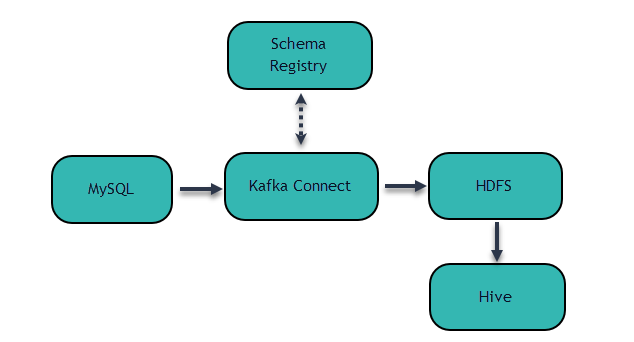
# **Build a Scalable ETL Pipeline with Kafka Connect**

****

To demonstrate Kafka Connect, we’ll build a simple data pipeline tying together a few common systems: **MySQL → Kafka → HDFS → Hive.** The pipeline captures changes from the database and loads the change history into the data warehouse, in this case Hive.

In the MySQL database, we have a user table which stores the current state of user profiles. In addition to common user profile information, the user table has a unique id column and a modified column which stores the timestamp of the most recent user profile change.

We simulate user profile changes by updating the corresponding entry in the user table. The JDBC Kafka connector will automatically capture those user profile changes and write each as an event to Kafka. Then, the HDFS Kafka connector will write those events to files in HDFS.

**Data Preparation**

Now we need to create some data in the MySQL database. We will create a users table to represent the user profiles. The auto increment id column is the primary key and the modified column saves the timestamp of the most recent update of each user profile. The modified column will be filled with the current timestamp if we omit the value during .insert.

|  |
| --- |
| $ mysql -uroot -ppassword mysql> CREATE DATABASE test; mysql> USE test; mysql> CREATE TABLE user (  id serial NOT NULL PRIMARY KEY,  name varchar(100),  email varchar(200),  department varchar(200),  modified timestamp default CURRENT\_TIMESTAMP NOT NULL,  INDEX `modified\_index` (`modified`)  ); mysql> INSERT INTO user (name, email, department) VALUES ('alice', 'alice@abc.com', 'engineering'); mysql> INSERT INTO user (name, email, department) VALUES ('bob', 'bob@abc.com', 'sales'); |

**Copy MySQL Connector Jar.**

Add the jar to existing Kafka Connect JDBC Jars **/usr/share/java/kafka-connect-jdbc**

|  |
| --- |
| sudo cp /usr/share/java/mysql-connector-java-5.1.42.jar /usr/share/java/kafka-connect-jdbc/ |

**Using Kafka Connect to Ingest Data**

Create a directory  **temp\_config**

|  |
| --- |
| **~$ mkdir temp\_config** |

Create a **hdfs.properties** and **mysql.properties** file and paste the following line:

|  |
| --- |
| **temp\_config$ gedit mysql.properties** |

name=test-mysql-jdbc

connector.class=io.confluent.connect.jdbc.JdbcSourceConnector

tasks.max=1

connection.url=jdbc:mysql://127.0.0.1:3306/test?user=root&password=password

mode=timestamp+incrementing

incrementing.column.name=id

timestamp.column.name=modified

topic.prefix=test\_jdbc\_

|  |
| --- |
| **temp\_config$ gedit hdfs.properties** |

name=hdfs-sink

connector.class=io.confluent.connect.hdfs.HdfsSinkConnector

tasks.max=1

topics=test\_jdbc\_user

hdfs.url=hdfs://<hostname>:8020/user/datacouch21

flush.size=2

hive.metastore.uris=thrift://<hostname>:9083

hive.integration=true

schema.compatibility=BACKWARD

partitioner.class=io.confluent.connect.hdfs.partitioner.FieldPartitioner

partition.field.name=department

Now we start Kafka Connect to ingest data from the MySQL database to Kafka then to HDFS. Note that data of the MySQL database will remain in Kafka after ingesting HDFS.

|  |
| --- |
| **temp\_config$ sudo connect-standalone /etc/schema-registry/connect-avro-standalone.properties mysql.properties hdfs.properties** |

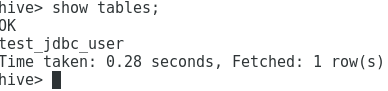
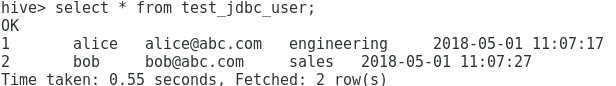
The JDBC connector creates a Kafka topic with the name **test\_jdbc\_users**. The HDFS connector reads data from **test\_jdbc\_users** and writes data to HDFS under **topics/test\_jdbc\_users** and creates a Hive external table with name **test\_jdbc\_users**.

Open a new terminal Kafka Consumer

|  |
| --- |
| $ sudo kafka-avro-console-consumer --topic test\_jdbc\_user --bootstrap-server localhost:9092 --from-beginning |

Once the connector finishes ingesting data to HDFS, open another terminal and check that the data is available in HDFS using Hive:

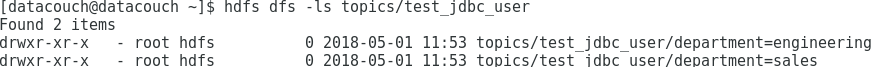
|  |
| --- |
| $ beeline -u jdbc:hive2://localhost:10000 -n $USER  0: jdbc:hive2://localhost:10000> SHOW TABLES; |

|  |
| --- |
| 1: jdbc:hive2://localhost:10000> SELECT \* FROM test\_jdbc\_user; |

In the test, we used the department as the partition field. To check the data in HDFS are actually partitioned by department:

|  |
| --- |
| **~$ hdfs dfs -ls topics/test\_jdbc\_user** |



We see that under **topics/test\_jdbc\_users/**, we have two directories and each directory name has the form of **department=value**. The value part is determined by the department column in the MySQL user table.

The **mysql.properties** and **hdfs.properties** control the behavior of the JDBC and HDFS connectors respectively.

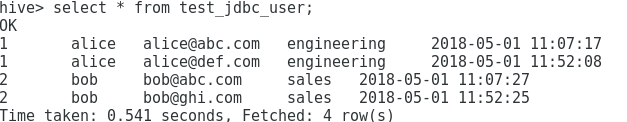
**Database Change Capture**

In the MySQL **user** table, the modified column stores the timestamp for last modification of a row. We change the user profiles by modifying the email column and set modified to current timestamp:

|  |
| --- |
| mysql>UPDATE user SET email = 'alice@def.com', modified = CURRENT\_TIMESTAMP WHERE name='alice'; mysql>UPDATE user SET email = 'bob@ghi.com', modified = CURRENT\_TIMESTAMP WHERE name='bob'; |

The JDBC connector detects that data in the users table have been changed as the modified column has updated values and will fetch modified data from the database and copied to Kafka and eventually to HDFS. To verify that the modified data is in HDFS:

|  |
| --- |
| **hive> SELECT \* FROM test\_jdbc\_user;** |

****

Note that there are two more records for the Hive external table **test\_jdbc\_users**. The two new records match the new content of the users table in the mysql database.

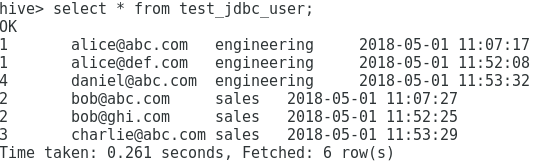
**Schema Migration**

To make a backward compatible change in our source JDBC system, we drop a column in the user table in the database, effectively removing a field from the data.

|  |
| --- |
| mysql> ALTER TABLE users DROP COLUMN name; mysql> INSERT INTO users (email, department) VALUES ('charlie@abc.com', 'sales'); mysql> INSERT INTO users (email, department) VALUES ('daniel@abc.com', 'engineering'); |

We see that all the six records are in **test\_jdbc\_users** and the name column no longer appears. While it still exists in the old data files, it is ignored by the Hive query because the field is not included in the latest schema.

|  |
| --- |
| hive> SELECT \* FROM test\_jdbc\_user; |

****