**Apache Kafka**

* **Introduction**

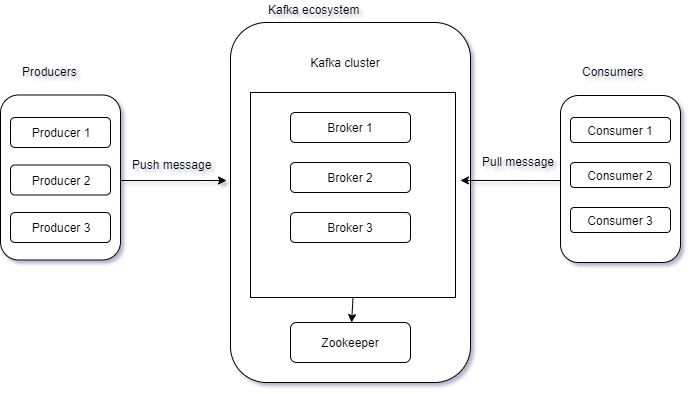
In Big Data, an enormous volume of data is used. Regarding data, we have two main challenges. The first challenge is how to collect large volume of data and the second challenge is to analyze the collected data. To overcome these challenges, you must need a messaging system.

Kafka is designed for distributed high throughput systems. Kafka tends to work very well as a replacement for a more traditional message broker. In comparison to other messaging systems, Kafka has better throughput, built-in partitioning, replication and inherent fault-tolerance, which makes it a good fit for large-scale message processing applications.

Apache Kafka is a community distributed event streaming platform capable of handling trillions of events a day. It is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables you to pass messages from one end-point to another. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. Kafka is built on top of the ZooKeeper synchronization service.

Thousands of companies are built on Kafka that are Airbnb, Netflix, Linked in, Microsoft, Target etc.

* **Cluster Architecture**



A Brief explanation about Kafka core components are given below,

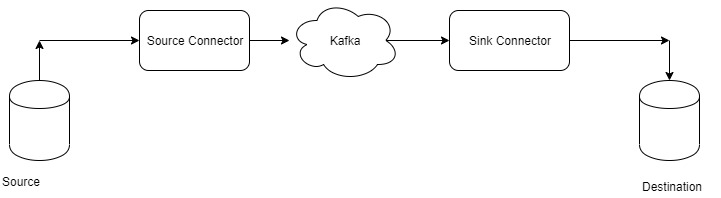
1. **Broker**: Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.
2. **ZooKeeper:** ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system.
3. **Producers:** Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
4. **Consumers:** Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

The docker-compose-file for kafka cluster setup is https://github.com/confluentinc/cp-all-in-one/blob/7.0.1-post/cp-all-in-one/docker-compose.yml

* **Building Data pipelines**

In this section presumes that you know what is kafka and its core components. This section focus on how to build simple data pipeline using kafka. To Build data pipeline we need

1. Broker
2. Zookeeper
3. Connect
4. Database



Here the requirement is to collect all the data from source side to kafka topic and then move that data to destination side. In between these actions we have to analyse the stored information in kafka topic before writing to destination side.

Let’s see how to solve this,

So our first requirement is to collect all the data from source side, to achieve this we need a “***Source Connector***”. The responsibility of source connector is to collect the data from source data source and push that information to kafka topic. In Data Pipeline Source connectors are the “***Publishers***”. There are two types of source connectors namely JDBC Source connector and Debezium Source Connector. In this looking into JDBC source connector it helps transfer data from database to kafka.

Our next requirement is to transfer data from kafka to external database(destination), to achieve this we need a “***Sink Connector***”. In Data Pipeline Sink connectors are the “***Subscribers***”.

In a nutshell, JDBC source connector helps transfer data from database to kafka, while JDBC sink connector transfer data from kafka to any external databases.

* **Source Connector**

The role of source connector in Kafka is to import data from any relational or non-relational database to Kafka topic. Based on the connector internal working we can classify the source connectors in to two categories

1. JDBC
2. Debezium

In JDBC source connector it will periodically executing a sql query and output record for each row in the result set. One limitation of this connector is that it cannot detect deleted records. Which means when a table row get delete it will not publish message to respective Kafka topic.

Debezium is **a set of source connectors for Apache Kafka Connect**. Each connector ingests changes from a different database by using that database's features for change data capture (CDC). This connector can detect all the table transactions like insert, update and delete.

In the rest of the section will explain how to create a JDBC Source connector (for confluent) for a sqlserver database table.

(Note: To create JDBC source connector first you should have a running kafka cluster).

To create a source connector we can use kafka connect rest endpoint. The endpoint details are given below

Endpoint: {kafka connect url}/connectors

Http verb: POST

Payload: {

“name: “{{your connector name}}”,

“config: {

"connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",

//Provider your configurations

}

}

Here I have a up and running Kafka cluster, where the connect api is running at <http://localhost:8083>

1. JDBC Source connector with incrementing mode

This source connector use a strictly incrementing column on each table to detect only new rows. Note that this will not detect modifications or deletions of existing rows.

URL: <http://localhost:8083/connectors>

Http verb: Post

Request payload:

{

"name": "jdbc\_simple\_car\_connector",

"config": {

"connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",

"connection.url": "jdbc:sqlserver://sqlserverkafkaplayground.database.windows.net:1433;database=Kafka\_Db",

"connection.user": "kafkauser",

"connection.password": "Kafka123",

"table.whitelist": "Cars",

**"mode": "incrementing",**

**"incrementing.column.name": "Id",**

"key.converter": "io.confluent.connect.json.JsonSchemaConverter",

"key.converter.schema.registry.url": "http://schema-registry:8081",

"value.converter": "io.confluent.connect.json.JsonSchemaConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081"

}

}

Where,

“name” : It is the connector name

“connection.url”: The jdbc sql server connection url

“connection.user”: The jdbc sql server connection user

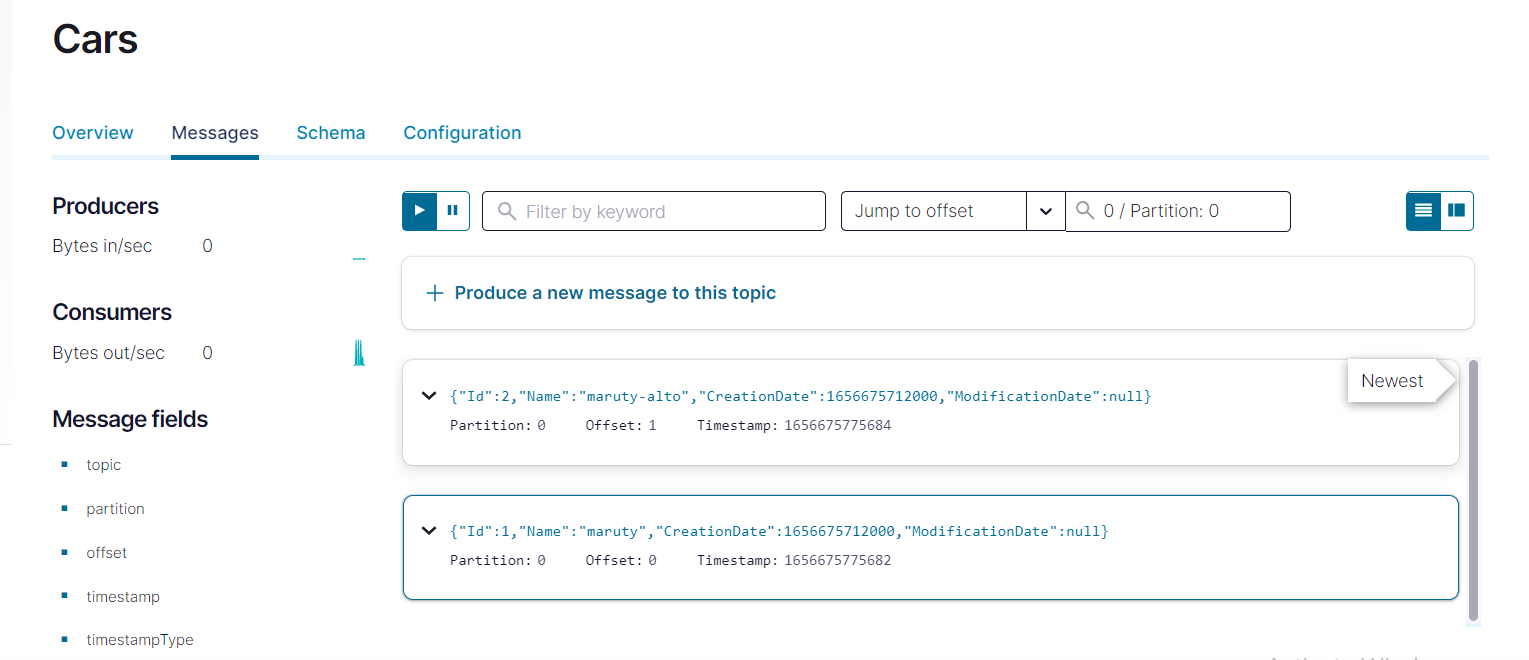
“table.whitelist”: List of table to include in copying

“connection.password”: The jdbc sql server connection password

“mode” : The mode for updating a table each time is polled. In the above example, its value is “incrementing” it will detect only new rows

“incrementing.column.name”: The name of the strictly incrementing column to use to detect new rows. Here the value is “id” which is the primary key of Cars table

This connector will automatically create a new topic and the name of topic will be the table name that specified in the table.whitelist. By using control-center or kafdrop you can check the topic messages.



The above screenshot is take from control-center UI. Here we can find the message payload. When you add a new row to Cars table, it will push a new message to “Cars” topic. However, if you update existing row it will not publish any message.

1. JDBC Source connector with incrementing+timestamp mode

In this mode use two columns, a timestamp column that detects new and modified rows and a strictly incrementing column which provides a globally unique ID for updates so each row can be assigned a unique stream offset.

Briefly, in this mode the connector will detect new rows as well as the modified rows.

URL: <http://localhost:8083/connectors>

Http verb: Post

Request payload:

{

"name": "jdbc\_car\_connector",

"config": {

"connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",

"connection.url": "jdbc:sqlserver://sqlserverkafkaplayground.database.windows.net:1433;database=Kafka\_Db",

"connection.user": "kafkauser",

"connection.password": "Kafka123",

"table.whitelist": "Cars",

**"mode": "timestamp+incrementing",**

**"incrementing.column.name": "Id",**

**"timestamp.column.name": "ModificationDate,CreationDate",**

"key.converter": "io.confluent.connect.json.JsonSchemaConverter",

"key.converter.schema.registry.url": "http://schema-registry:8081",

"value.converter": "io.confluent.connect.json.JsonSchemaConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081"

}

}

"timestamp.column.name": It is a Comma-separated list of one or more timestamp columns to detect new or modified rows using the COALESCE SQL function. Rows whose first non-null timestamp value is greater than the largest previous timestamp value seen will be discover with each poll. At least one column should not be nullable.

At the moment of creating this connector, It will create a new topic(if it’s not exist), the name of the topic will be the tables specified in the “table.whitelist”.

This source connector is able to publish message to corresponding topic at creating new rows as well as updating existing one. One condition is that the update action should update table value of any of the columns specified in the "timestamp.column.name".

* **Sink Connectors**

Sink connector is a service, which reads data from at least one topic in kafka and write to a target system.

The Kafka Connect JDBC Sink connector allows you to export data from Apache Kafka topics to any relational database with a JDBC driver. This connector can support a wide variety of databases. 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 support.

For creating sink connector, we can use the same connector creation rest endpoint. Let us check configurations of a jdbc sink connector that pointing to sql server database at the destination.

{

"name": "jdbc\_sink\_car\_connector",

"config": {

"connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",

"connection.url": "jdbc:sqlserver://sqlserverkafkaplayground.database.windows.net:1433;database=Kafka\_SinkDb",

"connection.user": "kafkauser",

"connection.password": "Kafka123",

"insert.mode": "insert",

"auto.create": true,

"pk.mode": "record\_value",

"pk.fields": "Id",

"topics": "Cars",

"key.converter": "io.confluent.connect.json.JsonSchemaConverter",

"key.converter.schema.registry.url": "http://schema-registry:8081",

"value.converter": "io.confluent.connect.json.JsonSchemaConverter",

"value.converter.schema.registry.url": "http://schema-registry:8081"

}

}

Where,

“name”: Connector name

“connector.class” : Connector class name

“connection.url”: JDBC connection url

“connection.user”: JDBC connection user

“connection.password”: JDBC connection password

“insert.mode”: The insertion mode to use, valid values are insert, update and upsert

“auto.create”: Whether to automatically create the destination table based on record schema if it is found to be missing, default value is false

“pk.mode”: The primary key mode. Valid values are none, kafka, record\_key, record\_value

“pk.fields”: List of comma-separated primary key field names

“topic”: Topic want to subscribe

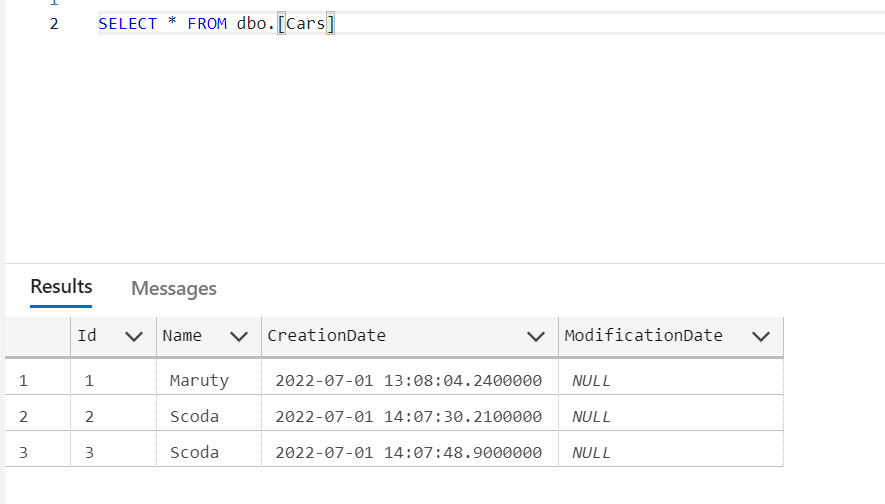
The above configuration will create a new jdbc sink connector. That will subscribe to “Cars” topic and Write message to Cars table in the Kafka\_SinkDb.

Here the configuration having “auto.create” is true, which means it will create Cars table at the time of connector creation. The name of the table columns are corresponding to topic message fields. The primary key column of the table will be to the value specified in the “pk.fields” property.

Here “insert.mode” is set as insert, which means the connector will insert new rows only. Which means it will not update the existing records in the destination table.



This above figure showing the “Cars” topic message in Kafka. After the sink connector creation, the destination table result will be given below



* **References**

1. <https://docs.confluent.io/kafka-connect-jdbc/current/sink-connector/index.html> - Jdbc sink connector documentation
2. <https://docs.confluent.io/kafka-connect-jdbc/current/source-connector/index.html> - Jdbc source connector documentation
3. <https://docs.confluent.io/platform/current/connect/references/restapi.html> - Connect rest api documentation
4. https://docs.confluent.io/platform/current/quickstart/ce-docker-quickstart.html?utm\_medium=sem&utm\_source=google&utm\_campaign=ch.sem\_br.brand\_tp.prs\_tgt.confluent-brand\_mt.mbm\_rgn.india\_lng.eng\_dv.all\_con.confluent-docker&utm\_term=%2Bconfluent%20%2Bdocker&creative=&device=c&placement=&gclid=Cj0KCQiAmKiQBhClARIsAKtSj-nl8v2dj\_6wR\_xV8whuuDBDX0hOSiKi0Um3CS7hHjS68JdnPXRCqmcaAsg7EALw\_wcB – Kafka quick start
5. <https://github.com/confluentinc/cp-all-in-one/blob/7.0.1-post/cp-all-in-one/docker-compose.yml> - docker-compose-file for kafka cluster