Lab 8. Building ETL Pipelines Using Kafka

In this lab, we will look into creating an ETL pipeline using these tools and Kafka Connect use cases and examples. In this lab, we will cover Kafka Connect in detail.

Using Kafka Connect

Kafka Connect provides us with various Connectors, and we can use the Connectors based on our use case requirement. It also provides an API that can be used to build your own Connector. We will go through a few examples in this section.

JDBC Source Connector

The JDBC source connector allows you to import data from any relational database with a JDBC driver into Kafka topics. By using JDBC, this connector can support a wide variety of databases without requiring custom code for each one.

Data is loaded by periodically executing a SQL query and creating an output record for each row in the result set. By default, all tables in a database are copied, each to its own output topic. The database is monitored for new or deleted tables and adapts automatically. When copying data from a table, the connector can load only new or modified rows by specifying which columns should be used to detect new or modified data.

Task

To see the basic functionality of the connector, you'll copy a single table from a local SQLite database. In this quick start, you can assume each entry in the table is assigned a unique ID and is not modified after creation.

Create SQLite Database and Load Data

1. Create a SQLite database with this command:

```
$ sqlite3 test.db
```

Your output should resemble:

```
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite>
```

2. In the SQLite command prompt, create a table and seed it with some data:

```
sqlite> CREATE TABLE accounts(id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name
VARCHAR(255));
```

```
sqlite> INSERT INTO accounts(name) VALUES('alice');
```

```
sqlite> INSERT INTO accounts(name) VALUES('bob');
```

Tip: You can run `SELECT * from accounts;` to verify your table has been created.

Load the JDBC Source Connector

Load the predefined JDBC source connector.

1. Optional: View the available predefined connectors with this command:

```
confluent list connectors
```

Your output should resemble:

```
Bundled Predefined Connectors (edit configuration under etc/):
    elasticsearch-sink
    file-source
    file-sink
    jdbc-source
    jdbc-sink
    hdfs-sink
    s3-sink
```

2. Load the the `jdbc-source` connector. The `test.db` file must be in the same directory where Connect is started.

```
confluent load jdbc-source
```

Your output should resemble:

```
{
    "name": "jdbc-source",
    "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "tasks.max": "1",
    "connection.url": "jdbc:sqlite:test.db",
    "mode": "incrementing",
    "incrementing.column.name": "id",
    "topic.prefix": "test-sqlite-jdbc-",
    "name": "jdbc-source"
    },
    "tasks": [],
    "type": null
}
```

To check that it has copied the data that was present when you started Kafka Connect, start a console consumer, reading from the beginning of the topic:

```
./bin/kafka-avro-console-consumer --bootstrap-server localhost:9092 --topic test-
sqlite-jdbc-accounts --from-beginning

{"id":1,"name":{"string":"alice"}}
{"id":2,"name":{"string":"bob"}}
```

The output shows the two records as expected, one per line, in the JSON encoding of the Avro records. Each row is represented as an Avro record and each column is a field in the record. You can see both columns in the table, `id` and `name`. The IDs were auto-generated and the column is of type `INTEGER NOT NULL`, which can be encoded directly as an integer. The `name` column has type `STRING` and can be `NULL`. The JSON encoding of Avro encodes the strings in the format `{"type": value}`, so you can see that both rows have `string` values with the names specified when you inserted the data.

Add a Record to the Consumer

Add another record via the SQLite command prompt:

```
sqlite> INSERT INTO accounts(name) VALUES('cathy');
```

You can switch back to the console consumer and see the new record is added and, importantly, the old entries are not repeated:

```
{"id":3,"name":{"string":"cathy"}}
```

Note that the default polling interval is five seconds, so it may take a few seconds to show up. Depending on your expected rate of updates or desired latency, a smaller poll interval could be used to deliver updates more quickly.

JDBC Sink Connector

The JDBC sink connector allows you to export data from Kafka topics to any relational database with a JDBC driver. By using JDBC, this connector can support a wide variety of databases without requiring a dedicated connector for each one. The connector polls data from Kafka to write to the database based on the topics subscription. It is possible to achieve idempotent writes with upserts. Auto-creation of tables, and limited auto-evolution is also supported.

Task

To see the basic functionality of the connector, we'll be copying Avro data from a single topic to a local SQLite database. This example assumes you are running Kafka and Schema Registry locally on the default ports.

Let's create a configuration file for the connector. This file is included with the connector in `./etc/kafka-connect-jdbc/sink-quickstart-sqlite.properties` and contains the following settings:

```
name=test-sink
connector.class=io.confluent.connect.jdbc.JdbcSinkConnector
tasks.max=1
topics=orders
connection.url=jdbc:sqlite:test.db
auto.create=true
```

The first few settings are common settings you'll specify for all connectors, except for `topics` which is specific to sink connectors like this one. The `connection.url` specifies the database to connect to, in this case a local SQLite database file. Enabling `auto.create` allows us to rely on the connector for creating the table.

Now we can run the connector with this configuration.

```
./bin/connect-standalone etc/schema-registry/connect-avro-standalone.properties
etc/kafka-connect-jdbc/sink-quickstart-sqlite.properties
```

Now, we will produce a record into the orders topic.

```
bin/kafka-avro-console-producer \
--broker-list localhost:9092 --topic orders \
--property value.schema='{"type":"record","name":"myrecord","fields":
[{"name":"id","type":"int"},{"name":"product", "type": "string"}, {"name":"quantity",
"type": "int"}, {"name":"price",
"type": "float"}]}'
```

The console producer is waiting for input. Copy and paste the following record into the terminal:

```
{"id": 999, "product": "foo", "quantity": 100, "price": 50}
```

Now if we query the database, we will see that the orders table was automatically created and contains the record.

```
$ sqlite3 test.db

sqlite> select * from orders;
```

Output: `foo|50.0|100|999`

Stop Confluent Platform

When you are done working with the local install, you can stop Confluent Platform. Open new terminal and go to following directory:

`cd /headless/kafka-advanced/confluent-6.1.1/bin`

1. Stop Confluent Platform using the Confluent CLI.

   ```
   ./confluent local services stop
   ```

2. Destroy the data in the Confluent Platform instance with the confluent local destroy command.

   ```
   ./confluent local destroy
   ```

Summary

In this lab, we learned about Kafka Connect in detail. In the next lab, you will learn about Kafka Stream in detail, and we will also see how we can use Kafka stream API to build our own streaming application. We will explore the Kafka Stream API in detail and focus on its advantages.