Dbvisit Replicate Connector for Kafka documentation

Release 2.9.00-SNAPSHOT

Dbvisit Software Limited

Aug 02, 2017

Contents

Contents:

CHAPTER 1

Dbvisit Replicate Connector for Kafka

The **Dbvisit Replicate Connector for Kafka** is a SOURCE connector for the Kafka Connect utility. Kafka Connect, as a tool, makes it easy to get data in and out of Kafka. It is open source software and can be downloaded from the Apache Kafka project site, or simply and conveniently run within the Confluent Platform.

This SOURCE connector enables you to stream DML change data records (inserts, deletes, updates) from an Oracle database, which have been made available in PLOG (parsed log) files generated by the Dbvisit Replicate application, through into Kafka topics.

You can find the Dbvisit Replicate Connector for Kafka project on GitHub, and which you can build from source, or you can download the prebuilt bundle including the Connector and OSFM JARs and properties file, from this location.

Overview

The Oracle Source

Change data is identified and captured on the Oracle RDBMS by the Dbvisit Replicate application, using proprietary technology to mine the online redo logs in real time. Committed changes (pessimistic commit mode) made on the Oracle database are streamed in real time, in a custom binary format called a PLOG file, to a location where the **Dbvisit Replicate Connector for Kafka** picks them up, processes and ingests to Kafka topics. Note that only changes made to tables or schemas you are interested in listening for, and which have been configured to be included in the replication, are delivered to the PLOG files.

Please refer to the Dbvisit Replicate online user guide for information and instructions on setting and configuring that application to produce and deliver the PLOG file stream.

The Kafka Target

The **Dbvisit Replicate Connector for Kafka** polls the directory where the PLOGs will be delivered, picking up and streaming the changes identified in these files into Kafka, via the Kafka Connect framework.

By default all changes for a table are delivered to a single topic in Kafka. Topics are automatically generated, with a single partition. Topics can be pre-created if you so desire, but it is important to note that the mapping is as follows:

```
Oracle table -> Kafka topic
```

Note: Each topic receiving change data event messages from Oracle should always be created with a single partition if retaining the global change ordering from the source in Kafka is important for your use of the data on that side.

In addition to this the **Dbvisit Replicate Connector for Kafka** will also automatically create and write to a meta-data topic (the name of which is `TX.INFO` by default, and can be configured in `topic.name.transaction.info`) in Kafka, which lists out Oracle transaction information from across all the tables that changes have been configured to listen for. This can be utilized by Kafka consumers to reconstruct the precise global ordering of changes, across the various topics, as they occurred in order on the Oracle database.

The **Dbvisit Replicate Connector for Kafka** works with the open source Avro converters and Schema Registry metadata service, provided by Confluent, to govern the shape (and evolution) of the messages delivered to Kafka. This is a natural fit when working with highly structured RDBMS data, and the recommended approach for deployment.

Quickstart

To demonstrate the operation and functionality of the connector, we have provided a couple of example PLOG file sets. These were generated by running the Swingbench load generation utility, against an Oracle XE 11g database, and extracting the changes with Dbvisit Replicate 2.9.00. The PLOGs containing a smaller dataset can be downloaded from this location, and contain the following number of change records, across the tables included in the replication:

```
SOE.CUSTOMERS:     Mine:364
SOE.ADDRESSES:     Mine:364
SOE.CARD_DETAILS:  Mine:357
SOE.ORDER_ITEMS:   Mine:2853
SOE.ORDERS:        Mine:2356
SOE.INVENTORIES:   Mine:2755
SOE.LOGON:         Mine:2749
```

This corresponds to the following number of transactions recorded by the Connector in the metadata topic in Kafka:

```
TX.META:           2880
```

A set of PLOGs containing a larger dataset, and which also utilises the LOAD function, can be downloaded from the location. This contains the following number of change records, across the tables included in the replication:

```
SOE.CUSTOMERS:             Mine:218944
SOE.ADDRESSES:             Mine:220034
SOE.CARD_DETAILS:          Mine:219006
SOE.WAREHOUSES:            Mine:1000
SOE.ORDER_ITEMS:           Mine:1497866
SOE.ORDERS:                Mine:582198
SOE.INVENTORIES:           Mine:899874
SOE.PRODUCT_INFORMATION:   Mine:1000
SOE.LOGON:                 Mine:1596723
SOE.PRODUCT_DESCRIPTIONS:  Mine:1000
SOE.ORDERENTRY_METADATA :  Mine:4
```

This corresponds to the following number of transactions recorded by the Connector in the metadata topic in Kafka:

```
TX.META:          606
```

You can download the Dbvisit Replicate Connector properties file (that you can also see on GitHub), which contains sensible starting configuration parameters, from this location.

Using these example files as a starting point means that you do not have to setup and configure the Dbvisit Replicate application to produce a stream of PLOG files. This will enable you to get the Dbvisit Replicate Connector for Kafka up and running quickly. From there you can see it ingest Oracle change data to Kafka, and view via consumers, or route to some other end target. Of course this limited change set means that you will not see new changes flowing through from an Oracle source once this dataset has been processed - but it is a good place to begin in terms of understanding the connector functionality and operation.

To move beyond the Quickstart please refer to the Dbvisit Replicate online user guide for information and instructions on setting and configuring that application to produce and deliver the PLOG file stream.

We also recommend reviewing the Confluent Kafka Connect Quickstart guide which is an excellent reference in terms of understanding source/sink data flows and providing background context for Kafka Connect itself.

Once the Zookeeper, Kafka server and Schema Registry processes have been initiated, start the Replicate Connector, running in Kafka Connect in standalone mode. This will then ingest and process the PLOG files, writing the change data record messages to Kafka. These can be viewed on the other side with an Avro consumer provided with the Confluent Platform, or the default JSON consumer in the Kafka Connect framework.

Steps

1. Download the Confluent Platform

```
The only requirement is Oracle Java >= 1.7. Java installation
#Download the software from the Confluent website, version 3.x
#Install onto your test server: i.e: /usr/confluent
 unzip confluent-3.1.1-2.11.zip
```

2. Install the Replicate Connector JAR file

```
#Create the following directory
 mkdir $CONFLUENT_HOME/share/java/kafka-connect-dbvisit
#Build the Replicate Connector JAR file from the Github Repo (or download as per␣
↪instructions above)
#Install the JAR file to the location just created above
```

3. Install the Replicate Connector "Quickstart" properties file

```
#Create the following directory
 mkdir $CONFLUENT_HOME/etc/kafka-connect-dbvisit
#Install the Quickstart properties file (download link above) to the location just␣
↪created
```

Note: When working with the Connector JAR bundle `dbvisit_replicate_connector_for_kafka-2.9.00-linux_x86_64-jar.zip` (see download link above) extracting the zip file in your CONFLUENT_HOME directory will create these directories for you, and move the files (including OSFM) into place. For example:

```
[oracle@dbvrep01 confluent-3.2.1]$ unzip dbvisit_replicate_connector_for_kafka-2.9.00-
↪linux_x86_64-jar.zip
Archive:  dbvisit_replicate_connector_for_kafka-2.9.00-linux_x86_64-jar.zip
creating: share/java/kafka-connect-dbvisit/
```

```
inflating: share/java/kafka-connect-dbvisit/kafka-connect-dbvisitreplicate-2.9.00.jar
inflating: share/java/kafka-connect-dbvisit/jackson-core-2.7.1.jar
inflating: share/java/kafka-connect-dbvisit/jackson-databind-2.7.1.jar
inflating: share/java/kafka-connect-dbvisit/jackson-annotations-2.7.1.jar
inflating: share/java/kafka-connect-dbvisit/replicate-connector-lib-2.9.00.jar
inflating: share/java/kafka-connect-dbvisit/slf4j-log4j12-1.7.6.jar
inflating: share/java/kafka-connect-dbvisit/log4j-1.2.17.jar
inflating: share/java/kafka-connect-dbvisit/kafka-connect-dbvisit-admin-2.9.00.jar
inflating: share/java/kafka-connect-dbvisit/commons-cli-1.3.1.jar
creating: share/doc/kafka-connect-dbvisit/
inflating: share/doc/kafka-connect-dbvisit/NOTICE
inflating: share/doc/kafka-connect-dbvisit/LICENSE
creating: etc/kafka-connect-dbvisit/
inflating: etc/kafka-connect-dbvisit/dbvisit-replicate.properties
inflating: bin/run-admin-class.sh
inflating: bin/obsolete-source-file-manager.sh
```

4. Work with the example PLOG files

```
#Create a directory to hold the example PLOG files, e.g:
 mkdir /usr/dbvisit/replicate/demo/mine
#Upload and unzip the example PLOG files (download links for small and large datasets␣
↪provided above) to the location just created
#Edit the plog.location.uri parameter in the Quickstart dbvisit-replicate.properties␣
↪example configuration file to point to the location where the example PLOG files␣
↪are located: e.g;
 plog.location.uri=file:/usr/dbvisit/replicate/demo/mine
```

5. Start the Zookeeper, Kafka and Schema Registry processes

```
#Start Zookeeper
 $CONFLUENT_HOME/bin/zookeeper-server-start -daemon $CONFLUENT_HOME/etc/kafka/
↪zookeeper.properties
#Start Kafka
 $CONFLUENT_HOME/bin/kafka-server-start -daemon $CONFLUENT_HOME/etc/kafka/server.
↪properties
#Start the Schema Registry
 $CONFLUENT_HOME/bin/schema-registry-start -daemon $CONFLUENT_HOME/etc/schema-
↪registry/schema-registry.properties
#Start the REST Proxy (optional)
 $CONFLUENT_HOME/bin/kafka-rest-start -daemon $CONFLUENT_HOME/etc/kafka-rest/kafka-
↪rest.properties
```

Note: This default configuration is run on a single server with local Zookeeper, Kafka, Schema Registry and REST Proxy services.

As an alternative, for ease of use, these commands can be wrapped in a script and then invoked to start the processes. Name and save this script to a location of your choice, being sure to set CONFLUENT_HOME correctly within it. Note that version 3.3.0 of the Confluent Platform now comes with a 'CLI utility <http://docs.confluent.io/current/quickstart.html#clihelp>'_ in order to simplify service startup and management, and this is worth exploring as an alternative when running versions > 3.3.0.

```
#! /bin/bash

echo $(hostname)
CONFLUENT_HOME=/usr/confluent/confluent-3.1.1
```

```
echo "INFO Starting Zookeeper"
$CONFLUENT_HOME/bin/zookeeper-server-start -daemon $CONFLUENT_HOME/etc/kafka/
↪zookeeper.properties
sleep 10

echo "INFO Starting Kafka Server"
$CONFLUENT_HOME/bin/kafka-server-start -daemon $CONFLUENT_HOME/etc/kafka/server.
↪properties
sleep 10

echo "INFO Starting Schema Registry"
$CONFLUENT_HOME/bin/schema-registry-start -daemon $CONFLUENT_HOME/etc/schema-registry/
↪schema-registry.properties
#sleep 10

echo "INFO Starting REST Proxy"
$CONFLUENT_HOME/bin/kafka-rest-start -daemon $CONFLUENT_HOME/etc/kafka-rest/kafka-
↪rest.properties
sleep 10
```

And run this as follows:

```
./kafka-init.sh
```

6. Run Kafka Connect, and the Replicate Connector

To run the Replicate Connector in Kafka Connect standalone mode open another terminal window to your test server and execute the following from your CONFLUENT_HOME location:

```
./bin/connect-standalone ./etc/schema-registry/connect-avro-standalone.properties ./
↪etc/kafka-connect-dbvisit/dbvisit-replicate.properties
```

You should see the process start up, log some messages, and locate and begin processing PLOG files. The change records will be extracted and written in batches, sending the results through to Kafka.

7. View the messages in Kafka with the default Consumer utilities

Default Kafka consumers (clients for consuming messages from Kafka) are provided in the Confluent Platform for both Avro and Json encoding, and they can be invoked as follows:

```
./bin/kafka-avro-console-consumer --new-consumer --bootstrap-server localhost:9092 --
↪topic SOE.CUSTOMERS --from-beginning

{"XID":"0000.68d6.00000002","TYPE":"INSERT","CHANGE_ID":1021010014941,"CUSTOMER_ID
↪":205158,"CUST_FIRST_NAME":"connie","CUST_LAST_NAME":"prince","NLS_LANGUAGE":{
↪"string":"th"},"NLS_TERRITORY":{"string":"THAILAND"},"CREDIT_LIMIT":{"bytes":
↪"\u0006´\u0004"},"CUST_EMAIL":{"string":"connie.prince@oracle.com"},"ACCOUNT_MGR_ID
↪":{"long":158},"CUSTOMER_SINCE":{"long":1477566000000},"CUSTOMER_CLASS":{"string":
↪"Occasional"},"SUGGESTIONS":{"string":"Music"},"DOB":{"long":247143600000},"MAILSHOT
↪":{"string":"Y"},"PARTNER_MAILSHOT":{"string":"N"},"PREFERRED_ADDRESS":{"long
↪":205220},"PREFERRED_CARD":{"long":205220}}
```

This expected output shows the SOE.CUSTOMERS table column data in the JSON encoding of the Avro records. The JSON encoding of Avro encodes the strings in the format `{"type": value}`, and a column of type `STRING` can be `NULL`. So each row is represented as an Avro record and each column is a field in the record. Included also are the Transaction ID (XID) that the change to this particular record occurred in, the TYPE of DML change made (insert, delete or update), and the specific CHANGE_ID as recorded for this in Dbvisit Replicate.

Note: To use JSON encoding and the JSON consumer please see our notes on JsonConverter settings later in this guide.

If there are more PLOGS to process you should see changes come through the consumers in real-time, and the following "Processing PLOG" messages in the Replicate Connector log file output:

```
[2016-12-03 09:28:13,557] INFO Processing PLOG: 1695.plog.1480706183 (com.dbvisit.
↪replicate.kafkaconnect.ReplicateSourceTask:587)
[2016-12-03 09:28:17,517] INFO Reflections took 22059 ms to scan 265 urls, producing␣
↪14763 keys and 113652 values  (org.reflections.Reflections:229)
[2016-12-03 09:29:04,836] INFO Finished WorkerSourceTask{id=dbvisit-replicate-0}␣
↪commitOffsets successfully in 9 ms (org.apache.kafka.connect.runtime.
↪WorkerSourceTask:356)
[2016-12-03 09:29:04,838] INFO Finished WorkerSourceTask{id=dbvisit-replicate-1}␣
↪commitOffsets successfully in 1 ms (org.apache.kafka.connect.runtime.
↪WorkerSourceTask:356)
[2016-12-03 09:29:04,839] INFO Finished WorkerSourceTask{id=dbvisit-replicate-2}␣
↪commitOffsets successfully in 1 ms (org.apache.kafka.connect.runtime.
↪WorkerSourceTask:356)
[2016-12-03 09:29:04,840] INFO Finished WorkerSourceTask{id=dbvisit-replicate-3}␣
↪commitOffsets successfully in 1 ms (org.apache.kafka.connect.runtime.
↪WorkerSourceTask:356)
```

Ctrl-C to stop the consumer processing further, and which will then show a count of how many records (messages) the consumer has processed:

```
^CProcessed a total of 156 messages
```

You can then start another consumer session as follows (or alternatively use a new console window), to see the changes delivered to the TX.META topic, which contains the meta-data about all the changes made on the source.

```
 ./bin/kafka-avro-console-consumer --new-consumer --bootstrap-server localhost:9092 --
↪topic TX.META --from-beginning

 {"XID":"0000.68d9.00000000","START_SCN":24893566,"END_SCN":24893566,"START_TIME
↪":1479626569000,"END_TIME":1479626569000,"START_CHANGE_ID":1060010003361,"END_
↪CHANGE_ID":1060010003468,"CHANGE_COUNT":100,"SCHEMA_CHANGE_COUNT_ARRAY":[{"SCHEMA_
↪NAME":"SOE.WAREHOUSES","CHANGE_COUNT":100}]}
```

In this output we can see details relating to specific transactions (XID) including the total CHANGE_COUNT made within this to tables we are interested in, and these are then cataloged for convenience in SCHEMA_CHANGE_COUNT_ARRAY.

Features

Dbvisit Replicate Connector supports the streaming of Oracle database change data with a variety of Oracle data types, varying batch sizes and polling intervals, the dynamic addition/removal of tables from a Dbvisit Replicate configuration, and other settings.

When beginning with this connector the majority of the default settings will be more than adequate to start with, although `plog.location.uri`, which is where PLOG files will be read from, will need to be set according to your system and the specific location for these files.

All the features of Kafka Connect, including offset management and fault tolerance, work with the Replicate Connector. You can restart and kill the processes and they will pick up where they left off, copying only new data.

CDC Format

Two types of change data capture format are supported for Kafka messages. See the configuration option `connector.publish.cdc.format`

Change Row publishing

`connector.publish.cdc.format=changerow`

Dbvisit Replicate Connector will attempt to assemble a complete view of the row record, based on the information made available in a PLOG, once the change has been made and committed on the source. This is done by merging the various components of the change into one complete record that conforms to an Avro schema definition, which itself is a verbatim copy of the Oracle source table definition. This includes merging LOB change vectors, emitted as separate change records, to its data counterpart to ensure the complete record view.

- INSERT - NEW fields
- UPDATE - NEW and OLD fields merged into one record
- DELETE - OLD fields

When a Schema Registry is used to perform schema validation all records must conform to its source schema definition. This includes DELETEs which publishes the record's last known state prior to deletion.

This type of change record is useful when the latest version of the data is all that's needed, irrespective of the change vector. However with state-full stream processing the change vectors are implicit and can be easily extracted.

`For the 2.8.04 version the only mode of operation supported was to publish complete change rows to a unique topic per table.`

To illustrate we create a simple table on the Oracle source database, as follows, and perform an insert, update and delete:

```
create table SOE.TEST2 (
user_id number (6,0),
user_name varchar2(100),
user_role varchar2(100));
```

The default Kafka Connect JSON consumer can be invoked as follows (see the notes below on JSON encoding). Note that using the default Avro encoding with the supplied Avro consumers produces output that does not include the JSON schema information, and effectively begins from XID as follows:

```
[oracle@dbvrep01 confluent-3.1.1]$ ./bin/kafka-console-consumer --new-consumer --
↪bootstrap-server localhost:9092 --topic SOE.TEST2 --from-beginning
```

Inserts

insert into SOE.TEST2 values (1, 'Matt Roberts', 'Clerk'); commit;

```
{"schema":{"type":"struct","fields":[{"type":"string","optional":false,"field":"XID"},
↪{"type":"string","optional":false,"field":"TYPE"},{"type":"int64","optional":false,
↪"field":"CHANGE_ID"},{"type":"int32","optional":true,"field":"USER_ID"},{"type":
↪"string","optional":true,"field":"USER_NAME"},{"type":"string","optional":true,
↪"field":"USER_ROLE"}],"optional":false,"name":"REP-SOE.TEST2"},"payload":{"XID":
↪"0003.014.00009447","TYPE":"INSERT","CHANGE_ID":1416010010667,"USER_ID":1,"USER_NAME
↪":"Matt Roberts","USER_ROLE":"Clerk"}}
```

Updates

update SOE.TEST2 set user_role = 'Senior Partner' where user_id=1; commit;


```
{"schema":{"type":"struct","fields":[{"type":"string","optional":false,"field":"XID"},
↪{"type":"string","optional":false,"field":"TYPE"},{"type":"int64","optional":false,
↪"field":"CHANGE_ID"},{"type":"int32","optional":true,"field":"USER_ID"},{"type":
↪"string","optional":true,"field":"USER_NAME"},{"type":"string","optional":true,
↪"field":"USER_ROLE"}],"optional":false,"name":"REP-SOE.TEST2"},"payload":{"XID":
↪"0004.012.00007357","TYPE":"UPDATE","CHANGE_ID":1417010001808,"USER_ID":1,"USER_NAME
↪":"Matt Roberts","USER_ROLE":"Senior Partner"}}
```


Note that a complete row is represented as a message delivered to Kafka. This is obtained by merging the existing and changed values to produce the current view of the record as it stands.

Deletes

delete from SOE.TEST2 where user_id=1; commit;


```
{"schema":{"type":"struct","fields":[{"type":"string","optional":false,"field":"XID"},
↪{"type":"string","optional":false,"field":"TYPE"},{"type":"int64","optional":false,
↪"field":"CHANGE_ID"},{"type":"int32","optional":true,"field":"USER_ID"},{"type":
↪"string","optional":true,"field":"USER_NAME"},{"type":"string","optional":true,
↪"field":"USER_ROLE"}],"optional":false,"name":"REP-SOE.TEST2"},"payload":{"XID":
↪"0007.01b.000072b4","TYPE":"DELETE","CHANGE_ID":1418010000537,"USER_ID":1,"USER_NAME
↪":"Matt Roberts","USER_ROLE":"Senior Partner"}}
```


Note that the detail for a delete shows the row values as they were at the time this operation was performed.

Change Set publishing

`connector.publish.cdc.format=changeset`

This option provides the change set as separate fields in the Kafka message, and no attempt is made to merge the fields into one consistent view. The message payload uses a Map with the following keys, each of which uses the same optional schema definition:

- key - supplementally logged key fields for the change (similar to a WHERE clause)
- old - the unchanged or previous message fields
- new - the new values for message fields
- lob - all changes to LOB fields are emitted separately

For change records each DML action will result in different change sets (examples include transaction fields):

- INSERT - new and/or lob (no key or old)


```
{
   "XID":"0007.00b.000002a2",
   "TYPE":"INSERT",
   "CHANGE_ID":21010000011,
   "CHANGE_DATA":{
      "NEW":{
         "ID":{
            "int":1
         },
```

```
            "TEST_NAME":{
               "string":"TEST INSERT"
            }
         }
      }
}
```

- UPDATE - key, old, new and/or lob

```
{
   "XID":"0003.015.000003bb",
   "TYPE":"UPDATE",
   "CHANGE_ID":22010000008,
   "CHANGE_DATA":{
      "NEW":{
         "ID":null,
         "TEST_NAME":{
            "string":"TEST UPDATE"
         }
      },
      "KEY":{
         "ID":{
            "int":1
         },
         "TEST_NAME":{
            "string":"TEST INSERT"
         }
      }
   }
}
```

- DELETE - key and/or old

```
{
   "XID":"0005.007.00000395",
   "TYPE":"DELETE",
   "CHANGE_ID":23010000008,
   "CHANGE_DATA":{
      "KEY":{
         "ID":{
            "int":1
         },
         "TEST_NAME":{
            "string":"TEST UPDATE"
         }
      }
   }
}
```

Message Keys

`connector.publish.keys=true`

Publishes the values for columns recorded as part of a key constraint (either primary or unique key) in the source Oracle table as the key values for a Kafka message. When no key constraint exists in the source table it will publish all columns, excluding LOB or RAW fields, as the key values. However it is recommended to create a primary or unique key in the source table instead.

Message Timestamps

All Kafka data messages (not aggregate transaction messages) publish the time stamps of their source change record as the message timestamps. This is not the timestamp of when the message was published to Kafka, but when the change was originally recorded in the source system.

Metadata Topic

`connector.publish.transaction.info=true`

Dbvisit Replicate Connector for Kafka automatically creates and writes a meta-data topic which lists out the Transactions (TX), and an ordered list of the changes contained with these. This can be utilized/cross-referenced within consumers or applications to reconstruct change ordering across different tables, and manifested in different topics. This is a means of obtaining an authoritative "global" view of the change order, as they occurred on the Oracle database, as may be important in specific scenarios and implementations.

So the output of a TX meta data record is as follows:

```
{"XID":"0002.019.00008295","START_SCN":24914841,"END_SCN":24914850,"START_TIME
↪":1479630708000,"END_TIME":1479630710000,"START_CHANGE_ID":1066010000608,"END_
↪CHANGE_ID":1066010000619,"CHANGE_COUNT":1,"SCHEMA_CHANGE_COUNT_ARRAY":[{"SCHEMA_NAME
↪":"SOE.TEST2","CHANGE_COUNT":1}]}
```

Explanation:

1. **XID**: Transaction ID from the Oracle RDBMS
2. **START_SCN**: SCN of first change in transaction
3. **END_SCN**: SCN of last change in transaction
4. **START_TIME**: Time when transaction started
5. **END_TIME**: Time when transaction ended
6. **START_CHANGE_ID**: ID of first change record in transaction
7. **END_CHANGE_ID**: ID of last change record in transaction
8. **CHANGE_COUNT**: Number of data change records in transaction, not all changes are row level changes
9. **SCHEMA_CHANGE_COUNT_ARRAY: Number of data change records for each replicated table in the transaction, as a**

    (a) **SCHEMA_NAME**: Replicated table name (referred to as schema name because each table has their own Avro schema definition)

    (b) **CHANGE_COUNT**: Number of data records changed for table

Corresponding to this, each data message in all replicated table topics contain three additional fields in their payload, for example:

```
{"XID":"0003.007.00008168","TYPE":"INSERT","CHANGE_ID":1064010025000,"USER_ID":{"bytes
↪":"\u0000"},"USER_NAME":{"string":"Matt Roberts"},"USER_ROLE":{"string":"Clerk"}}
```

This allows linking it to the transaction meta data topic which holds the following transaction information aggregated from individual changes:

1. **XID**: Transaction ID - its parent transaction identifier
2. **TYPE**: the type of action that resulted in this change row record, eg. INSERT, UPDATE or DELETE

3. **CHANGE_ID**: its unique change ID in the replication

Topic Per Table

Data from each replicated table is published to their own un-partitioned topic, e.g. all change records for a replicated table will be published as Kafka messages in a single partition in a topic. The topic name is a verbatim copy of the fully qualified replicate schema name, eg. SOE.WAREHOUSES, except when a topic name space prefix is provided in the configuration option topic.prefix.

Topic Auto-creation

The automatic creation of topics is governed by the Kafka parameter `auto.create.topics.enable` which is TRUE by default. This means that, as far as the Dbvisit Replicate Connector goes, any new tables detected in the PLOG files it processes will have new topics (with a single partition) automatically generated for them – and change messages written to them without any additional intervention.

Data types

Information on the data types supported by Dbvisit Replicate, and so what can be delivered through to PLOG files for processing by the Replicate Connector for Kafka, can be found here.

Information on Dbvisit Replicate Connector for Kafka specific data type mappings and support can be found in the Configuration section of this documentation.

LOAD

Dbvisit Replicate's Load function can be used to instantiate or baseline all existing data in the Oracle database tables by generating special LOAD PLOG files, which can be processed by the **Dbvisit Replicate Connector for Kafka**. This function ensures that before any change data messages are delivered the application will write out all the current table data – effectively initializing or instantiating these data sets within the Kafka topics.

Regular and LOAD PLOGS on the file system.

```
1682.plog.1480557139
1671.plog.1480555838-000001-LOAD_26839-SOE.ADDRESSES-APPLY
1671.plog.1480555838-000010-LOAD_26845-SOE.PRODUCT_INFORMATION-APPLY
1680.plog.1480556667
1676.plog.1480556539
1677.plog.1480556547
1674.plog.1480555972
1672.plog.1480555970
1671.plog.1480555838-000008-LOAD_26842-SOE.ORDER_ITEMS-APPLY
1671.plog.1480555838-000006-LOAD_26848-SOE.ORDERENTRY_METADATA-APPLY
1671.plog.1480555838-000004-LOAD_26844-SOE.INVENTORIES-APPLY
1671.plog.1480555838-000009-LOAD_26847-SOE.PRODUCT_DESCRIPTIONS-APPLY
1671.plog.1480555838-000007-LOAD_26843-SOE.ORDERS-APPLY
1671.plog.1480555838-000013-LOAD_26841-SOE.WAREHOUSES-APPLY
1671.plog.1480555838
1678.plog.1480556557
1671.plog.1480555838-000002-LOAD_26840-SOE.CARD_DETAILS-APPLY
1671.plog.1480555838-000012-LOAD_42165-SOE.TEST2-APPLY
1673.plog.1480555971
1681.plog.1480556671
```

```
1679.plog.1480556659
1671.plog.1480555838-000011-LOAD_39367-SOE.TEST1-APPLY
1671.plog.1480555838-000003-LOAD_26838-SOE.CUSTOMERS-APPLY
```

The parameter `plog.global.scn.cold.start` can be invoked to specify a particular SCN that the connector should work from, before the LOAD operation was run to generate the LOAD plogs, to provide some known guarantees around the state of the tables on the Oracle source at this time.

Note: the system change number or SCN, is a stamp that defines a committed version of a database at a point in time. Oracle assigns every committed transaction a unique SCN.

LIMITATIONS

Replicate Stream Limitation

Each PLOG corresponds to a REDO LOG and inherits its sequence number which, combined with other properties, is used to uniquely identify the replicate offset of the change records it contains and their offset in their Kafka topic. Each replicated table publishes their data to their own topic identified by the fully qualified name (including user schema owner) of the replicated table. If more than one replicate process is mining the same REDO LOGs the PLOG sequences may overlap and the Kafka topics must be separated by adding a unique namespace identifier to the topic names in Kafka.

It is important to note this limitation and specify a unique topic prefix for each Kafka connector session using the same DDC replication stream in the configuration file provided when starting the Kafka connector. See the `topic.prefix` configuration property.

Replicating DELETEs

By default DELETE messages will conform to the schema definition (as per the source table definition) and will publish its state at the time of deletion (its PRE state). This is not useful when log compaction is enabled for Kafka or the JDBC Sink Connector is used and DELETEs are replicated to a relational target.

LOBs

Inline LOB

Only single-part inline LOBs (inline - LOB size < 4000 bytes) are replicated as complete fields. All others (larger/out-of-line LOBs) will be partial LOB writes.

Single mandatory LOB field

Only one mandatory (NOT-NULL) LOB field is supported in a single change row when integrated with Avro/Schema Registry.

Multiple LOB fields

When publishing change row messages (See CDC format) only tables with single LOB fields are fully supported. When multiple LOB fields are present in a source table the LOB messages will be emitted separately from the rest of the fields.

Schema Evolution

Note: The following limitations apply when using Avro and the Schema Registry. They are not relevant when working with JSON key/value serializers, for which there should be no issue.

Due to the limitations outlined below schema evolution is disabled by default for the Dbvisit Replicate source connector. To enable it set `connector.publish.no.schema.evolution` to false only if using Schema Registry 3.3 or newer.

Default Values:

When a DDL operation on the source table adds a new column and the changes are replicated a default value is added to the Kafka schema. These are limited to datum specific types and may not be suited for aggregation purposes.

Kafka Connect Data Type	Default Value
Type.STRING	Empty string (zero length)
Type.INT64	-1L
Type.INT32	-1
Type.BYTES	Empty byte array (zero length)
Timestamp	Epoch start
Decimal	BigDecimal.ZERO

Logical Data Type Defaults

An issue in older versions of Schema Registry (3.2 and earlier) may prevent publishing default values for the logical data types. This occurs when a DDL operation adds one of the following fields mapped to either Decimal or Timestamp:

- NUMBER
- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

DDL Support

At this point in time only there is only limited support for DDL.

New tables may be added to a replication, if enabled on the Dbvisit Replicate side, then these will automatically be detected by the Replicate Connector for Kafka, and written to a new topic. However, table/column renames, truncate and drop table statements are ignored, and will not impact on the existing associated Kafka topic.

The adding and removing of table columns is supported by default. Those records which existed prior to the addition of a new column will have default (empty) values assigned during their next operation, as in named EXTRA column in the following:

```
{"XID":"0004.012.00006500","TYPE":"UPDATE","CHANGE_ID":1076010000726,"USER_ID":{"bytes
↪":"\u0000"},"USER_NAME":{"string":"Matt Roberts"},"USER_ROLE":{"string":
↪"Administrator"},"EXTRA":{"string":""}}
```

Conversely any columns which are dropped will have null values assigned, as in the following, for any previously values which existed in the record set:

```
{"XID":"0005.013.0000826f","TYPE":"UPDATE","CHANGE_ID":1078010000324,"USER_ID":{"bytes
↪":"\u0000"},"USER_NAME":{"string":"Matt Roberts"},"USER_ROLE":{"string":"Senior␣
↪Partner"},"EXTRA":null}
```

Note: When working with a table without any keys defined, with key publishing enabled (`connector.publish.keys`) columns removed from the Oracle source will remain in the corresponding Kafka topic. The column will remain and be populated with null. As noted with `connector.publish.keys` it is, however, recommended to create a primary or unique key in the Oracle source table.

Delivery Semantics

The Replicate Connector for Kafka manages offsets committed by encoding then storing and retrieving them (see the log file extract below). This is done in order that the connector can start from the last committed offsets in case of failures and task restarts. The replicate offset object is serialized as JSON and stored as a String schema in Kafka offset storage. This method should ensure that, under normal circumstances, records delivered from Oracle are only written once to Kafka.

```
[2016-11-17 11:41:31,757] INFO Offset JSON - TX.META:{"plogUID":4030157521414,
↪"plogOffset":2870088} (com.dbvisit.replicate.kafkaconnect.ReplicateSourceTask:353)␣
↪[2016-11-17 11:41:31,761] INFO Kafka offset retrieved for schema: TX.META PLOG: 938.
↪plog.1478197766 offset: 2870088 (com.dbvisit.replicate.kafkaconnect.
↪ReplicateSourceTask:392) [2016-11-17 11:41:31,761] INFO Processing starting at␣
↪PLOG: 938.plog.1478197766 at file offset: 2870088 schemas: [TX.META] (com.dbvisit.
↪replicate.kafkaconnect.ReplicateSourceTask:409) [2016-11-17 11:41:31,762] INFO␣
↪Offset JSON - SCOTT.TEST2:{"plogUID":4030157521414,"plogOffset":2869872} (com.
↪dbvisit.replicate.kafkaconnect.ReplicateSourceTask:353) [2016-11-17 11:41:31,762]␣
↪INFO Kafka offset retrieved for schema: SCOTT.TEST2 PLOG: 938.plog.1478197766␣
↪offset: 2869872 (com.dbvisit.replicate.kafkaconnect.ReplicateSourceTask:392) [2016-
↪11-17 11:41:31,762] INFO Processing starting at PLOG: 938.plog.1478197766 at file␣
↪offset: 2853648 schemas: [SCOTT.TEST1, SCOTT.TEST2] (com.dbvisit.replicate.
↪kafkaconnect.ReplicateSourceTask:409)
```

JSON

To use JSON encoding, rather than the default Avro option, use the JSON Converter options supplied as part of the Kafka Connect framework, by setting them as follows in the $CONFLUENT_HOME/etc/schema-registry/connect-avro-standalone.properties or the $CONFLUENT_HOME /etc/schema-registry/connect-avro-distributed.properties parameter files:

```
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
```

The non-Avro consumer can be invoked as follows, and will then display output as follows (here for the SOE.TEST1 table):

```
 ./bin/kafka-console-consumer --new-consumer --bootstrap-server localhost:9092 --
↪topic REP-SOE.TEST1 --from-beginning
```

```
{"schema":{"type":"struct","fields":[{"type":"string","optional":false,"field":"XID"},
↪{"type":"string","optional":false,"field":"TYPE"},{"type":"int64","optional":false,
↪"field":"CHANGE_ID"},{"type":"string","optional":false,"field":"USERNAME"},{"type":
↪"bytes","optional":false,"name":"org.apache.kafka.connect.data.Decimal","version":1,
↪"parameters":{"scale":"130"},"field":"USER_ID"},{"type":"string","optional":true,
↪"field":"PASSWORD"},{"type":"string","optional":false,"field":"ACCOUNT_STATUS"},{
↪"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Timestamp",
↪"version":1,"field":"LOCK_DATE"},{"type":"int64","optional":true,"name":"org.apache.
↪kafka.connect.data.Timestamp","version":1,"field":"EXPIRY_DATE"},{"type":"string",
↪"optional":false,"field":"DEFAULT_TABLESPACE"},{"type":"string","optional":false,
↪"field":"TEMPORARY_TABLESPACE"},{"type":"int64","optional":false,"name":"org.apache.
↪kafka.connect.data.Timestamp","version":1,"field":"CREATED"},{"type":"string",
↪"optional":false,"field":"PROFILE"},{"type":"string","optional":true,"field":
↪"INITIAL_RSRC_CONSUMER_GROUP"},{"type":"string","optional":true,"field":"EXTERNAL_
↪NAME"},{"type":"string","optional":true,"field":"PASSWORD_VERSIONS"},{"type":"string
↪","optional":true,"field":"EDITIONS_ENABLED"},{"type":"string","optional":true,
↪"field":"AUTHENTICATION_TYPE"}],"optional":false,"name":"REP-SOE.TEST1"},"payload":{
↪"XID":"0000.99c7.00000009","TYPE":"INSERT","CHANGE_ID":1089010003413,"USERNAME":"OE
↪","USER_ID":
↪"K0eTCAjEbEWPIpQQtAXbWx2lg3tDC5jPtnJazeyICKB1bCluSpLAAAAAAAAAAAAAAAAAAAAAA==",
↪"PASSWORD":"","ACCOUNT_STATUS":"OPEN","LOCK_DATE":null,"EXPIRY_DATE":null,"DEFAULT_
↪TABLESPACE":"DATA","TEMPORARY_TABLESPACE":"TEMP","CREATED":1383722235000,"PROFILE":
↪"DEFAULT","INITIAL_RSRC_CONSUMER_GROUP":"DEFAULT_CONSUMER_GROUP","EXTERNAL_NAME":"",
↪"PASSWORD_VERSIONS":"10G 11G ","EDITIONS_ENABLED":"N","AUTHENTICATION_TYPE":
↪"PASSWORD"}}
```


Schema Evolution

See the limitations detailed above.

The Replicate Connector supports schema evolution when the Avro converter is used. When there is a change in a database table schema, the Replicate Connector can detect the change, create a new Kafka Connect schema and try to register a new Avro schema in the Schema Registry. Whether it is able to successfully register the schema or not depends on the compatibility level of the Schema Registry, which is backward by default.

For example, if you add or remove a column from a table, these changes are backward compatible by default (as mentioned above) and the corresponding Avro schema can be successfully registered in the Schema Registry.

You can change the compatibility level of Schema Registry to allow incompatible schemas or other compatibility levels by setting `avro.compatibility.level` in Schema Registry. Note that this is a global setting that applies to all schemas in the Schema Registry.

Administration

Obsolete Source File Manager

Overview

The ObsoleteSourceFileManager aims to identify when source PLOG files stored on disk (consumed by all Kafka tasks and published to topics) may be considered obsolete and can be safely discarded. To do so it needs access to:

- ZooKeeper to query for all available topics in Kafka
- Dbvisit Replicate Connector properties used to start the Replicate Source Connector for Kafka

- Worker (source task) properties used to start the Replicate Source Connector for Kafka
- Kafka Connect Offset backing store (different for standalone and distributed mode)
- Kafka Connect Offset reader implementation (using correct offset backing store)

Installation

To install:

- download the Connector JAR bundle `dbvisit_replicate_connector_for_kafka-2.9.00-linux_x86_64-jar.zip` (see download link at top of page)
- install it alongside Kafka and/or the Dbvisit Replicate Connector in $KAFKA_HOME OR unzip to CONFLUENT_HOME
- run the wrapper script

```
cd $KAFKA_HOME;
unzip dbvisit_replicate_connector_for_kafka-2.9.00-linux_x86_64-jar.zip
$KAFKA_HOME/bin/obsolete-source-file-manager.sh
```

OR

- install Kafka to standard location /usr/share/java
- download and unzip the Connector JAR bundle `dbvisit_replicate_connector_for_kafka-2.9.00-linux_x86_64-jar.zip` in a different location, eg. $HOME
- export $PATH
- run the wrapper script

Command Line Options

A shell script is provided to run the obsolete source file manager from the command line, either as normal script or as a background process.

```
usage: obsolete-source-file-manager.sh
   --connector <connector>   The source connector property file used to
                             start the Kafka connector. Note that this is
                             also required for distributed mode, as config
                             parameters cannot be passed in directly.
   --distributed             Source connector is running in distributed
                             mode
   --dryrun                  Do not delete any obsolete source files,
                             instead only report the source files that
                             are considered obsolete and may be deleted
   --interval <interval>     The hour interval between scanning for
                             obsolete source PLOG files, range: 1 to 596,
                             default: 3
   --standalone              Source connector is running in standalone
                             mode
   --worker <worker>         The standalone/distributed worker property
                             file used to start the Kafka connector
   --zookeeper <zookeeper>   Zookeeper connect string, eg. localhost:2181
   --daemon                  Run as background process
   --logdir                  Log directory to use when run as daemon
   --shutdown                Shutdown the obsolete source file manager,
                             zookeeper and offset backing store
   --debug                   Enable debug logging
   --help                    Print this message
```

Option	Description	Example	Mandatory	Default
connector	The source connector property file used to start the Kafka Connector	dbvisit-replicate.properties	Y	
worker	The standalone/distributed worker property file used to start the Kafka connector	connect-standalone.properties	Y	
zookeeper	Zookeeper connect string	localhost:2181	Y	
standaloneordistributed	Source connector is running in standalone or distributed mode)		Y	
interval	The hour interval between scanning for obsolete source PLOG files	1 - 596	N	3
dryrun	Don't delete obsolete source files, instead only report those considered obsolete and thus eligible for deletion		N	
daemon	Run as background process		N	
logdir	Log directory to use when run as daemon		N	
shutdown	Shutdown the obsolete source file manager, Zookeeper and offset backing store		N	
debug	Enable debug logging		N	

Note: The connector name property in the connector property file must match the name of connector used in either standalone or distributed mode.

Example

Run as background process to scan hourly for obsolete source PLOG files using local Zookeeper and Kafka Connect running in standalone mode. This is for a standard system installation of Kafka and ZooKeeper with the Dbvisit Replicate Connector for Kafka installed to /usr.

```
obsolete-source-file-manager.sh
   --standalone
   --connector /usr/etc/kafka-connect-dbvisitreplicate/dbvisit-replicate.properties
   --worker /etc/kafka/connect-standalone.properties
   --zookeeper "localhost:2181"
   --interval 1
   --daemon
   --logdir /var/log/kafka
```

In this example the console output will be written to /var/log/kafka/obsolete-source-file-manager.out

Operations Guidelines

Adding New Tables to a Replication

When adding new tables to a replication (done on the Oracle source side in conjunction with dbvpf) this operation triggers a rebalance within Kafka Connect, and a restart of the Dbvisit Replicate Connector. Depending on the activity and specifications of your Kafka Connect environment the Kafka Connect parameter `task.shutdown.graceful.timeout.ms` may need to be increased from it's default, in order to ensure a clean shutdown of the Dbvisit Replicate Connector, and the safe writing of its cache to its internal Kafka topic `connector.catalog.topic.name`.

Maintenance

Shutting down dbvpf (Dbvisit Replicate) and its mining operations should not affect the actual running of the Dbvisit Replicate Connector for Kafka. However, a dbvpf restart will generate multiple PLOG files of the same base number, differentiated by a timestamp. In the current implementation of the connector if left running it will sit waiting for more information in the older version of the PLOG file, without awareness of the freshly generated plog. We expect to make improvements in upcoming versions of the connector to implement a watch service that notifies of these changes, and enables seamless processing.

But for the moment the best current practice when a dbvpf restart is required, is to step through the following ordering:

1. shutdown Dbvisit Replicate for Kafka source connector
2. shutdown dbvpf (Dbvisit Replicate)
3. start dbvpf (Dbvisit Replicate)
4. start Dbvisit Replicate for Kafkasource connector

The connector will then restart scanning at the last PLOG in the *plog.location.uri* directory. It will notice the multi-part REDO log sequences and mark them as "one" file and resume processing.

Upgrading

To upgrade to a newer version of the Dbvisit Replicate Connector for Kafka simply stop this process running in Kafka Connect, and replace the associated JAR file in the following location:

```
$CONFLUENT_HOME/share/java/kafka-connect-dbvisit
```

Note: The 2.9 version of the Dbvisit Replicate Connector for Kafka requires version 2.9+ of Dbvisit Replicate, and is not backwards compatible.

Troubleshooting

Logging

To alter logging levels for the connector all you need to do is update the log4j.properties file used by the invocation of the Kafka Connect worker. You can either edit the default file directly (see bin/connect-distributed and bin/connect-standalone) or set the env variable KAFKA_LOG4J_OPTS before invoking those scripts. The exact syntax is:

```
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:${CFG_DIR}/dbvisitreplicate-log4j.
↪properties"
```

In the following example, the settings were set to DEBUG to increase the log level for this connector class (and other options are ERROR, WARNING and INFO):

```
log4j.rootLogger=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c:%L)%n
log4j.logger.org.apache.zookeeper=WARN
log4j.logger.org.I0Itec.zkclient=WARN
```

```
log4j.logger.dbvisit.replicate.kafkaconnect.ReplicateSourceTask=DEBUG
log4j.logger.dbvisit.replicate.kafkaconnect.ReplicateSourceConnector=DEBUG
```

CHAPTER 2

Configuration Options

Configuration Parameters

Key User Configuration Properties

Note: **plog.location.uri** - is the location from which Dbvisit Replicate Connector will search for and read PLOG files delivered by the Dbvisit Replicate application. This directory must be local to the filesystem on which Kafka Connect is running, or accessible to it via a mounted filesystem. It cannot be an otherwise remote filesystem.

Note: **plog.data.flush.size** - for low transactional volumes (and for testing) it is best to change the default value of `plog.data.flush.size` in the configuration file to a value less than your total number of change records, eg. for manual testing you can use 1 to verify that each and every record is emitted correctly. This configuration parameter is used as the internal PLOG reader's cache size which translates at run time to the size of the polled batch of Kafka messages. It is more efficient under high transactional load to publish to Kafka (particularly in distributed mode where network latency might be an issue) in batches. Please note that in this approach the data from the PLOG reader is only emitted once the cache is full (for the specific Kafka source task) and/or the PLOG is done, so an Oracle redo log switch has occurred. This means that if the `plog.data.flush.size` is greater than total number of LCRs in cache it will wait for more data to arrive or a log switch to occur.

All User Configuration Properties

Version 2.8.04

`tasks.max` Maximum number of tasks to start for processing PLOGs.

- Type: string
- Default: 4

`topic.prefix` Prefix for all topic names.

- Type: string
- Default:

plog.location.uri Replicate PLOG location URI, output of Replicate MINE.

- Type: string
- Default: file:/home/oracle/ktest/mine

plog.data.flush.size LCRs to cache before flushing, for connector this is the batch size, choose this value according to transactional volume, for high throughput to kafka the default value may suffic, for low or sporadic volume lower this value, eg. for testing use 1 which will not use cache and emit every record immediately.

- Type: string
- Default: 1000

plog.interval.time.ms Time in milliseconds for one wait interval, used by scans and health check.

- Type: string
- Default: 500

plog.scan.interval.count Number of intervals between scans, eg. 5 x 0.5s = 2.5s scan wait time.

- Type: string
- Default: 5

plog.health.check.interval Number of intervals between health checks, these are used when initially waiting for MINE to produce PLOGs, eg. 10 * 0.5s = 5.0s.

- Type: string
- Default: 10

plog.scan.offline.interval Default number of health check scans to decide whether or not replicate is offline, this is used as time out value. NOTE for testing use 1, i.e. quit after first health check 1 * 10 * 0.5s = 5s where 10 is plog.health.check.interval value and 0.5s is plog.interval.time.ms value.

- Type: string
- Default: 1000

topic.name.transaction.info Topic name for transaction meta data stream.

- Type: string
- Default: TX.META

plog.global.scn.cold.start Global SCN when to start loading data during cold start.

- Type: string
- Default: 0

Version 2.9.00

Includes all configuration options for 2.8.04.

connector.publish.cdc.format Set the output CDC format of the Replicate Connector. This determines what type of messages are published to Kafka, and the options are:

1. changerow - complete row, the view of the table record after the action was applied (Default)
2. changeset - publish the KEY, NEW, OLD and LOB change fields separately

- Type: string

- Default: changerow

connector.publish.transaction.info Set whether or not the transaction info topic `topic.name.transaction.info` should be populated. This includes adding three fields to each Kafka data message to link the individual change message to its parent transaction message:

- XID - transaction ID
- TYPE - type of change, e.g. INSERT, UPDATE or DELETE
- CHANGE_ID - unique ID of change

The options are:

1. true
2. false - do not publish the extra transaction info and fields

- Type: string
- Default: true

connector.publish.keys Set whether or not keys should be published to all table topics. Keys are either primary or unique table constraints. When none of these are available all columns with either character, numeric or date data types are used as the key. The latter is not ideal, so it is encouraged to use PK or unique key constraints on source table.

The options are:

1. true - publish key schema and values for all Kafka data messages (not transactional info message)
2. false - do not publish keys

- Type: string
- Default: false

connector.publish.no.schema.evolution If logical data types are used as default values certain versions of Schema Registry might fail validation due to an issue, see #556. This option is provided for disabling schema evolution for BACKWARDS compatible schemas, effectively forcing all messages to conform to the first schema version published by ignoring all subsequent DDL operations.

The options are:

1. true - disable schema evolution, ignore all DDL modifications
2. false - allow schema evolution for Schema Registry version 3.3 and newer

- Type: string
- Default: true

topic.static.schemas Define the source schemas, as a comma separated list of fully qualified source table names, that may be considered static or only receiving sporadic changes. The committed offsets of their last message can be safely ignored if the lapsed days between the source PLOG of a new message and that of a previous one exceeds topic.static.offsets.age.days

Example:

- SCHEMA.TABLE1,SCHEMA.TABLE2
- Type: string
- Default: none

topic.static.offsets.age.days The age of the last committed offset for a static schema topic.static.schemas, when it can be safely ignored during a task restart and stream rewind. A message that originated from a source PLOG older will be considered static and not restart at its original source PLOG stream offset, but instead at its next available message offset. This is intended for static look up tables that rarely change when their source PLOGs may have been flushed since their last update. Defaults to 7 days.

- Type: string
- Default: 7

connector.catalog.bootstrap.servers The Kafka bootstrap servers to use for establishing the initial connection to the Kafka cluster. This is needed for storing internal catalog records for the Dbvisit Replicate source connector.

- Type: string
- Default: localhost:9092

connector.catalog.topic.name The name of the internal catalog topic created by the Dbvisit Replicate Connector for Kafka, used for tracking all replicated schemas emitted in PLOG stream. The provision is made here to rename this, to avoid conflicts with existing tables. But otherwise this topic should not be interfered with.

- Type: string
- Default: REPLICATE-INFO

Data Types

Oracle Data Type	Connect Data Type	Default Value	Conversion Rule
NUMBER	Int32	-1	scale <= 0 and precision - scale < 10
NUMBER	Int64	-1L	scale <= 0 and precision - scale > 10 and < 20
NUMBER	Decimal	BigDecimal.ZERO	scale > 0 or precision - scale > 20
CHAR	Type.String	Empty string (zero length)	Encoded as UTF8 string
VARCHAR	“”	“”	“”
VARCHAR2	“”	“”	“”
LONG	“”	“”	“”
NCHAR	Type.String	Empty string (zero length)	Encoded as UTF8, attempt is made to auto-detect if national character set was UTF-16
NVARCHAR	“”	“”	“”
NVARCHAR2	“”	“”	“”
INTERVAL DAY TO SECOND	Type.String	Empty string (zero length)	
INTERVAL YEAR TO MONTH	“”	“”	
CLOB	Type.String	Empty string (zero length)	UTF8 string
NCLOB	“”	“”	“”
DATE	Timestamp	Epoch time	
TIMESTAMP	“”	“”	
TIMESTAMP WITH TIME ZONE	“”	“”	
TIMESTAMP WITH LOCAL TIME ZONE	“”	“”	
BLOB	Bytes	Empty byte array (zero length)	Converted from SerialBlob to bytes
RAW	Bytes	Empty byte array (zero length)	No conversion
LONG RAW	“”	“”	“”

Distributed Mode Settings

Use the following as a guide to starting Dbvisit Replicate Connector for Kafka in Distributed mode, once the Kafka Connect worker(s) has been started on the host node(s). Postman is an excellent utility for working with cUrl commands.

```
   curl -v -H "Content-Type: application/json" -X PUT 'http://localhost:8083/
↪connectors/kafka-connect-dbvisitreplicate/config' -d
'{
  "connector.class": "com.dbvisit.replicate.kafkaconnect.ReplicateSourceConnector",
  "tasks.max": "2",
  "topic.prefix": "REP-",
  "plog.location.uri": "file:/foo/bar",
  "plog.data.flush.size": "1",
  "plog.interval.time.ms": "500",
  "plog.scan.interval.count": "5",
```

```
  "plog.health.check.interval": "10",
  "plog.scan.offline.interval": "1000",
  "topic.name.transaction.info": "TX.META"
}'
```

Or save this to a file <json_file>:

```
{
  "name": "kafka-connect-dbvisitreplicate",
  "config": {
    "connector.class": "com.dbvisit.replicate.kafkaconnect.ReplicateSourceConnector",
  "tasks.max": "2",
  "topic.prefix": "REP-",
  "plog.location.uri": "file:/foo/bar",
  "plog.data.flush.size": "1",
  "plog.interval.time.ms": "500",
  "plog.scan.interval.count": "5",
  "plog.health.check.interval": "10",
  "plog.scan.offline.interval": "1000",
  "topic.name.transaction.info": "TX.META"
  }
}

 curl -X POST -H "Content-Type: application/json" http://localhost:8083 --data "@
↪<json_file>"
```

CHAPTER 3

Changelog

Replicate Source Connector

- Dec 2016 - Early Release
- 2nd Aug 2017 - 2.9.00 Release