Programming with Mati: KqlDB

Running Locally

We're deploying the following components with Docker compose:

- Zookeeper
- Kafka
- Create Topics container
- ksqlDB server
- ksqlDB CLI

Feel free to checkout the ksqlDB Server config to see how we're configuring the ksqlDB server in this tutorial. The configuration is pretty basic in this introductory tutorial, but we'll expand on this in later episodes 😃

When you're ready, you can start the above services by running the following command:

```
docker-compose up &
```

Log into the ksqlDB CLI using the following command:

```
docker-compose exec ksqldb-cli ksql http://ksqldb-server:8088
```

If you see a `Could not connect to the server` error in the CLI, wait a few seconds and try again. ksqlDB can take several seconds to start.

Then, run this command in the ksqldb-cli:

```
SHOW TOPICS;
```

Now, you're ready to run the ksqlDB statements for our hello, world tutorial.

1 | Set the Ksql consumer to `earliest`

We want to set the Ksql consumer to `earliest` so that we can see all messages that we will be producing. Run the following:

```
SET 'auto.offset.reset'='earliest';
```

2 | Create a Stream of users

```
CREATE STREAM users (
    ROWKEY INT KEY,
    USERNAME VARCHAR
) WITH (
    KAFKA_TOPIC='users',
    VALUE_FORMAT='JSON'
);
```

You should receive a message like this:

```
 Message
----------------
 Stream created
----------------
```

You can verify the Stream was created running:

```
SHOW STREAMS;
```

3 | Insert data into the Stream

```
INSERT INTO users (username) VALUES ('Mati');
INSERT INTO users (username) VALUES ('Michelle');
INSERT INTO users (username) VALUES ('John');
```

4 | Consume Stream

```
SELECT 'Hello, ' + USERNAME AS GREETING
FROM users
EMIT CHANGES;
```

You should see output similar to the following:

```
+--------------------+
|GREETING            |
+--------------------+
|Hello, Mati         |
|Hello, Michelle     |
|Hello, John         |
```

You can also open a Kafka Console Consumer and check what's happening in the `users` topic by running this in a new terminal:

```
docker-compose exec kafka kafka-console-consumer --bootstrap-server localhost:9092
--from-beginning --topic users
```

Once you're finished, tear everything down using the following command:

```
docker-compose down
```

KSQLDB: Kafka Connect Overview

Running Locally

We're deploying the following components with Docker compose:

- Zookeeper
- Kafka
- ksqlDB server (With Kafka Connect)
- ksqlDB CLI
- Schema Registry (To keep the schema of the data)
- MySQL
- Redis

KSQLDB and Kafka Connect

The KsqlDB server will run also Kafka Connect, so that we can create source and sink connectors. There are a couple of things to consider in the deployment:

1. We need to install the connectors required for our components. In this case we will use JDBC Source Connector for MySQL and Redis Sink Connector for Redis.
2. We also need to add the MySQL Driver to our ksqldb-server container.
3. The file run.sh contains all the commands to install the connectors and copy the MySQL driver to the right location.
4. In the ksqldb-server folder we also have the ksql-server.properties which is quite simple, but it also points to the connect.properties which is a bit more complex.
5. In the ksqldb-cli folder we have the SQL scripts that we will run manually to create the connectors to get data from MySQL into Kafka and from Kafka into Redis

MySQL deployment

The MySQL instance will have a database called `football`. Inside that DB, there will be a table called `players` and there will be `10` players already inserted in it. The init script can be found here: init.sql. This script creates the db, the table and the data.

Start the containers

To start running all the containers, just run:

```
docker-compose up &
```

Then run the following to connect to use the `ksql-cli`:

```
docker-compose exec ksqldb-cli ksql http://ksqldb-server:8088
```

Create the connectors

Once we are logged in to the ksqldb-cli, we can create the connectors that are found in the script all.sql.

First create the MySQL Source Connector:

```
CREATE SOURCE CONNECTOR mysql_source_connector
WITH (
  'connector.class' = 'io.confluent.connect.jdbc.JdbcSourceConnector',
  'connection.url' = 'jdbc:mysql://mysql:3306/football',
  'connection.user' = 'root',
  'connection.password' = 'root',
  'table.whitelist' = 'players',
  'mode' = 'incrementing',
  'incrementing.column.name' = 'id',
  'topic.prefix' = '',
  'key'='id'
);
```

This tells KsqlDB that we want to create a connector that will read data from the `players` table and will insert it into kafka. Because we declared the converter in the connect.properties file, we don't need to specify the converters here.

Then we can verify that our connector was created by running:

```
SHOW CONNECTORS;
```

By default, Kafka Connect will create a new topic and will call it the same name that the table has. We can verify that the `players` topic was created in kafka with this command:

```
SHOW TOPICS;
```

Now let's create the Redis Sink Connector. Run this script:

```
CREATE SINK CONNECTOR redis_sink WITH (

'connector.class'='com.github.jcustenborder.kafka.connect.redis.RedisSinkConnector
',
  'tasks.max'='1',
  'topics'='players',
  'redis.hosts'='redis:6379',
  'key.converter'='org.apache.kafka.connect.converters.ByteArrayConverter',
  'value.converter'='org.apache.kafka.connect.converters.ByteArrayConverter'
);
```

This creates a new Redis Sink Connector that will get the data from the `players` topic and put it into Redis. Notice that we define the converters for key and value as `ByteArrayConverter`, since we want to store the Avro into Redis. Since Avro is a binary format, we can use the `ByteArrayConverter` to save it into Redis as a Byte Array.

Verify the data is now in Redis

To do this, connect to the redis command line tool running this in a new terminal:

```
docker-compose exec redis redis-cli
```

Once logged in to the Redis server select the database 1 with this command:

```
SELECT 1
```

Finally, you can run the command to get the value corresponding to the key 1:

```
GET 1
```

You should see something like this:

```
"\x00\x00\x00\x00\x01\x02\x18Lionel Messi\x12Paris Saint-Germain\x16Argentinian"
```

Some of it is not readable. But because we have some string values, we can make sense of it and we know that this information belongs to the first record in our table, which is "Lionel Messi".

KSQLDB: Kafka Connect Overview

Running Locally

We're deploying the following components with Docker compose:

- Zookeeper
- Kafka
- ksqlDB server (With Kafka Connect)
- ksqlDB CLI
- MySQL

KSQLDB and Kafka Connect

The KsqlDB server will run also Kafka Connect, so that we can create source connector. There are a couple of things to consider in the deployment:

1. We need to install the connectors required for our components. In this case we will use JDBC Source Connector for MySQL.
2. We also need to add the MySQL Driver to our ksqldb-server container.
3. The file run.sh contains all the commands to install the connectors and copy the MySQL driver to the right location.
4. In the ksqldb-server folder we also have the ksql-server.properties which is quite simple, but it also points to the connect.properties which is a bit more complex.
5. In the ksqldb-cli folder we have the SQL scripts that we will run manually to create the connectors to get data from MySQL into Kafka

MySQL deployment

The MySQL instance will have a database called `football`. Inside that DB, there will be a table called `players` and there will be `10` players already inserted in it. The init script can be found here: init.sql. This script creates the db, the table and the data.

Start the containers

To start running all the containers, just run:

```
docker-compose up &
```

Then run the following to connect to use the `ksql-cli`:

```
docker exec -it ksqldb-cli ksql http://ksqldb-server:8088
```

Set Offset to earliest

We set the offset to earliest to be able to read the data from the beginning of the topic.

```
SET 'auto.offset.reset' = 'earliest';
```

Create the MySQL connector

Once we are logged in to the ksqldb-cli, we can create the connector that is found in the script all.sql.

```
CREATE
SOURCE CONNECTOR mysql_source_connector
WITH (
  'connector.class' = 'io.confluent.connect.jdbc.JdbcSourceConnector',
  'connection.url' = 'jdbc:mysql://mysql:3306/football',
  'connection.user' = 'root',
  'connection.password' = 'root',
  'table.whitelist' = 'players',
  'mode' = 'incrementing',
  'incrementing.column.name' = 'id',
  'topic.prefix' = '',
  'key'='id',
  'key.converter'='org.apache.kafka.connect.storage.StringConverter',
  'value.converter'='org.apache.kafka.connect.json.JsonConverter',
  'value.converter.schemas.enable' = false
);
```

This tells KsqlDB that we want to create a connector that will read data from the `players` table and will insert it into kafka.

Then we can verify that our connector was created by running:

```
SHOW CONNECTORS;
```

By default, Kafka Connect will create a new topic and will call it the same name that the table has. We can verify that the `players` topic was created in kafka with this command:

```
SHOW TOPICS;
```

Create a Source Table for Players

Now that we have the players in the topic, we can create a `source` table that will hold the information about the players in KSQLDB. To do this, we can do:

```
CREATE TABLE players
(
    id          VARCHAR PRIMARY KEY,
```

```
    name        VARCHAR(50),
    team        VARCHAR(50),
    nationality VARCHAR(50)
)
    WITH (
        KAFKA_TOPIC = 'players', -- The topic in which we added the players
        VALUE_FORMAT = 'JSON', -- The format in which the data is written
        PARTITIONS = 1
        );
```

Note that we have added all the columns we have in the table. If we wanted, we could have less columns, if we need less data.

This table is equivalent to a Kafka Stream's `KTable`. This means that players will be inserted in the table if they have a new key or they will be updated if a new record is inserted in the topic with the same key.

Running simple queries

Now that we have players in our table, we can run a simple query:

```
SELECT * FROM players EMIT CHANGES;
```

You should see something like this:

```
+----------------------------+----------------------------+----------------------
-----+----------------------------+
|ID                          |NAME                        |TEAM
|NATIONALITY                 |
+----------------------------+----------------------------+----------------------
-----+----------------------------+
|1                           |Lionel Messi                |Paris Saint-Germain
|Argentinian                 |
|2                           |Cristiano Ronaldo           |Al-Nassr
|Portuguese                  |
|3                           |Neymar Jr.                  |Paris Saint-Germain
|Brazilian                   |
|4                           |Kevin De Bruyne             |Manchester City
|Belgian                     |
|5                           |Kylian Mbappe               |Paris Saint-Germain
|French                      |
|6                           |Robert Lewandowski          |Barcelona
|Polish                      |
|7                           |Sadio Mane                  |Bayern Munich
|Senegalese                  |
|8                           |Virgil van Dijk             |Liverpool
|Dutch                       |
|9                           |Bernardo Silva              |Manchester City
|Portuguese                  |
```

```
|10                         |Raheem Sterling            |Chelsea
|English                    |
```

This is the classic `SELECT *` we use in any DB. The only difference here is that we need to add the `EMIT CHANGES` to be able to see the data in the standard output. This is called a **Push Query**. This kind of query can be used to push data into a Kafka topic. If we use the keyword `EMIT CHANGES` then we are telling the query to push the changes into the standard output.

Let's try some more useful queries.

Projections

We can use projections by enumerating the columns we want to see. For example:

```
SELECT name, team FROM players EMIT CHANGES;
```

We can only see players with their names and teams.

We can also modify the projections using functions. Let's put the team name in Upper Case with the function `UCASE()`.

```
SELECT name, UCASE(team) team FROM players EMIT CHANGES;
```

You should see the teams in upper case now.

Conditionals

You can also have conditional statements in your projections using the `CASE` keyword.

```
SELECT name,
       team,
       CASE
           WHEN name = 'Lionel Messi' THEN 'GOAT'
           ELSE 'PLAYER'
       END status
       FROM players
       EMIT CHANGES;
```

Now you should see a new column in the query result where in the case of Messi, `status` is `GOAT`. The `CASE` keyword is extremely useful.

Filtering

We can also filter results using the `WHERE` clause. Let's say we want all the players from Manchester City. We can get them like this:

```
SELECT * FROM players
WHERE team = 'Manchester City'
EMIT CHANGES;
```

As in regular SQL, we can add multiple conditions to our filter:

```
SELECT * FROM players
WHERE team = 'Manchester City'
AND nationality = 'Belgian'
EMIT CHANGES;
```

Creating a Stream

Streams are the equivalent of a Kafka Stream `KStream`. They are an unmodifiable stream of data. Values are never updated. Let's create a new source stream that will represent the events of a football match:

```
CREATE
STREAM match_event (
    id VARCHAR KEY, -- Streams don't have a primary key, but just a key
    event_type VARCHAR, -- This will be 'GOAL' or 'ASSIST'
    player_id VARCHAR,
    home boolean -- whether the event favors the home team or away
) WITH (
    KAFKA_TOPIC='match_event', -- topic doesn't exist so it will be created
    VALUE_FORMAT='JSON', -- the format in which data will be stored
    PARTITIONS=1
);
```

Querying a Stream

Streams can also be queried using a `SELECT` statement. This is also a `push query`. For example:

```
SELECT * FROM match_event EMIT CHANGES;
```

But we don't have any data in the stream. We can add data with an `INSERT` statement.

Inserting data in a Stream

To insert data, just use a regular `INSERT` statement. Let's insert a goal done by Messi on match 1 where he plays in the "home" team.

```
INSERT INTO match_event
VALUES ('1', 'GOAL', '1', true);
```

Now if we run the query again:

```
SELECT * FROM match_event EMIT CHANGES;
```

We can see this result:

```
+---------------------------+---------------------------+----------------------
-----+---------------------------+
|ID                         |EVENT_TYPE                 |PLAYER_ID
|HOME                       |
+---------------------------+---------------------------+----------------------
-----+---------------------------+
|1                          |GOAL                       |1
|true                       |
```

You can also note that this record has been inserted in the topic `match_event` and that's why we see it in the result of the query. Any new record inserted in the topic, through `INSERT` or directly into the topic with another Kafka producer, will be shown here.

Let's add an Asist now:

```
INSERT INTO match_event
VALUES ('1', 'ASSIST', '1', true);
```

You can now filter the stream with using the `WHERE` clause:

```
SELECT * FROM match_event
WHERE event_type = 'ASSIST'
EMIT CHANGES;
```

The result is:

```
+---------------------------+---------------------------+----------------------
-----+---------------------------+
|ID                         |EVENT_TYPE                 |PLAYER_ID
|HOME                       |
+---------------------------+---------------------------+----------------------
-----+---------------------------+
|1                          |ASSIST                     |1
|true                       |
```

Aggregate queries

You can create aggregate queries using `aggregate functions` and the `GROUP BY` clause. Let's say we want to count the home team goals of the matches. We should group by the match `id` and aggregate counting the amount of goals, also filtering the cases in which the event if for the home team. In SQL this would look like this:

```
SELECT
    id,
    COUNT(id) home_goals -- aggregate function to count goals
FROM match_event
WHERE home AND event_type = 'GOAL'
GROUP BY id
EMIT CHANGES;
```

In order to get some more interesting results, let's add more data into the match_event stream:

```
INSERT INTO match_event
VALUES ('1', 'GOAL', '1', true);
INSERT INTO match_event
VALUES ('1', 'GOAL', '2', false);
```

Now if we run the previous query, the select one, we will see this:

```
+-------------------------------------------------------------+--------------------
-------------------------------------+
|ID                                                           |HOME_GOALS
|
+-------------------------------------------------------------+--------------------
-------------------------------------+
|1                                                            |2
|
```

You can also query the away goals:

```
SELECT
    id,
    COUNT(id) away_goals
FROM match_event
WHERE NOT home AND event_type = 'GOAL'
GROUP BY id
EMIT CHANGES;
```

Writing data into Kafka

So far we've done many queries, but we haven't saved any of this data in Kafka. To do this, we need to create a `SINK` collection. This is a `TABLE` or a `STREAM` that will push data into Kafka. The ones we created so far read data from Kafka.

First we will add some more data to the stream:

```
INSERT INTO match_event
VALUES ('2', 'GOAL', '1', true);
INSERT INTO match_event
VALUES ('2', 'ASSIST', '2', false);
INSERT INTO match_event
VALUES ('2', 'GOAL', '2', false);
```

Second, we will create a new query which will tell us the result of the match:

```
SELECT id,
       sum(
               CASE
                   WHEN home AND event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) home_goals, -- Here we only count home goals with a conditional
statement
       sum(
               CASE
                   WHEN NOT home AND event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) away_goals -- Here we only count home goals with a conditional
statement
FROM match_event
GROUP BY id
EMIT CHANGES;
```

You should see this result:

```
+-------------------------------------+-------------------------------------+---
---------------------------------+
|ID                                   |HOME_GOALS
|AWAY_GOALS                           |
+-------------------------------------+-------------------------------------+---
---------------------------------+
|1                                    |2                                    |1
|
|2                                    |1                                    |1
|
```

Now we are going to create a SINK Table, to update the values of the match as new events come along:

```
CREATE TABLE match_results
WITH (
    KAFKA_TOPIC='match_results', -- new updates will be pushed to the topic
'match_results'
    VALUE_FORMAT='JSON' -- We indicate that the data will be pushed in JSON format
    ) AS
SELECT id,
       sum(
               CASE
                   WHEN home AND event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) home_goals, -- Here we only count home goals with a conditional
statement
       sum(
               CASE
                   WHEN NOT home AND event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) away_goals -- Here we only count home goals with a conditional
statement
FROM match_event
GROUP BY id; -- We copy the previous query, but now, we are not emitting changes.
```

Now we can query the table match_results directly:

```
SELECT * FROM match_results EMIT CHANGES;
```

And more importantly, the data was saved into kafka. To see this, in a different terminal run this:

```
 docker exec kafka kafka-console-consumer --bootstrap-server localhost:9092 --
from-beginning --topic match_results
```

You will see this:

```
{"HOME_GOALS":2,"AWAY_GOALS":1}
{"HOME_GOALS":1,"AWAY_GOALS":1}
```

You may wonder why the match id is missing. That is because the match id is actually the key of the message and we are not showing it. We can show the key like this:

```
docker exec kafka kafka-console-consumer --bootstrap-server localhost:9092 --from-beginning --topic match_results --property print.key=true --property key.separator=":"
```

We have added properties to the command so that it prints the keys with `:` as separator.

```
1:{"HOME_GOALS":2,"AWAY_GOALS":1}
2:{"HOME_GOALS":1,"AWAY_GOALS":1}
```

Initial Joins

As in Kafka Streams, we can join `TABLES` and `STREAMS` with the `JOIN` clause. There are some rules we need to follow when we do joins depending on the type of Join we do:

Join Type	Supported Joins	Windowed
Stream-Stream	INNER JOIN LEFT JOIN FULL JOIN	YES
Stream-Table	INNER JOIN LEFT JOIN	NO
Table-Table	INNER JOIN LEFT JOIN FULL JOIN	NO

There are some other rules to take into account:

- All columns referenced in the join expression must be of the same data type (STRING, INT, LONG, etc.).
- The partition count on each side of the join must be the same.
- The data in the underlying topics must have been written using the same partitioning strategy (usually, this means the producers will use the default parti- tioner, which creates a hash based on the input record's key).

Let's join the `match_event` table with the `players` to count the goals each player did:

```
SELECT p.id, p.name, count(me.id) goals
FROM match_event me
JOIN players p on me.player_id = p.id -- Join key for the stream match_events can be any column. But for the table we have to use the PRIMARY KEY
WHERE me.event_type = 'GOAL'
GROUP BY p.id, p.name
EMIT CHANGES;
```

This is great! But now, let's have more data. Let's add to our query assists and average goals per match:

```
SELECT p.id AS player_id,
       p.name AS name,
       p.nationality AS nationality,
       SUM(
               CASE
                   WHEN me.event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) goals, -- Obtain goals count summing all events that are goals
       CAST(SUM(
               CASE
                   WHEN me.event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           )
           AS DOUBLE) / cast(COUNT_DISTINCT((me.id)) AS DOUBLE) avg_goals, -- Sum
all goals, then divided by games played. We have to cast the results to DOUBLE to
be able to get a decimal number as result
       SUM(
               CASE
                   WHEN me.event_type = 'ASSIST' THEN 1
                   ELSE 0
                   END
           ) assists -- Count all events that are assists
FROM match_event me
         JOIN players p
              ON p.id = me.player_id
GROUP BY p.id, p.name, p.nationality
EMIT CHANGES;
```

This query looks complicated, but if you follow each part of the statement it will be easy to understand. The hardest part are the projections:

- `goals`: To get the count of all the goals, we need to only count events that are of type `"GOAL"` and ignore others. That's why we have the `CASE`. Also, instead of using `COUNT` we use `SUM`, so that we can ignore the `0`.
- `avg_goals`: Because we need to calculate the average per game, but we don't want to group by game, we simply calculate the amount of goals with the `SUM` and then we divide by the amount of matches the player has been in. To get that amount we use `COUNT_DISTINCT` with the match_id.
- `assists`: This is almost the same as `goals`, but we only SUM 1 when `event_type` is `"ASSIST"`.

You should see this on the output:

```
+------------------+------------------+------------------+------------------+-----
-------------+------------------+
|PLAYER_ID         |NAME              |NATIONALITY       |GOALS
|AVG_GOALS         |ASSISTS           |
+------------------+------------------+------------------+------------------+-----
-------------+------------------+
```

```
|1                |Lionel Messi      |Argentinian       |3                |1.5
|1                |
|2                |Cristiano Ronaldo |Portuguese        |2                |1.0
|1                |
```

Now let's write it into Kafka creating a new `SINK` table `player_stats`:

```
CREATE TABLE player_stats
WITH (
    KAFKA_TOPIC='player_stats',
    FORMAT='JSON',
    PARTITIONS=1
    ) AS
SELECT p.id AS player_id,
       p.name AS name,
       p.nationality AS nationality,
       SUM(
               CASE
                   WHEN me.event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           ) goals, -- Obtain goals count summing all events that are goals
       CAST(SUM(
               CASE
                   WHEN me.event_type = 'GOAL' THEN 1
                   ELSE 0
                   END
           )
           AS DOUBLE) / cast(COUNT_DISTINCT((me.id)) AS DOUBLE) avg_goals, -- Sum
all goals, then divided by games played. We have to cast the results to DOUBLE to
be able to get a decimal number as result
       SUM(
               CASE
                   WHEN me.event_type = 'ASSIST' THEN 1
                   ELSE 0
                   END
           ) assists -- Count all events that are assists
FROM match_event me
         JOIN players p
             ON p.id = me.player_id
GROUP BY p.id, p.name, p.nationality;
```

Now let's see if we have the data in the Kafka Topic:

```
docker exec kafka kafka-console-consumer --bootstrap-server localhost:9092 --from-
beginning --topic player_stats --property print.key=true --property
key.separator=":"
```

You will see this result:

```
{"PLAYER_ID":"1","NAME":"Lionel Messi","NATIONALITY":"Argentinian"}:
{"GOALS":3,"AVG_GOALS":1.5,"ASSISTS":1}
{"PLAYER_ID":"2","NAME":"Cristiano Ronaldo","NATIONALITY":"Portuguese"}:
{"GOALS":2,"AVG_GOALS":1.0,"ASSISTS":1}
```

Now the key is formatted as a `JSON` object because it's a composite key, formed with the columns used to group the data. There are ways to avoid this but we will see this in the future.