keys must be dynamically created then executed. Compressed updates mean that primary key updates are not possible so the result is that the Cassandra Handler will abend.

You must set the control properties, `gg.handler.name.compressedUpdates` and `gg.handler.name.compressedUpdatesfor`, so that the handler expects either compressed or full image updates.

The default value, `true`, means that the Cassandra Handler expects compressed updates. Prepared statements are not be used for updates and primary key updates cause the handler to abend.

Setting the value to `false` means that prepared statements are used for updates and primary key updates can be processed. When the source trail file does not contain full image data, it is dangerous and can lead to corrupted data. This is because columns for which the data is missing are considered null and the null value is pushed to Cassandra. If you have doubts about whether the source trail files contains compressed or full image data, then you should set `gg.handler.name.compressedUpdates` to `true`.

In addition, CLOB and BLOB data types do not propagate LOB data in updates unless the LOB column value changed. So if the source tables contain LOB data, then you should set `gg.handler.name.compressedUpdates` to `true`.

6.2.6 Primary Key Updates

Primary Key Updates

Primary key values for a row in Cassandra are immutable. An update operation that changes any primary key value for a Cassandra row must be treated as a delete and insert. The Cassandra Handler can process update operations that result in the change of a primary key in Cassandra only as a delete and insert. To successfully process this operation, the source trail file *must* contain the complete before and after change data images for all columns. The `gg.handler.name.compressed` configuration property of the Cassandra Handler must be set to `false` for primary key updates to be successfully processed.

6.3 Setting Up and Running the Cassandra Handler

You must configure the following:

Get the Driver Libraries

The Datastax Java Driver for Cassandra does not ship with Oracle GoldenGate for Big Data. The recommended version of the Datastax Java Driver for Cassandra is 3.1 and you download it at:

https://github.com/datastax/java-driver

Set the Classpath

You must configure the `gg.classpath` configuration property in the Java Adapter properties file to specify the JARs for the Datastax Java Driver for Cassandra.

```
gg.classpath={download_dir}/cassandra-java-driver-3.1.0/*:{download_dir}/cassandra-
java-driver-3.1.0/lib/*
```

Instructions for configuring the Cassandra Handler components and running the handler are described in the following sections.

6.3.1 Cassandra Handler Configuration

The following are the configurable values for the Cassandra Handler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 6-1 Cassandra Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handlerlist`	Required	Any string	None	Provides a name for the Cassandra Handler. The Cassandra Handler name is then becomes part of the property names listed in this table.
`gg.handler.`*`name`*`.type=cassandra`	Required	`cassandra`	None	Selects the Cassandra Handler for streaming change data capture into Cassandra.
`gg.handler.`*`name`*`.mode`	Optional	`op` | `tx`	`op`	The default is recommended. In `op` mode, operations are processed as received. In `tx` mode, operations are cached and processed at transaction commit. The `tx`mode is slower and creates a larger memory footprint.
`gg.handler.`*`name`*`.contactPoints=`	Optional	A comma separated list of host names that the Cassandra Handler will connect to.	`localhost`	A comma separated list of the Cassandra host machines for the driver to establish an initial connection to the Cassandra cluster. This configuration property does *not* need to include all the machines enlisted in the Cassandra cluster. By connecting to a single machine, the driver can learn about other machines in the Cassandra cluster and establish connections to those machines as required.
`gg.handler.`*`name`*`.username`	Optional	A legal username string.	None	A username for the connection to Cassandra. It is required if Cassandra is configured to require credentials.
`gg.handler.`*`name`*`.password`	Optional	A legal password string.	None	A password for the connection to Cassandra. It is required if Cassandra is configured to require credentials.

Table 6-1 (Cont.) Cassandra Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.compressedUpdates`	Optional	`true` | `false`	`true`	Sets the Cassandra Handler whether to or not to expect full image updates from the source trail file. Set to `true` means that updates in the source trail file only contain column data for the primary keys and for columns that changed. The Cassandra Handler executes updates as simple statements updating only the columns that changed. Setting it to `false` means that updates in the source trail file contain column data for primary keys and all columns regardless of whether the column value has changed. The Cassandra Handler is able to use prepared statements for updates, which can provide better performance for streaming data to Cassandra.
`gg.handler.name.ddlHandling`	Optional	`CREATE` | `ADD` | `DROP` in any combination with values delimited by a comma	None	Configures the Cassandra Handler for the DDL functionality to provide. Options include `CREATE`, `ADD`, and `DROP`. These options can be set in any combination delimited by commas. When `CREATE` is enabled the Cassandra Handler creates tables in Cassandra if a corresponding table does not exist. When `ADD` is enabled the Cassandra Handler adds columns that exist in the source table definition that do *not* exist in the corresponding Cassandra table definition. When `DROP` is enable the handler drops columns that exist in the Cassandra table definition that do *not* exist in the corresponding source table definition.
`gg.handler.name.cassandraMode`	Optional	`async` | `sync`	`async`	Sets the interaction between the Cassandra Handler and Cassandra. Set to `async` for asynchronous interaction. Operations are sent to Cassandra asynchronously and then flushed at transaction commit to ensure durability. Asynchronous will provide better performance. Set to `sync` for synchronous interaction. Operations are sent to Cassandra synchronously.

Table 6-1 (Cont.) Cassandra Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.consistencyLevel`	Optional	`ALL | ANY | EACH_QUORUM | LOCAL_ONE | LOCAL_QUORUM | ONE | QUORUM | THREE | TWO`	The Cassandra default.	Sets the consistency level for operations with Cassandra. It configures the criteria that must be met for storage on the Cassandra cluster when an operation is executed. Lower levels of consistency can provide better performance while higher levels of consistency are safer. You can review details about the write consistency levels for Cassandra at: https://docs.datastax.com/en/cassandra/3.x/cassandra/dml/dmlConfigConsistency.html

6.3.2 Sample Configuration

The following is sample configuration for the Cassandra Handler from the Java Adapter properties file:

```
gg.handlerlist=cassandra

#The handler properties
gg.handler.cassandra.type=cassandra
gg.handler.cassandra.mode=op
gg.handler.cassandra.contactPoints=localhost
gg.handler.cassandra.ddlHandling=CREATE,ADD,DROP
gg.handler.cassandra.compressedUpdates=true
gg.handler.cassandra.cassandraMode=async
gg.handler.cassandra.consistencyLevel=ONE
```

6.3.3 Security

The Cassandra Handler connection to the Cassandra Cluster can be secured using user name and password credentials. The user name and password credentials are configured as part of the Cassandra Handler configuration in the Adapter properties file, see Cassandra Handler Configuration.

6.4 Automated DDL Handling

When started, the Cassandra Handler performs the table check and reconciliation process the first time an operation for a source table is encountered. Additionally, a DDL event or a metadata change event causes the table definition in the Cassandra Handler to be marked as dirty so the next time an operation for the table is encountered the handler repeats the table check and reconciliation process as described in the following section.

Table Check and Reconciliation Process

6.4.1 Table Check and Reconciliation Process

The Cassandra Handler first interrogates the target Cassandra database to see if the target Cassandra keyspace exists. If the target Cassandra keyspace does not exist, then the Cassandra Handler abends. Keyspaces must be created by the user. The log file should contain the error of the exact keyspace name the Cassandra Handler is expecting.

Next, the Cassandra Handler interrogates the target Cassandra database for the table definition. If the table does not exist, the Cassandra Handler will do one of two things. If `gg.handler.name.ddlHandling` includes `CREATE`, then a table is created in Cassandra; otherwise the process abends. A message is logged that shows you the table that does not exist in Cassandra.

If the table exists in Cassandra, then the Cassandra Handler performs a reconciliation between the table definition from the source trail file and the table definition in Cassandra. This reconciliation process searches for columns that exist in the source table definition and not in the corresponding Cassandra table definition. If it locates columns fitting this criteria and the `gg.handler.name.ddlHandling` property includes `ADD`, then the Cassandra Handler alters the target table in Cassandra adding the new columns; otherwise it ignores these columns.

Next, the reconciliation process search for columns that exist in the target Cassandra table though do not exist in the source table definition. If the locates columns fitting this criteria and the `gg.handler.name.ddlHandling` property includes `DROP` then the Cassandra Handler alters the target table in Cassandra to drop these columns; otherwise those columns are ignored.

Finally, the prepared statements are built.

6.5 Performance Considerations

Configuring the Cassandra Handler for `async` mode will provide better performance than `sync` mode. The Replicat property `GROUPTRANSOPS` should be set to the default of a 1000.

Setting of the consistency level directly affects performance. The higher the consistency level, the more work must occur on the Cassandra cluster before the transmission of a given operation can be considered complete. You should select the minimum consistency level that still satisfies the requirements of your use case. Consistency level information is found at:

https://docs.datastax.com/en/cassandra/3.x/cassandra/dml/dmlConfigConsistency.html

The Cassandra Handler can work in either operation (`op`) or transaction (`tx`) mode. For the best performance operation mode is recommended:

```
gg.handler.name.mode=op
```

6.6 Additional Considerations

- Cassandra database requires at least one primary key and the value for any primary key cannot be null. Automated table creation fails for source tables that do not have a primary key.
- When `gg.handler.name.compressedUpdates=false` is set it means that the Cassandra Handler expects to update full before and after images of the data.

Using this property setting with a source trail file with partial image updates results in null values being updated for columns for which the data is missing. This configuration is incorrect and update operations pollute the target data with null values in columns that did not change.

- The Cassandra Handler does *not* process DDL from the source database even if the source database provides DDL. Instead it performs a reconciliation process between the source table definition and the target Cassandra table definition. A DDL statement executed at the source database changing a column name appears to the Cassandra Handler the same as if a column was dropped from the source table and a new column was added to the source table. This behavior is dependent on how the `gg.handler.name.ddlHandling` property is configured:

`gg.handler.name.ddlHandling` Configuration	**Behavior**
Not configured for `ADD` or `DROP`	Old column name and data maintained in Cassandra. New column is not created in Cassandra so no data is replicated for the new column name from the DDL change forward.
Configured for `ADD` only	Old column name and data maintained in Cassandra. New column iscreated in Cassandra and data replicated for the new column name from the DDL change forward. Column mismatch of where the data is located before and after the DDL change.
Configured for `DROP` only	Old column name and data dropped in Cassandra. New column is not created in Cassandra so no data replicated for the new column name.
Configured for `ADD` and `DROP`	Old column name and data dropped in Cassandra. New column is created in Cassandra and data replicated for the new column name from the DDL change forward.

6.7 Troubleshooting

This section contains information to help you troubleshoot various issues. Review the following topics for additional help:

Java Classpath

Logging

Write Timeout Exception

6.7.1 Java Classpath

The most common initial error is an incorrect classpath to include all the required client libraries and creates a `ClassNotFound` exception in the log file. You can

troubleshoot by setting the Java Adapter logging to `DEBUG`, and then rerun the process. At the debug level, the logging includes information of which JARs were added to the classpath from the `gg.classpath` configuration variable. The `gg.classpath` variable supports the wildcard (`*`) character to select all JARs in a configured directory. For example, `/usr/cassandra/cassandra-java-driver-3.1.0/*:/usr/cassandra/cassandra-java-driver-3.1.0/lib/*`.

For more information about setting the classpath, see Setting Up and Running the Cassandra Handlerand Cassandra Handler Client Dependencies.

6.7.2 Logging

The Cassandra Handler logs the state of its configuration to the Java log file. This is helpful because you can review the configuration values for the Cassandra Handler. An sample of the logging of the state of the configuration follows:

```
**** Begin Cassandra Handler - Configuration Summary ****
  Mode of operation is set to op.
  The Cassandra cluster contact point(s) is [localhost].
  The handler has been configured for GoldenGate compressed updates (partial image
updates).
  Sending data to Cassandra in [ASYNC] mode.
  The Cassandra consistency level has been set to [ONE].
  Cassandra Handler DDL handling:
    The handler will create tables in Cassandra if they do not exist.
    The handler will add columns to Cassandra tables for columns in the source
metadata that do not exist in Cassandra.
    The handler will drop columns in Cassandra tables for columns that do not exist
in the source metadata.
**** End Cassandra Handler - Configuration Summary ****
```

6.7.3 Write Timeout Exception

When running the Cassandra handler, you may experience a `com.datastax.driver.core.exceptions.WriteTimeoutException` exception that causes the Replicat process to abend. It is likely to occur under some or all of the following conditions.

- The Cassandra Handler is processing large numbers of operations putting the Cassandra cluster under a significant processing load.
- `GROUPTRANSOPS` is configured higher than the 1000 default.
- The Cassandra Handler is configured in asynchronous mode.
- The Cassandra Handler is configured for a consistency level higher than `ONE`.

The problem is that the Cassandra Handler is streaming data faster than the Cassandra cluster can process it. The write latency in the Cassandra cluster then finally exceeds the write request timeout period, which in turn results in the exception.

The following are potential solutions:

- Increase the write request timeout period. This is controlled with the `write_request_timeout_in_ms` property in Cassandra and is located in the `cassandra.yaml` file in the *`cassandra_install`*`/conf` directory. The default is 2000 (2 seconds). You can increase this value to move past the error, and then restart the Cassandra node or nodes for the change to take affect.

- It is considered a good practice to also decrease the GROUPTRANSOPS configuration value of the Replicat process if this error occurs. Typically, decreasing the GROUPTRANSOPS configuration decreases the size of transactions processed and reduces the likelihood that the Cassandra Handler can overtax the Cassandra cluster.
- The consistency level of the Cassandra Handler can be reduced, which in turn reduces the amount of work the Cassandra cluster has to complete for an operation to be considered as written.

7

Using the Java Database Connectivity Handler

This chapter explains the Java Database Connectivity (JDBC) Handler and includes examples so that you can understand this functionality.

Topics:

7.1 Overview

The Generic Java Database Connectivity (JDBC) Handler provides the ability to replicate source transactional data to a target or database using a JDBC interface and is applicable for targets that support JDBC connectivity.

Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which the JDBC Handler was built. The JDBC handler with the JDBC metadata provider provide additional capability to use Replicat features, such as column mapping and column functions, see Using the Metadata Provider

See the Oracle Java JDBC API Documentation website for more information:

http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/index.html

7.2 Detailed Functionality

The JDBC Handler replicates source transactional data to a target (or database) using a JDBC interface.

Topics:

Redshift Database Data Types

7.2.1 Single Operation Mode

The JDBC Handler performs SQL operations on every single trail record (row operation) when the trail record is processed by the handler. The JDBC Handler does *not* use the `BATCHSQL` feature of the JDBC API to batch operations.

7.2.2 Oracle Database Data Types

The following column data types are supported for Oracle Database targets:

```
NUMBER
DECIMAL
INTEGER
FLOAT
REAL
DATE
TIMESTAMP
INTERVAL YEAR TO MONTH
INTERVAL DAY TO SECOND
CHAR
VARCHAR2
NCHAR
NVARCHAR2
RAW
CLOB
NCLOB
BLOB
TIMESTAMP WITH TIMEZONE[11]
TIME WITH TIMEZONE[22]
```

7.2.3 MySQL Database Data Types

The following column data types are supported for MySQL Database targets:

```
INT
REAL
FLOAT
DOUBLE
NUMERIC
DATE
DATETIME
TIMESTAMP
TINYINT
BOOLEAN
SMALLINT
BIGINT
MEDIUMINT
DECIMAL
```

1 Time zone with a two digit hour and a two digit minimum offset.
2 Time zone with a two digit hour and a two digit minimum offset.

```
BIT
YEAR
ENUM
CHAR
VARCHAR
```

7.2.4 Netezza Database Data Types

The following column data types are supported for Netezza database targets:

```
byteint
smallint
integer
bigint
numeric(p,s)
numeric(p)
float(p)
Real
double
char
varchar
nchar
nvarchar
date
time
Timestamp
```

7.2.5 Redshift Database Data Types

The following column data types are supported for Redshift database targets:

```
SMALLINT
INTEGER
BIGINT
DECIMAL
REAL
DOUBLE
CHAR
VARCHAR
DATE
TIMESTAMP
```

7.3 Setting Up and Running the JDBC Handler

Instructions for configuring the JDBC Handler components and running the handler are described in the following sections.

Note:

You should use the JDBC Metadata Provider with the JDBC Handler to obtain better data type mapping, column mapping, and column function features.

Topics:

7.3.1 Java Classpath

The JDBC Java Driver location must be included in the class path of the handler using the `gg.classpath` property.

For example, the configuration for a MySQL database could be:

```
gg.classpath= /path/to/jdbc/driver/jar/mysql-connector-java-5.1.39-bin.jar
```

7.3.2 Handler Configuration

You configure the JDBC Handler operation using the properties file. To enable the selection of the JDBC handler, one must first configure the handler type by specifying `gg.handler.`*`name`*`.type=jdbc` and the other JDBC properties as follows:

Table 7-1 JDBC Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.`*`name`*`.type`	Required	`jdbc`	None	Selects the JDBC Handler for streaming change data capture into JDBC.
`gg.handler.`*`name`*`.connectionURL`	Required	A valid JDBC connection URL.	None	The target specific JDBC connection URL.
`gg.handler.`*`name`*`.DriverClass`	Target database dependent.	The target specific JDBC driver class name.	None	The target specific JDBC driver class name.
`gg.handler.`*`name`*`.userName`	Target database dependent.	A valid user name.	None	The user name used for the JDBC connection to the target database.

Table 7-1 (Cont.) JDBC Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.password`	Target database dependent.	A valid password.	None	The password used for the JDBC connection to the target database.
`gg.handler.name.maxActiveStatements`	Optional	Unsigned integer.	Target database dependent	If this property is not specified, the JDBC Handler queries the target dependent database metadata indicating maximum number of active prepared SQL statements. Some targets do not provide this metadata so then the default value of 256 active SQL statements is used. If this property is specified, the JDBC Handler will not query the target database for such metadata and use the property value provided in the configuration. In either case, when the JDBC handler finds that the total number of active SQL statements is about to be exceeded, the oldest SQL statement is removed from the cache to add one new SQL statement.

7.3.3 Statement Caching

Typically, JDBC driver implementations allow multiple statements to be cached in order to speed up the execution of the DML operations. This avoids repreparing the statement for operations that share the same profile or template.

The JDBC Handler uses statement caching to speed up the process and caches as many statements as supported by the underlying JDBC driver. The cache is implemented by using an **LRU cache** where the **key** is the profile of the operation (stored internally in the memory as an instance of `StatementCacheKey` class), and the **value** is the `PreparedStatement` object itself.

A `StatementCacheKey` object contains the following information for the various DML profiles that are supported in the JDBC Handler:

DML operation type	`StatementCacheKey` contains a tuple of:
`INSERT`	(table name, operation type, ordered after-image column indices)
`UPDATE`	(table name, operation type, ordered after-image column indices)
`DELETE`	(table name, operation type)

DML operation type	StatementCacheKey contains a tuple of:
TRUNCATE	(table name, operation type)

7.3.4 Setting Up Error Handling

The JDBC Handler supports using the REPERROR and HANDLECOLLISIONS Oracle GoldenGate parameters, see *Reference for Oracle GoldenGate for Windows and UNIX*.

Additional configuration is required in the handler properties file to define the mapping of different error codes for the target database as follows:

gg.error.duplicateErrorCodes
A comma-separated list of error codes defined in the target database that indicates a duplicate key violation error. Most the JDBC drivers return a valid error code so REPERROR actions can be configured based on the error code configured. For example:

```
gg.error.duplicateErrorCodes=1062,1088,1092,1291,1330,1331,1332,1333
```

gg.error.notFoundErrorCodes
A comma-separated list of error codes that indicate missed DELETE or UPDATE operations on target database.

In some cases, the JDBC driver errors when an UPDATE or DELETE operation does not modify any rows in the target database so no additional handling is required by the JDBC Handler.

Most JDBC drivers do not return an error when a DELETE or UPDATE is affecting zero rows so the JDBC handler automatically detects a missed UPDATE or DELETE operation and triggers an error to indicate a not-found error to the Replicat process. The Replicat process can then execute the specified REPERROR action.

The default error code used by the handler is the value zero. When you configure this property to a non-zero value, the configured error code value is used when the handler triggers a not found error. For example:

```
gg.error.notFoundErrorCodes=1222
```

gg.error.deadlockErrorCodes
A comma-separated list of error codes that indicate a deadlock error in the target database. For example:

```
gg.error.deadlockErrorCodes=1213
```

Sample Oracle Database Target Error Codes

```
gg.error.duplicateErrorCodes=1
gg.error.notFoundErrorCodes=0
gg.error.deadlockErrorCodes=60
```

Sample MySQL Database Target Error Codes

```
gg.error.duplicateErrorCodes=1022,1062
gg.error.notFoundErrorCodes=1329
gg.error.deadlockErrorCodes=1213,1614
```

7.4 Sample Configurations

The following sections contain sample configurations for the databases supported by the JDBC Handler from the Java Adapter properties file:

Topics:

7.4.1 Sample Oracle Database Target

```
gg.handlerlist=jdbcwriter
gg.handler.jdbcwriter.type=jdbc

#Handler properties for Oracle database target
gg.handler.jdbcwriter.DriverClass=oracle.jdbc.driver.OracleDriver
gg.handler.jdbcwriter.connectionURL=jdbc:oracle:thin:@<DBServer address>:
1521:<database name>
gg.handler.jdbcwriter.userName=<dbuser>
gg.handler.jdbcwriter.password=<dbpassword>
gg.classpath=/path/to/oracle/jdbc/driver/ojdbc5.jar
goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE
gg.log=log4j
gg.log.level=INFO
gg.report.time=30sec
javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=.:ggjava/ggjava.jar:./
dirprm
```

7.4.2 Sample Oracle Database Target with JDBC Metadata Provider

```
gg.handlerlist=jdbcwriter
gg.handler.jdbcwriter.type=jdbc

#Handler properties for Oracle database target with JDBC Metadata provider
gg.handler.jdbcwriter.DriverClass=oracle.jdbc.driver.OracleDriver
gg.handler.jdbcwriter.connectionURL=jdbc:oracle:thin:@<DBServer address>:
1521:<database name>
gg.handler.jdbcwriter.userName=<dbuser>
gg.handler.jdbcwriter.password=<dbpassword>
gg.classpath=/path/to/oracle/jdbc/driver/ojdbc5.jar
#JDBC Metadata provider for Oracle target
gg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:oracle:thin:@<DBServer address>:1521:<database name>
gg.mdp.DriverClassName=oracle.jdbc.driver.OracleDriver
gg.mdp.UserName=<dbuser>
gg.mdp.Password=<dbpassword>
goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE
gg.log=log4j
```

```
gg.log.level=INFO
gg.report.time=30sec
javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=.:ggjava/ggjava.jar:./
dirprm
```

7.4.3 Sample MySQL Database Target

```
gg.handlerlist=jdbcwriter
gg.handler.jdbcwriter.type=jdbc

#Handler properties for MySQL database target
gg.handler.jdbcwriter.DriverClass=com.mysql.jdbc.Driver
gg.handler.jdbcwriter.connectionURL=jdbc:<a target="_blank"
href="mysql://">mysql://</a><DBServer address>:3306/<database name>
gg.handler.jdbcwriter.userName=<dbuser>
gg.handler.jdbcwriter.password=<dbpassword>
gg.classpath=/path/to/mysql/jdbc/driver//mysql-connector-java-5.1.39-bin.jar

goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE
gg.log=log4j
gg.log.level=INFO
gg.report.time=30sec
javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=.:ggjava/ggjava.jar:./
dirprm
```

7.4.4 Sample MySQL Database Target with JDBC Metadata Provider

```
gg.handlerlist=jdbcwriter
gg.handler.jdbcwriter.type=jdbc

#Handler properties for MySQL database target with JDBC Metadata provider
gg.handler.jdbcwriter.DriverClass=com.mysql.jdbc.Driver
gg.handler.jdbcwriter.connectionURL=jdbc:<a target="_blank"
href="mysql://">mysql://</a><DBServer address>:3306/<database name>
gg.handler.jdbcwriter.userName=<dbuser>
gg.handler.jdbcwriter.password=<dbpassword>
gg.classpath=/path/to/mysql/jdbc/driver//mysql-connector-java-5.1.39-bin.jar
#JDBC Metadata provider for MySQL target
gg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:<a target="_blank" href="mysql://">mysql://</a><DBServer
address>:3306/<database name>
gg.mdp.DriverClassName=com.mysql.jdbc.Driver
gg.mdp.UserName=<dbuser>
gg.mdp.Password=<dbpassword>

goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE
gg.log=log4j
gg.log.level=INFO
gg.report.time=30sec
javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=.:ggjava/ggjava.jar:./
dirprm
```

8

Using the MongoDB Handler

This chapter explains the MongoDB Handler and includes examples so that you can understand this functionality.

Topics:

8.1 Overview

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

See the MongoDB website for more information:

https://www.mongodb.com/

You can use the MongoDB Handler to replicate the transactional data from Oracle GoldenGate trail to a target MongoDB database.

8.2 Detailed Functionality

The MongoDB Handler takes operations from the source trail file and creates corresponding documents in the target MongoDB database.

A **record** in MongoDB is a **Binary JSON (BSON) document**, which is a data structure composed of field and value pairs. A BSON data structure is a binary representation of JSON documents. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

A **collection** is a grouping of MongoDB documents and is the equivalent of an RDBMS table. In MongoDB, databases hold collections of documents. Collections do not enforce a schema. MongoDB documents within a collection can have different fields.

Topics:

8.2.1 Document Key Column

MongoDB databases require every document (row) to have a column named `_id` whose value should be unique in a collection (table). This is similar to a primary key for RDBMS tables. If a document does not contain a top-level `_id` column during an insert, the MongoDB driver adds this column.

The MongoDB Handler builds custom `_id` field values for every document based on the primary key column values in the trail record. This custom `_id` is built using all the key column values concatenated by a `:` (colon) separator. For example:

```
KeyColValue1:KeyColValue2:KeyColValue3
```

The MongoDB Handler enforces uniqueness based on these custom `_id` values. This means that every record in the trail must be unique based on the primary key columns values. Existence of non-unique records for the same table results in a MongoDB Handler failure and in Replicat abending with a duplicate key error.

The behavior of the `_id` field is:

- By default, MongoDB creates a unique index on the column during the creation of a collection.
- It is always the first column in a document.
- It may contain values of any BSON data type except an array.

8.2.2 Primary Key Update Operation

MongoDB databases do not allow the `_id` column to be modified. This means a primary key update operation record in the trail needs special handling. The MongoDB Handler converts a primary key update operation into a combination of a `DELETE` (with old key) and an `INSERT` (with new key). To perform the `INSERT`, a complete before-image of the update operation in trail is recommended. You can generate the trail to populate a complete before image for update operations by enabling the Oracle GoldenGate `GETUPDATEBEFORES` and `NOCOMPRESSUPDATES` parameters, see *Reference for Oracle GoldenGate for Windows and UNIX*.

8.2.3 MongoDB Trail Data Types

The MongoDB Handler supports delivery to the BSON data types as follows:

- 32-bit integer
- 64-bit integer
- Double
- Date
- String
- Binary data

8.3 Setting Up and Running the MongoDB Handler

Instructions for configuring the MongoDB Handler components and running the handler are described in the following sections.

Topics:

8.3.1 Classpath Configuration

The MongoDB Java Driver is required for Oracle GoldenGate for Big Data to connect and stream data to MongoDB. The recommended version of MongoDB Java Driver is 3.2.2. The MongoDB Java Driver is not included in the packaging of Oracle GoldenGate for Big Data so you must download the driver from:

https://docs.mongodb.com/ecosystem/drivers/java/#download-upgrade

Select “mongo-java-driver" and the "3.2.2" version to download the recommended driver JAR file.

You must configure the `gg.classpath` variable to load the MongoDB Java Driver JAR at runtime. For example: `gg.classpath=/home/mongodb/mongo-java-driver-3.2.2.jar`

8.3.2 MongoDB Handler Configuration

The following are the configurable values for the MongoDB Handler. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 8-1 MongoDB Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.`*`name`*`.type`	Required	`mongodb`	None	Selects the MongoDB Handler for use with Replicat.
`gg.handler.`*`name`*`.bulkWrite`	Optional	`true` | `false`	`true`	Set to `true`, the handler caches operations until a commit transaction event is received. When committing the transaction event, all the cached operations are written out to the target MongoDB database, which provides improved throughput. Set to `false`, there is no caching within the handler and operations are immediately written to the MongoDB database.

Table 8-1 (Cont.) MongoDB Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.WriteConcern`	Optional	`{"w": "value" , "wtimeout": "number" }`	None	Sets the required write concern for all the operations performed by the MongoDB Handler. The property value is in JSON format and can only accept keys as `"w"` and `"wtimeout"`. For more information about write concerns, see https://docs.mongodb.com/manual/reference/write-concern/.
`gg.handler.name.username`	Optional	A legal username string.	None	Sets the authentication username to be used. Use with the `AuthenticationMechanism` property.
`gg.handler.name.password`	Optional	A legal password string.	None	Sets the authentication password to be used. Use with the `AuthenticationMechanism` property.
`gg.handler.name.ServerAddressList`	Optional	`IP:PORT` with multiple port values delimited by a comma	None	Enables the connection to a list of Replicat set members or a list of MongoDB databases. This property accepts a comma separated list of `[hostnames:port]`. For example, `localhost1:27017,localhost2:27018,localhost3:27019`. For more information, see http://api.mongodb.com/java/3.0/com/mongodb/MongoClient.html#MongoClient-java.util.List-java.util.List-com.mongodb.MongoClientOptions-.
`gg.handler.name.AuthenticationMechanism`	Optional	Comma separated list of authentication mechanism	None	Sets the authentication mechanism which is a process of verifying the identity of a client. The input would be a comma separated list of various authentication options. For example, `GSSAPI,MONGODB_CR,MONGODB_X509,PLAIN,SCRAM_SHA_1`. For more information about authentication options, see http://api.mongodb.com/java/3.0/com/mongodb/MongoCredential.html,
`gg.handler.name.source`	Optional	Valid authentication source	None	Sets the source of the user name, typically the name of the database where the user is defined. Use with the `AuthenticationMechanism` property.

Table 8-1 (Cont.) MongoDB Handler Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.clientURI`	Optional	Valid MongoDB client URI	None	Sets the MongoDB client URI. A client URI can also be used to set other MongoDB connection properties, such as authentication and `WriteConcern`. For example, `mongodb://localhost:27017/` For more details about the format of the client URI, see http://api.mongodb.com/java/3.0/com/mongodb/MongoClientURI.html
`gg.handler.name.Host`	Optional	Valid MongoDB server name or IP address	None	Sets the MongoDB database hostname to connect to based on a (single) MongoDB node see http://api.mongodb.com/java/3.0/com/mongodb/MongoClient.html#MongoClient-java.lang.String-.
`gg.handler.name.Port`	Optional	Valid MongoDB port	None	Sets the MongoDB database instance port number. Use with the `Host` property.
`gg.handler.name.CheckMaxRowSizeLimit`	Optional	`true` | `false`	`false`	Set to `true`, the handler always checks the size of the BSON document inserted or modified to be within the limits defined by MongoDB database. Calculating the size involves the use of a default codec to generate a `RawBsonDocument` leading to a small degradation in the throughput of the MongoDB Handler. If the size of the document exceeds the MongoDB limit, an exception occurs and Replicat abends.

8.3.3 Connecting and Authenticating

You can use various connection and authentication properties which can be configured in the handler properties file. When multiple connection properties are specified, the MongoDB Handler chooses the properties based on the following priority order:

Priority 1:

```
AuthentictionMechanism
UserName
Password
Source
Write Concern
```

Priority 2:

```
ServerAddressList
AuthentictionMechanism
UserName
Password
Source
```

Priority 3:

```
clientURI
```

Priority 4:

```
Host
Port
```

Priority 5:

```
Host
```

If none of the connection and authentication properties are specified, the handler tries to connect to `localhost` on port `27017`.

8.3.4 Using Bulk Write

The MongoDB Handler uses the `GROUPTRANSOPS` parameter to retrieve the batch size. A batch of trail records are converted to a batch of MongoDB documents then written in one request to the database.

You can enable bulk write for better apply throughput using the `BulkWrite` handler property . By default, this is enabled and this is the recommended setting for the best performance of the handler..

You use the `gg.handler.`*`handler`*`.BulkWrite=`*`true | false`* property to enable or disable bulk write. The Oracle GoldenGate for Big Data default property, `gg.handler.`*`handler`*`.mode=op | tx`, is *not* used in the MongoDB Handler.

Oracle recommends that you use bulk write.

8.3.5 Using Write Concern

Write concern describes the level of acknowledgement requested from MongoDB for write operations to a standalone MongoDB, replica sets, and sharded-clusters. With sharded clusters, mongos instances will pass the write concern on to the shards.

Use the following configuration:

```
w: value
wtimeout: number
```

Related Topics:

https://docs.mongodb.com/manual/reference/write-concern/

8.3.6 Using Three-Part Table Names

An Oracle GoldenGate trail may have data for sources that support three-part table names, such as `Catalog.Schema.Table`. MongoDB only supports two-part names, such as `DBName.Collection`. To support the mapping of source three-part names to

MongoDB two-part names, the source *Catalog* and *Schema* is concatenated with an underscore delimiter to construct the Mongo *DBName*.

For example, *Catalog.Schema.Table* would become *catalog1_schema1.table1*.

8.3.7 Using Undo Handling

The MongoDB Handler can recover from bulk write errors using a lightweight undo engine. This engine does not provide the functionality provided by typical RDBMS undo engines, rather the best effort to assist you in error recovery. The error recovery works well when there are primary violations or any other bulk write error where the MongoDB database is able to provide information about the point of failure through the `BulkWriteException`.

Table 8-2lists the requirements to make the best use of this functionality.

Table 8-2 Undo Handling Requirements

Operation to Undo	Require Full Before Image in the Trail?
`INSERT`	No
`DELETE`	Yes
`UPDATE`	No (Before image of fields in the `SET` clause.)

If there are errors during undo operations, it may be not possible to get the MongoDB collections to a consistent state so you would have to do a manual reconciliation of data.

8.4 Sample Configuration

The following is sample configuration for the MongoDB Handler from the Java Adapter properties file:

```
gg.handlerlist=mongodb
gg.handler.mongodb.type=mongodb

#The following handler properties are optional.
#Please refer to the Oracle GoldenGate for BigData documentation
#for details about the configuration.
#gg.handler.mongodb.clientURI=mongodb://localhost:27017/
#gg.handler.mongodb.Host=<MongoDBServer address>
#gg.handler.mongodb.Port=<MongoDBServer port>
#gg.handler.mongodb.WriteConcern={ w: <value>, wtimeout: <number> }
#gg.handler.mongodb.AuthenticationMechanism=GSSAPI,MONGODB_CR,MONGODB_X509,PLAIN,SCRA
M_SHA_1
#gg.handler.mongodb.UserName=<Authentication username>
#gg.handler.mongodb.Password=<Authentication password>
#gg.handler.mongodb.Source=<Authentication source>
#gg.handler.mongodb.ServerAddressList=localhost1:27017,localhost2:27018,localhost3:27
019,...
#gg.handler.mongodb.BulkWrite=<false|true>
#gg.handler.mongodb.CheckMaxRowSizeLimit=<true|false>

goldengate.userexit.timestamp=utc
goldengate.userexit.writers=javawriter
javawriter.stats.display=TRUE
```

```
javawriter.stats.full=TRUE
gg.log=log4j
gg.log.level=INFO
gg.report.time=30sec

#Path to MongoDB Java driver.
# maven co-ordinates
# <dependency>
# <groupId>org.mongodb</groupId>
# <artifactId>mongo-java-driver</artifactId>
# <version>3.2.2</version>
# </dependency>
gg.classpath=/path/to/mongodb/java/driver/mongo-java-driver-3.2.2.jar
javawriter.bootoptions=-Xmx512m -Xms32m -Djava.class.path=.:ggjava/ggjava.jar:./
dirprm
```

9

Using the Pluggable Formatters

Formatters provide the functionality to convert operations from the Oracle GoldenGate trail file info formatted messages that can then be sent to Big Data targets by one of the Oracle GoldenGate for Big Data Handlers.

Topics:

9.1 Operation vs. Row Based Formatting

The Oracle GoldenGate for Big Data formatters are operations-based and row-based formatters.

Operation-based represent the individual insert, update, and delete events that occur on table data in the source database. Insert operations only provide after change data (or images) since a new row is being added to the source database. Update operations provide both before and after change data that shows how existing row data is modified. Delete operations only provide before change data to provide identification of the row being deleted. The operation-based formatters model the operation as it is exists in the source trail file. Operation-based formats include fields for the before and after images.

The row-based formatters model the row data as it exists after the operation data is applied. Row based formatters only contain a single image of the data. The following sections describe what data is displayed for both the operation-based and row based formatters.

9.1.1 Operation Formatters

The formatters that support operation-based formatting are JSON, Avro Operation, and XML. The output of operation-based formatters are as follows:

- Insert operation - Before image data is NULL. After image data is output.
- Update operation - Both before and after image data is output.

- Delete operation - Before image data is output. After image data is NULL.
- Truncate operation - Both before and after image data is NULL.

9.1.2 Row Formatters

The formatters that support row-based formatting area Delimited Text and Avro Row. Row-based formatters output the following information for the following operations.

- Insert operation - After image data only.
- Update operation - After image data only. Primary key updates are a special case which will be discussed in individual sections for the specific formatters.
- Delete operation - Before image data only.
- Truncate operation - Table name is provided but both before and after image data are NULL. Truncate table is a DDL operation and it may not support different database implementations. Refer to the Oracle GoldenGate documentation for your database implementation.

9.1.3 Table Row or Column Value States

- In an RDBMS, table data for a specific row and column can only have one of two states. Either the table row/column value has a value or the row/column value is NULL. However; when data is transferred to the Oracle GoldenGate trail file by the Oracle GoldenGate capture process, this can expand to three possible states: the table row/column has a value, the row/column value is NULL, or the row/column value is missing.
- For an insert operation, the after image contains data for all column values whether that column has a value or is NULL. However, the data included for update and delete operations may not always contain complete data for all columns. When replicating data to an RDBMS for an update operation the only data that is required to modify the data in the target database are the primary key values and the values of the columns that changed. In addition, for a delete operation it is only necessary to have the primary key values to delete the row from the target database. Therefore, even though table row/column values have a value in the source database, the values may be missing in the source trail file. Because it is possible for row/column data in the source trail file to have three states, the Plugable Formatters must also be able to represent data in the three states.
- What row/column data is available in the Oracle GoldenGate trail file will have an impact on Big Data integrations. It is important for you to understand what data is required. You typically have control on the data that is included for operations in the Oracle GoldenGate trail file. For Oracle Databases, this is controlled by the supplemental logging level. Refer to the Oracle GoldenGate documentation for your specific source database implementation to understand how to control the row and olumn values that are included in the Oracle GoldenGate trail file.

9.2 Delimited Text Formatter

The Delimited Text Formatter is a row-based formatter. It formats database operations from the source trail file into a delimited text output. Each insert, update, delete, or truncate operation from the source trail will be formatted into an individual delimited message. Delimited text output is a fixed number of fields for each table separated by a field delimiter and terminated by a line delimiter. The fields are positionally

relevant. Many Big Data analytical tools including Hive work well with HDFS files containing delimited text.

Column values for an operation from the source trail file can have one of three states: column has a value, column value is NULL, or column value is missing. By default the delimited text maps these column value states into the delimited text output as follows:

- Column has a value - The column value is output.
- Column value is NULL - The default output value is `NULL`. The output for the case of a NULL column value is configurable.
- Column value is missing - The default output value is "". The output for the case of a missing column value is configurable.

This section contains the following topics:

9.2.1 Message Formatting Details

The default format for output of data is the following, which is delimited by a semi-colon:

First is the row metadata:

```
operation_type;fully_qualified_table_name;operation_timestamp;current_timestamp;trail
_position;tokens;
```

Next is the row data:

```
column_1_value;column_n_value_then_line_delimeter
```

Optionally, the column name can be included before each column value that changes the output format for the row data:

```
column_1_name;column_1_value;column_n_name;column_n_value_then_line_delimeter
```

Operation Type - Operation type is the indicator of the type of database operation from the source trail file. Default values are `I` for insert, `U` for update, `D` for delete, `T` for truncate. Output of this field is suppressible.

Fully Qualified Table name - The fully qualified table name is the source database table include including the catalog name and schema name. The format of the fully qualified table name is `catalog_name.schema_name.table_name`. Output of this field is suppressible.

Operation Timestamp - The operation timestamp is the commit record timestamp from the source system. All operations in a transaction (unbatched transaction) should

have the same operation timestamp. This timestamp is fixed and the operation timestamp will be the same if the trail file is replayed. Output of this field is suppressible.

Current Timestamp - The current timestamp is a timestamp of the current time when delimited text formatter processes the current operation record. This timestamp follows the ISO-8601 format and includes microsecond precision. Replaying the trail file will *not* result in the same timestamp for the same operation. Output of this field is suppressible.

Trail Position - This is the concatenated sequence number and RBA number from the source trail file. The trail position provides traceability of the operation back to the source trail file. The sequence number is the source trail file number. The RBA number is the offset in the trail file. Output of this field is suppressible.

Tokens - The tokens are the token key value pairs from the source trail file. The output of this field in the delimited text output is suppressed if the `includeTokens` configuration property on the corresponding handler is *not* explicitly set to `true`.

9.2.2 Sample Formatted Messages

The following sections contain sample messages from the Delimited Text Formatter. The default field delimiter has been changed to a pipe character, `|`, to more clearly display the message.

Sample Insert Message

Sample Update Message

Sample Delete Message

Sample Truncate Message

9.2.2.1 Sample Insert Message

```
I|GG.TCUSTORD|2013-06-02
22:14:36.000000|2015-09-18T13:23:01.612001|00000000000000001444|R=AADPkvAAEAAEqL2A
AA|WILL|1994-09-30:15:33:00|CAR|144|17520.00|3|100
```

9.2.2.2 Sample Update Message

```
U|GG.TCUSTORD|2013-06-02
22:14:41.000000|2015-09-18T13:23:01.987000|00000000000000002891|R=AADPkvAAEAAEqLzA
AA|BILL|1995-12-31:15:00:00|CAR|765|14000.00|3|100
```

9.2.2.3 Sample Delete Message

```
D|GG.TCUSTORD|2013-06-02
22:14:41.000000|2015-09-18T13:23:02.000000|00000000000000004338|L=206080450,6=9.0.
80330,R=AADPkvAAEAAEqLzAAC|DAVE|1993-11-03:07:51:35|PLANE|600|||
```

9.2.2.4 Sample Truncate Message

```
T|GG.TCUSTORD|2013-06-02
22:14:41.000000|2015-09-18T13:23:02.001000|00000000000000004515|R=AADPkvAAEAAEqL2A
AB|||||||
```

9.2.3 Additional Considerations

You should exercise care when choosing field and line delimiters. It is important to choose delimiter values that will not occur in the content of the data.

The Java Adapter configuration functionally trims leading and trailing characters from configuration values that are determined to be whitespace. You may want field delimiters, line delimiters, null value representations, and missing value representations that include or are fully considered to be whitespace. In these cases, you must employ specialized syntax in the Java Adapter configuration file to preserve the whitespace. Wrap the configuration value in a `CDATA[]` wrapper to preserve the whitespace when your configuration values contain leading or trailing characters that are considered whitespace. For example, a configuration value of `\n` should be configured as `CDATA[\n]`.

You can search column values using regular expressions then replace matches with a specified value. This search and replace functionality can be utilized in conjunction with the Delimited Text Formatter to ensure that there are no collisions between column value contents and field and line delimiters. For more information, see Using Regular Expression Search and Replace.

Big Data applications differ from RDBMSs in how data is stored. Update and delete operations in an RDBMS result in a change to the existing data. In contrast, data is not changed in Big Data applications rather appended to existing data. Therefore, the current state of a given row becomes a consolidation of all of the existing operations for that row in the HDFS system. This leads to some special scenarios as described in the following sections.

Primary Key Updates

Data Consolidation

9.2.3.1 Primary Key Updates

Primary key update operations require special consideration and planning for Big Data integrations. Primary key updates are update operations that modify one or more of the primary keys for the given row from the source database. Since data is simply appended in Big Data applications a primary key update operation looks more like a new insert than an update without any special handling. The Delimited Text formatter provides specialized handling for primary keys that is configurable to you. These are the configurable behaviors:

Table 9-1 Configurable Behavior

Value	Description
`abend`	The default behavior is that the delimited text formatter will abend in the case of a primary key update.
`update`	With this configuration the primary key update will be treated just like any other update operation. This configuration alternative should only be selected if you can guarantee that the primary key that is being changed is not being used as the selection criteria when selecting row data from a Big Data system.

Table 9-1 (Cont.) Configurable Behavior

Value	Description
`delete-insert`	Using this configuration the primary key update is treated as a special case of a delete using the before image data and an insert using the after image data. This configuration may more accurately model the effect of a primary key update in a Big Data application. However, if this configuration is selected it is important to have full supplemental logging enabled on replication at the source database. Without full supplemental logging, the delete operation will be correct, but the insert operation will not contain all of the data for all of the columns for a full representation of the row data in the Big Data application.

9.2.3.2 Data Consolidation

As previously stated, Big Data applications simply append data to the underlying storage. Analytic tools generally spawn MapReduce programs that traverse the data files and consolidate all the operations for a given row into a single output. Therefore, it is important to have an indicator of the order of operations. The Delimited Text formatter provides a number of metadata fields to fulfill this need. The operation timestamp may be sufficient to fulfill this requirement. However, two update operations may have the same operation timestamp especially if they share a common transaction. The trail position can provide a tie breaking field on the operation timestamp. Lastly, the current timestamp may provide the best indicator of order of operations in Big Data.

9.2.4 Output Format Summary Log

The Java `log4j` logging logs a summary of the delimited text output format if `INFO` level logging is enabled. A summary of the delimited fields is logged for each source table encountered and occurs when the first operation for that table is received by the Delimited Text formatter. You may find this detailed explanation of the fields of the delimited text output useful when performing an initial setup. With a metadata change event, the summary of the delimited fields is regenerated and logged again at the first operation for that table after the metadata change event.

9.2.5 Delimited Text Format Configuration

Table 9-2 Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.`*`name`*`.format.includeColumnNames`	Optional	`true` | `false`	`false`	Controls the output of writing the column names as a delimited field preceding the column value. If true output is like: *`COL1_Name`*`|`*`COL1_Value`*`|`*`COL2_Name`*`|`*`COL2_Value`* If the false output is like: *`COL1_Value`*`|I`

Table 9-2 (Cont.) Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.includeOpTimestamp`	Optional	`true` | `false`	`true`	A `false` value suppresses the output of the operation timestamp from the source trail file in the output.
`gg.handler.name.format.includeCurrentTimestamp`	Optional	`true` | `false`	`true`	A `false` value suppresses the output of the current timestamp in the output.
`gg.handler.name.format.includeOpType`	Optional	`true` | `false`	`true`	A `false` value suppresses the output of the operation type in the output.
`gg.handler.name.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation.
`gg.handler.name.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.
`gg.handler.name.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a delete operation.
`gg.handler.name.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be inserted into the output record to indicate a truncate operation.
`gg.handler.name.format.encoding`	Optional	Any encoding name or alias supported by Java.	The native system encoding of the machine hosting the Oracle Golden Gate process.	Determines the encoding of the output delimited text.

Table 9-2 (Cont.) Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.fieldDelimiter`	Optional	Any String	`ASCII 001` (the default Hive delimiter)	The delimiter used between delimited fields. This value supports `CDATA[]` wrapping.
`gg.handler.name.format.lineDelimiter`	Optional	Any String	Newline (the default Hive delimiter)	The delimiter used between records. This value supports `CDATA[]` wrapping.
`gg.handler.name.format.includeTableName`	Optional	`true` | `false`	`true`	Use `false` to suppress the output of the table name in the output delimited data.
`gg.handler.name.format.keyValueDelimiter`	Optional	Any string	`=`	Provides a delimiter between keys and values in a map. Key1=value1. Tokens are mapped values. Configuration value supports `CDATA[]` wrapping.
`gg.handler.name.format.keyValuePairDelimiter`	Optional	Any string	`,`	Provides a delimiter between key value pairs in a map. *`Key1=Value1,Key2=Value2`*. Tokens are mapped values. Configuration value supports `CDATA[]` wrapping.
`gg.handler.name.format.pkUpdateHandling`	Optional	`abend` | `update` | `delete-insert`	`abend`	Provides configuration for how the formatter should handle update operations that change a primary key. Primary key operations can be problematic for the text formatter and require special consideration by you. • `abend` - indicates the process will abend • `update` - indicates the process will treat this as a normal update • `delete-insert` - indicates the process will treat this as a delete and an insert. Full supplemental logging needs to be enabled for this to work. Without full before and after row images the insert data will be incomplete.
`gg.handler.name.format.nullValueRepresentation`	Optional	Any string	NULL	Allows you to configure what will be included in the delimited output in the case of a NULL value. Configuration value supports `CDATA[]` wrapping.

Table 9-2 (Cont.) Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.missingValueRepresentation`	Optional	Any string	`""` (no value)	Allows you to configure what will be included in the delimited text output in the case of a missing value. Configuration value supports `CDATA[]` wrapping.
`gg.handler.name.format.includePosition`	Optional	`true` | `false`	`true`	Allows you to suppress the output of the operation position from the source trail file.
`gg.handler.name.format.iso8601Format`	Optional	`true` | `false`	`true`	Controls the format of the current timestamp. The default is the ISO 8601 format. Set to `false` removes the `T` between the date and time in the current timestamp, which outputs a space instead.

9.2.6 Sample Configuration

The following is the sample configuration for the Delimited Text formatter from the Java Adapter configuration file:

```
gg.handler.hdfs.format.includeColumnNames=false
gg.handler.hdfs.format.includeOpTimestamp=true
gg.handler.hdfs.format.includeCurrentTimestamp=true
gg.handler.hdfs.format.insertOpKey=I
gg.handler.hdfs.format.updateOpKey=U
gg.handler.hdfs.format.deleteOpKey=D
gg.handler.hdfs.format.truncateOpKey=T
gg.handler.hdfs.format.encoding=UTF-8
gg.handler.hdfs.format.fieldDelimiter=CDATA[\u0001]
gg.handler.hdfs.format.lineDelimiter=CDATA[\n]
gg.handler.hdfs.format.includeTableName=true
gg.handler.hdfs.format.keyValueDelimiter=CDATA[=]
gg.handler.hdfs.format.kevValuePairDelimiter=CDATA[,]
gg.handler.hdfs.format.pkUpdateHandling=abend
gg.handler.hdfs.format.nullValueRepresentation=NULL
gg.handler.hdfs.format.missingValueRepresentation=CDATA[]
gg.handler.hdfs.format.includePosition=true
gg.handler.hdfs.format=delimitedtext
```

9.2.7 Metadata Change Events

Oracle GoldenGate for Big Data now handles metadata change events at runtime. This assumes the replicated database and upstream replication processes are propagating metadata change events. The Delimited Text Formatter changes the output format to accommodate the change and continue running.

It is important to understand that a metadata change may impact downstream applications. Delimited text formats are comprised of a fixed number of fields that are positionally relevant. Deleting a column in the source table can be handled seamlessly during Oracle GoldenGate runtime, but results in a change in the total number of fields and potentially the positional relevance of some fields. Adding an additional column or columns is probably the least impactful metadata change event assuming the new column is added to the end. You should consider the impact of a metadata

change event before executing the event. When metadata change events will be frequent, Oracle recommends that you consider a more flexible and self describing format, such as, JSON or XML.

9.3 JSON Formatter

The JavaScripts Object Notation (JSON) formatter can output operations from the source trail file in either row based format or operation based format. It formats operation data from the source trail file into a JSON objects. Each individual insert, update, delete, and truncate operation is formatted into an individual JSON message.

This section contains the following topics:

9.3.1 Operation Metadata Formatting Details

JSON objects generated by the JSON Formatter contain the following metadata fields at the beginning of each message:

Table 9-3 JSON Metadata

Value	Description
`table`	Contains fully qualified table name. The format of the fully qualified table name is: *`CATALOG NAME.SCHEMA NAME.TABLE NAME`*
`op_type`	Contains the operation type that is the indicator of the type of database operation from the source trail file. Default values are `I` for insert, `U` for update, `D` for delete, and `T` for truncate.
`op_ts`	The operation timestamp is the timestamp of the operation from the source trail file. Since this timestamp is from the source trail, it is fixed. Replaying the trail file results in the same timestamp for the same operation.
`current_ts`	The current timestamp is a timestamp of the current time when delimited text formatter processes the current operation record. This timestamp follows the ISO-8601 format and includes microsecond precision. Replaying the trail file will *not* result in the same timestamp for the same operation.

Table 9-3 (Cont.) JSON Metadata

Value	Description
`pos`	This is the trail file position with is the concatenated sequence number and RBA number from the source trail file. The trail position provides traceability of the operation back to the source trail file. The sequence number is the source trail file number. The RBA number is the offset in the trail file.
`primary_keys`	An array variable holding the column names of the primary keys of the source table. The `primary_keys` field is only include in the JSON output if the `includePrimaryKeys` configuration property is set to `true`.
`tokens`	The tokens field is only included in the output if the `includeTokens` handler configuration property is set to `true`.

9.3.2 Operation Data Formatting Details

JSON messages first contain the operation metadata fields, which are followed by the operation data fields. This data is represented by `before` and `after` members that are objects. These objects contain members with the keys being the column names and the values being the column values.

Operation data is modeled as follows:

- Inserts – Includes the after image data.
- Updates – Includes both the before and after image data.
- Deletes – Includes the before image data.

Column values for an operation from the source trail file can have one of three states: column has a value, column value is NULL, or column value is missing. The JSON Formatter maps these column value states into the created JSON objects as follows:

- Column has a value - The column value is output. In the following example the member `STATE` has a value.

```
    "after":{        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",        "STATE":"CO"    }
```

- Column value is NULL - The default output value is a JSON NULL. In the following example the member `STATE` is NULL.

```
    "after":{        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",        "STATE":null    }
```

- Column value is missing - The JSON will contain no element for a missing column value. In the following example the member `STATE` is missing.

```
    "after":{        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",    }
```

The default setting of the JSON Formatter is to map the data types from the source trail file to the associated JSON data type. JSON supports few data types so this functionality largely results in the mapping of numeric fields from the source trail file

to members typed as numbers. This data type mapping is configurable to alternatively treat all data as strings.

9.3.3 Row Data Formatting Details

JSON messages first contain the operation metadata fields, which are followed by the operation data fields. For row data formatting this is the source column names and source column values as JSON key value pairs. This data is represented by `before` and `after` members that are objects. These objects contain members with the keys being the column names and the values being the column values.

Row data is modeled as follows:

- Inserts – Includes the after image data.
- Updates – Includes the after image data.
- Deletes – Includes the before image data.

Column values for an operation from the source trail file can have one of three states: column has a value, column value is NULL, or column value is missing. The JSON Formatter maps these column value states into the created JSON objects as follows:

- Column has a value - The column value is output. In the following example the member `STATE` has a value.

```
        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",        "STATE":"CO"    }
```

- Column value is NULL - The default output value is a JSON NULL. In the following example the member `STATE` is NULL.

```
        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",        "STATE":null    }
```

- Column value is missing - The JSON will contain no element for a missing column value. In the following example the member `STATE` is missing.

```
        "CUST_CODE":"BILL",        "NAME":"BILL'S USED CARS",
"CITY":"DENVER",    }
```

The default setting of the JSON Formatter is to map the data types from the source trail file to the associated JSON data type. JSON supports few data types so this functionality largely results in the mapping of numeric fields from the source trail file to members typed as numbers. This data type mapping is configurable to alternatively treat all data as strings.

9.3.4 Sample JSON Messages

The following topics are sample JSON messages created by the JSON Formatter for insert, update, delete, and truncate operations.

Sample Operation Modeled JSON Messages

Sample Flattened Operation Modeled JSON Messages

Sample Row Modeled JSON Messages

Sample Primary Key Output JSON Message

9.3.4.1 Sample Operation Modeled JSON Messages

Insert:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"I",
    "op_ts":"2015-11-05 18:45:36.000000",
    "current_ts":"2016-10-05T10:15:51.267000",
    "pos":"00000000000000002928",
    "after":{
        "CUST_CODE":"WILL",
        "ORDER_DATE":"1994-09-30:15:33:00",
        "PRODUCT_CODE":"CAR",
        "ORDER_ID":144,
        "PRODUCT_PRICE":17520.00,
        "PRODUCT_AMOUNT":3,
        "TRANSACTION_ID":100
    }
}
```

Update:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"U",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:15:51.310002",
    "pos":"00000000000000004300",
    "before":{
        "CUST_CODE":"BILL",
        "ORDER_DATE":"1995-12-31:15:00:00",
        "PRODUCT_CODE":"CAR",
        "ORDER_ID":765,
        "PRODUCT_PRICE":15000.00,
        "PRODUCT_AMOUNT":3,
        "TRANSACTION_ID":100
    },
    "after":{
        "CUST_CODE":"BILL",
        "ORDER_DATE":"1995-12-31:15:00:00",
        "PRODUCT_CODE":"CAR",
        "ORDER_ID":765,
        "PRODUCT_PRICE":14000.00
    }
}
```

Delete:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"D",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:15:51.312000",
    "pos":"00000000000000005272",
    "before":{
        "CUST_CODE":"DAVE",
        "ORDER_DATE":"1993-11-03:07:51:35",
        "PRODUCT_CODE":"PLANE",
```

```
        "ORDER_ID":600,
        "PRODUCT_PRICE":135000.00,
        "PRODUCT_AMOUNT":2,
        "TRANSACTION_ID":200
    }
}
```

Truncate:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"T",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:15:51.312001",
    "pos":"00000000000000005480",
}
```

9.3.4.2 Sample Flattened Operation Modeled JSON Messages

Insert:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"I",
    "op_ts":"2015-11-05 18:45:36.000000",
    "current_ts":"2016-10-05T10:34:47.956000",
    "pos":"00000000000000002928",
    "after.CUST_CODE":"WILL",
    "after.ORDER_DATE":"1994-09-30:15:33:00",
    "after.PRODUCT_CODE":"CAR",
    "after.ORDER_ID":144,
    "after.PRODUCT_PRICE":17520.00,
    "after.PRODUCT_AMOUNT":3,
    "after.TRANSACTION_ID":100
}
```

Update:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"U",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:34:48.192000",
    "pos":"00000000000000004300",
    "before.CUST_CODE":"BILL",
    "before.ORDER_DATE":"1995-12-31:15:00:00",
    "before.PRODUCT_CODE":"CAR",
    "before.ORDER_ID":765,
    "before.PRODUCT_PRICE":15000.00,
    "before.PRODUCT_AMOUNT":3,
    "before.TRANSACTION_ID":100,
    "after.CUST_CODE":"BILL",
    "after.ORDER_DATE":"1995-12-31:15:00:00",
    "after.PRODUCT_CODE":"CAR",
    "after.ORDER_ID":765,
    "after.PRODUCT_PRICE":14000.00
}
```

Delete:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"D",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:34:48.193000",
    "pos":"00000000000000005272",
    "before.CUST_CODE":"DAVE",
    "before.ORDER_DATE":"1993-11-03:07:51:35",
    "before.PRODUCT_CODE":"PLANE",
    "before.ORDER_ID":600,
    "before.PRODUCT_PRICE":135000.00,
    "before.PRODUCT_AMOUNT":2,
    "before.TRANSACTION_ID":200
}
```

Truncate:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"D",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T10:34:48.193001",
    "pos":"00000000000000005480",
    "before.CUST_CODE":"JANE",
    "before.ORDER_DATE":"1995-11-11:13:52:00",
    "before.PRODUCT_CODE":"PLANE",
    "before.ORDER_ID":256,
    "before.PRODUCT_PRICE":133300.00,
    "before.PRODUCT_AMOUNT":1,
    "before.TRANSACTION_ID":100
}
```

9.3.4.3 Sample Row Modeled JSON Messages

Insert:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"I",
    "op_ts":"2015-11-05 18:45:36.000000",
    "current_ts":"2016-10-05T11:10:42.294000",
    "pos":"00000000000000002928",
    "CUST_CODE":"WILL",
    "ORDER_DATE":"1994-09-30:15:33:00",
    "PRODUCT_CODE":"CAR",
    "ORDER_ID":144,
    "PRODUCT_PRICE":17520.00,
    "PRODUCT_AMOUNT":3,
    "TRANSACTION_ID":100
}
```

Update:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"U",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T11:10:42.350005",
    "pos":"00000000000000004300",
```

```
    "CUST_CODE":"BILL",
    "ORDER_DATE":"1995-12-31:15:00:00",
    "PRODUCT_CODE":"CAR",
    "ORDER_ID":765,
    "PRODUCT_PRICE":14000.00
}
```

Delete:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"D",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T11:10:42.351002",
    "pos":"00000000000000005272",
    "CUST_CODE":"DAVE",
    "ORDER_DATE":"1993-11-03:07:51:35",
    "PRODUCT_CODE":"PLANE",
    "ORDER_ID":600,
    "PRODUCT_PRICE":135000.00,
    "PRODUCT_AMOUNT":2,
    "TRANSACTION_ID":200
}
```

Truncate:

```
{
    "table":"QASOURCE.TCUSTORD",
    "op_type":"T",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-10-05T11:10:42.351003",
    "pos":"00000000000000005480",
}
```

9.3.4.4 Sample Primary Key Output JSON Message

```
{
    "table":"DDL_OGGSRC.TCUSTMER",
    "op_type":"I",
    "op_ts":"2015-10-26 03:00:06.000000",
    "current_ts":"2016-04-05T08:59:23.001000",
    "pos":"00000000000000006605",
    "primary_keys":[
        "CUST_CODE"
    ],
    "after":{
        "CUST_CODE":"WILL",
        "NAME":"BG SOFTWARE CO.",
        "CITY":"SEATTLE",
        "STATE":"WA"
    }
}
```

9.3.5 JSON Schemas

By default, JSON schemas are generated for each source table encountered. JSON schemas are generated on a just in time basis when an operation for that table is first encountered. A JSON schema is not required to parse a JSON object. However, many JSON parsers can use a JSON schema to perform a validating parse of a JSON object. Alternatively, you can review the JSON schemas to understand the layout of output

JSON objects. The JSON schemas are created in the *GoldenGate_Home*/dirdef directory by default and are named by the following convention:

FULLY_QUALIFIED_TABLE_NAME.schema.json

The generation of the JSON schemas is suppressible.

Following is a JSON schema example for the JSON object listed in Sample Operation Modeled JSON Messages:

```
{
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"QASOURCE.TCUSTORD",
    "description":"JSON schema for table QASOURCE.TCUSTORD",
    "definitions":{
        "row":{
            "type":"object",
            "properties":{
                "CUST_CODE":{
                    "type":[
                        "string",
                        "null"
                    ]
                },
                "ORDER_DATE":{
                    "type":[
                        "string",
                        "null"
                    ]
                },
                "PRODUCT_CODE":{
                    "type":[
                        "string",
                        "null"
                    ]
                },
                "ORDER_ID":{
                    "type":[
                        "number",
                        "null"
                    ]
                },
                "PRODUCT_PRICE":{
                    "type":[
                        "number",
                        "null"
                    ]
                },
                "PRODUCT_AMOUNT":{
                    "type":[
                        "integer",
                        "null"
                    ]
                },
                "TRANSACTION_ID":{
                    "type":[
                        "number",
                        "null"
                    ]
                }
```

```
            },
            "additionalProperties":false
        },
        "tokens":{
            "type":"object",
            "description":"Token keys and values are free form key value pairs.",
            "properties":{
            },
            "additionalProperties":true
        }
    },
    "type":"object",
    "properties":{
        "table":{
            "description":"The fully qualified table name",
            "type":"string"
        },
        "op_type":{
            "description":"The operation type",
            "type":"string"
        },
        "op_ts":{
            "description":"The operation timestamp",
            "type":"string"
        },
        "current_ts":{
            "description":"The current processing timestamp",
            "type":"string"
        },
        "pos":{
            "description":"The position of the operation in the data source",
            "type":"string"
        },
        "primary_keys":{
            "description":"Array of the primary key column names.",
            "type":"array",
            "items":{
                "type":"string"
            },
            "minItems":0,
            "uniqueItems":true
        },
        "tokens":{
            "$ref":"#/definitions/tokens"
        },
        "before":{
            "$ref":"#/definitions/row"
        },
        "after":{
            "$ref":"#/definitions/row"
        }
    },
    "required":[
        "table",
        "op_type",
        "op_ts",
        "current_ts",
        "pos"
    ],
```

```
    "additionalProperties":false
}
```

Following is a JSON schema example for the JSON object listed in Sample Flattened Operation Modeled JSON Messages:

```
{
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"QASOURCE.TCUSTORD",
    "description":"JSON schema for table QASOURCE.TCUSTORD",
    "definitions":{
        "tokens":{
            "type":"object",
            "description":"Token keys and values are free form key value pairs.",
            "properties":{
            },
            "additionalProperties":true
        }
    },
    "type":"object",
    "properties":{
        "table":{
            "description":"The fully qualified table name",
            "type":"string"
        },
        "op_type":{
            "description":"The operation type",
            "type":"string"
        },
        "op_ts":{
            "description":"The operation timestamp",
            "type":"string"
        },
        "current_ts":{
            "description":"The current processing timestamp",
            "type":"string"
        },
        "pos":{
            "description":"The position of the operation in the data source",
            "type":"string"
        },
        "primary_keys":{
            "description":"Array of the primary key column names.",
            "type":"array",
            "items":{
                "type":"string"
            },
            "minItems":0,
            "uniqueItems":true
        },
        "tokens":{
            "$ref":"#/definitions/tokens"
        },
        "before.CUST_CODE":{
            "type":[
                "string",
                "null"
            ]
        },
        "before.ORDER_DATE":{
```

```
        "type":[
            "string",
            "null"
        ]
    },
    "before.PRODUCT_CODE":{
        "type":[
            "string",
            "null"
        ]
    },
    "before.ORDER_ID":{
        "type":[
            "number",
            "null"
        ]
    },
    "before.PRODUCT_PRICE":{
        "type":[
            "number",
            "null"
        ]
    },
    "before.PRODUCT_AMOUNT":{
        "type":[
            "integer",
            "null"
        ]
    },
    "before.TRANSACTION_ID":{
        "type":[
            "number",
            "null"
        ]
    },
    "after.CUST_CODE":{
        "type":[
            "string",
            "null"
        ]
    },
    "after.ORDER_DATE":{
        "type":[
            "string",
            "null"
        ]
    },
    "after.PRODUCT_CODE":{
        "type":[
            "string",
            "null"
        ]
    },
    "after.ORDER_ID":{
        "type":[
            "number",
            "null"
        ]
    },
    "after.PRODUCT_PRICE":{
```

```
            "type":[
                "number",
                "null"
            ]
        },
        "after.PRODUCT_AMOUNT":{
            "type":[
                "integer",
                "null"
            ]
        },
        "after.TRANSACTION_ID":{
            "type":[
                "number",
                "null"
            ]
        }
    },
    "required":[
        "table",
        "op_type",
        "op_ts",
        "current_ts",
        "pos"
    ],
    "additionalProperties":false
}
```

Following is a JSON schema example for the JSON object listed in Sample Row Modeled JSON Messages:

```
{
    "$schema":"http://json-schema.org/draft-04/schema#",
    "title":"QASOURCE.TCUSTORD",
    "description":"JSON schema for table QASOURCE.TCUSTORD",
    "definitions":{
        "tokens":{
            "type":"object",
            "description":"Token keys and values are free form key value pairs.",
            "properties":{
            },
            "additionalProperties":true
        }
    },
    "type":"object",
    "properties":{
        "table":{
            "description":"The fully qualified table name",
            "type":"string"
        },
        "op_type":{
            "description":"The operation type",
            "type":"string"
        },
        "op_ts":{
            "description":"The operation timestamp",
            "type":"string"
        },
        "current_ts":{
            "description":"The current processing timestamp",
```

```
        "type":"string"
    },
    "pos":{
        "description":"The position of the operation in the data source",
        "type":"string"
    },
    "primary_keys":{
        "description":"Array of the primary key column names.",
        "type":"array",
        "items":{
            "type":"string"
        },
        "minItems":0,
        "uniqueItems":true
    },
    "tokens":{
        "$ref":"#/definitions/tokens"
    },
    "CUST_CODE":{
        "type":[
            "string",
            "null"
        ]
    },
    "ORDER_DATE":{
        "type":[
            "string",
            "null"
        ]
    },
    "PRODUCT_CODE":{
        "type":[
            "string",
            "null"
        ]
    },
    "ORDER_ID":{
        "type":[
            "number",
            "null"
        ]
    },
    "PRODUCT_PRICE":{
        "type":[
            "number",
            "null"
        ]
    },
    "PRODUCT_AMOUNT":{
        "type":[
            "integer",
            "null"
        ]
    },
    "TRANSACTION_ID":{
        "type":[
            "number",
            "null"
        ]
    }
```

```
        },
        "required":[
            "table",
            "op_type",
            "op_ts",
            "current_ts",
            "pos"
        ],
        "additionalProperties":false
    }
```

9.3.6 JSON Formatter Configuration

Table 9-4 JSON Formatter Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.`*`name`*`.format`	Optional	`json` | `json_row`	None	Controls whether the generated JSON output messages are operation modeled or row modeled. Set to `json` for operation modeled or`json_row` for row modeled.
`gg.handler.`*`name`*`.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation.
`gg.handler.`*`name`*`.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.
`gg.handler.`*`name`*`.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a delete operation.
`gg.handler.`*`name`*`.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be inserted into the output record to indicate a truncate operation.
`gg.handler.`*`name`*`.format.prettyPrint`	Optional	`true` | `false`	`false`	Controls the output format of the JSON data. True is pretty print, formatted with white space to be more easily read by humans. False is not pretty print, more compact but very difficult for humans to read.
`gg.handler.`*`name`*`.format.jsonDelimiter`	Optional	Any string	`""` (no value)	Allows you to insert an optional delimiter between generated JSONs to allow them to be more easily parsed out of a continuous stream of data. Configuration value supports `CDATA[]` wrapping.
`gg.handler.`*`name`*`.format.generateSchema`	Optional	`true` | `false`	`true`	Controls the generation of JSON schemas for the generated JSON documents. JSON schemas are generated on a table by table basis. A JSON schema is not required to parse a JSON document. However, a JSON schema can provide you an indication of what the JSON documents will look like and can be used for a validating JSON parse.

Table 9-4 (Cont.) JSON Formatter Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.format.schemaDirectory`	Optional	Any legal, existing file system path	`./dirdef`	Controls the output location of generated JSON schemas.
`gg.handler.name.format.treatAllColumnsAsStrings`	Optional	`true | false`	`false`	Controls the output typing of generated JSON documents. If set to false then the formatter will attempt to map Oracle GoldenGate types to the corresponding JSON type. If set to true then all data will be treated as Strings in the generated JSONs and JSON schemas.
`gg.handler.name.format.encoding`	Optional	Any legal encoding name or alias supported by Java.	`UTF-8` (the JSON default)	Controls the output encoding of generated JSON schemas and documents.
`gg.handler.name.format.versionSchemas`	Optional	`true | false`	`false`	Controls the version of created schemas. Schema versioning causes a schema with a timestamp to be created in the schema directory on the local file system every time a new schema is created. True enables schema versioning. False disables schema versioning.
`gg.handler.name.format.iso8601Format`	Optional	`true | false`	`true`	Controls the format of the current timestamp. The default is the ISO 8601 format. Set to `false` removes the `"T"` between the date and time in the current timestamp, which outputs `" "` instead.
`gg.handler.name.format.includePrimaryKeys`	Optional	`true | false`	`false`	Set this configuration property to `true` to include an array of the primary key column names from the source table in the JSON output.
`gg.handler.name.format.flatten`	Optional	`true | false`	`false`	This property is only applicable to Operation Formatted JSON (`gg.handler.name.format=json`). Controls sending flattened JSON formatted data to the target entity. This must be set to `true` for the following property to work.
`gg.handler.name.format.flattenDelimiter`	Optional	Any legal character or character string for a JSON field name.	`.`	Controls the delimiter for concatenated JSON element names. It supports `CDATA[]` wrapping to preserve whitespace. It is only relevant when `gg.handler.name.format.flatten` is set to `true`.

Table 9-4 (Cont.) JSON Formatter Configuration Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.handler.name.format.beforeObjectName`	Optional	Any legal character or character string for a JSON field name.	Any legal JSON attribute name.	This property is only applicable to Operation Formatted JSON (`gg.handler.name.format=json`). Allows you to set whether the JSON element, that contains the before change column values, can be renamed.
`gg.handler.name.format.afterObjectName`	Optional	Any legal character or character string for a JSON field name.	Any legal JSON attribute name.	This property is only applicable to Operation Formatted JSON (`gg.handler.name.format=json`). Allows you to set whether the JSON element, that contains the after change column values, can be renamed.
`gg.handler.name.format.pkUpdateHandling`	Optional	`abend` | `update` | `delete-insert`	`abend`	Provides configuration for how the formatter should handle update operations that change a primary key. Primary key operations can be problematic for the JSON formatter and require special consideration by you. You can only use this property in conjunction with the row modeled JSON output messages. This property is only applicable to Row Formatted JSON (`gg.handler.name.format=json_row`). • `abend` - indicates that the process will abend. • `update` - indicates that the process will treat this as a normal update. • `delete` or `insert` - indicates that the process will treat this as a delete and an insert. Full supplemental logging needs to be enabled for this to work. Without full before and after row images the insert data will be incomplete.

9.3.7 Sample Configuration

The following is sample configuration for the JSON Formatter from the Java Adapter configuration file:

```
gg.handler.hdfs.format=json
gg.handler.hdfs.format.insertOpKey=I
gg.handler.hdfs.format.updateOpKey=U
gg.handler.hdfs.format.deleteOpKey=D
gg.handler.hdfs.format.truncateOpKey=T
gg.handler.hdfs.format.prettyPrint=false
gg.handler.hdfs.format.jsonDelimiter=CDATA[]
gg.handler.hdfs.format.generateSchema=true
```

```
gg.handler.hdfs.format.schemaDirectory=dirdef
gg.handler.hdfs.format.treatAllColumnsAsStrings=false
```

9.3.8 Metadata Change Events

Metadata change events are handled at runtime. A metadata change event for a given table results in the regeneration of the JSON schema the next time an operation for that table is encountered. The content of created JSON messages is changed to reflect the metadata change. For example, if the metadata change is to add an additional column, the new column will be included in created JSON messages after the metadata change event.

9.3.9 JSON Primary Key Updates

When the JSON formatter is configured to model operation data, the primary key updates require no special treatment and are treated like any other update.. The before and after values reflect the change in the primary key.

When the JSON formatter is configured to model row data, the primary key updates are a concern. The default behavior is to abend. You can configure the JSON formatter to model row data using the `gg.handler.name.format.pkUpdateHandling` configuration property to treat primary key updates as either a regular update, or as delete and then insert operations. If configured to operate as a delete and insert operations, Oracle recommends that you configure your replication stream to contain the complete before and after image data for updates. Otherwise, the generated insert operation for a primary key update will be missing data for fields that did not change.

9.3.10 Integrating Oracle Stream Analytics

You can integrate Oracle GoldenGate for Big Data with Oracle Stream Analytics (OSA) by sending operation modelled JSON messages to the Kafka Handler. This only works when the JSON formatter is configured to output operation modelled JSON messages.

OSA requires flattened JSON objects so a new feature was added to the JSON formatter generate flattened JSONs. You can use this feature by setting the JSON formatter property, `gg.handler.`*`name`*`.format.flatten=false` to `true`; `false` is the default. Following is an example of a flattened JSON file:

```
{
    "table":"QASOURCE.TCUSTMER",
    "op_type":"U",
    "op_ts":"2015-11-05 18:45:39.000000",
    "current_ts":"2016-06-22T13:38:45.335001",
    "pos":"00000000000000005100",
    "before.CUST_CODE":"ANN",
    "before.NAME":"ANN'S BOATS",
    "before.CITY":"SEATTLE",
    "before.STATE":"WA",
    "after.CUST_CODE":"ANN",
    "after.CITY":"NEW YORK",
    "after.STATE":"NY"
}
```

9.4 Avro Formatter

Apache Avro is an open source data serialization and deserialization framework known for its flexibility, compactness of serialized data, and good serialization and deserialization performance. Apache Avro is commonly used in Big Data applications.

Topics:

9.4.1 Avro Row Formatter

The Avro Row Formatter formats operation data from the source trail file into messages in an Avro binary array format. Each individual insert, update, delete, and truncate operation is formatted into an individual Avro message. The source trail file contains the before and after images of the operation data. The Avro Row Formatter takes that before and after image data and formats the data into an Avro binary representation of the operation data.

The Avro Row Formatter formats operations from the source trail file into a format that represents the row data. This format is more compact than the output from the Avro Operation Formatter for that the Avro messages model the change data operation.

The Avro Row Formatter may be a good choice when streaming Avro data to HDFS. Hive supports data files in HDFS in an Avro format.

This section contains the following topics:

9.4.1.1 Operation Metadata Formatting Details

Avro messages generated by the Avro Row Formatter contain the following seven metadata fields that begin the message:

Table 9-5 Avro Formatter Metadata

Value	Description
`table`	The fully qualified table name. The format of the fully qualified table name is: *`CATALOG_NAME.SCHEMA_NAME.TABLE_NAME`*
`op_type`	The operation type that is the indicator of the type of database operation from the source trail file. Default values are `I` for insert, `U` for update, `D` for delete, and `T` for truncate.

Table 9-5 (Cont.) Avro Formatter Metadata

Value	Description
`op_ts`	The operation timestamp is the timestamp of the operation from the source trail file. Since this timestamp is from the source trail it is fixed. Replaying the trail file results in the same timestamp for the same operation.
`current_ts`	The current timestamp is the current time when the formatter processed the current operation record. This timestamp follows the ISO-8601 format and includes microsecond precision. Replaying the trail file will *not* result in the same timestamp for the same operation.
`pos`	The trail file position is the concatenated sequence number and rba number from the source trail file. The trail position provides traceability of the operation back to the source trail file. The sequence number is the source trail file number. The rba number is the offset in the trail file.
`primary_keys`	An array variable holding the column names of the primary keys of the source table.
`tokens`	A map variable holding the token key value pairs from the source trail file.

9.4.1.2 Operation Data Formatting Details

The data following the operation metadata is the operation data. This data is represented as individual fields identified by the column names.

Column values for an operation from the source trail file can have one of three states: column has a value, column value is NULL, or column value is missing. Avro attributes only support two states, column has a value or column value is NULL. Missing column values will be treated the same as NULL values. Oracle recommends that when using the Avro Row Formatter, you configure the Oracle GoldenGate capture process to provide full image data for all columns in the source trail file.

The default setting of the Avro Row Formatter is to map the data types from the source trail file to the associated Avro data type. Avro provides limited support for data types so source columns map into Avro long, double, float, binary, or string data types. This data type mapping is configurable to alternatively treat all data as strings.

9.4.1.3 Sample Avro Row Messages

Avro messages are binary so not human readable. The following topics are sample messages and the JSON representation of the messages are displayed in them.

Sample Insert Message

Sample Update Message

Sample Delete Message

Sample Truncate Message

9.4.1.3.1 Sample Insert Message

```
{"table": "GG.TCUSTORD",
"op_type": "I",
"op_ts": "2013-06-02 22:14:36.000000",
"current_ts": "2015-09-18T10:13:11.172000",
"pos": "00000000000000001444",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"],
"tokens": {"R": "AADPkvAAEAAEqL2AAA"},
"CUST_CODE": "WILL",
"ORDER_DATE": "1994-09-30:15:33:00",
"PRODUCT_CODE": "CAR",
"ORDER_ID": "144",
"PRODUCT_PRICE": 17520.0,
"PRODUCT_AMOUNT": 3.0,
"TRANSACTION_ID": "100"}
```

9.4.1.3.2 Sample Update Message

```
{"table": "GG.TCUSTORD",
"op_type": "U",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:13:11.492000",
"pos": "00000000000000002891",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"R": "AADPkvAAEAAEqLzAAA"},
"CUST_CODE": "BILL",
"ORDER_DATE": "1995-12-31:15:00:00",
"PRODUCT_CODE": "CAR",
"ORDER_ID": "765",
"PRODUCT_PRICE": 14000.0,
"PRODUCT_AMOUNT": 3.0,
"TRANSACTION_ID": "100"}
```

9.4.1.3.3 Sample Delete Message

```
{"table": "GG.TCUSTORD",
"op_type": "D",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:13:11.512000",
"pos": "00000000000000004338",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"L": "206080450", "6": "9.0.80330", "R": "AADPkvAAEAAEqLzAAC"}, "CUST_CODE":
 "DAVE",
"ORDER_DATE": "1993-11-03:07:51:35",
"PRODUCT_CODE": "PLANE",
"ORDER_ID": "600",
"PRODUCT_PRICE": null,
"PRODUCT_AMOUNT": null,
"TRANSACTION_ID": null}
```

9.4.1.3.4 Sample Truncate Message

```
{"table": "GG.TCUSTORD",
"op_type": "T",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:13:11.514000",
"pos": "00000000000000004515",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"R": "AADPkvAAEAAEqL2AAB"},
"CUST_CODE": null,
```

```
"ORDER_DATE": null,
"PRODUCT_CODE": null,
"ORDER_ID": null,
"PRODUCT_PRICE": null,
"PRODUCT_AMOUNT": null,
"TRANSACTION_ID": null}
```

9.4.1.4 Avro Schemas

Avro uses JSONs to represent schemas. Avro schemas define the format of generated Avro messages and are required to serialize and deserialize Avro messages. Schemas are generated on a just in time basis when the first operation for a table is encountered. Generated Avro schemas are specific to a table definition that means that a separate Avro schema is generated for every table encountered for processed operations. By default, Avro schemas are written to the *GoldenGate_Home*/dirdef directory although the write location is configurable. Avro schema file names adhere to the following naming convention: *Fully_Qualified_Table_Name*.avsc.

The following is a sample Avro schema for the Avro Row Format for the previous references examples:

```
{
  "type" : "record",
  "name" : "TCUSTORD",
  "namespace" : "GG",
  "fields" : [ {
    "name" : "table",
    "type" : "string"
  }, {
    "name" : "op_type",
    "type" : "string"
  }, {
    "name" : "op_ts",
    "type" : "string"
  }, {
    "name" : "current_ts",
    "type" : "string"
  }, {
    "name" : "pos",
    "type" : "string"
  }, {
    "name" : "primary_keys",
    "type" : {
      "type" : "array",
      "items" : "string"
    }
  }, {
    "name" : "tokens",
    "type" : {
      "type" : "map",
      "values" : "string"
    },
    "default" : { }
  }, {
    "name" : "CUST_CODE",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "ORDER_DATE",
    "type" : [ "null", "string" ],
    "default" : null
```

```
  }, {
    "name" : "PRODUCT_CODE",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "ORDER_ID",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "PRODUCT_PRICE",
    "type" : [ "null", "double" ],
    "default" : null
  }, {
    "name" : "PRODUCT_AMOUNT",
    "type" : [ "null", "double" ],
    "default" : null
  }, {
    "name" : "TRANSACTION_ID",
    "type" : [ "null", "string" ],
    "default" : null
  } ]
}
```

9.4.1.5 Avro Row Configuration

Table 9-6 Avro Row Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.`*`name`*`.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation.
`gg.handler.`*`name`*`.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.
`gg.handler.`*`name`*`.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a delete operation.
`gg.handler.`*`name`*`.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be inserted into the output record to indicate a truncate operation.
`gg.handler.`*`name`*`.format.encoding`	Optional	Any legal encoding name or alias supported by Java.	UTF-8 (the JSON default)	Controls the output encoding of generated Avro schema that is a JSON. The JSON default is UTF-8. Avro messages are binary and support their own internal representation of encoding.

Table 9-6 (Cont.) Avro Row Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.treatAllColumnsAsStrings`	Optional	`true` | `false`	`false`	Controls the output typing of generated Avro messages. If set to false then the formatter will attempt to map Oracle GoldenGate types to the corresponding AVRO type. If set to true then all data will be treated as Strings in the generated Avro messages and schemas.
`gg.handler.name.format.pkUpdateHandlingformat.pkUpdateHandling`	Optional	`abend` | `update` | `delete-insert`	`abend`	Provides configuration for how the formatter should handle update operations that change a primary key. Primary key operations can be problematic for the Avro Row formatter and require special consideration by you. • `abend` - indicates the process will abend. • `update` - indicates the process will treat this as a normal update. • `delete` or `insert` - indicates the process will treat this as a delete and an insert. Full supplemental logging needs to be enabled for this to work. Without full before and after row images the insert data will be incomplete.
`gg.handler.name.format.lineDelimiter`	Optional	Any string	no value	Optionally allows a user to insert a delimiter after each Avro message. This is not considered the best practice but in certain use cases customers may wish to parse a stream of data and extract individual Avro messages from the stream. This property allows the customer that option. Select a unique delimiter that cannot occur in any Avro message. This property supports `CDATA[]` wrapping.

Table 9-6 (Cont.) Avro Row Configuration Options

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.versionSchemas`	Optional	`true|false`	`false`	The created Avro schemas always follow the convention *`fully_qualified_table_name`*`.avsc`. Setting this property to `true` creates an additional Avro schema in the schema directory named *`fully_qualified_table_name_current_timestamp`*`.avsc`. The additional Avro schema does not get destroyed or removed and thereby provides a history of schema evolution.
`gg.handler.name.format.wrapMessageInGenericAvroSchema`	Optional	`true|false`	`false`	Provides functionality to wrap the Avro messages for operations from the source trail file in a generic Avro wrapper message. For more information, see Generic Wrapper Functionality.
`gg.handler.name.format.schemaDirectory`	Optional	Any legal, existing file system path.	`./dirdef`	Controls the output location of generated Avro schemas.
`gg.handler.name.schemaFilePath=`	Optional	Any legal encoding name or alias supported by Java.	`./dirdef`	Controls the configuration property to a file directory inside of HDFS where you want schemas to be output. A metadata change event causes the schema to be overwritten when the next operation for the associated table is encountered. Schemas follow the same naming convention as schemas written to the local file system, *`catalog.schema.table`*`.avsc`.
`gg.handler.name.format.iso8601Format`	Optional	`true|false`	`true`	Controls the format of the current timestamp. The default is the ISO 8601 format. Set to `false` removes the `T` between the date and time in the current timestamp, which outputs a space instead.

9.4.1.6 Sample Configuration

The following is sample configuration for the Avro Row Formatter from the Java Adapter properties file:

```
gg.handler.hdfs.format=avro_row
gg.handler.hdfs.format.insertOpKey=I
gg.handler.hdfs.format.updateOpKey=U
gg.handler.hdfs.format.deleteOpKey=D
gg.handler.hdfs.format.truncateOpKey=T
gg.handler.hdfs.format.encoding=UTF-8
gg.handler.hdfs.format.pkUpdateHandling=abend
gg.handler.hafs.format.wrapMessageInGenericAvroMessage=false
```

9.4.1.7 Metadata Change Events

The Avro Row Formatter is capable of taking action in the case of a metadata change event. This assumes that the replicated database and upstream Oracle GoldenGate replication process can propagate metadata change events. Metadata change events are of particular importance when formatting using Avro due to the tight dependency of Avro messages to its corresponding schema.

Metadata change events are handled seamlessly by the Avro Row Formatter and an updated Avro schema will be generated upon the first encounter of an operation of that table after the metadata change event. You should understand the impact of a metadata change event and change downstream targets to the new Avro schema. The tight dependency of Avro messages to Avro schemas may result in compatibility issues. Avro messages generated before the schema change may not be able to be deserialized with the newly generated Avro schema.

Conversely, Avro messages generated after the schema change may not be able to be deserialized with the previous Avro schema. It is a best practice to use the same version of the Avro schema that was used to generate the message. Consult the Apache Avro documentation for more details.

9.4.1.8 Special Considerations

This sections describes these special considerations:

Troubleshooting

Primary Key Updates

Generic Wrapper Functionality

9.4.1.8.1 Troubleshooting

Avro is a binary format so is not human readable. Since Avro messages are in binary format, it is difficult to debug any issue so the Avro Row Formatter provides a special feature to mitigate this issue. When the `log4j` Java logging level is set to `TRACE` the created Avro messages are deserialized and displayed in the log file as a JSON object. This allows you to view the structure and contents of the created Avro messages. `TRACE` should never be enabled in a production environment as it has substantial negative impact on performance. Alternatively, you may want to consider switching to use a formatter that produces human readable content for content troubleshooting. The XML or JSON formatters both produce content in human readable format that may facilitate troubleshooting.

9.4.1.8.2 Primary Key Updates

Primary key update operations require special consideration and planning for Big Data integrations. Primary key updates are update operations that modify one or more of the primary keys for the given row from the source database. Since data is simply appended in Big Data applications, a primary key update operation looks more like a

new insert than an update without any special handling. The Avro Row Formatter provides specialized handling for primary keys that is configurable by you as follows:

Table 9-7 Configurable behavior

Value	Description
`abend`	The default behavior is that the delimited text formatter abends with a primary key update.
`update`	With this configuration the primary key update will be treated just like any other update operation. This configuration alternative should only be selected if you can guarantee that the primary key that is being changed is not being used as the selection criteria when selecting row data from a Big Data system.
`delete-insert`	Using this configuration the primary key update is treated as a special case of a delete using the before image data and an insert using the after image data. This configuration may more accurately model the effect of a primary key update in a Big Data application. However, if this configuration is selected it is important to have full supplemental logging enabled on Replication at the source database. Without full supplemental logging the delete operation will be correct, however, the insert operation will not contain all of the data for all of the columns for a full representation of the row data in the Big Data application.

9.4.1.8.3 Generic Wrapper Functionality

Avro messages are not self describing, which means that the receiver of the message must know the schema associated with the message before the message can be deserialized. Avro messages are binary and provide no consistent or reliable way to inspect the message contents in order to ascertain the message type. Therefore, Avro can be especially troublesome when messages are interlaced into a single stream of data like Kafka.

The Avro formatter provides a special feature to wrap the Avro message in a generic Avro message. This functionality is enabled by setting the following configuration property.

```
gg.handler.name.format.wrapMessageInGenericAvroMessage=true
```

The generic message is Avro message wrapping the Avro payload message that is common to all Avro messages that are output. The schema for the generic message is name `generic_wrapper.avsc` and is written to the output schema directory. This message has the following three fields.

- `table_name` - The fully qualified source table name.
- `schema_fingerprint` - The fingerprint of the Avro schema of the wrapped message. The fingerprint is generated using the Avro `SchemaNormalization.parsingFingerprint64(schema)` call.
- `payload` - The wrapped Avro message.

The following is the Avro Formatter generic wrapper schema.

```
{
  "type" : "record",
```

```
  "name" : "generic_wrapper",
  "namespace" : "oracle.goldengate",
  "fields" : [ {
    "name" : "table_name",
    "type" : "string"
  }, {
    "name" : "schema_fingerprint",
    "type" : "long"
  }, {
    "name" : "payload",
    "type" : "bytes"
  } ]
}
```

9.4.2 Avro Operation Formatter

The Avro Operation Formatter formats operation data from the source trail file into messages in an Avro binary array format. Each individual insert, update, delete, and truncate operation will be formatted into an individual Avro message. The source trail file will contain the before and after images of the operation data. The Avro Operation Formatter takes that before and after image data and formats the data into an Avro binary representation of the operation data.

The Avro Operation Formatter formats operations from the source trail file into a format that represents the operation data. This format is more verbose than the output from the Avro Row Formatter for which the Avro messages model the row data.

This section contains the following topics:

9.4.2.1 Operation Metadata Formatting Details

Avro messages, generated by the Avro Operation Formatter, contain the following metadata fields that begin the message:

Table 9-8 Avro Messages and its Metadata

Fields	Description
`table`	*`CATALOG_NAME.SCHEMA NAME.TABLE NAME`*The fully qualified table name. The format of the fully qualified table name is the following:
`op_type`	The operation type that is the indicator of the type of database operation from the source trail file. Default values are `I`for insert, `U`for update, `D`for delete, and `T`for truncate.

Table 9-8 (Cont.) Avro Messages and its Metadata

Fields	Description
op_ts	The operation timestamp is the timestamp of the operation from the source trail file. Since this timestamp is from the source trail, it is fixed. Replaying the trail file results in the same timestamp for the same operation.
current_ts	The current timestamp is the current time when the formatter processed the current operation record. This timestamp follows the ISO-8601 format and includes microsecond precision. Replaying the trail file will *not* result in the same timestamp for the same operation.
pos	The trail file position with is the concatenated sequence number and rba number from the source trail file. The trail position provides traceability of the operation back to the source trail file. The sequence number is the source trail file number. The rba number is the offset in the trail file.
primary_keys	An array variable holding the column names of the primary keys of the source table.
tokens	A map variable holding the token key value pairs from the source trail file.

9.4.2.2 Operation Data Formatting Details

The operation data is represented as individual fields identified by the column names.

Column values for an operation from the source trail file can have one of three states: column has a value, column value is NULL, or column value is missing. Avro attributes only support two states, column has a value or column value is NULL. The Avro Operation Formatter contains an additional Boolean field for each column as a special indicator if the column value is missing or not. This Boolean field is named, *COLUMN_NAME*_isMissing. Using the combination of the *COLUMN_NAME* field, all three states can be defined.

- State 1: Column has a value

  COLUMN_NAME field has a value

  *COLUMN_NAME*_isMissing field is false

- State 2: Column value is NULL

  COLUMN_NAME field value is NULL

  *COLUMN_NAME*_isMissing field is false

- State 3: Column value is missing

  COLUMN_NAME field value is NULL

  *COLUMN_NAME*_isMissing field is true

The default setting of the Avro Row Formatter is to map the data types from the source trail file to the associated Avro data type. Avro supports few data types so this functionality largely results in the mapping of numeric fields from the source trail file

to members typed as numbers. This data type mapping is configurable to alternatively treat all data as strings.

9.4.2.3 Sample Avro Operation Messages

Avro messages are binary and therefore not human readable. The following topics are sample messages the JSON representation of the messages displayed:

Sample Insert Message

Sample Update Message

Sample Delete Message

Sample Truncate Message

9.4.2.3.1 Sample Insert Message

```
{"table": "GG.TCUSTORD",
"op_type": "I",
"op_ts": "2013-06-02 22:14:36.000000",
"current_ts": "2015-09-18T10:17:49.570000",
"pos": "00000000000000001444",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"R": "AADPkvAAEAAEqL2AAA"},
"before": null,
"after": {
"CUST_CODE": "WILL",
"CUST_CODE_isMissing": false,
"ORDER_DATE": "1994-09-30:15:33:00",
"ORDER_DATE_isMissing": false,
"PRODUCT_CODE": "CAR",
"PRODUCT_CODE_isMissing": false,
"ORDER_ID": "144", "ORDER_ID_isMissing": false,
"PRODUCT_PRICE": 17520.0,
"PRODUCT_PRICE_isMissing": false,
"PRODUCT_AMOUNT": 3.0, "PRODUCT_AMOUNT_isMissing": false,
"TRANSACTION_ID": "100",
"TRANSACTION_ID_isMissing": false}}
```

9.4.2.3.2 Sample Update Message

```
{"table": "GG.TCUSTORD",
"op_type": "U",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:17:49.880000",
"pos": "00000000000000002891",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"R": "AADPkvAAEAAEqLzAAA"},
"before": {
"CUST_CODE": "BILL",
"CUST_CODE_isMissing": false,
"ORDER_DATE": "1995-12-31:15:00:00",
"ORDER_DATE_isMissing": false,
"PRODUCT_CODE": "CAR",
"PRODUCT_CODE_isMissing": false,
"ORDER_ID": "765",
"ORDER_ID_isMissing": false,
"PRODUCT_PRICE": 15000.0,
"PRODUCT_PRICE_isMissing": false,
"PRODUCT_AMOUNT": 3.0,
"PRODUCT_AMOUNT_isMissing": false,
```

```
"TRANSACTION_ID": "100",
"TRANSACTION_ID_isMissing": false},
"after": {
"CUST_CODE": "BILL",
"CUST_CODE_isMissing": false,
"ORDER_DATE": "1995-12-31:15:00:00",
"ORDER_DATE_isMissing": false,
"PRODUCT_CODE": "CAR",
"PRODUCT_CODE_isMissing": false,
"ORDER_ID": "765",
"ORDER_ID_isMissing": false,
"PRODUCT_PRICE": 14000.0,
"PRODUCT_PRICE_isMissing": false,
"PRODUCT_AMOUNT": 3.0,
"PRODUCT_AMOUNT_isMissing": false,
"TRANSACTION_ID": "100",
"TRANSACTION_ID_isMissing": false}}
```

9.4.2.3.3 Sample Delete Message

```
{"table": "GG.TCUSTORD",
"op_type": "D",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:17:49.899000",
"pos": "00000000000000004338",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"L": "206080450", "6": "9.0.80330", "R": "AADPkvAAEAAEqLzAAC"}, "before": {
"CUST_CODE": "DAVE",
"CUST_CODE_isMissing": false,
"ORDER_DATE": "1993-11-03:07:51:35",
"ORDER_DATE_isMissing": false,
"PRODUCT_CODE": "PLANE",
"PRODUCT_CODE_isMissing": false,
"ORDER_ID": "600",
"ORDER_ID_isMissing": false,
"PRODUCT_PRICE": null,
"PRODUCT_PRICE_isMissing": true,
"PRODUCT_AMOUNT": null,
"PRODUCT_AMOUNT_isMissing": true,
"TRANSACTION_ID": null,
"TRANSACTION_ID_isMissing": true},
"after": null}
```

9.4.2.3.4 Sample Truncate Message

```
{"table": "GG.TCUSTORD",
"op_type": "T",
"op_ts": "2013-06-02 22:14:41.000000",
"current_ts": "2015-09-18T10:17:49.900000",
"pos": "00000000000000004515",
"primary_keys": ["CUST_CODE", "ORDER_DATE", "PRODUCT_CODE", "ORDER_ID"], "tokens":
 {"R": "AADPkvAAEAAEqL2AAB"},
"before": null,
"after": null}
```

9.4.2.4 Avro Schema

Avro schemas are represented as JSONs. Avro schemas define the format of generated Avro messages and are required to serialize and deserialize Avro messages. Avro schemas are generated on a just in time basis when the first operation for a table is encountered. Avro schemas are specific to a table definition, which means that a

separate Avro schema is generated for every table encountered for processed operations. By default, Avro schemas are written to the *GoldenGate_Home*/dirdef directory although the write location is configurable. Avro schema file names adhere to the following naming convention: *Fully_Qualified_Table_Name*.avsc directory although the write location is configurable. Avro schema file names adhere to the following naming convention: .

The following is a sample Avro schema for the Avro Operation Format for the samples in the preceding sections:

```
{
  "type" : "record",
  "name" : "TCUSTORD",
  "namespace" : "GG",
  "fields" : [ {
    "name" : "table",
    "type" : "string"
  }, {
    "name" : "op_type",
    "type" : "string"
  }, {
    "name" : "op_ts",
    "type" : "string"
  }, {
    "name" : "current_ts",
    "type" : "string"
  }, {
    "name" : "pos",
    "type" : "string"
  }, {
    "name" : "primary_keys",
    "type" : {
      "type" : "array",
      "items" : "string"
    }
  }, {
    "name" : "tokens",
    "type" : {
      "type" : "map",
      "values" : "string"
    },
    "default" : { }
  }, {
    "name" : "before",
    "type" : [ "null", {
      "type" : "record",
      "name" : "columns",
      "fields" : [ {
        "name" : "CUST_CODE",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "CUST_CODE_isMissing",
        "type" : "boolean"
      }, {
        "name" : "ORDER_DATE",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "ORDER_DATE_isMissing",
        "type" : "boolean"
```

```
      }, {
        "name" : "PRODUCT_CODE",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "PRODUCT_CODE_isMissing",
        "type" : "boolean"
      }, {
        "name" : "ORDER_ID",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "ORDER_ID_isMissing",
        "type" : "boolean"
      }, {
        "name" : "PRODUCT_PRICE",
        "type" : [ "null", "double" ],
        "default" : null
      }, {
        "name" : "PRODUCT_PRICE_isMissing",
        "type" : "boolean"
      }, {
        "name" : "PRODUCT_AMOUNT",
        "type" : [ "null", "double" ],
        "default" : null
      }, {
        "name" : "PRODUCT_AMOUNT_isMissing",
        "type" : "boolean"
      }, {
        "name" : "TRANSACTION_ID",
        "type" : [ "null", "string" ],
        "default" : null
      }, {
        "name" : "TRANSACTION_ID_isMissing",
        "type" : "boolean"
      } ]
    } ],
    "default" : null
  }, {
    "name" : "after",
    "type" : [ "null", "columns" ],
    "default" : null
  } ]
}
```

9.4.2.5 Avro Operation Formatter Configuration

Table 9-9 Configuration Options

Properties	Optional Y/N	Legal Values	Default	Explanation
`gg.handler.name.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation
`gg.handler.name.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.

Table 9-9 (Cont.) Configuration Options

Properties	Optional Y/N	Legal Values	Default	Explanation
`gg.handler.name.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a delete operation.
`gg.handler.name.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be inserted into the output record to indicate a truncate operation.
`gg.handler.name.format.encoding`	Optional	Any legal encoding name or alias supported by Java	UTF-8 (the JSON default)	Controls the output encoding of generated Avro schema that is a JSON. JSON default is UTF-8. Avro messages are binary and support their own internal representation of encoding.
`gg.handler.name.format.treatAllColumnsAsStrings`	Optional	`true` | `false`	`false`	Controls the output typing of generated Avro messages. If set to `false`, then the formatter attempts to map Oracle GoldenGate types to the corresponding Avro type. If set to `true`, then all data is treated as Strings in the generated Avro messages and schemas.
`gg.handler.name.format.lineDelimiter`	Optional	Any string	no value	Optionally allows a user to insert a delimiter after each Avro message. This is not considered the best practice but in certain use cases customers may wish to parse a stream of data and extract individual Avro messages from the stream. This property allows the customer that option. Select a unique delimiter that cannot occur in any Avro message. This property supports `CDATA[]` wrapping.
`gg.handler.name.format.schemaDirectory`	Optional	Any legal, existing file system path.	`./dirdef`	Controls the output location of generated Avro schemas.

Table 9-9 (Cont.) Configuration Options

Properties	Optional Y/N	Legal Values	Default	Explanation
`gg.handler.name.format.wrapMessageInGenericAvroSchema`	Optional	`true|false`	`false`	Provides functionality to wrap the Avro messages for operations from the source trail file in a generic Avro wrapper message. For more information, see Generic Wrapper Functionality.
`gg.handler.name.format.iso8601Format`	Optional	`true | false`	`true`	Controls the format of the current timestamp. The default is the ISO 8601 format. Set to `false` removes the `T` between the date and time in the current timestamp, which outputs a space instead.

9.4.2.6 Sample Configuration

The following is a sample configuration for the Avro Operation Formatter from the Java Adapter `properg.handlerties` file:

```
gg.hdfs.format=avro_op
gg.handler.hdfs.format.insertOpKey=I
gg.handler.hdfs.format.updateOpKey=U
gg.handler.hdfs.format.deleteOpKey=D
gg.handler.hdfs.format.truncateOpKey=T
gg.handler.hdfs.format.encoding=UTF-8
gg.handler.hafs.format.wrapMessageInGenericAvroMessage=false
```

9.4.2.7 Metadata Change Events

The Avro Operation Formatter is capable of taking action with a metadata change event. This assumes that the replicated database and upstream Oracle GoldenGate replication process can propagate metadata change events. Metadata change events are of particular importance when formatting using Avro due to the tight dependency of Avro messages to its corresponding schema. Metadata change events are handled seamlessly by the Avro Operation Formatter and an updated Avro schema is generated upon the first encounter of an operation of that table after the metadata change event. You should understand the impact of a metadata change event and change downstream targets to the new Avro schema. The tight dependency of Avro messages to Avro schemas may result in compatibility issues. Avro messages generated before the schema change may not be able to be deserialized with the newly generated Avro schema. Conversely, Avro messages generated after the schema change may not be able to be deserialized with the previous Avro schema. It is a best practice to use the same version of the Avro schema that was used to generate the message. Consult the Apache Avro documentation for more details.

9.4.2.8 Special Considerations

This section describes these special considerations:

Troubleshooting

9.4.2.8.1 Troubleshooting

Avro is a binary format so is not human readable. Since Avro messages are in binary format, it is difficult to debug any issues. When the `log4j` Java logging level is set to `TRACE`, the created Avro messages are deserialized and displayed in the log file as a JSON object. This allows you to view the structure and contents of the created Avro messages. `TRACE` should never be enabled in a production environment as it has a substantial impact on performance.

9.4.2.8.2 Primary Key Updates

The Avro Operation Formatter creates messages with complete data of before and after images for update operations. Therefore, the Avro Operation Formatter requires no special treatment for primary key updates.

9.4.2.8.3 Generic Wrapper Message

Avro messages are not self describing, which means the receiver of the message must know the schema associated with the message before the message can be deserialized. Avro messages are binary and provide no consistent or reliable way to inspect the message contents in order to ascertain the message type. Therefore, Avro can be especially troublesome when messages are interlaced into a single stream of data like Kafka.

The Avro formatter provides a special feature to wrap the Avro message in a generic Avro message. This functionality is enabled by setting the following configuration property.

```
gg.handler.name.format.wrapMessageInGenericAvroMessage=true
```

The generic message is Avro message wrapping the Avro payload message that is common to all Avro messages that are output. The schema for the generic message is name `generic_wrapper.avsc` and is written to the output schema directory. This message has the following three fields.

- `table_name` - The fully qualified source table name.
- `schema_fingerprint` - The fingerprint of the of the Avro schema generating the messages. The fingerprint is generated using the `parsingFingerprint64(Schema s)` method on the `org.apache.avro.SchemaNormalization` class.
- `payload` - The wrapped Avro message.

The following is the Avro Formatter generic wrapper schema:

```
{
  "type" : "record",
  "name" : "generic_wrapper",
  "namespace" : "oracle.goldengate",
  "fields" : [ {
    "name" : "table_name",
    "type" : "string"
  }, {
    "name" : "schema_fingerprint",
    "type" : "long"
  }, {
```

```
    "name" : "payload",
    "type" : "bytes"
  } ]
}
```

9.4.3 Avro Object Container File Formatter

Oracle GoldenGate for Big Data can write to HDFS in Avro Object Container File (OCF) format. Using Avro OCF is a good choice for data formatting into HDFS because it handles schema evolution more efficiently than other formats. Compression and decompression is also supported in the Avro OCF Formatter to allow more efficient use of disk space.

The HDFS Handler integration with the Avro formatters to write files to HDFS in Avro OCF format is a specialized use case of the HDFS Handler. The Avro OCF format is required for Hive to be able to read Avro data in HDFS. The Avro OCF format is detailed in the Avro specification.

http://avro.apache.org/docs/current/spec.html#Object+Container+Files

Another important feature is that you can configure the HDFS Handler to stream data in Avro OCF format, generate table definitions in Hive, and update table definitions in Hive in the case of a metadata change event using the following:

Avro OCF Formatter Configuration

9.4.3.1 Avro OCF Formatter Configuration

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation.
`gg.handler.name.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.
`gg.handler.name.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be truncated into the output record to indicate a truncate operation.
`gg.handler.name.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a truncate operation.

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.encoding`	Optional	Any legal encoding name or alias supported by Java.	`UTF-8`	Controls the output encoding of generated Avro schema, which is a JSON. JSON default is UTF-8. Avro messages are binary and support their own internal representation of encoding.
`gg.handler.name.format.treatAllColumnsAsStrings`	Optional	`true` | `false`	`false`	Controls the output typing of generated Avro messages. If set to `false`, then the formatter attempts to map Oracle GoldenGate types to the corresponding Avro type. If set to `true`, then all data is treated as strings in the generated Avro messages and schemas.

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.pkUpdateHandling`	Optional	`abend` | `update` | `delete-insert`	`abend`	Controls how the formatter should handle update operations that change a primary key. Primary key operations can be problematic for the Avro Row formatter and require special consideration by you. • `abend` - indicates the process will abend • `update` - indicates the process will treat this as a normal update • `delete` and `insert` - indicates the process will treat this as a delete and an insert. The full before image is required for this feature to work properly. This can be achieved by using full supplemental logging in Oracle. Without full before and after row images the insert data will be incomplete.

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.generateSchema`	Optional	`true` | `false`	`true`	Schemas must be generated for Avro serialization so this property can be set to `false` to suppress the writing of the generated schemas to the local file system.
`gg.handler.name.format.schemaDirectory`	Optional	Any legal, existing file system path	`./dirdef`	Controls the output location of generated Avro schemas to the local file system. This property does not control where the Avro schema is written to in HDFS; that is controlled by an HDFS Handler property.
`gg.handler.name.format.iso8601Format`	Optional	`true` | `false`	`true`	The default format for the current timestamp is ISO8601. Set to `false` to remove the `T` between the date and time in the current timestamp and output a space instead.

Properties	Optional / Required	Legal Values	Default	Explanation
`gg.handler.name.format.versionSchemas`	Optional	`true` | `false`	`false`	If set to true, an Avro schema is created in the schema directory and versioned by a time stamp. The format of the schema is the following: *`fully_qualifiedtable_name_timestamp`*`.avsc`

9.5 XML Formatter

The XML Formatter formats operation data from the source trail file into a XML documents. The XML Formatter takes that before and after image data and formats the data into an XML document representation of the operation data. The format of the XML document is effectively the same as the XML format in the previous releases of the Oracle GoldenGate Java Adapter product.

This section contains the following topics:

9.5.1 Message Formatting Details

The XML formatted messages contain the following information:

Table 9-10 XML formatting details

Value	Description
`table`	The fully qualified table name.
`type`	The operation type.

Table 9-10 (Cont.) XML formatting details

Value	Description
`current_ts`	The current timestamp is the time when the formatter processed the current operation record. This timestamp follows the ISO-8601 format and includes micro second precision. Replaying the trail file does not result in the same timestamp for the same operation.
`pos`	The position from the source trail file.
`numCols`	The total number of columns in the source table.
`col`	The `col` element is a repeating element that contains the before and after images of operation data.
`tokens`	The `tokens` element contains the token values from the source trail file.

9.5.2 Sample XML Messages

This sections provides the following sample XML messages:

Sample Insert Message

Sample Update Message

Sample Delete Message

Sample Truncate Message

9.5.2.1 Sample Insert Message

```
<?xml version='1.0' encoding='UTF-8'?>
<operation table='GG.TCUSTORD' type='I' ts='2013-06-02 22:14:36.000000'
current_ts='2015-10-06T12:21:50.100001' pos='00000000000000001444' numCols='7'>
 <col name='CUST_CODE' index='0'>
   <before missing='true'/>
   <after><![CDATA[WILL]]></after>
 </col>
 <col name='ORDER_DATE' index='1'>
   <before missing='true'/>
   <after><![CDATA[1994-09-30:15:33:00]]></after>
 </col>
 <col name='PRODUCT_CODE' index='2'>
   <before missing='true'/>
   <after><![CDATA[CAR]]></after>
 </col>
 <col name='ORDER_ID' index='3'>
   <before missing='true'/>
   <after><![CDATA[144]]></after>
 </col>
 <col name='PRODUCT_PRICE' index='4'>
   <before missing='true'/>
   <after><![CDATA[17520.00]]></after>
 </col>
 <col name='PRODUCT_AMOUNT' index='5'>
   <before missing='true'/>
   <after><![CDATA[3]]></after>
```

```
 </col>
 <col name='TRANSACTION_ID' index='6'>
   <before missing='true'/>
   <after><![CDATA[100]]></after>
 </col>
 <tokens>
   <token>
     <Name><![CDATA[R]]></Name>
     <Value><![CDATA[AADPkvAAEAAEqL2AAA]]></Value>
   </token>
 </tokens>
</operation>
```

9.5.2.2 Sample Update Message

```
<?xml version='1.0' encoding='UTF-8'?>
<operation table='GG.TCUSTORD' type='U' ts='2013-06-02 22:14:41.000000'
current_ts='2015-10-06T12:21:50.413000' pos='00000000000000002891' numCols='7'>
 <col name='CUST_CODE' index='0'>
   <before><![CDATA[BILL]]></before>
   <after><![CDATA[BILL]]></after>
 </col>
 <col name='ORDER_DATE' index='1'>
   <before><![CDATA[1995-12-31:15:00:00]]></before>
   <after><![CDATA[1995-12-31:15:00:00]]></after>
 </col>
 <col name='PRODUCT_CODE' index='2'>
   <before><![CDATA[CAR]]></before>
   <after><![CDATA[CAR]]></after>
 </col>
 <col name='ORDER_ID' index='3'>
   <before><![CDATA[765]]></before>
   <after><![CDATA[765]]></after>
 </col>
 <col name='PRODUCT_PRICE' index='4'>
   <before><![CDATA[15000.00]]></before>
   <after><![CDATA[14000.00]]></after>
 </col>
 <col name='PRODUCT_AMOUNT' index='5'>
   <before><![CDATA[3]]></before>
   <after><![CDATA[3]]></after>
 </col>
 <col name='TRANSACTION_ID' index='6'>
   <before><![CDATA[100]]></before>
   <after><![CDATA[100]]></after>
 </col>
 <tokens>
   <token>
     <Name><![CDATA[R]]></Name>
     <Value><![CDATA[AADPkvAAEAAEqLzAAA]]></Value>
   </token>
 </tokens>
</operation>
```

9.5.2.3 Sample Delete Message

```
<?xml version='1.0' encoding='UTF-8'?>
<operation table='GG.TCUSTORD' type='D' ts='2013-06-02 22:14:41.000000'
current_ts='2015-10-06T12:21:50.415000' pos='00000000000000004338' numCols='7'>
 <col name='CUST_CODE' index='0'>
   <before><![CDATA[DAVE]]></before>
```

```
  <after missing='true'/>
 </col>
 <col name='ORDER_DATE' index='1'>
   <before><![CDATA[1993-11-03:07:51:35]]></before>
   <after missing='true'/>
 </col>
 <col name='PRODUCT_CODE' index='2'>
   <before><![CDATA[PLANE]]></before>
   <after missing='true'/>
 </col>
 <col name='ORDER_ID' index='3'>
   <before><![CDATA[600]]></before>
   <after missing='true'/>
 </col>
 <col name='PRODUCT_PRICE' index='4'>
  <missing/>
 </col>
 <col name='PRODUCT_AMOUNT' index='5'>
  <missing/>
 </col>
 <col name='TRANSACTION_ID' index='6'>
  <missing/>
 </col>
 <tokens>
   <token>
     <Name><![CDATA[L]]></Name>
     <Value><![CDATA[206080450]]></Value>
   </token>
   <token>
     <Name><![CDATA[6]]></Name>
     <Value><![CDATA[9.0.80330]]></Value>
   </token>
   <token>
     <Name><![CDATA[R]]></Name>
     <Value><![CDATA[AADPkvAAEAAEqLzAAC]]></Value>
   </token>
 </tokens>
</operation>
```

9.5.2.4 Sample Truncate Message

```
<?xml version='1.0' encoding='UTF-8'?>
<operation table='GG.TCUSTORD' type='T' ts='2013-06-02 22:14:41.000000'
current_ts='2015-10-06T12:21:50.415001' pos='00000000000000004515' numCols='7'>
 <col name='CUST_CODE' index='0'>
   <missing/>
 </col>
 <col name='ORDER_DATE' index='1'>
   <missing/>
 </col>
 <col name='PRODUCT_CODE' index='2'>
   <missing/>
 </col>
 <col name='ORDER_ID' index='3'>
   <missing/>
 </col>
 <col name='PRODUCT_PRICE' index='4'>
  <missing/>
 </col>
 <col name='PRODUCT_AMOUNT' index='5'>
  <missing/>
```

```
 </col>
 <col name='TRANSACTION_ID' index='6'>
  <missing/>
 </col>
 <tokens>
   <token>
     <Name><![CDATA[R]]></Name>
     <Value><![CDATA[AADPkvAAEAAEqL2AAB]]></Value>
   </token>
 </tokens>
</operation>
```

9.5.3 XML Schema

An XML schema (XSD) is not generated as part of the XML Formatter functionality. The XSD is generic to all messages generated by the XML Formatter. An XSD defining the structure of output XML documents is defined as follows:

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="operation">
     <xs:complexType>
       <xs:sequence>
          <xs:element name="col" maxOccurs="unbounded" minOccurs="0">
           <xs:complexType>
             <xs:sequence>
               <xs:element name="before" minOccurs="0">
                 <xs:complexType>
                   <xs:simpleContent>
                     <xs:extension base="xs:string">
                       <xs:attribute type="xs:string" name="missing"
use="optional"/>
                     </xs:extension>
                   </xs:simpleContent>
                 </xs:complexType>
               </xs:element>
               <xs:element name="after" minOccurs="0">
                 <xs:complexType>
                   <xs:simpleContent>
                     <xs:extension base="xs:string">
                       <xs:attribute type="xs:string" name="missing"
use="optional"/>
                     </xs:extension>
                   </xs:simpleContent>
                 </xs:complexType>
               </xs:element>
               <xs:element type="xs:string" name="missing" minOccurs="0"/>
             </xs:sequence>
             <xs:attribute type="xs:string" name="name"/>
             <xs:attribute type="xs:short" name="index"/>
           </xs:complexType>
         </xs:element>
         <xs:element name="tokens" minOccurs="0">
           <xs:complexType>
             <xs:sequence>
               <xs:element name="token" maxOccurs="unbounded" minOccurs="0">
                 <xs:complexType>
                   <xs:sequence>
                     <xs:element type="xs:string" name="Name"/>
                     <xs:element type="xs:string" name="Value"/>
                   </xs:sequence>
```

```
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="table"/>
      <xs:attribute type="xs:string" name="type"/>
      <xs:attribute type="xs:string" name="ts"/>
      <xs:attribute type="xs:dateTime" name="current_ts"/>
      <xs:attribute type="xs:long" name="pos"/>
      <xs:attribute type="xs:short" name="numCols"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

9.5.4 XML Configuration

Table 9-11 Configuration Options

Properties	Optional Y/N	Legal Values	Default	Explanation
`gg.handler.name.format.insertOpKey`	Optional	Any string	`I`	Indicator to be inserted into the output record to indicate an insert operation.
`gg.handler.name.format.updateOpKey`	Optional	Any string	`U`	Indicator to be inserted into the output record to indicate an update operation.
`gg.handler.name.format.deleteOpKey`	Optional	Any string	`D`	Indicator to be inserted into the output record to indicate a delete operation.
`gg.handler.name.format.truncateOpKey`	Optional	Any string	`T`	Indicator to be inserted into the output record to indicate a truncate operation.
`gg.handler.name.format.encoding`	Optional	Any legal encoding name or alias supported by Java.	UTF-8 (the XML default)	Controls the output encoding of generated XML documents.

Table 9-11 (Cont.) Configuration Options

Properties	Optional Y/N	Legal Values	Default	Explanation
`gg.handler.name.format.includeProlog`	Optional	`true` | `false`	`false`	Controls the output of an XML prolog on generated XML documents. The XML prolog is optional for well formed XML. Sample XML prolog looks like: `<?xml version='1.0' encoding='UTF-8' ?>`
`gg.handler.name.format.iso8601Format`	Optional	`true` | `false`	`true`	Controls the format of the current timestamp in the XML message. Set to `false` to suppress the `T` between the date and time and instead include blank space.

9.5.5 Sample Configuration

The following is sample configuration for the XML Formatter from the Java Adapter properties file:

```
gg.handler.hdfs.format=xml
gg.handler.hdfs.format.insertOpKey=I
gg.handler.hdfs.format.updateOpKey=U
gg.handler.hdfs.format.deleteOpKey=D
gg.handler.hdfs.format.truncateOpKey=T
gg.handler.hdfs.format.encoding=ISO-8859-1
gg.handler.hdfs.format.includeProlog=false
```

9.5.6 Metadata Change Events

The XML Formatter will seamlessly handle metadata change events. The format of the XML document is such that a metadata change event does not even result in a change to the XML schema. The XML schema is designed to be generic so that the same schema represents the data of any operation from any table.

The XML Formatter is capable of taking action with a metadata change event. This assumes that the replicated database and upstream Oracle GoldenGate replication process can propagate metadata change events. The format of the XML document is such that a metadata change event does not result in a change to the XML schema. The XML schema is generic so that the same schema represents the data of any operation form any table. The resulting changes in the metadata will be reflected in messages after the metadata change event. For example in the case of adding a column, the new column and column data will begin showing up in XML messages for that table after the metadata change event.

9.5.7 Primary Key Updates

Updates to a primary key require no special handling by the XML formatter. The XML formatter creates messages that model the database operations. For update operations, this includes before and after images of column values. Primary key changes are simply represented in this format as a change to a column value just like a change to any other column value

10

Using the Metadata Provider

This chapter explains the Metadata Provider functionality, different types of Metadata Providers, and examples that can be used to understand the functionality.

Topics:

10.1 About the Metadata Provider

The Metadata Provider is valid only if handlers are configured to run with a Replicat process.

The Replicat process provides functionality to perform source table to target table and source column to target column mapping using syntax in the Replicat configuration file. The source metadata definitions are included in the Oracle GoldenGate trail file (or by source definitions files for Oracle GoldenGate releases 12.2 and later). When the replication target is a database, the Replicat process obtains the target metadata definitions from the target database. However, this is a shortcoming when pushing data to Big Data applications or Java Delivery in general. Big Data applications generally provide no target metadata so the Replicat mapping is not possible. The Metadata Provider exists to address this deficiency. The Metadata Provider can be used to define target metadata using either Avro or Hive which in turn enables source table to target table and source column to target column Replicat mapping.

The use of the Metadata Provider is optional and is enabled if the `gg.mdp.type` property is specified in the Java Adapter Properties file.. If the metadata included in the source Oracle GoldenGate trail file is acceptable for the output, then do not use the Metadata Provider. The Metadata Provider should be used in the following cases:

- The requirement is for mapping source table names into target table names that do not match.
- The requirement is for mapping of source column names into target column name that do not match.
- The requirement is for the inclusion of certain columns from the source trail file and omitting other columns.

Replicat mapping has a general limitation in that the mapping defined in the Replicat configuration file is static. Oracle GoldenGate 12.2 and later provides functionality for DDL propagation when using an Oracle Database as the source. The proper handling of schema evolution can be problematic when the Metadata Provider and Replicat

mapping are being used. You should consider your use cases for schema evolution and plan for how you want to update the Metadata Provider and the Replicat mapping syntax for required changes.

For every table mapped in `REPLICAT` using `COLMAP`, the metadata will be retrieved from a configured metadata provider and retrieved metadata will then be used by `REPLICAT` for column mapping functionality.

You have choice of configuring one Metadata Provider implementation. Currently Hive and Avro Metadata Providers are supported.

Scenarios - When to use Metadata Provider

1. The following scenarios do *not* require the Metadata Provider to be configured:

   The mapping of schema name whereby the source schema named `GG` is mapped to the target schema named `GGADP`.*

   The mapping of schema and table name whereby the schema `GG.TCUSTMER` is mapped to the table name `GGADP.TCUSTMER_NEW`

   ```
   MAP GG.*, TARGET GGADP.*;
   (OR)
   MAP GG.TCUSTMER, TARGET GG_ADP.TCUSTMER_NEW;
   ```

2. The following scenario requires Metadata Provider to be configured:

   The mapping of column names whereby the source column name does not match the target column name. For example source column `CUST_CODE` mapped to target column `CUST_CODE_NEW`

   ```
   MAP GG.TCUSTMER, TARGET GG_ADP.TCUSTMER_NEW, COLMAP(USEDEFAULTS,
   CUST_CODE_NEW=CUST_CODE, CITY2=CITY);
   ```

10.2 Avro Metadata Provider

The Avro Metadata Provider is used to retrieve the table metadata from Avro Schema files. For every table mapped in `REPLICAT` using `COLMAP`, the metadata will be retrieved from Avro Schema and retrieved metadata will then be used by `REPLICAT` for column mapping.

This section contains the following:

10.2.1 Detailed Functionality

The Avro Metadata Provider uses Avro schema definition files to retrieve metadata. The Avro schemas are defined using the JSON. For each table mapped in *process_name*.prm file, a corresponding The Avro schema definition file should be created. More information on defining Avro schemas is found at:

http://avro.apache.org/docs/current/gettingstartedjava.html#Defining+a+schema

Avro Metadata Provider Schema definition syntax:

```
{"namespace": "[$catalogname.]$schemaname",
"type": "record",
"name": "$tablename",
"fields": [
     {"name": "$col1", "type": "$datatype"},
     {"name": "$col2 ",  "type": "$datatype ", "primary_key":true},
     {"name": "$col3", "type": "$datatype ", "primary_key":true},
     {"name": "$col4", "type": ["$datatype","null"]}
   ]
}

namespace            - name of catalog/schema being mapped
name                 - name of the table being mapped
fields.name          - array of column names
fields.type          - datatype of the column
fields.primary_key   - indicates the column is part of primary key.

Representing nullable and not nullable columns:

"type":"$datatype" - indicates the column is not nullable, where "$datatype" is the
actual datatype.
"type": ["$datatype","null"] - indicates the column is nullable, where "$datatype"
is the actual datatype
```

The file naming convention for Avro schema files accessed by the Avro Metadata Provider must be in the following format:

```
[$catalogname.]$schemaname.$tablename.mdp.avsc

$catalogname    - name of the catalog if exists
$schemaname   - name of the schema
$tablename        - name of the table
.mdp.avsc           -  constant, which should be appended always
```

Supported Avro Data Types:

- boolean
- bytes
- double
- float
- int
- long

- string

For more information on Avro data types, see https://avro.apache.org/docs/1.7.5/spec.html#schema_primitive.

10.2.2 Runtime Prerequisites

The Avro schema definitions should be created for all tables mapped in Replicat's parameter file before starting the Replicat process.

10.2.3 Classpath Configuration

There is no additional classpath setting required for Avro Metadata Provider.

10.2.4 Avro Metadata Provider Configuration

The configuration properties of Oracle GoldenGate Avro Metadata Provider are detailed in this section.

Property	Required/ Optional	Legal Values	Default	Explanation
`gg.mdp.type`	Required	`avro`	`-`	Selects the Avro Metadata Provider
`gg.mdp.schemaFilesPath`	Required	Example for a legal value could be `/home/user/ggadp/avroschema/`	`-`	Path to Avro schema files directory
`gg.mdp.charset`	Optional	Valid character set	`UTF-8`	Specifies the character set of the column with character data type. This is used to convert the source data from trail file to the correct target character set.
`gg.mdp.nationalCharset`	Optional	Valid character set	`UTF-8`	Specifies the character set of the column with character data type. This is used to convert the source data from trail file to the correct target character set. Example: Indicates character set of columns like `NCHAR`, `NVARCHAR`with an Oracle Database.

10.2.5 Sample Configuration

This section provides an example for configuring the Avro Metadata Provider. Consider a source with following table:

```
TABLE GG.TCUSTMER {
     CUST_CODE VARCHAR(4) PRIMARY KEY,
     NAME VARCHAR(100),
     CITY VARCHAR(200),
     STATE VARCHAR(200)
}
```

Mapping column`CUST_CODE (GG.TCUSTMER)` in source to `CUST_CODE2 (GG_AVRO.TCUSTMER_AVRO)` on target and column `CITY (GG.TCUSTMER)` in source to `CITY2 (GG_AVRO.TCUSTMER_AVRO)` on target. Thus, the mapping in *`process_name`*`.prm` file is:

```
MAP GG.TCUSTMER, TARGET GG_AVRO.TCUSTMER_AVRO, COLMAP(USEDEFAULTS,
CUST_CODE2=CUST_CODE, CITY2=CITY);
```

Mapping definition in this example:

- Source schema `GG` is mapped to target schema `GG_AVRO`.
- Source column `CUST_CODE` is mapped to target column `CUST_CODE2`.
- Source column `CITY` is mapped to target column `CITY2`.
- `USEDEFAULTS` specifies that rest of the columns names are same on both source and target (`NAME` and `STATE` columns).

The Avro schema definition file for the preceding example:

File path: `/home/ggadp/avromdpGG_AVRO.TCUSTMER_AVRO.mdp.avsc`

```
{"namespace": "GG_AVRO",
"type": "record",
"name": "TCUSTMER_AVRO",
"fields": [
     {"name": "NAME", "type": "string"},
    {"name": "CUST_CODE2",  "type": "string", "primary_key":true},
     {"name": "CITY2", "type": "string"},
     {"name": "STATE", "type": ["string","null"]}
]
}
```

The configuration in the Java Adapter properties file includes the following:

```
gg.mdp.type = avro
gg.mdp.schemaFilesPath = /home/ggadp/avromdp
```

Following is the sample output using delimited text formatter with a semi-colon as the delimiter for the preceding example.

```
I;GG_AVRO.TCUSTMER_AVRO;2013-06-02 22:14:36.000000;NAME;BG SOFTWARE
CO;CUST_CODE2;WILL;CITY2;SEATTLE;STATE;WA
```

The Oracle GoldenGate for Big Data installation include a sample Replicat configuration file, a sample Java Adapter properties file, and sample Avro schemas at:

GoldenGate_install_directory/AdapterExamples/big-data/metadata_provider/avro

10.2.6 Metadata Change Event

The Avro schema definitions and the mappings in the Replicat configuration file may need to be modified if there is a DDL change in the source database tables. You may want to stop or suspend the Replicat process in the case of a metadata change event. The Replicat process can be stopped by adding the following to the Replicat configuration file (*process_name*.prm):

```
DDL INCLUDE ALL, EVENTACTIONS (ABORT)
```

Alternatively, the Replicat process can be suspended by adding the following to the Replication configuration file.

```
DDL INCLUDE ALL, EVENTACTIONS (SUSPEND)
```

10.2.7 Limitations

Avro bytes data type cannot be used as primary key.

The source to target mapping defined in the Replicat configuration file is static. Oracle GoldenGate 12.2 and later supports DDL propagation and source schema evolution for Oracle Databases as the replication source. However, evolution of the source schemas may be problematic the static mapping configured in the Replicat configuration file.

10.2.8 Troubleshooting

This section contains the following:

Invalid Schema Files Location

Invalid Schema File Name

Invalid Namespace in Schema File

Invalid Table Name in Schema File

10.2.8.1 Invalid Schema Files Location

The Avro schema files directory location specified by the configuration property `gg.mdp.schemaFilesPath` should be a valid directory. Failure to configure a valid directory in `gg.mdp.schemaFilesPath` property leads to following exception:

```
oracle.goldengate.util.ConfigException: Error initializing Avro metadata provider
Specified schema location does not exist. {/path/to/schema/files/dir}
```

10.2.8.2 Invalid Schema File Name

For every table mapped in the *process_name*.prm file, a corresponding Avro schema file must be created in the directory specified in `gg.mdp.schemaFilesPath`.

For example, consider the following scenario:

Mapping:

```
MAP GG.TCUSTMER, TARGET GG_AVRO.TCUSTMER_AVRO, COLMAP(USEDEFAULTS,
cust_code2=cust_code, CITY2 = CITY);
```

Property:

```
gg.mdp.schemaFilesPath=/home/usr/avro/
```

A file called GG_AVRO.TCUSTMER_AVRO.mdp.avsc must be created in the /home/usr/avro/ directory. that is, /home/usr/avro/GG_AVRO.TCUSTMER_AVRO.mdp.avsc

Failing to create the /home/usr/avro/GG_AVRO.TCUSTMER_AVRO.mdp.avsc file results in the following exception:

```
java.io.FileNotFoundException: /home/usr/avro/GG_AVRO.TCUSTMER_AVRO.mdp.avsc
```

10.2.8.3 Invalid Namespace in Schema File

The target schema name specified in REPLICAT mapping must be same as namespace in the Avro schema definition file.

For example, consider the following scenario:

Mapping:

```
MAP GG.TCUSTMER, TARGET GG_AVRO.TCUSTMER_AVRO, COLMAP(USEDEFAULTS, cust_code2 =
cust_code, CITY2 = CITY);

Avro Schema Definition:

{
"namespace": "GG_AVRO",
..
}
```

In this scenario, REPLICAT abends with following exception if the target schema name specified in Replicat mapping does not match with Avro schema namespace:

```
Unable to retrieve table matadata. Table : GG_AVRO.TCUSTMER_AVRO
Mapped [catalogname.]schemaname (GG_AVRO) does not match with the schema namespace
{schema namespace}
```

10.2.8.4 Invalid Table Name in Schema File

The target table name specified in Replicat mapping must be same as name in the Avro schema definition file.

For example, consider the following scenario:

Mapping:

```
MAP GG.TCUSTMER, TARGET GG_AVRO.TCUSTMER_AVRO, COLMAP(USEDEFAULTS, cust_code2 =
cust_code, CITY2 = CITY);
```

Avro Schema Definition:

```
{
"namespace": "GG_AVRO",
"name": "TCUSTMER_AVRO",
..
}
```

In this scenario, REPLICAT abends with following exception if the target table name specified in Replicat mapping does not match with Avro schema name.

```
Unable to retrieve table matadata. Table : GG_AVRO.TCUSTMER_AVRO
Mapped table name (TCUSTMER_AVRO) does not match with the schema table name {table
name}
```

10.3 Java Database Connectivity Metadata Provider

The Java Database Connectivity (JDBC) Metadata Provider is used to retrieve the table metadata from any target database that supports a JDBC connection and has a database schema. The JDBC Metadata Provider should be the preferred metadata provider for any target database that is an RDBMS, although there are various other non-RDBMS targets that also provide a JDBC driver

This section contains the following:

10.3.1 JDBC Detailed Functionality

The JDBC Metadata Provider uses the JDBC Driver provided with your target database. The metadata is retrieved using the JDBC Driver for every target table mapped in the Replicat properties file. Replicat processes use the retrieved target metadata for the column mapping functionality.

You can enable this feature for JDBC Handler by configuring the `REPERROR` in your Replicat parameter file. In addition, you need to define the error codes specific to your RDBMS JDBC target in the JDBC Handler properties file as follows:

Table 10-1 JDBC REPERROR Codes

Property	Value	Required
`gg.error.duplicateErrorCodes`	Comma-separated integer values of error codes that mean duplicate errors	No
`gg.error.notFoundErrorCodes`	Comma-separated integer values of error codes that mean duplicate errors	No
`gg.error.deadlockErrorCodes`	Comma-separated integer values of error codes that mean duplicate errors	No

For example:

```
#ErrorCode
gg.error.duplicateErrorCodes=1062,1088,1092,1291,1330,1331,1332,1333
gg.error.notFoundErrorCodes=0
gg.error.deadlockErrorCodes=1213
```

To understand how the various JDBC types are mapped to database-specific SQL types, review the specifics at:

https://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/getstart/mapping.html#table1

10.3.2 Java Classpath

The JDBC Java Driver location must be included in the class path of the handler using the `gg.classpath` property.

For example, the configuration for a MySQL database could be:

```
gg.classpath= /path/to/jdbc/driver/jar/mysql-connector-java-5.1.39-bin.jar
```

10.3.3 JDBC Metadata Provider Configuration

The following are the configurable values for the JDBC Metadata Provider. These properties are located in the Java Adapter properties file (not in the Replicat properties file).

Table 10-2 JDBC Metadata Provider Properties

Properties	Required/ Optional	Legal Values	Default	Explanation
`gg.mdp.type`	Required	`jdbc`	None	Entering `jdbc` at a command prompt activates the use of the JDBC Metadata Provider.
`gg.mdp.ConnectionUrl`	Required	`jdbc:`*`subprotocol:subname`*	None	The target database JDBC URL.
`gg.mdp.DriverClassName`	Required	Java class name of the JDBC driver	None	The fully qualified Java class name of the JDBC driver.
`gg.mdp.userName`	Optional	A legal username string.	None	The user name for the JDBC connection. Alternatively, you can provide the user name using the `ConnectionURL` property.
`gg.mdp.password`	Optional	A legal password string.	None	The password for the JDBC connection. Alternatively, you can provide the user name using the `ConnectionURL` property.

10.3.4 Sample Configuration

This section provides examples for configuring the JDBC Metadata Provider.

MySQL Driver configuration:

```
gg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:oracle:thin:@myhost:1521:orcl
gg.mdp.DriverClassName=oracle.jdbc.driver.OracleDriver
gg.mdp.UserName=username
```

```
gg.mdp.Password=password
```

Oracle Thin Driver configuration:

```
gg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:mysql://localhost/databaseName?
user=username&password=password
gg.mdp.DriverClassName=com.mysql.jdbc.Driver
gg.mdp.UserName=username
gg.mdp.Password=password
```

Oracle OCI Driver configuration:

```
ggg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:oracle:oci:@myhost:1521:orcl
gg.mdp.DriverClassName=oracle.jdbc.driver.OracleDriver
gg.mdp.UserName=username
gg.mdp.Password=password
```

Oracle Teradata Driver configuration:

```
gg.mdp.type=jdbc
gg.mdp.ConnectionUrl=jdbc:teradata://10.111.11.111/USER=username,PASSWORD=password
gg.mdp.DriverClassName=com.teradata.jdbc.TeraDrivergg.mdp.UserName=username
gg.mdp.UserName=username
gg.mdp.Password=password
```

10.4 Hive Metadata Provider

The Hive Metadata Provider is used to retrieve the table metadata from a Hive metastore. The metadata will be retrieved from Hive for every target table mapped in the Replicat properties file using the COLMAP parameter. The retrieved target metadata is used by Replicat for the column mapping functionality.

This sections contains the following:

10.4.1 Detailed Functionality

The Hive Metadata Provider uses both Hive JDBC and HCatalog interfaces to retrieve metadata from the Hive metastore. For each table mapped in the *process_name*.prm file, a corresponding table should be created in Hive.

The default Hive configuration starts an embedded and local metastore Derby database. Apache Derby is designed to be an embedded database and only allows a single connection. The single connection limitation of the Derby Database as the Hive Metastore implementation means that it cannot function when working with the Hive Metadata Provider. To overcome this, you must configure Hive with a remote metastore database. More information on configuring Hive with remote metastore database can found at:

https://cwiki.apache.org/confluence/display/Hive/AdminManual+MetastoreAdmin#AdminManualMetastoreAdmin-RemoteMetastoreDatabase

Hive does not support Primary Key semantics, so the metadata retrieved from Hive metastore will not include any primary key definition. Replicat's KEYCOLS parameter should be used to define primary keys when you use the Hive Metadata Provider.

KEYCOLS

The Replicat mapping KEYCOLS parameter must be used to define primary keys in the target schema. The Oracle GoldenGate HBase Handler requires primary keys. Therefore, setting primary keys in the target schema is required when Replicat mapping is employed with HBase as the target.

Additionally, the output of the Avro Formatters includes an Array field to hold the primary column names. If Replicat mapping is employed with the Avro Formatters you should consider using KEYCOLS to identify the primary key columns.

Examples of configuring KEYCOLS is described in Sample Configuration.

Supported Hive Data types:

- BIGINT
- BINARY
- BOOLEAN
- CHAR
- DATE
- DECIMAL
- DOUBLE
- FLOAT
- INT
- SMALLINT
- STRING
- TIMESTAMP

- TINYINT
- VARCHAR

For more information on Hive data types, see https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types.

10.4.2 Configuring Hive with a Remote Metastore Database

A list of supported databases that can be used to configure remote Hive metastore can be found at https://cwiki.apache.org/confluence/display/Hive/AdminManual+MetastoreAdmin#AdminManualMetastoreAdmin-SupportedBackendDatabasesforMetastore.

In the following example, a MySQL database is configured as the Hive metastore using the following properties in the ${HIVE_HOME}/conf/hive-site.xml Hive configuration file:

Note:

The ConnectionURL and driver class used in this example are specific to MySQL database. Change the values appropriately if any database other than MySQL is chosen.

```
<property>
         <name>javax.jdo.option.ConnectionURL</name>
         <value>jdbc:mysql://MYSQL_DB_IP:MYSQL_DB_PORT/DB_NAME?
createDatabaseIfNotExist=false</value>
 </property>

 <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
 </property>

 <property>
          <name>javax.jdo.option.ConnectionUserName</name>
     <value>MYSQL_CONNECTION_USERNAME</value>
 </property>

 <property>
         <name>javax.jdo.option.ConnectionPassword</name>
         <value>MYSQL_CONNECTION_PASSWORD</value>
 </property>
```

The list of parameters to be configured in the hive-site.xml file for a remote metastore can be found at https://cwiki.apache.org/confluence/display/Hive/AdminManual+MetastoreAdmin#AdminManualMetastoreAdmin-RemoteMetastoreDatabase.

Note:

MySQL JDBC connector JAR must be added in the Hive classpath:

1. In *HIVE_HOME*/lib/ directory. *DB_NAME* should be replaced by a valid database name created in MySQL.
2. Start the Hive Server:

   HIVE_HOME/bin/hiveserver2/bin/hiveserver2
3. Start the Hive Remote Metastore Server:

   HIVE_HOME/bin/hive --service metastore

10.4.3 Classpath Configuration

You must configure two things in the gg.classpath configuration variable in order for the Hive Metadata Provider to connect to Hive and run. The first is the hive-site.xml file and the second are the Hive and HDFS client jars. The client JARs must match the version of Hive that the Hive Metadata Provider is connecting.

1. Create hive-site.xml file with the following properties:

```
<configuration>
<!-- Mandatory Property -->
<property>
<name>hive.metastore.uris</name>
<value>thrift://HIVE_SERVER_HOST_IP:9083</value>
<property>

<!-- Optional Property. Default value is 5 -->
<property>
<name>hive.metastore.connect.retries</name>
<value>3</value>
</property>

<!-- Optional Property. Default value is 1 -->
<property>
<name>hive.metastore.client.connect.retry.delay</name>
<value>10</value>
</property>

<!-- Optional Property. Default value is 600 seconds -->
<property>
<name>hive.metastore.client.socket.timeout</name>
<value>50</value>
</property>

 </configuration>
```

Note:

For example, if the hive-site.xml file is created in the /home/user/oggadp/dirprm directory, then gg.classpath entry is gg.classpath=/home/user/oggadp/dirprm/

2. The default location of the Hive and HDFS client jars are the following directories:

```
HIVE_HOME/hcatalog/share/hcatalog/*
HIVE_HOME/lib/*
HIVE_HOME/hcatalog/share/webhcat/java-client/*
HADOOP_HOME/share/hadoop/common/*
HADOOP_HOME/share/hadoop/common/lib/*
HADOOP_HOME/share/hadoop/mapreduce/*
```

Configure the `gg.classpath` exactly as shown in the preceding step. Creating a path to the `hive-site.xml` should contain the path with no wildcard appended. The inclusion of the * wildcard in the path to the `hive-site.xml` file causes it not to be picked up. Conversely, creating a path to the dependency JARs should include the * wildcard character to include all of the JAR files in that directory in the associated classpath. Do *not* use `*.jar`.

10.4.4 Hive Metadata Provider Configuration

The configuration properties of the Hive Metadata Provider are detailed in this section.

Property	Required/ Optional	Legal Values	Default	Explanation
`gg.mdp.type`	Required	`hive`	-	Selects Hive Metadata Provider
`gg.mdp.connectionUrl`	Required	Format without Kerberos Authentication: `jdbc:hive2://HIVE_SERVER_IP:HIVE_JDBC_PORT/HIVE_DB` Format with Kerberos Authentication: `jdbc:hive2://HIVE_SERVER_IP:HIVE_JDBC_PORT/HIVE_DB;principal=user/FQDN@MY.REALM`	-	JDBC Connection URL of Hive Server
`gg.mdp.driverClassName`	Required	`org.apache.hive.jdbc.HiveDriver`	-	Fully qualified Hive JDBC Driver class name.
`gg.mdp.userName`	Optional	Valid username	""	User name to connect to the Hive Database. The `userName` property is not required when Kerberos Authentication is used. The Kerberos principal should be specified in the connection URL as specified in `connectionUrl` property's legal values.

Property	Required/ Optional	Legal Values	Default	Explanation
gg.mdp.password	Optional	Valid Password	""	Password to connect to Hive Database
gg.mdp.charset	Optional	Valid character set	UTF-8	Specifies the character set of the column with character data type. This is used to convert the source data from trail file to the correct target character set.
gg.mdp.nationalCharset	Optional	Valid character set	UTF-8	Specifies the character set of the column with character data type. This is used to convert the source data from trail file to the correct target character set. For example, Indicates character set of columns like NCHAR, NVARCHAR in an Oracle Database.
gg.mdp.authType	Optional	kerberos	none	
gg.mdp.kerberosKeytabFile	Optional (Required if authType=kerberos)	Relative or absolute path to a Kerberos keytab file.	-	The keytab file allows Hive to access a password to perform kinit operation for Kerberos security.
gg.mdp.kerberosPrincipal	Optional (Required if authType=kerberos)	A legal Kerberos principal name(user/FQDN@MY.REALM)	-	The Kerberos principal name for Kerberos authentication.

10.4.5 Sample Configuration

The following is an example for configuring the Hive Metadata Provider. Consider a source with following table:

```
TABLE GG.TCUSTMER {
     CUST_CODE VARCHAR(4)   PRIMARY KEY,
     NAME VARCHAR(100),
     CITY VARCHAR(200),
     STATE VARCHAR(200)}
```

The example maps the column CUST_CODE (GG.TCUSTMER) in the source to CUST_CODE2 (GG_HIVE.TCUSTMER_HIVE) on the target and column CITY (GG.TCUSTMER) in the source to CITY2 (GG_HIVE.TCUSTMER_HIVE)on the target.

Mapping configuration in the *process_name*.prm file:

```
MAP GG.TCUSTMER, TARGET GG_HIVE.TCUSTMER_HIVE, COLMAP(USEDEFAULTS,
CUST_CODE2=CUST_CODE, CITY2=CITY) KEYCOLS(CUST_CODE2);
```

The mapping definition for this example is:

- Source schema GG is mapped to target schema GG_HIVE
- Source column CUST_CODE is mapped to target column CUST_CODE2
- Source column CITY is mapped to target column CITY2
- USEDEFAULTS specifies that rest of the columns names are same on both source and target (NAME and STATE columns).
- KEYCOLS is used to specify that CUST_CODE2 should be treated as primary key.

Since primary keys cannot be specified in Hive DDL, the KEYCOLS parameter is used to specify the primary keys.

Create schema and tables in Hive for the preceding example:

Note:

You can choose any schema name and are not restricted to the gg_hive schema name. The Hive schema can be pre-existing or newly created. You do this by modifying the connection URL (gg.mdp.connectionUrl) in the Java Adapter properties file and the mapping configuration in the Replicat.prm file. Once the schema name is changed, the connection URL (gg.mdp.connectionUrl) and mapping in the Replicat.prm file should be updated.

To start the Hive CLI type the following command:

HIVE_HOME/bin/hive

To create a schema, GG_HIVE, in Hive, use the following command:

```
hive> create schema gg_hive;
OK
Time taken: 0.02 seconds
```

To create a table TCUSTMER_HIVE in GG_HIVE database type the following command:

```
hive> CREATE EXTERNAL TABLE `TCUSTMER_HIVE`(
    >   "CUST_CODE2" VARCHAR(4),
    >   "NAME" VARCHAR(30),
    >   "CITY2" VARCHAR(20),
    >   "STATE" STRING);
OK
Time taken: 0.056 seconds
```

Configuration in the .properties file can be like the following:

```
gg.mdp.type=hive
gg.mdp.connectionUrl=jdbc:hive2://HIVE_SERVER_IP:10000/gg_hive
gg.mdp.driverClassName=org.apache.hive.jdbc.HiveDriver
```

Following is the sample output using delimited text formatter with a comma as the delimiter for the preceding example.

```
I;GG_HIVE.TCUSTMER_HIVE;2015-10-07T04:50:47.519000;cust_code2;WILL;name;BG SOFTWARE
CO;city2;SEATTLE;state;WA
```

A sample Replicat configuration file, Java Adapter properties file, and a Hive create table SQL script are included with the installation, and located at:

GoldenGate_install_directory/AdapterExamples/big-data/metadata_provider/hive

10.4.6 Security

The Hive server can be secured using Kerberos Authentication. Refer to the Hive documentation for your specific Hive release for instructions on how to secure the Hive server. The Hive Metadata Provider can connect to a Kerberos secured Hive server.

The HDFS `core-site.xml` and `hive-site.xml` should be in handler's classpath.

Enable the following properties in `core-site.xml`:

```
<property>
<name>hadoop.security.authentication</name>
<value>kerberos</value>
</property>

<property>
<name>hadoop.security.authorization</name>
<value>true</value>
</property>
```

Enable the following properties in `hive-site.xml`

```
<property>
<name>hive.metastore.sasl.enabled</name>
<value>true</value>
</property>

<property>
<name>hive.metastore.kerberos.keytab.file</name>
<value>/path/to/keytab</value> <!-- Change this value -->
</property>

<property>
<name>hive.metastore.kerberos.principal</name>
<value>Kerberos Principal</value> <!-- Change this value -->
</property>

<property>
   <name>hive.server2.authentication</name>
    <value>KERBEROS</value>
</property>

<property>
   <name>hive.server2.authentication.kerberos.principal</name>
    <value>Kerberos Principal</value> <!-- Change this value -->
</property>

<property>
    <name>hive.server2.authentication.kerberos.keytab</name>
```

```
    <value>/path/to/keytab</value> <!-- Change this value -->
</property>
```

10.4.7 Metadata Change Event

Tables in Hive metastore should be updated/altered/created manually if there is a change in source database tables. You may wish to abort or suspend the Replicat process in the case of a metadata change event. The Replicat process can be aborted by adding the following to the Replicat configuration file (*process_name*.prm):

```
DDL INCLUDE ALL, EVENTACTIONS (ABORT)
```

Alternatively, the Replicat process can be suspended by adding the following to the Replication configuration file (*process_name*.prm):

```
DDL INCLUDE ALL, EVENTACTIONS (SUSPEND)
```

10.4.8 Limitations

Columns with binary data type cannot be used as primary key.

The source to target mapping defined in the Replicat configuration file is static. Oracle GoldenGate 12.2 and later supports DDL propagation and source schema evolution for Oracle Databases as the replication source. However, evolution of the source schemas may be problematic the static mapping configured in the Replicat configuration file.

10.4.9 Additional Considerations

The most common problems encountered are the Java classpath issues. The Hive Metadata Provider requires certain Hive and HDFS client libraries to be resolved in its classpath as a prerequisite.

The required client JAR directories are listed in Classpath Configuration. Hive and HDFS client jars do not ship with Oracle GoldenGate for Big Data product. The client JARs should be the same version as the Hive version to which Hive Metadata Provider is connecting.

To establish a connection to the Hive server, the `hive-site.xml` file must be in the classpath.

10.4.10 Troubleshooting

The Replicat process will abend with a *"Table metadata resolution exception"* if the mapped target table does not exist in Hive.

For example, consider the following mapping:

```
MAP GG.TCUSTMER, TARGET GG_HIVE.TCUSTMER_HIVE, COLMAP(USEDEFAULTS,
CUST_CODE2=CUST_CODE, CITY2=CITY) KEYCOLS(CUST_CODE2);
```

The preceding mapping requires a table called `TCUSTMER_HIVE` to be created in schema `GG_HIVE` in the Hive metastore. Failure to create the `GG_HIVE.TCUSTMER_HIVE` table in Hive will result in following exception:

```
ERROR [main) - Table Metadata Resolution Exception
Unable to retrieve table matadata. Table : GG_HIVE.TCUSTMER_HIVE
NoSuchObjectException(message:GG_HIVE.TCUSTMER_HIVE table not found)
```

A

HDFS Handler Client Dependencies

This appendix lists the HDFS client dependencies for Apache Hadoop. The `hadoop-client-x.x.x.jar` is not distributed with Apache Hadoop nor is it mandatory to be in the classpath. The `hadoop-client-x.x.x.jar` is an empty maven project with the purpose of aggregating all of the Hadoop client dependencies.

Maven groupId: `org.apache.hadoop`

Maven atifactId: `hadoop-client`

Maven version: the HDFS version numbers listed for each section

A.1 Hadoop Client Dependencies

This section lists the Hadoop client dependencies for each HDFS version.

A.1.1 HDFS 2.7.1

HDFS 2.7.1 (HDFS 2.7.0 is effectively the same, simply substitute 2.7.0 on the libraries versioned as 2.7.1)

```
activation-1.1.jar
apacheds-i18n-2.0.0-M15.jar
apacheds-kerberos-codec-2.0.0-M15.jar
api-asn1-api-1.0.0-M20.jar
```

```
api-util-1.0.0-M20.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.1.3.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
curator-client-2.7.1.jar
curator-framework-2.7.1.jar
curator-recipes-2.7.1.jar
gson-2.2.4.jar
guava-11.0.2.jar
hadoop-annotations-2.7.1.jar
hadoop-auth-2.7.1.jar
hadoop-client-2.7.1.jar
hadoop-common-2.7.1.jar
hadoop-hdfs-2.7.1.jar
hadoop-mapreduce-client-app-2.7.1.jar
hadoop-mapreduce-client-common-2.7.1.jar
hadoop-mapreduce-client-core-2.7.1.jar
hadoop-mapreduce-client-jobclient-2.7.1.jar
hadoop-mapreduce-client-shuffle-2.7.1.jar
hadoop-yarn-api-2.7.1.jar
hadoop-yarn-client-2.7.1.jar
hadoop-yarn-common-2.7.1.jar
hadoop-yarn-server-common-2.7.1.jar
htrace-core-3.1.0-incubating.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.9.13.jar
jackson-jaxrs-1.9.13.jar
jackson-mapper-asl-1.9.13.jar
jackson-xc-1.9.13.jar
jaxb-api-2.2.2.jar
jersey-client-1.9.jar
jersey-core-1.9.jar
jetty-util-6.1.26.jar
jsp-api-2.1.jar
jsr305-3.0.0.jar
leveldbjni-all-1.8.jar
log4j-1.2.17.jar
netty-3.7.0.Final.jar
netty-all-4.0.23.Final.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
servlet-api-2.5.jar
slf4j-api-1.7.10.jar
slf4j-log4j12-1.7.10.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xercesImpl-2.9.1.jar
xml-apis-1.3.04.jar
```

```
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.6.jar
```

A.1.2 HDFS 2.6.0

```
activation-1.1.jar
apacheds-i18n-2.0.0-M15.jar
apacheds-kerberos-codec-2.0.0-M15.jar
api-asn1-api-1.0.0-M20.jar
api-util-1.0.0-M20.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.1.3.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
curator-client-2.6.0.jar
curator-framework-2.6.0.jar
curator-recipes-2.6.0.jar
gson-2.2.4.jar
guava-11.0.2.jar
hadoop-annotations-2.6.0.jar
hadoop-auth-2.6.0.jar
hadoop-client-2.6.0.jar
hadoop-common-2.6.0.jar
hadoop-hdfs-2.6.0.jar
hadoop-mapreduce-client-app-2.6.0.jar
hadoop-mapreduce-client-common-2.6.0.jar
hadoop-mapreduce-client-core-2.6.0.jar
hadoop-mapreduce-client-jobclient-2.6.0.jar
hadoop-mapreduce-client-shuffle-2.6.0.jar
hadoop-yarn-api-2.6.0.jar
hadoop-yarn-client-2.6.0.jar
hadoop-yarn-common-2.6.0.jar
hadoop-yarn-server-common-2.6.0.jar
htrace-core-3.0.4.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.9.13.jar
jackson-jaxrs-1.9.13.jar
jackson-mapper-asl-1.9.13.jar
jackson-xc-1.9.13.jar
jaxb-api-2.2.2.jar
jersey-client-1.9.jar
jersey-core-1.9.jar
jetty-util-6.1.26.jar
jsr305-1.3.9.jar
leveldbjni-all-1.8.jar
log4j-1.2.17.jar
netty-3.6.2.Final.jar
```

```
paranamer-2.3.jar
protobuf-java-2.5.0.jar
servlet-api-2.5.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xercesImpl-2.9.1.jar
xml-apis-1.3.04.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.6.jar
```

A.1.3 HDFS 2.5.2

HDFS 2.5.2 (HDFS 2.5.1 and 2.5.0 are effectively the same, simply substitute 2.5.1 or 2.5.0 on the libraries versioned as 2.5.2)

```
activation-1.1.jar
apacheds-i18n-2.0.0-M15.jar
apacheds-kerberos-codec-2.0.0-M15.jar
api-asn1-api-1.0.0-M20.jar
api-util-1.0.0-M20.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.1.3.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
guava-11.0.2.jar
hadoop-annotations-2.5.2.jar
adoop-auth-2.5.2.jar
hadoop-client-2.5.2.jar
hadoop-common-2.5.2.jar
hadoop-hdfs-2.5.2.jar
hadoop-mapreduce-client-app-2.5.2.jar
hadoop-mapreduce-client-common-2.5.2.jar
hadoop-mapreduce-client-core-2.5.2.jar
hadoop-mapreduce-client-jobclient-2.5.2.jar
hadoop-mapreduce-client-shuffle-2.5.2.jar
hadoop-yarn-api-2.5.2.jar
hadoop-yarn-client-2.5.2.jar
hadoop-yarn-common-2.5.2.jar
hadoop-yarn-server-common-2.5.2.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.9.13.jar
jackson-jaxrs-1.9.13.jar
jackson-mapper-asl-1.9.13.jar
jackson-xc-1.9.13.jar
jaxb-api-2.2.2.jar
jersey-client-1.9.jar
```

```
jersey-core-1.9.jar
jetty-util-6.1.26.jar
jsr305-1.3.9.jar
leveldbjni-all-1.8.jar
log4j-1.2.17.jar
netty-3.6.2.Final.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
servlet-api-2.5.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.6.jar
```

A.1.4 HDFS 2.4.1

HDFS 2.4.1 (HDFS 2.4.0 is effectively the same, simply substitute 2.4.0 on the libraries versioned as 2.4.1)

```
activation-1.1.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.1.3.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
guava-11.0.2.jar
hadoop-annotations-2.4.1.jar
hadoop-auth-2.4.1.jar
hadoop-client-2.4.1.jar
hadoop-hdfs-2.4.1.jar
hadoop-mapreduce-client-app-2.4.1.jar
hadoop-mapreduce-client-common-2.4.1.jar
hadoop-mapreduce-client-core-2.4.1.jar
hadoop-mapreduce-client-jobclient-2.4.1.jar
hadoop-mapreduce-client-shuffle-2.4.1.jar
hadoop-yarn-api-2.4.1.jar
hadoop-yarn-client-2.4.1.jar
hadoop-yarn-common-2.4.1.jar
hadoop-yarn-server-common-2.4.1.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jaxb-api-2.2.2.jar
jersey-client-1.9.jar
jersey-core-1.9.jar
jetty-util-6.1.26.jar
jsr305-1.3.9.jar
```

```
log4j-1.2.17.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
servlet-api-2.5.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.5.jar
hadoop-common-2.4.1.jar
```

A.1.5 HDFS 2.3.0

```
activation-1.1.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.1.3.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
guava-11.0.2.jar
hadoop-annotations-2.3.0.jar
hadoop-auth-2.3.0.jar
hadoop-client-2.3.0.jar
hadoop-common-2.3.0.jar
hadoop-hdfs-2.3.0.jar
hadoop-mapreduce-client-app-2.3.0.jar
hadoop-mapreduce-client-common-2.3.0.jar
hadoop-mapreduce-client-core-2.3.0.jar
hadoop-mapreduce-client-jobclient-2.3.0.jar
hadoop-mapreduce-client-shuffle-2.3.0.jar
hadoop-yarn-api-2.3.0.jar
hadoop-yarn-client-2.3.0.jar
hadoop-yarn-common-2.3.0.jar
hadoop-yarn-server-common-2.3.0.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jaxb-api-2.2.2.jar
jersey-core-1.9.jar
jetty-util-6.1.26.jar
jsr305-1.3.9.jar
log4j-1.2.17.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
servlet-api-2.5.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
snappy-java-1.0.4.1.jar
```

```
stax-api-1.0-2.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.5.jar
```

A.1.6 HDFS 2.2.0

```
activation-1.1.jar
aopalliance-1.0.jar
asm-3.1.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-httpclient-3.1.jar
commons-io-2.1.jar
commons-lang-2.5.jar
commons-logging-1.1.1.jar
commons-math-2.1.jar
commons-net-3.1.jar
gmbal-api-only-3.0.0-b023.jar
grizzly-framework-2.1.2.jar
grizzly-http-2.1.2.jar
grizzly-http-server-2.1.2.jar
grizzly-http-servlet-2.1.2.jar
grizzly-rcm-2.1.2.jar
guava-11.0.2.jar
guice-3.0.jar
hadoop-annotations-2.2.0.jar
hadoop-auth-2.2.0.jar
hadoop-client-2.2.0.jar
hadoop-common-2.2.0.jar
hadoop-hdfs-2.2.0.jar
hadoop-mapreduce-client-app-2.2.0.jar
hadoop-mapreduce-client-common-2.2.0.jar
hadoop-mapreduce-client-core-2.2.0.jar
hadoop-mapreduce-client-jobclient-2.2.0.jar
hadoop-mapreduce-client-shuffle-2.2.0.jar
hadoop-yarn-api-2.2.0.jar
hadoop-yarn-client-2.2.0.jar
hadoop-yarn-common-2.2.0.jar
hadoop-yarn-server-common-2.2.0.jar
jackson-core-asl-1.8.8.jar
jackson-jaxrs-1.8.3.jar
jackson-mapper-asl-1.8.8.jar
jackson-xc-1.8.3.jar
javax.inject-1.jar
javax.servlet-3.1.jar
javax.servlet-api-3.0.1.jar
jaxb-api-2.2.2.jar
jaxb-impl-2.2.3-1.jar
jersey-client-1.9.jar
jersey-core-1.9.jar
jersey-grizzly2-1.9.jar
jersey-guice-1.9.jar
jersey-json-1.9.jar
```

```
jersey-server-1.9.jar
jersey-test-framework-core-1.9.jar
jersey-test-framework-grizzly2-1.9.jar
jettison-1.1.jar
jetty-util-6.1.26.jar
jsr305-1.3.9.jar
log4j-1.2.17.jar
management-api-3.0.0-b012.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
snappy-java-1.0.4.1.jar
stax-api-1.0.1.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.5.jar
```

B

HBase Handler Client Dependencies

This appendix lists the HBase client dependencies for Apache HBase. The `hbase-client-x.x.x.jar` is not distributed with Apache HBase nor is it mandatory to be in the classpath. The `hbase-client-x.x.x.jar` is an empty maven project with the purpose of aggregating all of the HBase client dependencies.

- **Maven groupId**: `org.apache.hbase`
- **Maven atifactId**: `hbase-client`
- **Maven version**: the HBase version numbers listed for each section

B.1 HBase Client Dependencies

This section lists the Hadoop client dependencies for each HBase version.

B.1.1 HBase 1.1.1

HBase 1.1.1 (HBase 1.1.0.1 is effectively the same, simply substitute 1.1.0.1 on the libraries versioned as 1.1.1)

```
activation-1.1.jar
apacheds-i18n-2.0.0-M15.jar
apacheds-kerberos-codec-2.0.0-M15.jar
api-asn1-api-1.0.0-M20.jar
api-util-1.0.0-M20.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.9.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-el-1.0.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
commons-logging-1.2.jar
```

```
commons-math3-3.1.1.jar
commons-net-3.1.jar
findbugs-annotations-1.3.9-1.jar
guava-12.0.1.jar
hadoop-annotations-2.5.1.jar
hadoop-auth-2.5.1.jar
hadoop-common-2.5.1.jar
hadoop-mapreduce-client-core-2.5.1.jar
hadoop-yarn-api-2.5.1.jar
hadoop-yarn-common-2.5.1.jar
hamcrest-core-1.3.jar
hbase-annotations-1.1.1.jar
hbase-client-1.1.1.jar
hbase-common-1.1.1.jar
hbase-protocol-1.1.1.jar
htrace-core-3.1.0-incubating.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.9.13.jar
jackson-mapper-asl-1.9.13.jar
jaxb-api-2.2.2.jar
jcodings-1.0.8.jar
jdk.tools-1.7.jar
jetty-util-6.1.26.jar
joni-2.1.2.jar
jsch-0.1.42.jar
jsr305-1.3.9.jar
junit-4.11.jar
log4j-1.2.17.jar
netty-3.6.2.Final.jar
netty-all-4.0.23.Final.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.6.jar
```

B.1.2 HBase 1.0.1.1

```
activation-1.1.jar
apacheds-i18n-2.0.0-M15.jar
apacheds-kerberos-codec-2.0.0-M15.jar
api-asn1-api-1.0.0-M20.jar
api-util-1.0.0-M20.jar
avro-1.7.4.jar
commons-beanutils-1.7.0.jar
commons-beanutils-core-1.8.0.jar
commons-cli-1.2.jar
commons-codec-1.9.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-configuration-1.6.jar
commons-digester-1.8.jar
commons-el-1.0.jar
commons-httpclient-3.1.jar
commons-io-2.4.jar
commons-lang-2.6.jar
```

```
commons-logging-1.2.jar
commons-math3-3.1.1.jar
commons-net-3.1.jar
findbugs-annotations-1.3.9-1.jar
guava-12.0.1.jar
hadoop-annotations-2.5.1.jar
hadoop-auth-2.5.1.jar
hadoop-common-2.5.1.jar
hadoop-mapreduce-client-core-2.5.1.jar
hadoop-yarn-api-2.5.1.jar
hadoop-yarn-common-2.5.1.jar
hamcrest-core-1.3.jar
hbase-annotations-1.0.1.1.jar
hbase-client-1.0.1.1.jar
hbase-common-1.0.1.1.jar
hbase-protocol-1.0.1.1.jar
htrace-core-3.1.0-incubating.jar
httpclient-4.2.5.jar
httpcore-4.2.4.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jaxb-api-2.2.2.jar
jcodings-1.0.8.jar
jdk.tools-1.7.jar
jetty-util-6.1.26.jar
joni-2.1.2.jar
jsch-0.1.42.jar
jsr305-1.3.9.jar
junit-4.11.jar
log4j-1.2.17.jar
netty-3.6.2.Final.jar
netty-all-4.0.23.Final.jar
paranamer-2.3.jar
protobuf-java-2.5.0.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
snappy-java-1.0.4.1.jar
stax-api-1.0-2.jar
xmlenc-0.52.jar
xz-1.0.jar
zookeeper-3.4.6.jar
```

C

Flume Handler Client Dependencies

This appendix lists the Flume client dependencies for Apache Flume.

Maven groupId: `org.apache.flume`

Maven atifactId: `hadoop-ng-skd`

Maven version: the Flume version numbers listed for each section

C.1 Flume Client Dependencies

This section lists the Flume client dependencies for each Flume version.

- Flume 1.6.0
- Flume 1.5.2
- "Flume 1.4.0"

C.1.1 Flume 1.6.0

```
avro-1.7.4.jar
avro-ipc-1.7.4.jar
commons-codec-1.4.jar
commons-collections-3.2.1.jar
commons-compress-1.4.1.jar
commons-lang-2.5.jar
commons-logging-1.1.1.jar
flume-ng-sdk-1.6.0.jar
httpclient-4.1.3.jar
httpcore-4.1.3.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jetty-6.1.26.jar
jetty-util-6.1.26.jar
libthrift-0.9.0.jar
netty-3.5.12.Final.jar
paranamer-2.3.jar
slf4j-api-1.6.4.jar
snappy-java-1.0.4.1.jar
velocity-1.7.jar
xz-1.0.jar
```

C.1.2 Flume 1.5.2

```
avro-1.7.3.jar
avro-ipc-1.7.3.jar
commons-codec-1.3.jar
commons-collections-3.2.1.jar
commons-lang-2.5.jar
commons-logging-1.1.1.jar
flume-ng-sdk-1.5.2.jar
httpclient-4.0.1.jar
httpcore-4.0.1.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jetty-6.1.26.jar
jetty-util-6.1.26.jar
libthrift-0.7.0.jar
netty-3.5.12.Final.jar
paranamer-2.3.jar
slf4j-api-1.6.4.jar
snappy-java-1.0.4.1.jar
velocity-1.7.jar
```

C.1.3 Flume 1.4.0

```
avro-1.7.3.jar
avro-ipc-1.7.3.jar
commons-codec-1.3.jar
commons-collections-3.2.1.jar
commons-lang-2.5.jar
commons-logging-1.1.1.jar
flume-ng-sdk-1.4.0.jar
httpclient-4.0.1.jar
httpcore-4.0.1.jar
jackson-core-asl-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jetty-6.1.26.jar
jetty-util-6.1.26.jar
libthrift-0.7.0.jar
netty-3.4.0.Final.jar
paranamer-2.3.jar
slf4j-api-1.6.4.jar
snappy-java-1.0.4.1.jar
velocity-1.7.jar
```

D

Kafka Handler Client Dependencies

This appendix lists the Kafka client dependencies for Apache Kafka.

Maven groupId: `org.apache.kafka`

Maven atifactId: `kafka-clients`

Maven version: the Kafka version numbers listed for each section

D.1 Kafka Client Dependencies

This section lists the Kafka client dependencies for each Kafka version.

D.1.1 Kafka 0.9.0.1

```
kafka-clients-0.9.0.1.jar
lz4-1.2.0.jar
slf4j-api-1.7.6.jar
snappy-java-1.1.1.7.jar
```

D.1.2 Kafka 0.10.0.1

```
kafka-clients-0.10.0.1.jar
lz4-1.3.0.jar
slf4j-api-1.7.21.jar
snappy-java-1.1.2.6.jar
```

E

Cassandra Handler Client Dependencies

This appendix lists the Cassandra client dependencies for Apache Cassandra.

Maven groupId: org.apache.cassandra

Maven atifactId: cassandra-clients

Maven version: the Cassandra version numbers listed for each section

E.1 Cassandra Datastax Java Driver 3.1.0

```
cassandra-driver-core-3.1.0.jar
cassandra-driver-extras-3.1.0.jar
cassandra-driver-mapping-3.1.0.jar
asm-5.0.3.jar
asm-analysis-5.0.3.jar
asm-commons-5.0.3.jar
asm-tree-5.0.3.jar
asm-util-5.0.3.jar
guava-16.0.1.jar
HdrHistogram-2.1.9.jar
jackson-annotations-2.6.0.jar
jackson-core-2.6.3.jar
jackson-databind-2.6.3.jar
javax.json-api-1.0.jar
jffi-1.2.10.jar
jffi-1.2.10-native.jar
jnr-constants-0.9.0.jar
jnr-ffi-2.0.7.jar
jnr-posix-3.0.27.jar
jnr-x86asm-1.0.2.jar
joda-time-2.9.1.jar
lz4-1.3.0.jar
metrics-core-3.1.2.jar
netty-buffer-4.0.37.Final.jar
netty-codec-4.0.37.Final.jar
netty-common-4.0.37.Final.jar
netty-handler-4.0.37.Final.jar
netty-transport-4.0.37.Final.jar
slf4j-api-1.7.7.jar
snappy-java-1.1.2.6.jar
```

F

MongoDB Handler Client Dependencies

Oracle GoldenGate recommends that you use the 3.2.2 MongoDB Java Driver integration with MongoDB using `mongo-java-driver-3.2.2.jar`. You can download this driver from:

http://mongodb.github.io/mongo-java-driver/

MongoDB Java Driver 3.2.2

F.1 MongoDB Java Driver 3.2.2

The MongoDB Handler uses the native Java driver release 3.2.2 APIs (`mongo-java-driver-3.2.2.jar`). The handler should be compatible with the 3.X.X driver. You must include the path to the MongoDB java driver in the `gg.classpath` property. The Java driver can be automatically downloaded from maven central repository by adding the following in the `pom.xml` file as in the following example:

```
<!-- https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver -->
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.2.2</version>
</dependency>
```