Connecting Java Applications to Big Data Targets Using BDGlue

User Guide

Oracle Data Integration Solutions

Version 1.2.0

Contents

Change Log

Version	Date	Comments
1.0	8/3/2015	Initial Release
1.0.1	9/24/2015	Added code to support replacing newline characters in string fields.
	10/07/2015	Added support for before images of data in JSON encoding.
	10/23/2015	Fixed issue with negative hash value in ParallelPublisher.
	10/27/2015	Changes to make Kafka topics and message keys customizable.
	12/07/2015	Added prompt for password in schemadef.
	12/09/2015	Added support for including table name in encoded data.
	12/10/2015	Added support for pass through of properties from bdglue.properties to Kafka.
	12/16/2015	Added code to better deal with compressed records and null column values.
	01/13/2016	Added code to ignore updates where nothing changed.
	01/22/2016	Added support for the Kafka Schema Registry.
	01/26/2016	Added calls to trim() when retrieving properties.
	03/01/2016	Changes to support GGforBD 12.2 and schema change events.
	03/10/2016	Automatic generation of Avro schemas based on schema change events.
	03/23/2016	Support specifying numeric output type in Avro schema generation.
	04/07/2016	Added default null for dynamic avro schemas. Fixed issue with avro table name valid chars.
	04/09/2016	Reworked schema registry encoding to pass Avro GenericRecord to serializers.
1.1.0	4/21/2016	Refactored code for publishers to potentially allow for more selective building at some point. Added new KafkaRegistryPublisher that supports registering avro schemas with the schema registry.
	5/15/2016	Added Cassandra Support
	5/27/2016	Improved shutdown logic to clean up more quickly and not block on take() calls.
	6/1/2016	Changed queue “take” logic to use drainTo() to hopefully reduce latch waits
	6/8/2016	Changed dynamic avro schema generation to make all columns nullable.
1.2.0	6/8/2016	Initial release to GitHub.

Introduction

"Big Data Glue" (a.k.a. BDGlue) was developed as a general purpose library for delivering data from Java applications into various Big Data targets in a number of different data formats. The idea was to create a "one stop shop" of sorts to facilitate easy exploration of different technologies to help users identify what might be the most appropriate approach in any particular case. The overarching goal was to allow this experimentation to occur without the user having to write any Big Data-specific code. Big Data targets include Flume, Kafka, HDFS, Hive, HBase, Oracle NoSQL, Cassandra, and others.

Hadoop and other Big Data technologies are by their very natures constantly evolving and infinitely configurable. It is unlikely that BDGlue will exactly meet the requirements of a user's intended production architecture, but it will hopefully provide a good starting point for many and at a minimum should prove sufficient for early point proving exercises.

The code was developed using Oracle's "Big Data Lite" virtual machine[1] to ensure compatibility with Oracle's engineered Big Data solution, the Big Data Appliance (BDA)[2]. If you are not familiar with the BDA, it is an extremely well thought-out and cost effective solution that is certainly worthy of consideration as you start to scale from the lab and into production. However, BDGlue does not leverage any capabilities specific to BDA and will work equally well with any standard Hadoop distribution, including those from Cloudera, Hortonworks, MapR, and Apache.

Source Code

The source code for BDGlue is freely available and may be found on GitHub at:

http://github.com/bdglue/bdglue

Licensing

BDGlue is developed as open source and released under the **Apache License, Version 2.0**. Most external components that it interfaces with are licensed in this fashion as well, with a few exceptions which are called out explicitly in the LICENSE and NOTICE files that accompany the source code. In those situations, their corresponding licenses have been deemed compatible with Apache 2.0. A copy of the LICENSE and NOTICE files are also included at the end of this document.

There is no license fee associated with BDGlue itself. It is up to the user to determine if source and/or target environments are subject to license fees from their vendors. For example, the GoldenGate Adapter for Java must be licensed if you make use of the GoldenGate source as described at the end of this document, as well as the GoldenGate CDC capabilities in the source environment. In short, just because BDGlue provides an interface to a technology, it doesn't imply that access is inherently free.

[1] The Big Data Lite virtual machine may be downloaded from http://www.oracle.com/technetwork/database/bigdata-appliance/oracle-bigdatalite-2104726.html.

[2] For information on the Big Data Appliance, see https://www.oracle.com/engineered-systems/big-data-appliance/index.html

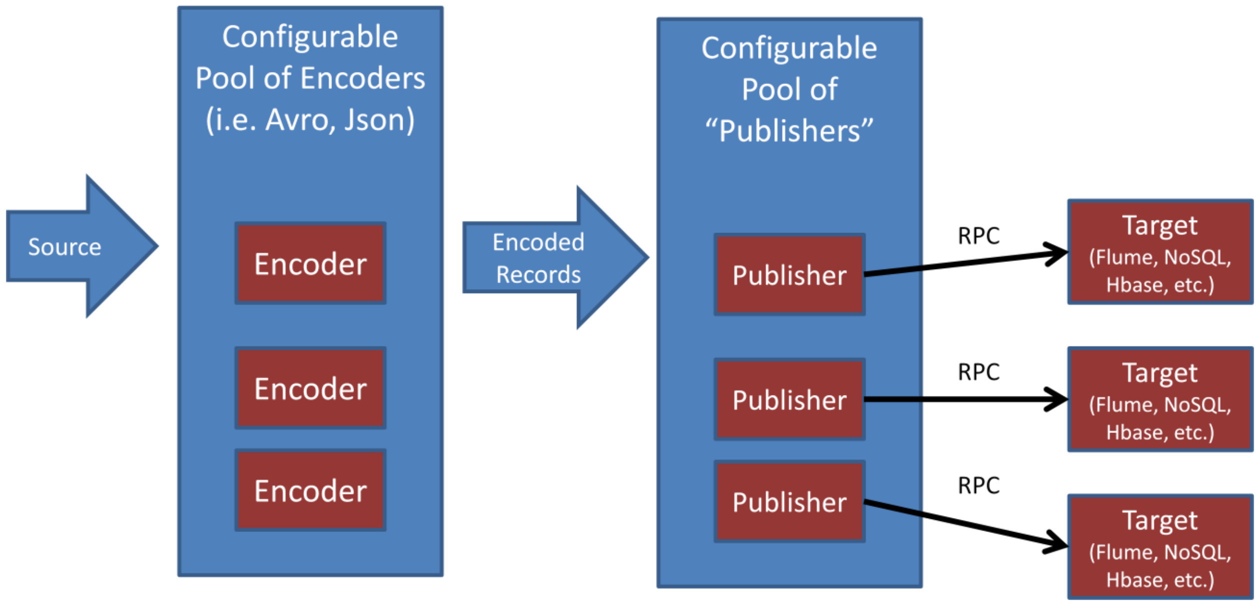
Disclaimer

While it is intended to be useful out of the box, this code is not formally supported by Oracle. The code is provided as is and as an example only. While quite functional, it is not warranted or supported as “production ready” and responsibility for making it so remains the customer’s responsibility. Source code is available so that customers may alter it as needed to meet their specific needs.

Architectural Approach

The BDGlue architecture is modular in its approach, based on the idea of sources, encoders, and publishers, with the goal of mitigating the impact of change as new capabilities are added. For example, “encoders” are independent of their upstream source and their downstream publisher so that new encoding formats can be implemented without requiring change elsewhere in the code.

The following diagram illustrates the high level structure of BDGlue.


Source
Configurable Pool of Encoders (i.e. Avro, Json)
Encoder
Encoder
Encoder
Encoded Records
Configurable Pool of “Publishers”
Publisher
Publisher
Publisher
RPC
RPC
RPC
Target (Flume, NoSQL, Hbase, etc.)
Target (Flume, NoSQL, Hbase, etc.)
Target (Flume, NoSQL, Hbase, etc.)

Encoding is the process of translating data received from the source into a particular format to facilitate downstream use. Publishing is the process of writing data to a target environment via RPC. You will note that BDGlue was designed to support two separate and distinct thread pools to provide scalability for the “encoding” and “publishing” processes, each of which can be somewhat time-consuming.

Note that while the process of encoding data is multi-threaded and essentially asynchronous, the encoded records are actually delivered to the publishers in the same order that they were received from the source. This is to ensure that data anomalies don’t get introduced as a result of a race condition that might arise if multiple changes to a particular record occur in rapid succession, but it will likely have a small impact on encoder throughput

In the same way, data is delivered to individual publishers based on a hash of a string value … either

- the “table” name, ensuring all records from a particular “table” will be processed in order by the same publisher; or
- based on the “key” associated with the record, in this case ensuring that records based on the same key value are processed in order by the same publisher.

The targets themselves are completely external to BDGlue. In most cases, they are accessed via an RPC connection (i.e. a socket opened on a specified port). A “target” might be a streaming technology such as Flume or Kafka, or it might be an actual big data repository such as HBase, Oracle NoSQL, Cassandra, etc. From Flume we can deliver encoded data at a very granular level to both HDFS files and Hive.

Last but not least in the BDGlue conversation has to do with sources. BDGlue was designed initially with the idea of delivering data sourced from a relational database, and leveraging Oracle GoldenGate in particular. We quickly came to realize that BDGlue had the potential of being more generally useful than that, so we made a deliberate effort to decouple the data sources specific to GoldenGate from the rest of BDGlue to the greatest degree possible. BDGlue looks at things from the perspective of table-like structures – essentially tables with a set of columns – but the reality is that any sort of data source could likely be mapped into them without requiring a lot of imagination or effort.

Installing BDGlue

For convenience, BDGlue can be obtained from GitHub in source form and can easily be compiled from there. The net result of the compilation process will be a bdglue-specific *.jar file, jar file dependencies needed to compile and execute, as well as documentation, example properties files, etc.

Note that BDGlue is configured to build with Maven, and a suitable pom.xml file is included for this purpose. For those unfamiliar with Maven, there is a traditional Makefile provided which invokes Maven under the covers. Being somewhat old school, while Maven is great for compiling everything and assembling the dependencies, we prefer calling Maven from *make* (*gmake* actually) as the install step is a bit more straightforward to comprehend as it copies all of the relevant build artifacts to a "deploy" directory.

Note in either case, you will need to set two environment variables:

```
# GGBD_HOME is the directory where GG for Big Data in installed.
# For example, if GG for Big Data is installed at /u01/ggbd12_2, then you would set

export GGBD_HOME=/u01/ggbd12_2
```

And

```
# GGBD_VERSION is an environment variable set to the version of the
# ggdbutil-VERSION.jar file found in the $GGBD_HOME/ggjava/resources/lib directory.
# For example, if the file is named ggdbutil-12.2.0.1.0.012.jar, then you would set

export GGBD_VERSION=12.2.0.1.0.012
```

Download.

```
# create a directory where you want to install the files
[ogg@bigdatalite ~]$ mkdir bdglue
[ogg@bigdatalite ~]$ cd bdglue
[ogg@bigdatalite ~]$ git clone https://github.com/bdglue/bdglue
[ogg@bigdatalite ~]$ export GGBD_HOME=/path/to/gg4bigdata
[ogg@bigdatalite ~]$ export GGBD_VERSION=12.2.0.1.0.12
```

Build with Make:

```
[ogg@bigdatalite ~]$ make
mvn package -Dggbd.VERSION=12.2.0.1.0.012 -Dggbd.HOME=/u01/ggbd12_2
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Building bdglue 1.2.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ bdglue ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] Copying 2 resources
< -- snip -- >
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.987 s
[INFO] Finished at: 2016-06-08T16:02:03-04:00
[INFO] Final Memory: 40M/314M
[INFO] ------------------------------------------------------------------------
mkdir –p ./deploy/lib/dependencies ./deploy/doc
cp ./target/bdglue*.jar ./deploy/lib
cp ./target/dependencies/*.jar ./deploy/lib/dependencies
cp -R ./target/apidocs ./deploy/doc
cp ./*.pdf ./deploy/doc
[ogg@bigdatalite ~]$
```

Build with Maven:

```
[ogg@bigdatalite ~]$
[ogg@bigdatalite ~]$ mvn package -Dggbd.VERSION=$GGBD_VERSION -Dggbd.HOME=$GGBD_HOME
[ogg@bigdatalite ~]$
```

CAUTION: Be sure that the versions of the java dependencies (i.e. Kafka, Avro, Cassandra, HBase, etc.) that BDGlue builds with are compatible with the version of those target solutions deployed in your environment. In many cases, the dependencies will be forward / backward compatible, but not always. If you have difficulties at run time, whether exceptions related to methods not being found, or unidentifiable failures, this could very well be the cause. You may need to alter the java dependencies in the pom.xml file (used for building), or install newer versions of the target solution in your environment.

Finally, note that if you are working directly with your Oracle sales team, you might be provided with a newer version of bdglue.jar than the one provided in this installation. If that is the case, you will want to **replace** the version of bdglue.jar in the installation directory with the version provided to you. You can confirm the version of bdglue.jar by typing the following from the command line:

```
[ogg@bigdatalite ~]$ java –jar bdglue*.jar
BDGlue Version: 1.2.0.0  Date: 2016/06/08 13:45
```

Encoders

There are a number of encoders that are inherently part of BDGlue: null, delimited text, Avro, and JSON. For the most part, these should prove to be sufficient for just about any use case, but BDGlue was designed to be extended with additional encoders if needed. This can be accomplished simply by implementing a Java interface. New Encoders can be developed and deployed without requiring changes to BDGlue itself. More information pertaining to creating new encoders can be found in ***Building a Custom Encoder*** in the "Developers Guide" section later in this document.

The "Null" Encoder

The "null" encoder is just what it sounds like … it actually does no encoding at all. It is designed to simply take the data that was provided by the source, encapsulate it with a little meta-data related to the work that needs to be done downstream, and then pass the data along to the publisher. Consequently, the null encoder is the most lightweight of the encoders and is intended for use against those targets that

- BDGlue will connect to directly (i.e. not via Flume, Kafka, etc.); and
- Require data to be applied via API at the field (or column) level rather than at the record level.

Targets for which the null encoder is appropriate include HBase, the Oracle NoSQL "table" API, Cassandra, etc.

To tell BDGlue to make use of the Null Encoder, simply specify the encoder in the *bdglue.properties* file as follows:

```
bdglue.encoder.class = com.oracle.bdglue.encoder.NullEncoder
```

The Delimited Text Encoder

The "delimited text encoder" is also just what it sounds like … it is designed to take the data that is passed in from the source and encode it into a data "delimited text" fashion.

Delimited text is the simplest and most straight forward way to transmit the data from BDGlue to a target. It is also likely the least useful. The column values are added to a buffer that will become the "body" of the data that is sent downstream by a publisher. Columns are added to the buffer in the order they are represented in the table metadata, with each column separated from the one preceding it in the buffer by a delimiter. By default, the delimiter is the default delimiter recognized by Hive, which is \001 (^A). That value can be overridden in the *bdglue.properties* file by specifying the *bdglue.encoder.delimiter* property.

Delimited text is fast and somewhat compact, but it contains no metadata regarding the structure of the data (i.e. the names of the columns). This requires downstream consumers of the data to know the structure when it comes time to make use of the data later. This could in theory be a challenge; particularly if the schema has evolved over time.

To tell BDGlue to make use of the Delimited Text Encoder, simply specify the encoder in the *bdglue.properties* file as follows:

```
bdglue.encoder.class = com.oracle.bdglue.encoder.DelimitedTextEncoder
```

The JSON Encoder

"JSON" is short for "JavaScript Object Notation". It is a lightweight data interchange format that uses human-readable text to transmit data comprised of attribute-value pairs. It is language-independent and has proven to be quite useful in many Big Data use cases. In the case of this encoder, the "attributes" are the column/field names, and the "values" are the actual data values associated with those names.

Here is an example of JSON-encoded data:

```
{"ID":"2871","NAME":"Dane Nash","GENDER":"Male","CITY":"Le Grand-Quevilly",
"PHONE":"(874) 373-6196","OLD_ID":"1","ZIP":"81558-771","CUST_DATE":"2014/04/13"}
```

Column/field names are ID, NAME, GENDER, CITY, and so on.

To tell BDGlue to make use of the JSON Encoder, simply specify the encoder in the *bdglue.properties* file as follows:

```
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
```

The Avro Encoder

This data encoding is a bit more advanced than the others. Avro is a data serialization format that supports rich data structures in a compact binary data format. It has proven to be quite useful, and is understood directly by Hive, Oracle NoSQL, and other targets. Avro also supports the notion of "schema evolution", albeit in a more limited sense than might be supported by a relational database.

Unlike JSON, which is text-based and self-describing, Avro data is actually transmitted downstream to recipients in a more compact binary format based on an "Avro schema" that describes the contents. Like JSON-formatted data, this data also has a clearly defined structure, but it is different in that the “schema” that describes the data must be made available to the recipient so that the data can be understood. Avro schemas are actually defined using JSON.

Here is an example of what an Avro schema file looks like. As mentioned, it is a JSON format that describes the columns and their data types. Notice the “union” entries that contain “null” and a data type. These indicate that those columns may be null. Note also the specification of default values: “null” for columns that may be null; -1 for the OLD_ID column which in this case may not be null; etc. Inclusion of the null column information and default values is optional and specified in the properties file. It is recommended that these always be enabled as the information assists the target repository (HDFS, Hive, NoSQL, etc.) in the schema evolution process.

```
 {
  "type" : "record",
  "name" : "CUST_INFO",
  "namespace" : "bdglue",
  "doc" : "SchemaDef",
  "fields" : [ {
    "name" : "ID",
    "type" : "int",
    "doc" : "keycol"
  }, {
    "name" : "NAME",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "GENDER",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "CITY",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "PHONE",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "OLD_ID",
    "type" : "int",
    "default" : -1
  }, {
    "name" : "ZIP",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "CUST_DATE",
    "type" : [ "null", "string" ],
    "default" : null
  } ]
}
```

For relational database sources, a utility "*SchemaDef*" is provided with BDGlue that will generate the Avro schema files that would correspond to a table from the table's metadata. *SchemaDef* will also generate meta-information in other formats as well. *SchemaDef* is described later in this document.

To tell BDGlue to make use of the Avro Encoder, simply specify the encoder in the bdglue.properties file as follows:

```
bdglue.encoder.class = com.oracle.bdglue.encoder.AvroEncoder
```

Publishers

A publisher is responsible for understanding how to interface with an external "target". Another way of saying that is that a publisher is specific to its intended target. A publisher takes the data and associated meta-data handed off from the encoder and delivers it to the target. As mentioned previously, publishers are part of a "pool", with each publisher having its own independent connection to the target, most typically via an RPC.

In some cases, the publisher will deliver the data received from the encoder "as is" to the target. It will hand off these encoded records without really understanding their contents, just knowing that it needs to pass them along. Examples of publishers where encoded data would likely be passed along as provided by the encoder without further interpretation include Flume, Kafka, the Oracle NoSQL KV API, etc.

The primary exception to this would be data passed along via the "null encoder". In this particular case, it is intended that the publisher process the data field-by-field as it writes to the target. Examples of publishers that would leverage data passed along from the "null encoder" include HBase, the Oracle NoSQL Table API, Cassandra, etc. In each of these cases, data is added to stored records on a field-by-field basis, so a pre-formatted record based on JSON, Avro, etc. are likely not appropriate.

We will explore one publisher, the "Console Publisher", in the next section. We'll look at how to configure BDGlue publishers to deliver to supported targets later in the document.

Finally, just as it was designed to support development of new "encoders", BDGlue was designed to be extended to support new publishers as well. Just as with Encoders, this is done by implementing a Java interface, and just as with new encoders this can be done without the need to make changes elsewhere in the code. More information pertaining to creating new publishers can be found in ***Building a Custom Publisher*** in the "Developers Guide" section found later in this document.

The Console Publisher

The first, and simplest, publisher we will cover is the "Console Publisher." It was developed to assist with certain troubleshooting processes, particularly in areas pertaining to ensuring that things are configured properly before we actually start trying to "publish" data to a target. The Console Publisher simply takes the records that are passed to it and writes them to standard out (the "console"). Because it is writing to what could very well be a display screen, configuring the JSON Encoder when using the Console Publisher is probably best … records are more easily readable.

Here is how you might configure BDGlue's properties file to use the Console Publisher.

```
# bdglue.properties to make use of the ConsolePublisher.
#

bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = false
bdglue.event.header-avropath = false

bdglue.publisher.class = com.oracle.bdglue.publisher.console.ConsolePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = table
```

BDGlue Targets

In this section, we'll explore delivery to a variety of target environments supported by BDGlue. We'll start specifically with using Flume to deliver files to HDFS. Flume is actually quite flexible and powerful. There are a number of other targets we can deliver to from Flume. We'll look at specific information for using Flume to deliver to those targets in their respective sections.

Connecting to Targets via the Flume Publisher

Apache Flume is a streaming mechanism that fits naturally with the BDGlue architecture. Flume provides a number of out-of-the-box benefits that align well with the BDGlue use case:

- It supports RPC connections from locations that are not physically part of the Hadoop cluster.
- It is modular and thus extremely flexible in terms of how data "streams" are configured.
- There are many out-of-the-box components that can be leveraged directly without need for modification or customization.
- In particular, Flume does an outstanding job with its HDFS file handling. If there is a need to stream data from outside of Hadoop into files in HDFS, there may be no better mechanism for doing this.
- It provides a pluggable architecture that allows custom components to be developed and deployed when needed.

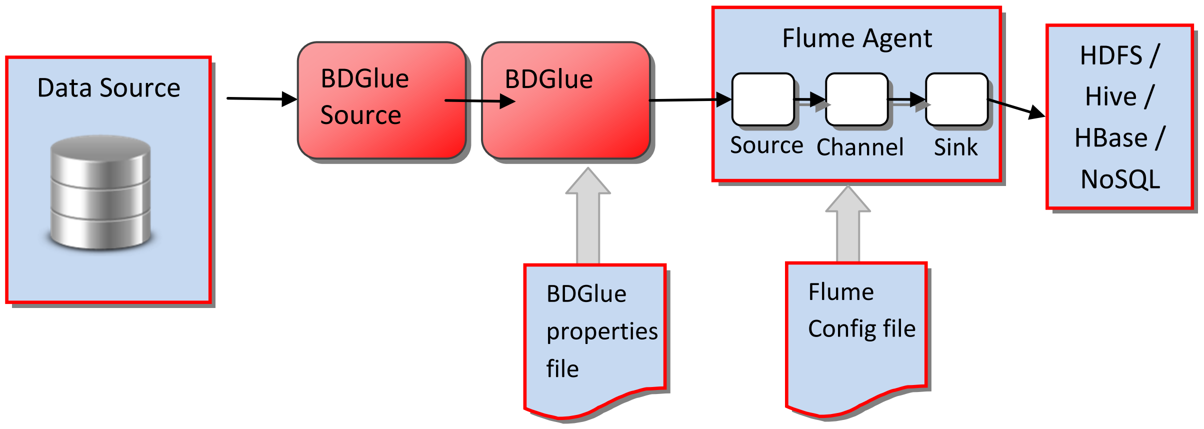


Figure 1: Basic architecture of the BDGlue Flume implementation

To really understand what is going on behind the scenes, it is important that the user have a good understanding of Flume and its various components.

A good reference on Flume is:

- *Apache Flume: Distributed Log Collection for Hadoop*, by Steve Hoffman (©2013 Packt Publishing)

While the book is focused predominantly on streaming data collected from log files, there is a lot of excellent information on configuring Flume to take advantage of the flexibility it offers. Despite the public perception, there is much more that Flume brings to the table than the scraping of log files.

An excellent introductory reference on Hadoop and Big Data in general is:

- *Hadoop: The Definitive Guide (Fourth Edition),* by Tom White (©2015 O'Reilly media)

This book not only provides information on Hadoop technologies such as HDFS, Hive, and HBase, but it also provides some good detail on Avro serialization, a technology that proved to be quite useful in practice in any number of customer environments.

Configuring Flume

As mentioned previously, Flume is incredibly flexible in terms of how it can be configured. Topologies can be arbitrarily complex, supporting fan-in, fan-out, etc. Data streams can be intercepted, modified in flight, and rerouted. There really is no end to what might be configured.

There are two basic topologies for Flume that we feel will be most commonly useful in BDGlue use cases:

- A single stream that will handle multiple tables via an agent that consists of a single source-channel-sink combination. This is easiest to implement, and we think will be most common.
- A multiplexing stream where a single source fans out into a separate channel and sink for each table being processed. This is more complex to implement, but might be an approach when there are a few high volume tables. There might be a separate channel and sink for each of the high volume tables, and then another "catch all" that handles the rest.

The Flume RPC Client

The BDGlue Flume publisher is implemented to support both Avro and Thrift for RPC communication with Flume. It is possible to switch between the two via a property in the *bdglue.properties* file. Most of the testing of BDGlue was done using Avro RPCs, and all examples in this user guide leverage Avro for RPC communication. If you wish to use Thrift instead, you will need to configure

- bdglue.flume.rpc.type = thrift-rpc in the *bdglue.properties* file; and
- bdglue.sources.<source>.type = thrift in the Flume configuration file.

Do not confuse Avro RPC with Avro Serialization, which we also make good use of in BDGlue. While they share a common portion of their name, the two are essentially independent of one another. For the examples in this user guide, we will configure

- bdglue.flume.rpc.type = avro-rpc in the *bdglue.properties* file.
- The bdglue.sources.<source>.type = avro in the Flume configuration file.

Flume Events

Note that data moves through a Flume Agent as a series of "events", where in the case of GoldenGate each event represents a captured database operation, or source record otherwise. We'll just generally refer to the data as "source data" or "source record" going forward. The body of the event contains an encoding of the contents of the source record. Several encodings are supported: Avro binary, JSON, and delimited text. All are configurable via the *bdglue.properties* file.

In addition to the body, each Flume event has a header that contains some meta-information about the event. For BDGlue, the header will always contain the table name. Depending on other options that are configured, additional meta-information will also be included.

Standard Flume Agent Configuration

As mentioned above, the most typical Flume agent configuration will be relatively simple: a single source-channel-sink combination that writes data to specific destinations for each table that is being processed.

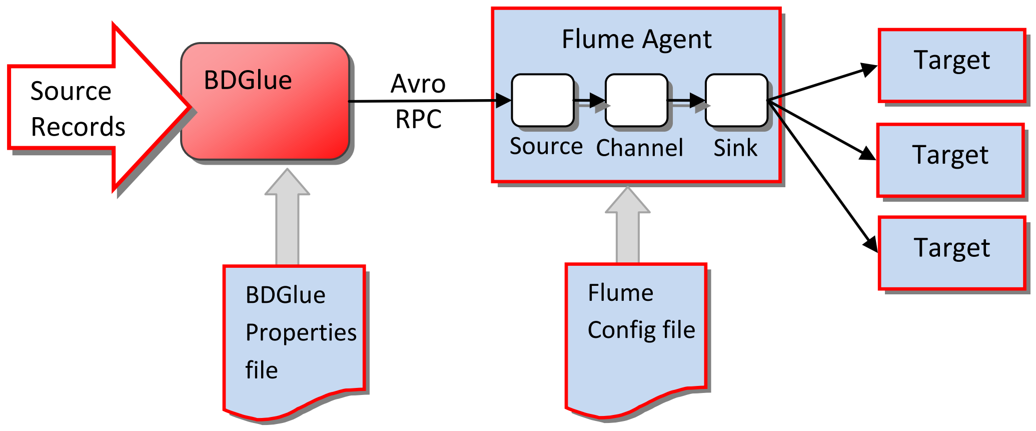


Figure 2: Typical Flume agent configuration

The various "Targets" might be files in HDFS, or perhaps Hive or HBase tables, based on how the properties and configuration files are set up. Our examples in subsequent sections will be based on this configuration and we'll look at the details of the *bdglue.properties* and Flume configuration files at that time.

Multiplexing Flume Agent Configuration

Before that, however, we'll take a quick look at one other configuration that might prove useful. This configuration is one where a single Flume "source" multiplexes data across multiple channels based on table name, and each channel has its own sink to write the data into Hadoop.

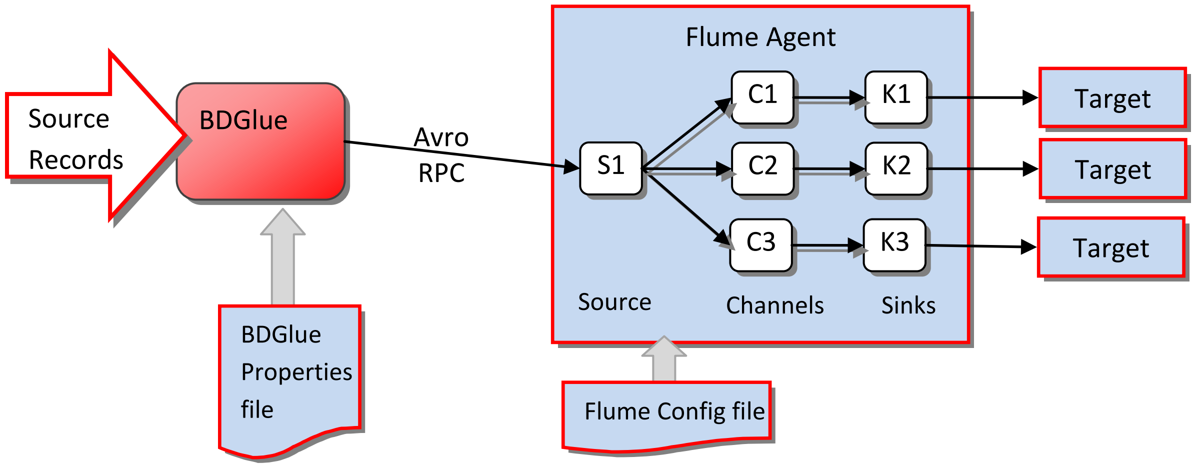


Figure 3: Multiplexing Flume Agent

To configure in this fashion, you'll need to specify a separate Flume configuration for each channel and sink. If there are a lot of tables that you want to process individually, this could get fairly complicated in a hurry. The following will give you an idea of what such a configuration file might look like. Note that this example is not complete, but it will give you an idea of what might be required to configure the example above.

```
# list the sources, channels, and sinks for the agent
bdglue.sources = s1
bdglue.channels = c1 c2 c3
bdglue.sinks = k1 k2 k3
# Map the channels to the source. One channel per table being captured.
bdglue.sources.s1.channels = c1 c2 c3
# Set the properties for the source
bdglue.sources.s1.type = avro
bdglue.sources.s1.bind = localhost
bdglue.sources.s1.port = 41414
bdglue.sources.s1.selector.type = multiplexing
bdglue.sources.s1.selector.header = table
bdglue.sources.s1.selector.mapping.default = c1
bdglue.sources.s1.selector.mapping.<table-1> = c2
bdglue.sources.s1.selector.mapping.<table-2> = c3
# Set the properties for the channels
# c1 is the default ... it will handle unspecified tables.
bdglue.channels.c1.type = memory
bdglue.channels.c1.capacity = 1000
bdglue.channels.c1.transactionCapacity = 100
bdglue.channels.c2.type = memory
bdglue.channels.c2.capacity = 1000
bdglue.channels.c2.transactionCapacity = 100
bdglue.channels.c3.type = memory
bdglue.channels.c3.capacity = 1000
bdglue.channels.c3.transactionCapacity = 100
# Set the properties for the sinks
# map the sinks to the channels
bdglue.sinks.k1.channel = c1
bdglue.sinks.k2.channel = c2
bdglue.sinks.k3.channel = c3
# k1 is the default. Logs instead of writes.
bdglue.sinks.k1.type = logger
bdglue.sinks.k2.type = hdfs
bdglue.sinks.k2.serializer = avro_event
bdglue.sinks.k2.serializer.compressionCodec = gzip
bdglue.sinks.k2.hdfs.path = hdfs://bigdatalite.localdomain/flume/gg-data/%{table}
bdglue.sinks.k2.hdfs.fileType = DataStream
# avro files must end in .avro to work in an Avro MapReduce job
bdglue.sinks.k2.hdfs.filePrefix = bdglue
bdglue.sinks.k2.hdfs.fileSuffix = .avro
bdglue.sinks.k2.hdfs.inUsePrefix = _
bdglue.sinks.k2.hdfs.inUseSuffix =
bdglue.sinks.k3.type = hdfs
bdglue.sinks.k3.serializer = avro_event
bdglue.sinks.k3.serializer.compressionCodec = gzip
bdglue.sinks.k3.hdfs.path = hdfs://bigdatalite.localdomain/flume/gg-data/%{table}
bdglue.sinks.k3.hdfs.fileType = DataStream
# avro files must end in .avro to work in an Avro MapReduce job
bdglue.sinks.k3.hdfs.filePrefix = bdglue
bdglue.sinks.k3.hdfs.fileSuffix = .avro
bdglue.sinks.k3.hdfs.inUsePrefix = _
bdglue.sinks.k3.hdfs.inUseSuffix =
```

In the example above, note that the configuration for channel/sink c1/s1 is configured as a "default" (i.e. catch all) channel. In this case, we are logging information as an exception. That channel could also be configured to process rather than log those tables, while still allowing "special" handling of channel/sinks c2/k2 and c3/k3.

Running Flume

Once it is actually time to start the flume agent, you'll do so by executing a statement similar to the following example.

```
flume-ng agent --conf conf --conf-file bdglue.conf --name bdglue
                  --classpath /path/to/lib/bdglue.jar
                   -Dflume.root.logger=info,console
```

Several things to note above:

- *bdglue.conf* is the name of your configuration file for Flume. It can have any name you wish.
- --*name bdglue*: "bdglue" is the name of your agent. It must exactly match the name of your agent in the configuration file. You'll note that each line in the example configuration file above begins with "bdglue". Your agent can have any name you wish, but this name must match.
- --*classpath *.jar* gives the name of your jar file that contains any custom source-channel-sink code you may have developed. It is not required otherwise. In the case of BDGlue, it will only be needed when delivering to HBase and Oracle NoSQL as custom sink logic was developed for those targets.

Using Flume to Deliver Data into HDFS files

Flume has actually proven to be an excellent way to deliver data into files stored within HDFS. This may seem counterintuitive in some ways, but unless you have an entire file ready to go at once, the idea of streaming data into those files actually makes a lot of sense, particularly if you might have the need to write to multiple files simultaneously. The Flume HDFS "sink" can support thousands of open files simultaneously (say, one for each table being delivered via transactional CDC – change data capture), and provides excellent control over directory structure, and when to roll to a new file based on size, number of records, and/or time.

BDGlue supports delivery to HDFS via Flume in several different encoded file formats:

- Delimited text. This is just as it sounds, with column values separated by a delimiter. By default, that delimiter is \001 (^A), which is the default delimiter for Hive, but that can be overridden in the *bdglue.properties* file.
- JSON-formatted text. This is basically a "key-value" description of the data where the key is the name of the column, and the value is the value stored within that column.

- Avro binary schema.

Each of these formats was described previously in the section on Encoders.

Delimited Text

```
# bdglue.properties
#
bdglue.encoder.class = com.oracle.bdglue.encoder.DelimitedTextEncoder
bdglue.encoder.threads = 2
bdglue.encoder.delimiter = 001
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

Here is the Flume configuration file:

```
# list the sources, channels, and sinks for the agent
bdglue.sources = s1
bdglue.channels = c1
bdglue.sinks = k1

# Map the channels to the source.
bdglue.sources.s1.channels = c1

# Set the properties for the source
bdglue.sources.s1.type = avro
bdglue.sources.s1.bind = localhost
bdglue.sources.s1.port = 41414
bdglue.sources.s1.selector.type = replicating

# Set the properties for the channels
bdglue.channels.c1.type = memory

# make capacity and transactionCapacity much larger
# (i.e. 10x or more) for production use
bdglue.channels.c1.capacity = 1000
bdglue.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
bdglue.sinks.k1.channel = c1

bdglue.sinks.k1.type = hdfs
bdglue.sinks.k1.serializer = text

# each table written to separate directory named ‘tablename’
bdglue.sinks.k1.hdfs.path = hdfs://bigdatalite.localdomain/user/flume/gg-
data/%{table}
bdglue.sinks.k1.hdfs.fileType = DataStream
bdglue.sinks.k1.hdfs.filePrefix = bdglue
bdglue.sinks.k1.hdfs.fileSuffix = .txt
bdglue.sinks.k1.hdfs.inUsePrefix = _
bdglue.sinks.k1.hdfs.inUseSuffix =

# number of records the sink will read per transaction.
# Higher numbers may yield better performance.
bdglue.sinks.k1.hdfs.batchSize = 10
# the size of the files in bytes.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollSize = 1048576
# roll to a new file after N records.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollCount = 100
# roll to a new file after N seconds.  0=disable
bdglue.sinks.k1.hdfs.rollInterval = 30
```

JSON Encoding

Under the covers, when writing to HDFS JSON-encoded data is handled in the same way as delimited text is handled. The fundamental difference is that the data is formatted in such a way that the column names are included along with their contents.

The *bdglue.properties* file needed for this might look something like the following.

```
# bdglue.properties
#
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

The corresponding Flume configuration file would look the same as it did for delimited text as the data is handled in exactly the same fashion by the Flume agent's sink when we are writing to HDFS. We will use JSON-formatted data again later to write data into HBase. There will definitely be differences in the configuration files at that point.

```
# list the sources, channels, and sinks for the agent
# list the sources, channels, and sinks for the agent
bdglue.sources = s1
bdglue.channels = c1
bdglue.sinks = k1

# Map the channels to the source.
bdglue.sources.s1.channels = c1

# Set the properties for the source
bdglue.sources.s1.type = avro
bdglue.sources.s1.bind = localhost
bdglue.sources.s1.port = 41414
bdglue.sources.s1.selector.type = replicating

# Set the properties for the channels
bdglue.channels.c1.type = memory

# make capacity and transactionCapacity much larger
# (i.e. 10x or more) for production use
bdglue.channels.c1.capacity = 1000
bdglue.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
bdglue.sinks.k1.channel = c1

bdglue.sinks.k1.type = hdfs
bdglue.sinks.k1.serializer = text

# each table written to separate directory named ‘tablename’
bdglue.sinks.k1.hdfs.path = hdfs://bigdatalite.localdomain/user/flume/gg-
data/%{table}
bdglue.sinks.k1.hdfs.fileType = DataStream
bdglue.sinks.k1.hdfs.filePrefix = bdglue
bdglue.sinks.k1.hdfs.fileSuffix = .txt
bdglue.sinks.k1.hdfs.inUsePrefix = _
bdglue.sinks.k1.hdfs.inUseSuffix =

# number of records the sink will read per transaction.
# Higher numbers may yield better performance.
bdglue.sinks.k1.hdfs.batchSize = 10
# the size of the files in bytes.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollSize = 1048576
# roll to a new file after N records.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollCount = 100
# roll to a new file after N seconds.  0=disable
bdglue.sinks.k1.hdfs.rollInterval = 30
```

Having the metadata transmitted with the column data is handy, but it does take up more space in HDFS when stored this way.

Configuring for Binary Avro Encoding

As mentioned earlier, and advantage to Avro encoding over JSON is that it is more compact, but it is also a little more complex as Avro schema files are required. It is possible to have BDGlue generate Avro schema files on the fly from the metadata that is passed in from the source, but this is not recommended as the files are needed downstream before data is actually landed. Instead, it is recommended that for data coming from relational database sources you utilize the SchemaDef utility to generate these schemas. See ***Generating Avro Schemas with SchemaDef*** later in this document for more information on how to do this.

Once we have the Avro schema files where we need them, we can think about configuring BDGlue and Flume to handle Avro encoded data.

The first step, as before, is to set the appropriate properties in the *bdglue.properties* file. You will see here that we are introducing a couple of new properties associated with the location of the *.avsc files locally and in HDFS.

```
# configuring BDGlue for Avro encoding
#
bdglue.encoder.class = com.oracle.bdglue.encoder.AvroEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = true
# The URI in HDFS where schemas will be stored.
# Required by the Flume sink event serializer.
bdglue.event.avro-hdfs-schema-path = hdfs:///user/flume/gg-data/avro-schema/
# local path where bdglue can find the avro *.avsc schema files
bdglue.event.avro-schema-path = /local/path/to/avro/schema/files

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

And of course, we also need to configure Flume to handle this data as well. Again, you’ll see some differences in the properties for the agent’s sink … specifically a non-default serializer that properly creates the *.avro files with the proper schema.

```
# list the sources, channels, and sinks for the agent
bdglue.sources = s1
bdglue.channels = c1
bdglue.sinks = k1

# Map the channels to the source. One channel per table being captured.
bdglue.sources.s1.channels = c1

# Set the properties for the source
bdglue.sources.s1.type = avro
bdglue.sources.s1.bind = localhost
bdglue.sources.s1.port = 41414
bdglue.sources.s1.selector.type = replicating

# Set the properties for the channels
# c1 is the default ... it will handle unspecified tables.
bdglue.channels.c1.type = memory

# make capacity and transactionCapacity much larger
# (i.e. 10x or more) for production use
bdglue.channels.c1.capacity = 1000
bdglue.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
bdglue.sinks.k1.channel = c1

bdglue.sinks.k1.type = hdfs
bdglue.sinks.k1.serializer = org.apache.flume.sink.hdfs.AvroEventSerializer$Builder

bdglue.sinks.k1.hdfs.path = hdfs://bigdatalite.localdomain/user/flume/gg-
data/%{table}
bdglue.sinks.k1.hdfs.fileType = DataStream
# avro files must end in .avro to work in an Avro MapReduce job
bdglue.sinks.k1.hdfs.filePrefix = bdglue
bdglue.sinks.k1.hdfs.fileSuffix = .avro
bdglue.sinks.k1.hdfs.inUsePrefix = _
bdglue.sinks.k1.hdfs.inUseSuffix =

# number of records the sink will read per transaction.
# Higher numbers may yield better performance.
bdglue.sinks.k1.hdfs.batchSize = 10
# the size of the files in bytes.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollSize = 1048576
# roll to a new file after N records.
# 0=disable (recommended for production)
bdglue.sinks.k1.hdfs.rollCount = 100
# roll to a new file after N seconds.  0=disable
bdglue.sinks.k1.hdfs.rollInterval = 30
```

And that's it. We are now all set to deliver Avro encoded data into *.avro files in HDFS.

Making Data Stored in HDFS Accessible to Hive

So now we have built and demonstrated the foundation for what comes next … making the data accessible via other Hadoop technologies. In this section, we'll look at accessing data from Hive.

You may be wondering why we went to the trouble we did in the previous section. It certainly seems like a lot of work just to put all that data into HDFS. The answer to that question is: "Hive." It turns out that once data has been properly serialized and stored in Avro format, Hive can make use of it directly … no need to Sqoop the data into Hive, etc. By approaching things this way, we save both an extra "Sqoop" step, and we eliminate any potential performance impact of writing the data directly into Hive tables on the fly. Of course, you can always choose to import the data into actual Hive storage later if you wish.

Configuration

The configuration for doing this is exactly the same as we did in the previous section. Since there are no differences in the Flume configuration, so we won't repeat it here. There are no differences in the *bdglue.properties* file either. We are repeating it here to highlight one property. The value of this property must match the corresponding value specified by the SchemaDef utility when generating the Hive Query Language DDL for the corresponding tables.

```
# configuring BDGlue to create HDFS-formatted files that
# can be accessed by Hive.
#
bdglue.encoder.class = com.oracle.bdglue.encoder.AvroEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = true
# The URI in HDFS where schemas will be stored.
# Required by the Flume sink event serializer.
bdglue.event.avro-hdfs-schema-path = hdfs:///user/flume/gg-data/avro-schema/
# local path where bdglue can find the avro *.avsc schema files
bdglue.event.avro-schema-path = /local/path/to/avro/schema/files

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

*Accessing *.avro Files From Hive*

Hive is smart enough to be able to access *.avro files where they live, and in this section we'll show you how that works.

The first, and only real step, is to create a table in Hive and tell it to read data from *.avro files. You'll notice a couple of key things:

- We do not need to specify the columns, their types, etc. All of this information is found in the Avro schema metadata, so all we have to do is point Hive to the schema and we're all set.
- This process is making use of Hive's Avro SerDe (serializer and deserializer) mechanism to decode the Avro data.

What is especially nice is that we can use the SchemaDef utility to generate the Hive table definitions like the following example. See ***Generating Hive Table Definitions for Use with Avro Schemas*** for more information.

```
DROP TABLE CUST_INFO;

CREATE EXTERNAL TABLE CUST_INFO
    COMMENT "A table backed by Avro data with the Avro schema stored in HDFS"
    ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
    STORED AS
    INPUTFORMAT  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
    LOCATION '/user/flume/gg-data/bdgluedemo.CUST_INFO/'
    TBLPROPERTIES (
        'avro.schema.url'=
        'hdfs:///user/flume/gg-data/avro-schema/bdgluedemo.CUST_INFO.avsc'
    );
```

And as a validation that this all works as expected, review the following.

```
hive> describe CUST_INFO;
OK
Id             int            from deserializer
Name           string         from deserializer
Gender         string         from deserializer
City           string         from deserializer
Phone          string         from deserializer
old_id         int            from deserializer
zip            string         from deserializer
cust_date      string         from deserializer
Time taken: 0.545 seconds, Fetched: 8 row(s)
hive> select * from CUST_INFO limit 5;
```

```
OK
1601  Dane Nash  Male  Le Grand-Quevilly  (874) 373-6196  1  81558-771 2014/04/13
1602  Serina Jarvis  Male  Carlton  (828) 764-7840   2  70179    2014/03/14
1603  Amos Fischer  Male  Fontaine-l'Evique  (141) 398-6160   3  9188  2015/02/06
1604  Hamish Mcpherson  Male  Edmonton  (251) 120-8238   4  T4M 1S9  2013/12/21
1605  Chadwick Daniels  Female Ansfelden (236) 631-9213   5 38076    2015/04/05
Time taken: 0.723 seconds, Fetched: 5 row(s)
Hive>
```

Delivering Data to Kafka

Kafka is a fast, scalable, and fault-tolerant publish-subscribe messaging system that is frequently used in place of more traditional message brokers in “Big Data” environments. As with traditional message brokers, Kafka has the notion of a “topic” to which events are published. Data is published by a Kafka “producer”.

Data written to a topic by a producer can further be partitioned by the notion of a “key”. The key serves two purposes: to aid in partitioning data that has been written to a topic for reasons of scalability, and in our case to aid downstream “consumers” in determining exactly what data they are looking at.

In the case of BDGlue, by default all data is written to a single topic, and the data is further partitioned by use of a key. The key in this case is the table name, which can be used to route data to particular consumers, and additional tell those consumer what exactly they are looking at.

Finally, Kafka supports the notion of “batch” or “bulk” writes using an asynchronous API that accepts many messages at once to aid in scalability. BDGlue takes advantage of this capability by writing batches of messages at once. The batch size is configurable, as is a timeout specified in milliseconds that will force a “flush” in the event that too much time passes before a batch is completed and written.

When publishing events, Kafka is expecting three bits of information:

- Topic – which will be the same for all events published by an instance of the Kafka Publisher
- Key – which will correspond to the table name that relates to the encoded data
- Body – the actual body of the message that is to be delivered. The format of this data may be anything. In the case of the Kafka publisher, any of the encoded types are supported: Delimited Text, JSON, and Avro.

Note that there are some additional java dependencies required to execute a Kafka publisher beyond those required to actually compile BDGlue and must be added to the classpath in the Java Adapter properties file. In this case, the specific order of the dependencies listed is very important. If you make a mistake here you will like find the wrong entry point into Kafka and results will be indeterminate.

```
#Adapter Logging parameters.
#log.logname=ggjavaue
#log.tofile=true
log.level=INFO


#Adapter Check pointing  parameters
goldengate.userexit.chkptprefix=GGHCHKP_
goldengate.userexit.nochkpt=true
# Java User Exit Property
goldengate.userexit.writers=javawriter

# this is one continuous line
javawriter.bootoptions= -Xms64m -Xmx512M
  -Dlog4j.configuration=ggjavaue-log4j.properties
  -Dbdglue.properties=bdglue.properties
  -Djava.class.path=./gghadoop:./ggjava/ggjava.jar
#

#Properties for reporting statistics
# Minimum number of {records, seconds} before generating a report
javawriter.stats.time=3600
javawriter.stats.numrecs=5000
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE

#Hadoop Handler.
gg.handlerlist=gghadoop
gg.handler.gghadoop.type=com.oracle.gghadoop.GG12Handler
gg.handler.gghadoop.mode=op
gg.classpath=./gghadoop/lib/*:/kafka/kafka_2.10-0.8.2.1/libs/kafka-clients-
0.8.2.1.jar:/kafka/kafka_2.10-0.8.2.1/libs/*
```

Configuring the Kafka Publisher

Configuring the Kafka Publisher is actually very straight-forward:

- Configure an encoder (note that the “NullEncoder” is not supported by this publisher). The encoder must be for one of the actual supported data formats: Avro, JSON, or Delimited.
- Configure the KafkaPublisher.

```
# bdglue.properties file for delivery to Kafka
#
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = false

bdglue.publisher.class = com.oracle.bdglue.publisher.kafka.KafkaPublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = table

bdglue.kafka.topic = goldengate
bdglue.kafka.batchSize = 3
bdglue.kafka.flushFreq = 500
bdglue.kafka.metadata.broker.list = localhost:9092
```

Note that at there are several "bdglue.kafka" properties located toward the bottom of the example above. Only one of those is actually required, and that is the broker list. This is defined in the Kafka documentation and tells the KafkaPublisher which Kafka broker(s) to deliver events to. Information about these and a few other Kafka-related properties can be found in the appendix at the end of this document.

Using Flume to Deliver Data to Kafka

While in most situations users will configure BDGlue to deliver data to Kafka directly, BDGlue also supports the delivery of data to Kafka by way of Flume. This approach might be useful if there more complicated flow of data required that neither BDGlue nor Kafka can provide on their own. Flume's ability to fork and merge data flows, or augment the flow with additional processors (called 'interceptors') can prove to be extremely powerful when defining the architecture of a data flow.

Configuring BDGlue

First we must configure BDGlue to deliver the data to Flume. Just as with the KafkaPublisher, the data must be encoded in one of the supported formats: Delimited Text, Avro, or JSON. You'll see that the bdglue.properties file is simpler than some as there isn't much for BDGlue to do other than encode the data and hand it on.

```
# Configuring BDGlue to deliver data to Kafka
# by way of Flume (bdglue.properties)
#
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

Configuring the BDGlue Sink for Kafka

You will see that for a number of targets that we can deliver to via Flume, we have developed a custom Flume “sink” to process the data as we expect. Configuration is still relatively straight forward.

```
# list the sources, channels, and sinks for the agent
ggflume.sources = s1
ggflume.channels = c1
ggflume.sinks = k1

# Map the channels to the source. One channel per table being captured.
ggflume.sources.s1.channels = c1

# Set the properties for the source
ggflume.sources.s1.type = avro
ggflume.sources.s1.bind = localhost
ggflume.sources.s1.port = 41414
ggflume.sources.s1.selector.type = replicating

# Set the properties for the channels
# c1 is the default ... it will handle unspecified tables.
ggflume.channels.c1.type = memory
ggflume.channels.c1.capacity = 1000
ggflume.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
ggflume.sinks.k1.channel = c1

ggflume.sinks.k1.type = com.oracle.bdglue.publisher.flume.sink.kafka.KafkaSink
ggflume.sinks.k1.batchSize = 3
ggflume.sinks.k1.brokerList = localhost:9092
ggflume.sinks.k1.topic = goldengate
```

You will see that there are some required properties specific to this Kafka delivery:

- *type* – identifies the Flume Sink we are calling
- *batchSize* – identifies the number of Flume events we should queue before actually delivering to Kafka. Note that this is a "pull" architecture and if the sink looks for and doesn't find another event to process it will deliver what it has accumulated to Kafka.
- *brokerList* – is required and identifies the broker(s) that we should deliver to.
- *topic* – provides the name of the Kafka topic that we should publish the events to.

And there are some optional properties as well (not specified in the example above):

- *requiredAcks* – defines the sort of acknowledgement we should expect before continuing. 0 = none, 1 = wait for acknowledgement from a single broker, and -1 = wait for acknowledgement from all brokers.
- *kafka.serializer.class* – override the default serializer when delivering the message body. This capability is present, but it is not likely that you will need to do so.
- *kafka.key.serializer.class* – override the default serializer used for encoding the key (table name in our case). This capability is present, but it is not likely that you will have reason to override this property.

Once configured, you simply start Flume and BDGlue as you otherwise would. See the next section to get ideas on how to validate that data is successfully being delivered.

Validating Delivery to Kafka

Note that for data to be delivered successfully, the Kafka broker must be running when BDGlue attempts to write to it. The broker may be installed and running as a service, or if not, will need to be started by hand. There is a script to do this that can be found in the "bin" directory of the Kafka installation, and a default set of properties can be found in the "config" directory:

```
./bin/kafka-server-start.sh config/server.properties
```

In the Kafka architecture, both the BDGlue KafkaPublisher and the Flume Kafka "sink" serve the role of "Kafka Producer". In order to see what has been delivered to Kafka, there will need to be a consumer. Kafka has a sample consumer, called the "Console Consumer" which is great for smoke testing the environment. The Console Consumer basically reads messages that have been posted to a topic and writes them to the screen.

```
./bin/kafka-console-consumer.sh --zookeeper localhost:2181  --topic goldengate --
from-beginning --property print.key=true
```

The “print.key” property causes the consumer to print the topic “key” (in our case, the table name) to the console along with the message. Note that if you are going to use the Console Consumer, it would probably be best to configure the JsonEncoder during this time as the data that is output will be in a text-based format. Data encoded by the AvroEncoder can contain binary data and will not be as legible on your screen.

Delivering Data to HBase

Apache HBase is a column-oriented key/value data store built to run on top of the Hadoop Distributed File System (HDFS). An HBase system comprises a set of tables. Each table contains rows and columns, much like a traditional database, and it also has an element defined as a key.

All access to HBase tables must use the defined key. While similar in nature to a primary key in a relational database, a key in HBase might be used a little differently … defined and based specifically on how the data will be accessed after it has been written.

An HBase column represents an attribute of an object and in our case likely a direct mapping of a column from a relational database. HBase allows for many columns to be grouped together into what are known as column families, such that the elements of a column family are all stored together. This is different from a row-oriented relational database, where all the columns of a given row are stored together.

With HBase you must predefine the table schema and specify the column families. However, it is very flexible in that new columns can be added to families at any time, making the schema flexible and therefore able to adapt to changing application requirements.

Currently in BDGlue, we map each source table into a single column family of a corresponding table in HBase, creating a key from the key on the relational side. This may not be the best approach in some circumstances, however. The very nature of HBase cries out for keys that are geared toward that actual way you are likely to access the data via map reduce (which is likely quite different than how you would access a relational table). In some cases, it would be most optimal to combine relational tables that share a common key on the relational side into a single table in HBase, having a separate column family for each mapped table. This is all possible in theory, and a future version of this code may support a JSON-based specification file to define the desired mappings.

Connecting to HBase via the Asynchronous HBase Publisher

Just as with Kafka, configuring the Asynchronous HBase Publisher very straight-forward:

- Configure the NullEncoder.
- Configure the AsyncHbasePublisher.

This is done by setting bdglue.properties as follows:

```
#
# bdglue.properties file for the AsyncHbasePublisher
#
bdglue.encoder.class = com.oracle.bdglue.encoder.NullEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = false

bdglue.publisher.class = com.oracle.bdglue.publisher.asynchbase.AsyncHbasePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.async-hbase.batchSize = 5
bdglue.async-hbase.timeout = 5000
```

Using Flume to Deliver Data to HBase

Connecting to HBase via Flume was a bit more complicated than working with HDFS and Hive. While we were able to make things work properly with multiple tables using the out-of-the-box Flume agent components with HDFS and Hive, we weren't able to do that with HBase. To accomplish our goal of supporting multiple tables with HBase via a single channel, we had to take advantage of the flexibility of Flume and implement a custom Flume sink and sink serializer. This wasn't particularly hard to accomplish, however.

As with our other examples, we first need to configure the *bdglue.properties* file with the appropriate properties. In this case, we will transmit the data in JSON format, which the custom sink was designed to expect, and we will specify HBase as the target.

```
# bdglue.properties for writing to HBase via Flume
#
```

```
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.json.text-only = false
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = false

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

With the exception of specifying the custom sink information, the Flume configuration properties are actually a little simpler than prior examples.

```
# list the sources, channels, and sinks for the agent
bdglue.sources = s1
bdglue.channels = c1
bdglue.sinks = k1

# Map the channels to the source. One channel per table being captured.
bdglue.sources.s1.channels = c1

# Set the properties for the source
bdglue.sources.s1.type = avro
bdglue.sources.s1.bind = localhost
bdglue.sources.s1.port = 41414
bdglue.sources.s1.selector.type = replicating

# Set the properties for the channels
bdglue.channels.c1.type = memory
bdglue.channels.c1.capacity = 1000
bdglue.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
bdglue.sinks.k1.channel = c1

bdglue.sinks.k1.type =
    com.oracle.bdglue.publisher.flume.sink.asynchbase.BDGlueAsyncHbaseSink
bdglue.sinks.k1.batchSize = 100
bdglue.sinks.k1.timeout = 6000
```

Finally, remember to add the bdglue.jar file to the Flume class path as described in ***Running Flume*** earlier in this document.

Basic HBase Administration

There are a couple of things we need to do to make sure that HBase is ready to receive data.

First off, we need to make sure that HBase is running. This requires 'sudo' access on Linux/Unix.

```
#>  sudo service hbase-master start
Starting HBase master daemon (hbase-master):               [  OK  ]
HBase master daemon is running
#>                           [  OK  ]
#>  sudo service hbase-regionserver start
Starting Hadoop HBase regionserver daemon: starting regionserver, logging to
/var/log/hbase/hbase-hbase-regionserver-bigdatalite.localdomain.out
hbase-regionserver.
#>
```

And we also need to create the tables in HBase to receive the information we want to write there. If you are not aware, HBase has something called “column families”. All columns reside within a column family, and each table can have multiple column families if desired. For the purpose of this adapter, we are assuming a default name of ‘data’ for the column family, and are putting all columns from the relational source in there.

The example below creates table CUST_INFO having a single column family called ‘data’. Before doing that, we check the status to be sure that we have a region server up and running.

```
[ogg@bigdatalite ~]$
[ogg@bigdatalite ~]$ hbase shell
2014-10-03 17:40:40,439 INFO  [main] Configuration.deprecation: Hadoop.native.lib is
deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.96.1.1-cdh5.0.3, rUnknown, Tue Jul  8 15:35:51 PDT 2014

hbase(main):001:0> status
1 servers, 0 dead, 3.0000 average load

hbase(main):007:0> create 'CUST_INFO', 'data'
0 row(s) in 0.8300 seconds

<add some test data ... >

hbase(main):006:0> scan 'CUST_INFO'
ROW                   COLUMN+CELL
 /7021                column=data:CITY, timestamp=1434739949881, value=Le Grand-
                      Quevilly
 /7021                column=data:CUST_DATE, timestamp=1434739949901, value=2014
                      /04/13
 /7021                column=data:GENDER, timestamp=1434739949899, value=Male
 /7021                column=data:ID, timestamp=1434739949843, value=7021
 /7021                column=data:NAME, timestamp=1434739949844, value=Dane Nash
 /7021                column=data:OLD_ID, timestamp=1434739949847, value=1
 /7021                column=data:PHONE, timestamp=1434739949846, value=(874) 37
                      3-6196
 /7021                column=data:ZIP, timestamp=1434739949847, value=81558-771
 /7022                column=data:CITY, timestamp=1434739949834, value=Carlton
 /7022                column=data:CUST_DATE, timestamp=1434739949838, value=2014
                      /03/14
 /7022                column=data:GENDER, timestamp=1434739949826, value=Male
 /7022                column=data:ID, timestamp=1434739949892, value=7022
 /7022                column=data:NAME, timestamp=1434739949825, value=Serina Ja
                      rvis
 /7022                column=data:OLD_ID, timestamp=1434739949835, value=2
 /7022                column=data:PHONE, timestamp=1434739949845, value=(828) 76
                      4-7840
 /7022                column=data:ZIP, timestamp=1434739949836, value=70179
```

```
totalEvents          column=data:eventCount, timestamp=1434739949985, value=\x0
                      0\x00\x00\x00\x00\x00\x00\x02
2 row(s) in 0.1020 seconds

hbase(main):009:0>hbase(main):009:0> exit
[ogg@bigdatalite ~]$
```

Delivering Data to Oracle NoSQL

The Oracle NoSQL database[3] is a leading player in the NoSQL space. Oracle NoSQL Database provides a powerful and flexible transaction model that greatly simplifies the process of developing a NoSQL-based application. It scales horizontally with high availability and transparent load balancing even when dynamically adding new capacity, bringing industrial strength into an arena where it is often found to be lacking.

Some key benefits that the product brings to the Big Data "table" are:

- Simple data model using key-value pairs with secondary indexes
- Simple programming model with ACID transactions, tabular data models, and JSON support
- Application security with authentication and session-level SSL encryption
- Integrated with Oracle Database, Oracle Wallet, and Hadoop
- Geo-distributed data with support for multiple data centers
- High availability with local and remote failover and synchronization
- Scalable throughput and bounded latency

Oracle NoSQL can be used with or without Hadoop. It supports two APIs for storing and retrieving data: the KV (key-value) API, and the Table API. Each API has its own strengths. The KV API is more "traditional", but the Table API is gaining a lot of momentum in the market. This adapter supports interfacing with Oracle NoSQL with both APIs.

KV API Support

The KV API writes data to Oracle NoSQL in key-value pairs, where the key is a text string that looks much like a file system path name, with each "node" of the key preceded by a slash ('/'). For example, keys based on customer names might look like:

```
/smith/john
/smith/patty
/hutchison/don
```

[3] More information on Oracle's NoSQL Database can be found here: http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html

BDGlue creates the key for each row by concatenating a string representation of each relational column that makes up the primary key in the order that the columns are listed in the relational table's metadata.

A "value" is data of some sort. It may be text-based, or binary. The structure obviously must be understood by the application. Oracle NoSQL itself is very powerful, however, and there is much database "work" it is able to do if it can understand the data. As it turns out, Oracle NoSQL supports Avro schemas, something that we have already discussed in the context of other Big Data targets. BDGlue makes good use of this Avro encoding.

Table API Support

The Oracle NoSQL Table API is a different way of storing and accessing data. In theory, you can leverage the same data via the KV and Table APIs, but we are not approaching things in that fashion. The Table API maps data in a "row" on a column-by-column basis.

BDGlue maps the source tables and their columns directly to tables in Oracle NoSQL of essentially the same structure. Key columns are also mapped one-for-one.

NoSQL Transactional Durability

Before we get to specific configurations, we should also mention at this point the "durability" property, which is applicable to all aspects of this adapter: direct to NoSQL, or via Flume; and for both the Table and KV APIs. Durability effectively addresses "guarantee" that data is safe and sound in the event of a badly timed failure. Oracle NoSQL supports different approaches to syncing transactions once they are committed (i.e. durability). BDGlue supports three sync models:

- SYNC : Commit onto disk at master and replicate to simple majority of replicas. This is the most durable. When the commit returns to the caller, you can be absolutely certain that the data will still be there no matter what the failure situation. It is also the slowest.
- WRITE_NO_SYNC: Commit onto disk at master but do not wait for data to replicate to other nodes. This is of medium performance as it writes to the master, but doesn't wait for the data to be replicated before returning to the caller after a commit.
- NO_SYNC: Commit only into master memory and do not wait for the data to replicate to other nodes. This is the fastest mode as it returns to the caller immediately upon handing the data to the NoSQL master. At that point, the data has not been synced to disk and could be lost in the event of a failure at the master.

Connecting Directly to Oracle NoSQL via the NoSQL Publisher

Connecting and delivering data to Oracle NoSQL is not particularly complicated.

Configuring for Delivery to the KV API

Delivery to the KV API is straight forward … the key is a concatenated string based on the columns from the source table that comprise the primary key, and the value is an Avro-encoded record containing all of the columns that have been captured, including the key columns.

The first step, obviously, is to configure the *bdglue.properties* file.

```
# bdglue.properties file for direct connection to
# Oracle NoSQL via the KV API.
#
bdglue.encoder.class = com.oracle.bdglue.encoder.AvroEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = true
bdglue.event.avro-schema-path = ./gghadoop/avro

bdglue.publisher.class = com.oracle.bdglue.publisher.nosql.NoSQLPublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.nosql.host = localhost
bdglue.nosql.port = 5000
bdglue.nosql.kvstore = kvstore
bdglue.nosql.durability = WRITE_NO_SYNC
bdglue.nosql.api = kv_api
```

The above properties are all that is required. See the admin section ***Basic Oracle NoSQL Administration*** for some basic information regarding how to define tables in Oracle NoSQL, etc.

Configuring for Delivery via the Table API

BDGlue maps the source tables and their columns directly to tables in Oracle NoSQL of essentially the same structure. Key columns are also mapped one-for-one.

Just as always, the first, and in this case the only thing we need to do is configure the adapter to format and process the data as we expect via the *bdglue.properties* file. For the NoSQL Table API, we configure the NullEncoder because BDGlue writes the data to NoSQL on a column-by-column basis.

```
# bdglue.properties for delivering directly to
# the Oracle NoSQL Table API.
#
bdglue.encoder.class = com.oracle.bdglue.encoder.NullEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = false

bdglue.publisher.class = com.oracle.bdglue.publisher.nosql.NoSQLPublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.nosql.host = localhost
bdglue.nosql.port = 5000
bdglue.nosql.kvstore = kvstore
bdglue.nosql.durability = WRITE_NO_SYNC
bdglue.nosql.api = table_api
```

Using Flume to Deliver Data into the Oracle NoSQL Database

Just as it did to integrate with HBase, BDGlue also requires a custom Flume sink in order to communicate with Oracle NoSQL. Communication with Oracle NoSQL occurs via RPC, and this document assumes that Oracle NoSQL is already up and running, and configured to listen on the specified port.

Configuring for Delivery via the KV API

As with the other target environments, the first step is to configure BDGlue itself via the *bdglue.properties* file.

```
# bdglue.properties for writing to NoSQL KV API via Flume
#
bdglue.encoder.class = com.oracle.bdglue.encoder.AvroEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-longname = true

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

It is pretty much the same as the other targets.

Next we need to configure Flume itself. Configuration of the Flume Source and Flume Channel is the same as before, but configuration of the sink is much different.

```
# list the sources, channels, and sinks for the agent
ggflume.sources = s1
ggflume.channels = c1
ggflume.sinks = k1

# Map the channels to the source. One channel per table being captured.
ggflume.sources.s1.channels = c1

# Set the properties for the source
ggflume.sources.s1.type = avro
ggflume.sources.s1.bind = localhost
ggflume.sources.s1.port = 41414
ggflume.sources.s1.selector.type = replicating

# Set the properties for the channels
ggflume.channels.c1.type = memory
ggflume.channels.c1.capacity = 1000
ggflume.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
ggflume.sinks.k1.channel = c1

ggflume.sinks.k1.type = com.oracle.bdglue.target.flume.sink.nosql.BDGlueNoSQLSin
k
ggflume.sinks.k1.kvHost = localhost
ggflume.sinks.k1.kvPort= 5000
ggflume.sinks.k1.kvStoreName = kvstore
ggflume.sinks.k1.durability = WRITE_NO_SYNC
# kv_api or table_api
ggflume.sinks.k1.kvapi= kv_api
```

Sink configuration is the same for both the KV and Table APIs. The only difference is the last line: *kv_api* in this case.

As mentioned previously, BDGlue assumes that Oracle NoSQL is up, running and listening on the specified port.

Configuring for Delivery via the Table API

Usage and access to Oracle NoSQL over Flume via the Table API is somewhat similar to how we interface with HBase. In fact, just as with HBase, we will pass the data into Flume in a JSON format so that we can manipulate it directly.

BDGlue maps the source tables and their columns directly to tables in Oracle NoSQL of essentially the same structure. Key columns are also mapped one-for-one.

Just as always, the first thing we need to do is configure the adapter to format and process the data as we expect via the *bdglue.properties* file.

```
# bdglue.properties for writing to NoSQL Table API via Flume
#
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tx-optype = false
bdglue.encoder.tx-timestamp = false
bdglue.encoder.user-token = false

bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-longname = false

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 41414
bdglue.flume.rpc.type = avro-rpc
```

The key difference to note for the Table API vs. the KV API is that we specified an encoding of “json” rather than “avro-binary”. The HBase sink implementation actually parses the JSON-formatted data to deliver the data to the NoSQL Table API column-by-column as the API expects.

And of course, we also need to configure Flume as well.

```
# list the sources, channels, and sinks for the agent
ggflume.sources = s1
ggflume.channels = c1
ggflume.sinks = k1

# Map the channels to the source. One channel per table being captured.
ggflume.sources.s1.channels = c1

# Set the properties for the source
ggflume.sources.s1.type = avro
ggflume.sources.s1.bind = localhost
ggflume.sources.s1.port = 41414
ggflume.sources.s1.selector.type = replicating

# Set the properties for the channels
ggflume.channels.c1.type = memory
ggflume.channels.c1.capacity = 1000
ggflume.channels.c1.transactionCapacity = 100

# Set the properties for the sinks
# map the sinks to the channels
ggflume.sinks.k1.channel = c1

ggflume.sinks.k1.type = ggflume.sink.nosql.GGFlumeNoSQLSink
ggflume.sinks.k1.kvHost = localhost
ggflume.sinks.k1.kvPort= 5000
ggflume.sinks.k1.kvStoreName = kvstore
ggflume.sinks.k1.durability = WRITE_NO_SYNC
# kv_api or table_api
ggflume.sinks.k1.kvapi= table_api
```

As mentioned previously, the only difference in the configuration between this and the KV API is the last line. In this case, we specify “table_api”.

Basic Oracle NoSQL Administration

Oracle NoSQL comes in two flavors, a “lite” version for basic testing, and an Enterprise version that contains more robust capabilities that enterprise deployments might require. Administratively, they are virtually identical as far as the features we are leveraging in BDGlue … so we keep it simple and test with the “lite” version, called KVLite.

Starting Oracle NoSQL from the Command Line

Oracle NoSQL runs as a Java process.

```
#
# starts the kvlite NoSQL instance, listening on default port of 5000
#
[nosqlhome]$ KVHOME="/u01/nosql/kv-ee"
```

```
[nosqlhome]$ java -Xmx256m -Xms256m -jar $KVHOME/lib/kvstore.jar kvlite
Opened existing kvlite store with config:
-root ./kvroot -store kvstore -host bigdatalite.localdomain -port 5000 -admin 5001
```

Running the KVLite Administration Command Line Interface

The command line utility is where you will define tables, review data stored in Oracle NoSQL, etc.

```
#
# run the kvlite command line interface
#
[nosqlhome]$ KVHOME="/u01/nosql/kv-ee"

[nosqlhome]$ java -Xmx256m -Xms256m -jar $KVHOME/lib/kvstore.jar runadmin -port 5000
-host localhost
kv->
kv-> connect store -name kvstore
Connected to kvstore
Connected to kvstore at localhost:5000.
kv->
```

KV API: Creating Tables in Oracle NoSQL

The KV API relies on delivery of key-value pairs. In the case of BDGlue, the key is a concatenation of the columns that make up the key in the relational database source. The "value" is an array of bytes that contain the data we are storing. In our case, we are choosing to encode the "value" in Avro format both because it is a compact representation of the data, and because it self-describes to Oracle NoSQL.

KV API: Preparing the Avro Schemas

Preparing the Avro schemas is a two step process:

- Generate the schemas from the source table meta data
- Load the schemas into Oracle NoSQL.

The steps to generate the schemas are exactly as described in ***Generating Avro Schemas***. Refer to that section for more information.

Loading the generated schemas into NoSQL is a relatively straight forward process. First you must log into NoSQL with the admin utility. Once you are logged in and have the command prompt, do the following:

```
kv-> ddl add-schema -file ./bdglue.CUST_INFO.avsc -force
Added schema: bdglue.CUST_INFO.1
8 warnings were ignored.    << Ignore if you see this: the result of not setting
default values
kv-> ddl add-schema -file ./bdglue.MYCUSTOMER.avsc -force
```

```
Added schema: bdglue.MYCUSTOMER.2
22 warnings were ignored. << Ignore if you see this: the result of not setting
default values
kv-> show schemas
bdglue.CUST_INFO
  ID: 1  Modified: 2014-10-21 19:19:22 UTC, From: bigdatalite.localdomain
bdglue.MYCUSTOMER
  ID: 2  Modified: 2014-10-21 19:19:53 UTC, From: bigdatalite.localdomain
kv->
```

Once complete, you are ready to capture data from a source database and deliver into the Oracle NoSQL data store.

KV API: Validating Your Data

Oracle NoSQL doesn't provide an easy way to query data stored in KV pairs from the command line. To do this, you need to have the key to the row you want to see.

```
kv-> get kv -key /2978
{
  "ID" : 2978,
  "NAME" : "Basia Foley",
  "GENDER" : "Female",
  "CITY" : "Ichtegem",
  "PHONE" : "(943) 730-2640",
  "OLD_ID" : 8,
  "ZIP" : "T1X 1M5",
  "CUST_DATE" : "2015/01/16"
}
kv->
```

In the example above, the key was "/2798" (note the preceding slash). Also note that Oracle NoSQL understood the structure of the stored value object. This is because we generated and used the Avro schema when writing the key-value pair to the database.

Table API: Creating Tables in Oracle NoSQL

Just as with a relational database, you have to create tables in Oracle NoSQL in order to use the Table API. Table creation commands can actually be quite cumbersome, but we have actually simplified the process somewhat by configuring the SchemaDef utility discussed later in this document in ***Generating Avro Schemas with SchemaDef*** to generate the NoSQL DDL for us. Just as for Avro, the utility connects to the source database via JDBC. Everything is essentially the same as before except for the output format.

Here is what the schemadef.properties file might look like:

```
# jdbc connection information
schemadef.jdbc.driver = com.mysql.jdbc.Driver
schemadef.jdbc.url = jdbc:mysql://localhost/bdglue

# Oracle JDBC connection info
#schemadef.jdbc.driver = oracle.jdbc.OracleDriver
#schemadef.jdbc.url = jdbc:oracle:thin:@//<host>:<port>/<service_name>

schemadef.jdbc.username = root
schemadef.jdbc.password = welcome1

# output format: avro, nosql
schemadef.output.format = nosql
schemadef.output.path = ./output

# encode numeric/decimal types as string, double, float
schemadef.numeric-encoding = double

schemadef.set-defaults = true
schemadef.tx-optype = false
schemadef.tx-timestamp = false

# whitespace delimited list of schema.table pairs
schemadef.jdbc.tables = bdglue.MYCUSTOMER bdglue.CUST_INFO \
                     bdglue.TCUSTORD
```

And the utility is executed just as before:

```
DIR=/path/to/jars

CLASSPATH="$DIR/bdglue.jar"
CLASSPATH="$CLASSPATH:$DIR/slf4j-api-1.6.1.jar"
CLASSPATH="$CLASSPATH:$DIR/slf4j-simple-1.7.7.jar"
CLASSPATH="$CLASSPATH:$DIR/commons-io-2.4.jar"
CLASSPATH="$CLASSPATH:$DIR/jackson-core-asl-1.9.13.jar"
CLASSPATH="$CLASSPATH:$DIR/mysql-connector-java-5.1.34-bin.jar"

java –Dschemadef.properties=schemadef.properties -cp $CLASSPATH
                com.oracle.bdglue.utility.schemadef.SchemaDef
```

Here is a sample of a generated output file. Each output file contains the script needed to create a table in Oracle NoSQL that corresponds to the source table.

```
## enter into table creation mode
table create -name CUST_INFO
add-field -type INTEGER -name ID
primary-key -field ID
add-field -type STRING -name NAME
add-field -type STRING -name GENDER
add-field -type STRING -name CITY
add-field -type STRING -name PHONE
add-field -type INTEGER -name OLD_ID
add-field -type STRING -name ZIP
add-field -type STRING -name CUST_DATE
## exit table creation mode
exit
## add the table to the store and wait for completion
plan add-table -name CUST_INFO –wait
```

And then we have to add the tables into Oracle NoSQL. We do this from the command prompt in the Oracle NoSQL admin utility.

```
kv->
kv-> load -file ./output/CUST_INFO.nosql
Table CUST_INFO built.
Executed plan 5, waiting for completion...
Plan 5 ended successfully

kv-> load -file ./output/MYCUSTOMER.nosql
Table MYCUSTOMER built.
Executed plan 6, waiting for completion...
Plan 6 ended successfully

kv->
```

And now we are ready to capture data and deliver it into Oracle NoSQL.

Table API: Validating Your Data

Looking at data stored with the Table API is a little easier than with data stored with the KV API, but don’t expect the power you might have with a SQL query.

Here is sample output from a single row stored in the CUST_INFO table.

```
kv-> get table -name CUST_INFO  -field ID -value 3204 -pretty
{
  "ID" : "3204",
  "NAME" : "Adria Bray",
  "GENDER" : "Female",
  "CITY" : "Anklam",
  "PHONE" : "(131) 670-1907",
  "OLD_ID" : "94",
  "ZIP" : "27665",
  "CUST_DATE" : "2014/06/01"
}

kv->
```

In this case, we knew the ID column's value was "3204". If you leave off the *–field* and *–value* options, you can get all rows in the table, by the way.

Just to further prove the point, here is some example output from the MYCUSTOMER table.

```
kv-> get table -name MYCUSTOMER  -field id -value 2864 -pretty
{
  "id" : "2864",
  "LAST_NAME" : "Barnes",
  "FIRST_NAME" : "Steel",
  "STREET_ADDRESS" : "Ap #325-5990 A Av.",
  "POSTAL_CODE" : "V0S 7A8",
  "CITY_ID" : "14819",
  "CITY" : "Reus",
  "STATE_PROVINCE_ID" : "316",
  "STATE_PROVINCE" : "CA",
  "COUNTRY_ID" : "137",
  "COUNTRY" : "Iran",
  "CONTINENT_ID" : "1",
  "CONTINENT" : "indigo",
  "AGE" : "25",
  "COMMUTE_DISTANCE" : "14",
  "CREDIT_BALANCE" : "2934",
  "EDUCATION" : "Zolpidem Tartrate",
  "EMAIL" : "feugiat.nec@ante.com",
  "FULL_TIME" : "YES",
  "GENDER" : "MALE",
  "HOUSEHOLD_SIZE" : "3",
  "INCOME" : "116452"
}

kv->
```

And there you have it. We have validated that we successfully delivered data into the Oracle NoSQL database via the Table API.

Delivering Data to Cassandra

Cassandra is a 'flavor' of NoSQL database that has a very tabular feel. In fact, the syntax for CQL (Cassandra Query Language) is very similar to SQL. Cassandra has become quite popular for a number of reasons:

- It has a peer-to-peer architecture rather than one based on master-slave configurations. Any number of server nodes can be added to the cluster in order to reliability as there is no single point of failure.
- It boasts elastic scalability by adding or removing nodes from the cluster.
- Cassandra's architecture delivers high availability and fault tolerance.
- Cassandra delivers very high performance on large sets of data.
- Cassandra is column-oriented, giving a tabular feel to things. Cassandra rows can be extremely wide.
- It has a tunable consistency model ranging from "eventual consistency" to "strong consistency" which ensures that updates are written to all nodes.

The BDGlue Cassandra Publisher makes use of the Cassandra Java API published as Open Source by DataStax. Make sure that the version of Cassandra you are using is compatible with the DataStax Java API. Currently, DataStax claims compatibility with the latest stable release, Cassandra 3.0.x. It is not known at this time if the monthly development releases (currently version 3.5) are compatible or not. Feel free to experiment.

Each column in a Cassandra table will correspond to a column of the same name found in the relational source. Data types are mapped as closely as possible, and default to 'text' in situations where there is no direct mapping. Key columns are also mapped directly and in the order they are specified in the DDL (and in the order they are returned by JDBC if you use the SchemaDef utility to generate the DDL). The source schema name will correspond to the Cassandra "key space."

Connecting to Cassandra via the Cassandra Publisher

To deliver data to Cassandra, you need to configure the Cassandra Publisher as follows:

- Configure the NullEncoder
- Configure the CassandraPublisher

This is done by setting the values in the bdglue.properties file as follows:

```
bdglue.encoder.class = com.oracle.bdglue.encoder.NullEncoder
bdglue.encoder.threads = 2
bdglue.encoder.tablename = false
bdglue.encoder.txid = false
bdglue.encoder.tx-optype = true
bdglue.encoder.tx-timestamp = false
bdglue.encoder.tx-position = false
bdglue.encoder.user-token = false
bdglue.encoder.include-befores = false
bdglue.encoder.ignore-unchanged = false

bdglue.event.generate-avro-schema = false
bdglue.event.header-optype = false
bdglue.event.header-timestamp = false
bdglue.event.header-rowkey = true
bdglue.event.header-avropath = false
bdglue.event.header-columnfamily = true
bdglue.event.header-longname = true

bdglue.publisher.class = com.oracle.bdglue.publisher.cassandra.CassandraPublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.cassandra.node = localhost
bdglue.cassandra.batch-size = 5
bdglue.cassandra.flush-frequency = 500
bdglue.cassandra.insert-only = false
```

Basic Cassandra Administration

This section briefly explains how to start Cassandra from the command line, run the CQL shell, generate DDL that corresponds to the relational source tables, and apply that DDL into Cassandra.

Running Cassandra and the CQL Shell

First off, we have to make sure that Cassandra is running. To run Cassandra from the command line, run Cassandra using 'sudo'.

```
#> cd ./apache-cassandra-3.0.5   # the Cassandra installation directory
#> sudo ./bin/cassandra –f       # runs in the console window. CTRL-C to end.
```

Running the Cassandra shell follows much the same process, but 'sudo' is not required.

```
#> cd ./apache-cassandra-3.0.5   # the Cassandra installation directory
#>./bin/cqlsh                    # runs in the console window. CTRL-C to end.
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.5 | CQL spec 3.4.0 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Note that 'cqlsh' is implemented using Python version 2.7. Most Linux systems currently have Python 2.6 installed as the default. Various utilities such as 'yum' seem to rely on this version of Python. Note that these versions of Python are not compatible. If you don't have version 2.7, you will have to install it. The easiest way is to download the Python 2.7 source, build, and then install it into /usr/local/bin. Be careful not to overwrite the default Python 2.6 (likely installed in /usr/bin/…).

Creating Tables in Cassandra

As with any "tabular" database, you need to create tables in Cassandra. Cassandra DDL looks much like DDL for a relational database and we have simplified the process of creating table definitions that correspond to the source tables that we will be capturing. This is done using the SchemaDef utility discussed later in this document. See ***The "SchemaDef" Utility*** for more information on how to configure and run SchemaDef for Cassandra and other targets.

Here is an example of generated Cassandra DDL:

```
CREATE KEYSPACE IF NOT EXISTS "bdglue"
    WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };

DROP TABLE IF EXISTS bdglue.CUST_INFO;

CREATE TABLE bdglue.CUST_INFO
 (
   txoptype text,
   ID int,
   NAME text,
   GENDER text,
   CITY text,
   PHONE text,
   OLD_ID int,
   ZIP text,
   CUST_DATE text,
   PRIMARY KEY (ID)
 );
```

To define the target tables in Cassandra to the following for each generated table definition:

```
#> cd <Cassandra installation directory>
#> bin/cqlsh < ~/ddl/bdglue.CUST_INFO.cql
#>
```

Validating Your Schema and Data

To check your schema in Cassandra, you simply do a "describe" as you would against a relational database:

```
cqlsh> describe bdglue.CUST_INFO;

CREATE TABLE bdglue.cust_info (
    id int PRIMARY KEY,
    city text,
    cust_date text,
    gender text,
    name text,
    old_id int,
    phone text,
    txoptype text,
    zip text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold':
'32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class':
'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';

cqlsh>
```

And to view some data:

```
cqlsh> select * from bdglue.CUST_INFO limit 2;

 id   | city               | cust_date  | gender | name         | old_id | phone
| txoptype | zip
------+-------------------+------------+--------+--------------+--------+------------
----+----------+-----------
 4460 | Le Grand-Quevilly | 2014/04/13 |   Male |    Dane Nash |      1 | (874) 373-
6196 |   INSERT | 81558-771
 4462 | Fontaine-l'Evique | 2015/02/06 |   Male | Amos Fischer |      3 | (141) 398-
6160 |   INSERT |      9188

(2 rows)
cqlsh>
```

Other Potential Targets

We have done some cursory examination and believe that publishers for other potential target technologies including Impala, MongoDB, Couchbase, Elastic Search, and others are feasible and might be developed in the future.

Source Configuration

The concept for BDGlue originated as we were building a Java-based adapter to Oracle GoldenGate to Big Data targets. It quickly became evident that BDGlue had the potential of being far more generally useful, however, so we made a deliberate effort to decouple the code that is used to tie BDGlue to a source from all of the downstream logic that interfaces with Big Data targets to encourage broader use of the solution.

However, BDGlue is of little use until it has been integrated with a data “source”. The data that is delivered to a Big Data target has to originate somewhere. While the “source” obviously comes first in any data pipeline, we have saved discussion of the source integrations until last in hope of truly decoupling the discussion related to configuring BDGlue from the discussion pertaining to configuring a source, whether GoldenGate at the present time, or some other source to be implemented in the future.

GoldenGate as a Source for BDGlue

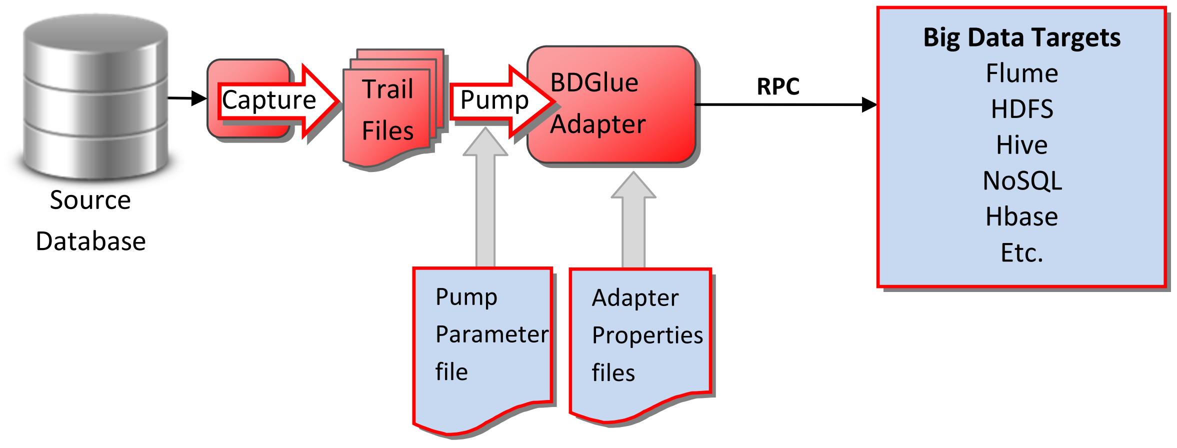
When linked with BDGlue, the GoldenGate Java Adapter becomes a fully functional GoldenGate Adapter that will deliver database operations that have been captured by Oracle GoldenGate from a relational database source into various target “Big Data” repositories and formats. Target repositories include HDFS, Hive, HBase, Oracle NoSQL, and others supported by BDGlue.

BDGlue is intended as a starting point for exploring Big Data architectures from the perspective of real-time change data capture (CDC) as provided by GoldenGate. As mentioned in the introduction, Hadoop and other Big Data technologies are by their very natures constantly evolving and infinitely configurable.

A GoldenGate Java Adapter is referred to as a “Custom Handler” in the GoldenGate Java Adapter documentation. This “custom handler” integration with BDGlue is developed using Oracle GoldenGate's Java API.

A custom handler is deployed as an integral part of an Oracle GoldenGate PUMP process. The PUMP and the custom handler are configured through a PUMP parameter file and the adapter's properties file. We will discuss the various properties files in more detail later in this document.

The PUMP process executes the adapter in its address space. The PUMP reads the trail file created by the Oracle GoldenGate EXTRACT process and passes the transactions into the adapter. Based on the configuration in the properties file, the adapter will write the transactions in one of several formats. Please refer to the *Oracle GoldenGate Adapters Administrator’s Guide for Java* (which can be found on http://docs.oracle.com) for details about the architecture and developing a custom adapter.



Configuring GoldenGate for BDGlue

There are three basic steps to getting GoldenGate properly configured to deliver data to BDGlue:

- Configure GoldenGate to capture the desired tables from the source database and write them to a trail file.
- Execute the “defgen” command to generate a sourcedefs file that defines the structure of the captured tables to the pump and Java Adapter.
- Configure the PUMP itself to reference the trail file, the sourcedefs file, and execute the Java Adapter.

Configure the GoldenGate EXTRACT

This User Guide makes no attempt to explain details of configuring GoldenGate itself. Please refer to the GoldenGate documentation for that information.

Simplistically speaking, however, a GoldenGate EXTRACT process has a parameter file that tells GoldenGate how to log into the source database to obtain table metadata, and what tables it should be concerned about capturing.

Here is a very basic example of a parameter file for connecting to a MySQL source database. The most important things to note are the tables we care about and the fact that there is nothing specific to configuration of the Java Adapter found there.

```
EXTRACT erdbms
DBOPTIONS HOST localhost, CONNECTIONPORT 3306
SOURCEDB bdgluedemo, USERID root, PASSWORD welcome1
EXTTRAIL ./dirdat/tc
GETUPDATEBEFORES
NOCOMPRESSDELETES
TRANLOGOPTIONS ALTLOGDEST /var/lib/mysql/log/bigdatalite-bin.index
TABLE bdgluedemo.MYCUSTOMER;
TABLE bdgluedemo.CUST_INFO;
TABLE bdgluedemo.TCUSTORD;
```

Please do make note of the parameters “GETUPDATEBEFORES” and “NOCOMPRESSDELETES”. If you think about it, in most cases it wouldn’t make sense to propagate a partial record downstream in the event of an update or a delete operation in the source database when dealing with Big Data targets. These parameters ensure that all columns are propagated downstream even if they are unchanged during an update operation on the source, and that all columns are propagated along with the key in the case of a delete.

Configure the GoldenGate PUMP

Unlike the EXTRACT, there are things specific to the Java Adapter found in the parameter file for the PUMP. This is because the Java Adapter is invoked by and runs as a part of the PUMP.

What follows is a simple PUMP parameter file. There are several things to note there:

- The CUSEREXIT statement which causes the PUMP to invoke code to run the Java Adapter we are providing. Note also on this statement the parameter “INCLUDEUPDATEBEFORES”. This ensures that the PUMP passes the “before image” of columns downstream along with the captured data.
- The actual tables that this PUMP will be capturing. Simplistically, this might be the same as the tables specified in the SOURCEDEFS file and in the EXTRACT, but in more complicated environments it is possible that we might configure multiple PUMPs, each handling a subset of the tables we are capturing.

```
extract ggjavaue
CUSEREXIT ./libggjava_ue.so CUSEREXIT PASSTHRU INCLUDEUPDATEBEFORES
TABLE bdgluedemo.MYCUSTOMER;
TABLE bdgluedemo.CUST_INFO;
TABLE bdgluedemo.TCUSTORD;
```

The Java Adapter itself has a properties file that has a bunch of configuration information that the Java Adapter needs to get going. Most of the information is fairly generic to the Java adapter itself and how it executes. The properties file resides in the GoldenGate “dirprm” directory along with the parameters for the various GoldenGate processes. Note that the name of the properties file is based on the name of the GoldenGate process it is associated with. In this case, the pump process is an instance of the Java Adapter called “ggjavaue”. The parameter file for the process is called “ggjavaue.prm”, and the properties file shown below would be called “bdglue.properties.”

There are a number of things to make specific note of:

- -Dbdglue.properties=./gghadoop/bdglue.properties (highlighted below). This defines a Java “system property” that the GoldenGate BDGlue “source” is looking for so that it can locate a properties file that is specific to what it needs to configure itself to run. If the system property is not defined, BDGlue will look for a file called bdglue.properties somewhere in a directory pointed to by the Java classpath. If the system property is used, calling the properties file “bdglue.properties” is not strictly required.
- gg.handlerlist=gghadoop gives a name to handler which is then used to identify properties to pass into it
- gg.handler.gghadoop.type=com.oracle.gghadoop.GG12Handler identifies the class that is the entry point into BDGlue. *[Note: as of this writing, the GG12Handler supports the GoldenGate 12.2 release of GoldenGate for Big Data.]*
- gg.handler.gghadoop.mode=op sets the Java Adapter to “operation mode” (rather than transaction mode) which is most appropriate for Big Data scenarios. Since all data in the trail file has been committed, this “eager” approach is far more efficient.
- gg.classpath=./gghadoop/lib/* points to a directory containing all of the Java dependencies for compiling and running BDGlue.

```
#Adapter Logging parameters.
#log.logname=ggjavaue
#log.tofile=true
log.level=INFO


#Adapter Check pointing  parameters
goldengate.userexit.chkptprefix=GGHCHKP_
goldengate.userexit.nochkpt=true

# Java Adapter Properties
goldengate.userexit.writers=javawriter
goldengate.userexit.utf8mode=true

# NOTE: bootoptions are all placed on a single line
javawriter.bootoptions= -Xms64m -Xmx512M
   -Dlog4j.configuration=ggjavaue-log4j.properties
   -Dbdglue.properties=./gghadoop/bdglue.properties
   -Djava.class.path=./gghadoop:./ggjava/ggjava.jar

#
```

```
#Properties for reporting statistics
# Minimum number of {records, seconds} before generating a report
javawriter.stats.time=3600
javawriter.stats.numrecs=5000
javawriter.stats.display=TRUE
javawriter.stats.full=TRUE

#Hadoop Handler.
gg.handlerlist=gghadoop
# the GG11Handler handler supports GoldenGate versions 11.2 and 12.1.2
gg.handler.gghadoop.type=com.oracle.gghadoop.GG12Handler
gg.handler.gghadoop.mode=op
# all dependent jar files should be placed here
gg.classpath=./gghadoop/lib/*
```

Here is a sample properties file that provides configuration properties for BDGlue:

```
# configure BDGlue properties
bdglue.encoder.threads = 3
bdglue.encoder.class = com.oracle.bdglue.encoder.JsonEncoder

bdglue.event.header-optype = true
bdglue.event.header-timestamp = true
bdglue.event.header-rowkey = true

bdglue.publisher.class = com.oracle.bdglue.publisher.flume.FlumePublisher
bdglue.publisher.threads = 2
bdglue.publisher.hash = rowkey

bdglue.flume.host = localhost
bdglue.flume.port = 5000
bdglue.flume.rpc.retries = 5
bdglue.flume.rpc.retry-delay = 10
```

We won’t go into details on the contents of this file here as they will vary a fair amount depending on what the target of BDGlue is: HDFS, Hive, HBase, NoSQL, etc. We’ll look at specific configurations in more detail subsequent sections of this document.

The “SchemaDef” Utility

SchemaDef is a java-based utility that connects to a source database via JDBC and generates metadata relevant to the BDGlue encoding process, the target repository, or both.

Running SchemaDef

The SchemaDef utility can be found in the jar file for BDGlue. Here is how you would run it:

```
DIR=/path/to/jars

CLASSPATH="$DIR/bdglue.jar"
CLASSPATH="$CLASSPATH:$DIR/slf4j-api-1.6.1.jar"
CLASSPATH="$CLASSPATH:$DIR/slf4j-simple-1.7.7.jar"
CLASSPATH="$CLASSPATH:$DIR/commons-io-2.4.jar"
CLASSPATH="$CLASSPATH:$DIR/jackson-core-asl-1.9.13.jar"
CLASSPATH="$CLASSPATH:$DIR/mysql-connector-java-5.1.34-bin.jar"

java –Dschemadef.properties=schemadef.properties -cp $CLASSPATH
                com.oracle.bdglue.utility.schemadef.SchemaDef
```

Note that the last jar file listed is specific to the database you will be connecting to. In this case, it is MySQL. Replace this jar file with the jar file that is appropriate for your database type and version. **NOTE: It is up to you to obtain the appropriate JDBC driver for your database platform and version, and to identify the appropriate connection URL and login credentials.**

Details on the various properties that can be configured for SchemaDef can be found in the appendix.

Generating Avro Schemas with SchemaDef

The first step in the process of generating binary encoded Avro data is to create the Avro schemas for the tables we want to capture. (Note that it is possible to let the BDGlue generate these schemas on the fly, but be aware that new schema files must be copied into HDFS before we start writing the Avro data there that corresponds to that version of the schema. The Avro (de)serialization process requires that the path to the schema file be included in the Avro event header information, and in turn it puts a copy of the schema in the meta data of each generated *.avro file. This allows various Big Data technologies (HDFS, Hive, and more) to always know the structure of the data, and in turn also allows Avro to support schema evolution on a table.

So, the best way to approach this is to generate the Avro schema files (*.avsc) and then copy them into place in HDFS before we start passing data through Flume to HDFS.

To facilitate generation of the Avro Schema files, we created a simple Java utility, SchemaDef, that parses connects to the source database via JDBC and generates the Avro schema files directly from the table definitions of the tables you specify.

Like everything else “java”, the utility is configured via a properties file that looks like this:

```
# jdbc connection information
schemadef.jdbc.driver = com.mysql.jdbc.Driver
schemadef.jdbc.url = jdbc:mysql://localhost/bdgluedemo

# Oracle JDBC connection info
#schemadef.jdbc.driver = oracle.jdbc.OracleDriver
#schemadef.jdbc.url = jdbc:oracle:thin:@//<host>:<port>/<service_name>

schemadef.jdbc.username = root
schemadef.jdbc.password = welcome1

# output format: avro, nosql, hive_avro
schemadef.output.format = avro
schemadef.output.path = ./avro

# encode numeric/decimal types as string, double, float
schemadef.numeric-encoding = double

schemadef.set-defaults = true
schemadef.tx-optype = false
schemadef.tx-timestamp = false
schemadef.user-token = false

# whitespace delimited list of schema.table pairs
schemadef.jdbc.tables = bdgluedemo.MYCUSTOMER bdgluedemo.CUST_INFO \
                     bdgluedemo.TCUSTORD
```

Details on the properties can be found in the Appendix.

Once the schema files have been created, you then need to copy them locally into HDFS. The following command will do that for you.

```
hdfs dfs -copyFromLocal -f ./output/*.avsc    /user/flume/gg-data/avro-schema
```

Generating Hive Table Definitions for Use with Avro Schemas

We can generate the Hive Table definitions (DDL) to read *.avro files written to HDFS by simply changing the output format from “avro” to “hive_avro” in the schemadef.properties file and rerun the utility.

```
# jdbc connection information
schemadef.jdbc.driver = com.mysql.jdbc.Driver
schemadef.jdbc.url = jdbc:mysql://localhost/bdgluedemo

# Oracle JDBC connection info
```

```
#schemadef.jdbc.driver = oracle.jdbc.OracleDriver
#schemadef.jdbc.url = jdbc:oracle:thin:@//<host>:<port>/<service_name>

schemadef.jdbc.username = root
schemadef.jdbc.password = welcome1

# output format: avro, nosql, hive_avro
schemadef.output.format = hive_avro
schemadef.output.path = ./avro

# encode numeric/decimal types as string, double, float
schemadef.numeric-encoding = double

schemadef.set-defaults = true
schemadef.tx-optype = false
schemadef.tx-timestamp = false
schemadef.user-token = false

schemadef.avro-url = hdfs:///user/flume/gg-data/avro-schema
schemadef.data-location = /user/flume/gg-data

# whitespace delimited list of schema.table pairs
schemadef.jdbc.tables = bdgluedemo.MYCUSTOMER bdgluedemo.CUST_INFO \
                     bdgluedemo.TCUSTORD
```

This will generate an *hql* file for each specified table that looks like this:

```
CREATE SCHEMA IF NOT EXISTS bdgluedemo;
USE bdgluedemo;
DROP TABLE IF EXISTS CUST_INFO;
CREATE EXTERNAL TABLE CUST_INFO
COMMENT "Table backed by Avro data with the Avro schema stored in HDFS"
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS
INPUTFORMAT  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/user/flume/gg-data/bdgluedemo.CUST_INFO/'
TBLPROPERTIES ( 'avro.schema.url'='hdfs:///user/flume/gg-data/avro-
schema/bdgluedemo.CUST_INFO.avsc' );
```

Generating Cassandra Table Definitions

SchemaDef is able to generate appropriate DDL that maps to the source tables for Cassandra just as it does for other targets. All you need to do is specify Cassandra as the target in the *schemadef.properties* file.

```
# jdbc connection information
# mysql
```

```
schemadef.jdbc.driver = com.mysql.jdbc.Driver
schemadef.jdbc.url = jdbc:mysql://localhost/bdglue
schemadef.jdbc.username = root
schemadef.jdbc.password = welcome1
#schemadef.jdbc.password = prompt
#
#schemadef.jdbc.driver = oracle.jdbc.OracleDriver
#schemadef.jdbc.url = jdbc:oracle:thin:@//<host>:<port>/<service_name>
#schemadef.jdbc.url = jdbc:oracle:thin:@//localhost:1521/orcl
#schemadef.jdbc.username = moviedemo
#schemadef.jdbc.password = welcome1


# output format: avro, nosql, hive_avro, cassandra
schemadef.output.format = cassandra
schemadef.output.path = ./ddl

schemadef.cassandra.replication-strategy = { 'class' : 'SimpleStrategy',
'replication_factor' : 1 }

schemadef.set-defaults = false
schemadef.tablename = false
schemadef.tx-optype = true
schemadef.tx-timestamp = false
schemadef.tx-position = false
schemadef.user-token = false

# whitespace delimited list of schema.table pairs
schemadef.jdbc.tables = bdglue.CUST_INFO     bdglue.MYCUSTOMER  \
                     bdglue.TCUSTORD bdglue.my$Table
```

This will generate a *cql* file for each table specified. A generated *cql* file will look like this:

```
CREATE KEYSPACE IF NOT EXISTS "bdglue"
    WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };

DROP TABLE IF EXISTS bdglue.CUST_INFO;

CREATE TABLE bdglue.CUST_INFO
 (
   txoptype text,
   ID int,
   NAME text,
   GENDER text,
   CITY text,
   PHONE text,
   OLD_ID int,
   ZIP text,
   CUST_DATE text,
   PRIMARY KEY (ID)
 );
```

BDGlue Developer’s Guide

This section will contain information related to building custom Encoders and Publishers.

Building a Custom Encoder

Information pertaining to building a custom Encoder will go here. Encoders are created by implementing the interface com.oracle.bdglue.encoder.BDGlueEncoder.

```
package com.oracle.bdglue.encoder;

import com.oracle.bdglue.meta.transaction.DownstreamOperation;

import java.io.IOException;

public interface BDGlueEncoder {
    /**
     * @param op
     * @return the encoded operation
     * @throws IOException
     */
    public EventData encodeDatabaseOperation(DownstreamOperation op) throws 
IOException;

    /**
     * @return the EncoderType for this encoder.
     */
    public EncoderType getEncoderType();
}
```

More specific details will follow in a future revision of this document. You can of course review the source code for examples.

Building a Custom Publisher

Information pertaining to building a custom Publisher will go here. Publishers are created by implementing the interface com.oracle.bdglue.publisher.BDGluePublisher.

```
package com.oracle.bdglue.publisher;

import com.oracle.bdglue.encoder.EventData;

public interface BDGluePublisher {
    /**
     * Connect to the target.
     */
    void connect();
    /**
     * Format the event and write it to the target.
     *
     * @param threadName the name of the calling thread.
     * @param evt the encoded event.
     */
    void writeEvent(String threadName, EventData evt);

    /**
     * Close connections and clean up as needed.
     */
    void cleanup();
}
```

More specific details will follow in a future revision of this document. You can of course review the source code for specific examples of how to implement a BDGlue publisher.

Prerequisite Requirements

Be sure you have taken care of the following before attempting to run the GoldenGate Java Adapter:

- Download, install and configure GoldenGate to capture from the source database.
    - Configure GoldenGate to capture all columns (uncompressed updates and deletes). This will some additional overhead to the capture process and require additional space in the trail files, but will eliminate the need to have to do any downstream reconciliation in the Hadoop environment later.
- Download, install, and configure the current version of GoldenGate for Big Data (version 12.2.x as of this writing).
    - This obviously requires Java to be installed and available in the GoldenGate environment. If it is not present, you will have to download and install it separately. The GoldenGate Java adapter requires Java SE 1.7 or later. BDGlue was built with Java SE 1.8. It is recommended that you use that version of Java. Refer to the documentation for the GoldenGate Java adapter and GoldenGate for Big Data for more information.
- Identify the target technology that you will be delivering data to, and ensure that the latest version of that technology has been installed and configured. You will likely need to know:
    - The host name and port number to which BDGlue will connect
    - The directory path where GoldenGate will write data if delivering to HDFS.
    - The directory path where we will place the Avro schema files in the HDFS environment if you will be configuring for Avro serialization.

Appendix

bdglue.properties

The following table lists the properties that can be specified in the *bdglue.properties* file.

Property	Required	Type	Default	Notes
bdglue.encoder.threads	No	Integer	2	The number of encoder threads to run in parallel.
bdglue.encoder.class	Yes	String	com.oracle.bdglue.encoder.JsonEncoder	The fully qualified class name (FQCN) of the class that will be called to encode the data. These Encoders, and any that are custom built, implement the interface com.oracle.bdglue.encoder.BDGlueEncoder. Built-in options are: • com.oracle.bdglue.encoder.AvroEncoder (encode in an Avro formatted byte array) • com.oracle.bdglue.encoder.AvroGenericRecordEncoder (encode an instance of an Avro GenericRecord) • com.oracle.bdglue.encoder.DelimtedTextEncoder (encode in delimited text format) • com.oracle.bdglue.encoder.JsonEncoder (encode in JSON format) • com.oracle.bdglue.encoder.NullEncoder (does not encode the data. This is used when the publisher will

Property	Required	Type	Default	Notes
				not pass along the data as encoded, and instead will apply the data to the target “column-by-column”. Example targets that approach things this way include HBase, Oracle NoSQL Table API, Cassandra, and others.
bdglue.encoder.delimiter	No	Integer	001	Default is ^A (001). Enter the numeric representation of the desired character (i.e. a semicolon is 073 in octal, 59 in decimal).
bdglue.encoder.tx-optype	No	Boolean	true	Include the transaction operation type in a column in the encoded data. Note that this configuration must match the corresponding property in the schemadef.properties file.
bdglue.encoder.tx-optype-name	No	String	txoptype	The name of the column to populate the operation type value in. Note that this configuration must match the corresponding property in the schemadef.properties file.
bdglue.encoder.tx-timestamp	No	Boolean	true	Include the transaction operation type in a column in the encoded data. Note that this configuration must match the corresponding property in the schemadef.properties file.

Property	Required	Type	Default	Notes
bdglue.encoder.tx-timestamp-name	No	String	txtimestamp	The name of the column to populate the transaction timestamp value in. Note that this configuration must match the corresponding property in the schemadef.properties file.
bdglue.encoder.tx-position	No	Boolean	true	Include information pertaining to the position of this operation in the transaction flow. This is used to allow sorting of operations when they are occurring more frequently than the granularity of the tx-timestamp.
bdglue.encoder.tx-position-name	No	String	txposition	The name of the column to populate the transaction position value in. Note that this configuration must match the corresponding property in the schemadef.properties file.
bdglue.encoder.user-token	No	Boolean	True	Populate a field that will contain a comma delimited list of any user tokens that accompany the record in the form of "token1=value, token2=value, …". This property must be the same as the corresponding property found for schemadef.
bdglue.encoder.user-token-name	No	String	usertokens	The name of the field that will contain the list of user-defined tokens. This property must be the same as the corresponding property found for schemadef.

Property	Required	Type	Default	Notes
bdglue.encoder.tablename	No	Boolean	False	Populate a field with the name of the source table. This will be the "long" table name in schema.table format.
bdglue.encoder.tablename-col	No	String	tablename	The name of the field to populate with the name of the source table.
bdglue.encoder.txid	No	Boolean	False	Populate a field with a transaction identifier.
bdglue.encoder.txid-col	No	String	txid	The name of the field to populate with the transaction identifier.
bdglue.encoder.replace-newline	No	Boolean	False	Replace newline characters found in string fields with another character. This is needed because newlines can cause problems in some downstream targets.
bdglue.encoder.newline-char	No	String	<space>	The character to substitute for newlines in string fields. The default is " " (a space). Override with another character if needed.
bdglue.encoder.json.text-only	No	Boolean	True	Whether or not to represent all column values as quoted text strings. When 'true', a numeric field would be represented as "ID":"789". When false, that same field would be represented as "ID":789, (no quotes around the value), which allows the downstream JSON parser to know to parse this as a number.

Property	Required	Type	Default	Notes
bdglue.encoder.include-befores	No	Boolean	False	Include the before images representation of all columns when encoding an operation. This option is only supported for JSON encoding at this time and will be ignored by other encoders.
bdglue.event.header-optype	No	Boolean	true	Include the operation type in the Flume event header
bdglue.event.header-timestamp	No	Boolean	true	Include the transaction timestamp in the Flume event header.
bdglue.event.header-rowkey	No	Boolean	true	Boolean as to whether or not to include a value for the row's key as a concatenation of the key columns in the event header information. HBase and NoSQL KV API need this. It is also needed if the publisher hash is based on key rather than table name.
bdglue.event.header-longname	No	Boolean	true	Boolean as to whether or not to include the "long" table name in the header. The long name is normally in the form of “schema.tablename”. FALSE will cause the "short" name (table name only) to be included. Most prefer the long name. HBase and NoSQL prefer the short name.
bdglue.event.header-columnfamily	No	Boolean	true	Boolean as to whether or not to include a "columnFamily" value in the header. This is needed for Hbase.

Property	Required	Type	Default	Notes
bdglue.event.header-avropath	No	Boolean	false	Boolean as to whether or not to include the path to the Avro schema file in the header. This is needed for Avro encoding where Avro-formatted files are created in HDFS, including those that will be leveraged by Hive.
bdglue.event.avro-hdfs-schema-path	No	String	hdfs:///user/flume/gg-data/avro-schema/	The URI in HDFS where Avro schemas can be found. This information is passed along as the header-avropath and is required by Flume when writing Avro-formatted files to HDFS.
bdglue.event.generate-avro-schema	No	Boolean	false	Boolean on whether or not to generate the avro schema on the fly. This is really intended for testing and should likely always be false. It might be useful at some point in the future to use to support Avro schema evolution. Note that current built-in schema generation capabilities are not on par with those in schemadef.
bdglue.event.avro-namespace	No	String	default	The namespace to use in avro schemas if the actual table schema name is not present. The table schema name will override.
bdglue.event.avro-schema-path	No	String	./gghadoop/avro	The path on local disk where we can find the avro schemas and/or where they will be written if we were to generate them on the fly.
bdglue.publisher.class	Yes	String	com.oracle.bdglue.publisher.console.ConsolePublisher	This is the fully qualified class name (FQCN) of the class that will be called to Publish the data. These Encoders, and any that are

Property	Required	Type	Default	Notes
				custom built, implement the interface com.oracle.bdglue.publisher.BDGluePublisher. Built-in options are: • com.oracle.bdglue.publisher.console.ConsolePublisher (writes the encoded data to the console. Useful for smoke testing upstream configurations before worrying about actually delivering data to a target. Json encoding is perhaps most useful for this. • com.oracle.bdglue.publisher.flume.FlumePublisher (delivers encoded data to Flume). • com.oracle.bdglue.publisher.hbase.HBasePublisher (delivers data to HBase. The NullEncoder should be used for this publisher). • com.oracle.bdglue.publisher.nosql.NoSQLPublisher (delivers to OracleNoSQL. Use the AvroEncoder for the KV API, and NullEncoder for the Table API). • com.oracle.bdglue.publisher.kafka.KafkaPublisher

Property	Required	Type	Default	Notes
				(delivers to Kafka. The AvroEncoder and JsonEncoder are perhaps most useful for this publisher). Note: this publisher uses an older Kafka API and is included for reasons of compatibility. • com.oracle.bdglue.publisher.kafka.KafkaRegistryPublisher (delivers to Kafka using the newer Kafka API. This publisher is also compatible with the Confluent "schema registry", although interfacing with the registry is not strictly required to use this publisher.) • com.oracle.bdglue.publisher.cassandra.CassandraPublisher (delivers data to Cassandra. The NullEncoder should be used for this publisher).
bdglue.publisher.threads	No	Integer	2	The number of publishers to run in parallel.

Property	Required	Type	Default	Notes
bdglue.publisher.hash	No	String	rowkey	Select the publisher thread to pass an encoded event to based on a hash of either the table name ("table") or row key ("rowkey"). This is to ensure that changes made to the same row are always handled by the same publisher to avoid any sort of race condition.
bdglue.nosql.host	No	String	localhost	The hostname that we will connect to for NoSQL
bdglue.nosql.port	No	String	5000	The port number where the NoSQL KVStore is listening.
bdglue.nosql.kvstore	No	String	kvstore	The name of the NoSQL KVStore to connect to.
bdglue.nosql.durability	No	String	WRITE_NO_SYNC	The NoSQL durability model for these transactions. Options are: • SYNC • WRITE_NO_SYNC • NO_SYNC
bdglue.nosql.api	No	String	kv_api	Specify whether to use the "kv_api" or "table_api" when writing to Oracle NoSQL.
bdglue.kafka.topic	No	String	goldengate	The name of the Kafka topic that GoldenGate will publish to.

Property	**Required**	**Type**	**Default**	**Notes**
bdglue.kafka.batchSize	No	Integer	100	The number of Kafka events to queue before publishing. The default value should be reasonable for most scenarios, but should be decreased to a smaller value for low volume situations, and perhaps made larger in extremely high volume situations. This property only applies to the KafkaPublisher as batching is handled by that publisher directly. Use bdglue.kafka.producer.batch.size for the KafkaRegistryPublisher as batching is handled by the actual Kafka producer logic in that case.
bdglue.kafka.flushFreq	No	Integer	500	The number of milliseconds to allow events to queue before forcing them to be written to Kafka in the event that 'batchSize' has not been reached.
bdglue.kafka.serializer.class	No	String	kafka.serializer.DefaultEncoder	The serializer to use when writing the event to Kafka. The DefaultEncoder passes the encoded data received verbatim to Kafka in a byte-for-byte fashion. It is not likely that there will be need to override the default value.
bdglue.kafka.key.serializer.class	No	String	kafka.serializer.StringEncoder	The serializer to use when encoding the Topic "key". It is not likely that the default value will need to be overridden.

Property	Required	Type	Default	Notes
bdglue.kafka.metadata.broker.list	Yes	String	localhost:9092	A comma-separated list of *host:port* pairs of Kafka brokers that may be published to. Note that this is for the Kafka broker, not for Zookeeper.
bdglue.kafka.metadata.helper.class	No	String	com.oracle.bdglue.publisher.kafka.KafkaMessageDefaultMeta	A simple class that implements the KafkaMessageHelper interface. Its purpose is to allow customization of message "topic" and message "key" behavior. Current built-in options: • com.oracle.bdglue.publisher.kafka.KafkaMessageDefaultMeta – writes all messages to a single topic specified in the properties file, and the key is the table name. • com.oracle.bdglue.publisher.kafka.KafkaMessageTableKey – publishes each table to a separate topic, where the topic name is the table name, and the message key is a concatenated version of the key columns from the table in this format: /key1/key2/…

Property	Required	Type	Default	Notes
bdglue.kafka.request.required.acks	No	Integer	1	0 – write and assume delivery. Don't wait for response (potentially unsafe). 1 – write and wait for the event to be accepted by at least one broker before continuing. -1 – write and wait for the event to be accepted by all brokers before continuing.
bdglue.cassandra.node	No	String	localhost	The Cassandra node to connect to.
bdglue.cassandra.batch-size	No	Integer	5	The number of operations to group together with each call to Cassandra.
bdglue.cassandra.flush-frequency	No	Integer	500	Force writing of any queued operations that haven't been flushed due to batch-size after this many milliseconds
bdglue.cassandra.insert-only	No	Boolean	false	Convert update and delete operations to an insert. Note that the default key generated by SchemaDef may need to be changed to include operation type and timestamp if this is set to 'true'.
bdglue.flume.host	Yes	String	localhost	The name of the target host that we will connect to.
bdglue.flume.port	Yes	Integer	5000	The port number on the host where the target is listening.
bdglue.flume.rpc.retries	No	Integer	5	The number of times to retry a connection after encountering an issue before aborting.
bdglue.flume.rpc.retry-delay	No	Integer	10	The number of seconds to delay after each attempt to connect before trying again.

Property	Required	Type	Default	Notes
bdglue.flume.rpc.type	No	String	avro-rpc	Currently only pertinent for Flume. Defines the type of event RPC protocol being used for communication. Options are avro-rpc and thrift-rpc. Avro is most common. Do not confuse avro RPC communication with avro encoding of data. Same name, different things entirely. One does not require the other.
schemadef.replace.invalid_char	No	String	_ (underscore)	Replace non-alphanumeric “special” characters that are supported in table and column names in some databases with the specified character or characters. This is needed because most of the big data targets are much more limited in terms of the characters that are supported. Note that this property begins with schemadef and should be identical to the property specified to the schemadef utility.
schemadef.replace.invalid_first_char	No	String	x (lower case x)	Prepend this string to table and column names that begin with anything other than an alpha character. This is needed because of limitations on the big data side of things. Set to a null value to avoid this functionality. Note that this property begins with schemadef and should be identical to the property specified to the schemadef utility.

Property	**Required**	**Type**	**Default**	**Notes**
schemadef.replace.regex	No	String	[^a-zA-Z0-9_\\.]	This is a regular expression that contains the characters that *are* supported in the target. (Note: the ^ is required just as in the default). All characters not in this list will be replaced by the character or characters specified in schemadef.replace.invalid_char. Note that this property begins with schemadef and should be identical to the property specified to the schemadef utility.

schemadef.properties

The following table lists the properties that can be specified in the *schemadef.properies* file.

Property	Required	Type	Default	Notes
schemadef.jdbc.driver	Yes	String	com.mysql.jdbc.Driver	The fully qualified class name of the jdbc driver.
schemadef.jdbc.url	Yes	String	jdbc:mysql://localhost /bdglue	The connection URL for JDBC
schemadef.jdbc.userna me	Yes	String	root	The database user that we will connect as.
schemadef.jdbc.passw ord	Yes	String	prompt	The database user's password. If this property is set to the value "prompt", SchemaDef will prompt the user to enter the password from the command line.
schemadef.jdbc.tables	Yes	String	N/A	A whitespace-delimited list of schema.table pairs that we should generate schema/ddl information for. More than one table may be specified per line, and a line may be continued by placing a backslash ('\') as the last character of the current line in the file.
schemadef.output.for mat	No	String	avro	The type of metadata / ddl to generate. Options are: *avro, hive_avro,* and *nosql*.
schemadef.output.path	No	String	./output	The directory where we should store the generated files.
schemadef.numeric-encoding	No	String	double	How to encode numeric, non-integer fields (decimal, numeric types) in the schema: *string*, *double*, *float*.
schemadef.set-defaults	No	Boolean	true	Whether or not to set default values in the generated Avro schema.
schemadef.tx-optype	No	Boolean	true	Include the transaction operation type in a column in the encoded data. Note that this configuration must match the corresponding property in the bdglue.properties file.

Property	Required	Type	Default	Notes
schemadef.tx-optype-name	No	String	txoptype	The name of the column to populate the operation type value in. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.tx-timestamp	No	Boolean	true	Include the transaction operation type in a column in the encoded data. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.tx-timestamp-name	No	String	txtimestamp	The name of the column to populate the transaction timestamp value in. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.tx-position	No	Boolean	true	Include details of the operation's position in the replication flow in a column in the encoded data to allow sorting when transactions are occurring more rapidly than the granularity of the transaction timestamp can support. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.tx-position-name	No	String	txposition	The name of the column to populate the transaction position information in. Note that this configuration must match the corresponding property in the bdglue.properties file.

Property	**Required**	**Type**	**Default**	**Notes**
schemadef.user-token	No	Boolean	true	Populate a field that will contain a comma delimited list of any user tokens that accompany the record in the form of “token1=value, token2=value, …”. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.user-token-name	No	String	usertokens	The name of the field that will contain the list of user-defined tokens. Note that this configuration must match the corresponding property in the bdglue.properties file.
schemadef.tablename	No	Boolean	false	Populate a field that will contain the long version of the table name (schema.table format).
schemadef.tablename-col	No	String	tablename	The name of the field that will contain the table name.
schemadef.txid	No	Boolean	false	Populate a field that will contain a transaction identifier.
schemadef.txid-col	No	String	txid	The name of the field that will contain the transaction identifier.
schemadef.avro-url	No	String	/path/to/avro/schema	Tells the Hive Avro SerDe where to find the avro schema for this table. Required for avro_hive schema generation
schemadef.data-location	No	String	/path/to/avro/data	Tells the Hive Avro SerDe where to find the avro-encoded data files for this table. Required for avro_hive schema generation.
schemadef.cassandra.replication-strategy	No	String	{ 'class' : 'SimpleStrategy', 'replication_factor' : 1 }	The replication strategy for the table. Note that this string is passed into SchemaDef and the corresponding CQL that is generated verbatim … it must be syntactically correct.

Property	Required	Type	Default	Notes
schemadef.replace.invalid_char	No	String	_ *(underscore)*	Replace non-alphanumeric "special" characters that are supported in table and column names in some databases with the specified character or characters. This is needed because most of the big data targets are much more limited in terms of the characters that are supported. This value must be the same as the value specified for the equivalent property in bdglue.properties.
schemadef.replace.invalid_first_char	No	String	x	Prepend this string to table and column names that begin with anything other than an alpha character. This is needed because of limitations on the big data side of things. Set to a null value to avoid this functionality. This value must be the same as the value specified for the equivalent property in bdglue.properties.
schemadef.replace.regex	No	String	[^a-zA-Z0-9_\\.]	This is a regular expression that contains the characters that *are* supported in the target. (Note: the ^ is required just as in the default). All characters not in this list will be replaced by the character or characters specified in schemadef.replace.invalid_char. This value must be the same as the value specified for the equivalent property in bdglue.properties.

Helpful Reference Sources

- Flume Developer Guide: https://flume.apache.org/FlumeDeveloperGuide.html
- Flume User Guide: https://flume.apache.org/FlumeUserGuide.html
- Hoffman, Steve. *Apache Flume Distributed Log Collection for Hadoop*. N.p.: Packt, 2013.
- White, Tom. *Hadoop: The Definitive Guide: (fourth edition)*. Beijing: O'Reilly, 2015.
- Oracle NoSQL Documentation: http://docs.oracle.com/cd/NOSQL/html/index.html

License and Notice Files

LICENSE

===============
async-1.4.0.jar and subsequent versions
===============

NOTICE

======================
kafka_2.10-0.8.2.1.jar and subsequent versions
======================

Kafka

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes jopt-simple, a library for parsing command line options (http://jopt-simple.sourceforge.net/).

This product includes junit, developed by junit.org.

This product includes zkclient, developed by Stefan Groschupf, http://github.com/sgroschupf/zkclient

This produce includes joda-time, developed by joda.org (joda-time.sourceforge.net)

This product includes the scala runtime and compiler (www.scala-lang.org) developed by EPFL, which includes the following license:

This product includes zookeeper, a Hadoop sub-project (http://hadoop.apache.org/zookeeper)

This product includes log4j, an Apache project (http://logging.apache.org/log4j)

This product includes easymock, developed by easymock.org (http://easymock.org)

This product includes objenesis, developed by Joe Walnes, Henri Tremblay, Leonardo Mesquita (http://code.google.com/p/objenesis)

==============
avro-1.7.7.jar and subsequent versions
==============

==========================
jackson-core-asl-1.9.13.jar and subsequent versions
==========================

# Jackson JSON processor

Jackson is a high-performance, Free/Open Source JSON processing library.
It was originally written by Tatu Saloranta (tatu.saloranta@iki.fi), and has
been in development since 2007.
It is currently developed by a community of developers, as well as supported
commercially by FasterXML.com.

## Licensing

Jackson core and extension components may licensed under different licenses.
To find the details that apply to this artifact see the accompanying LICENSE file.
For more information, including possible other licensing options, contact
FasterXML.com (http://fasterxml.com).

## Credits

A list of contributors may be found from CREDITS file, which is included
in some artifacts (usually source distributions); but is always available
from the source code management (SCM) system project uses.