Lab 7: Stream Processing

Exercise 1:

Note: Save your answers in a document for review (consultation).

To start **docker-compose**:

```
cmd> docker-compose up
```

To launch **ksqldb-cli**:

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

Monitoring Changes on Netflix

Netflix invests billions of dollars every year in video content. With so many movies and TV shows in production simultaneously, it is crucial to communicate updates to various systems whenever a production change occurs (e.g., release date changes, financial updates, etc.). Our goal is to solve this problem using ksqlDB.

The purpose of this application is simple. We need to consume a stream of production changes, filter and transform the data for processing, and enrich and aggregate the data for reporting purposes.

The type of change we will focus on is season length changes for a show (e.g., Stranger Things, season 4 might initially be planned for 12 episodes but could be revised into a season of 8 episodes). This example was chosen not only because it models a real-world problem but also because it addresses the most common tasks you will encounter in your own ksqlDB applications.

The architecture of the change-tracking application is shown in the figure below.

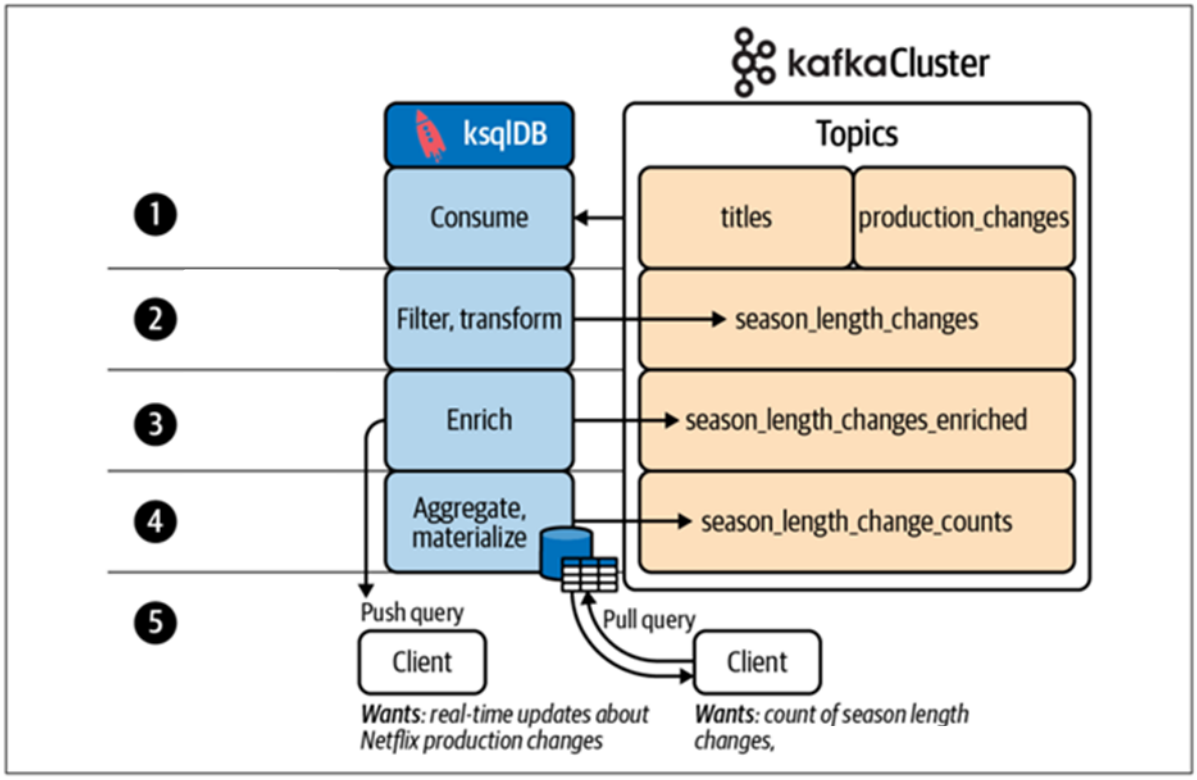

Figure 1: Architecture of the Netflix Change-Tracking Application.

Description of Steps:

Step 1: Our application will read from two topics:

- The **titles** topic, which contains the titles of videos (movies and TV shows) hosted on the Netflix service.
- The **production_changes** topic, which records any changes (budget, release date, season length, etc.) for a title that is in production.

Step 2: After consuming data from our source topics, we need to perform basic preprocessing to prepare the data from **production_changes** for enrichment. The preprocessed stream, which will only contain season length changes, will be written to a Kafka topic named **season_length_changes**.

Step 3: We will then perform data enrichment on the preprocessed data. Specifically, we will join the **season_length_changes** stream with the data from **titles** to create a combined record.

Step 4: Next, we will perform both windowed and non-windowed aggregations to count the number of changes over a given period. The resulting table will be materialized.

Step 5: Finally, we will make the enriched and aggregated data available to two types of clients:

- The first client will receive continuous updates via push queries.
- The second client will perform short-term pull queries (Pull queries return the current state to the client and then terminate. In this sense, they are much more like a SELECT statement executed on a traditional DBMS).

Source Topics

If you have data in a Kafka topic and want to process it using ksqlDB, you first need to examine the data in that topic and then determine how to model it in ksqlDB.

In our case, we have two main source topics: "titles" and "production_changes". An example of a record in each source topic is as follows:

Topic titles:

```
{
   "id": 1,
   "title": "Stranger Things"
}
```

Topic production_changes:

```
{
    "rowkey": "key1",
    "title_id": 1,
    "change_type": "season_length",
    "before": {
            "season_id": 1,
            "episode_count": 12
    },
    "after": {
            "season_id": 1,
            "episode_count": 8
    },
    "created_at": "2021-02-08 11:30:00"
}
```

Custom Types

1) Create a custom type in ksqlDB called **season_length** that represents the type of the two attributes **before** and **after**.

Once the type is created, you can view it using the following query:

```
ksql> SHOW TYPES ;
```

If you want to delete our custom type, execute the following statement:

```
ksql> DROP TYPE season_length;
```

Collections

2) Choose the most appropriate type of collection for **titles** and **production_changes**, justifying your choice.
3) Create the **titles** collection and the **production_changes** collection, keeping the following points in mind:
    - **id** is the identifier for the **titles** collection.
    - **rowkey** is the identifier for the **production_changes** collection.
    - The **created_at** column in **production_changes** contains the timestamp that ksqlDB should use for temporal operations (e.g., windowed aggregations and joins).
    - Assume the number of partitions = 4 in both collections.

Displaying Streams and Tables

The syntax for displaying information about all currently registered streams and tables is as follows (**LIST** and **SHOW** are interchangeable):

```
{ LIST | SHOW } { STREAMS | TABLES } [EXTENDED];
```

Example:

```
ksql> SHOW TABLES ;

ksql> SHOW STREAMS ;
```

If you need more information about the collections, you can use the **EXTENDED** variant of the **SHOW** command, as shown here:

```
ksql> SHOW TABLES EXTENDED ;

ksql> SHOW STREAMS EXTENDED ;
```

Describing Streams and Tables

Describing collections is similar to the **SHOW** command but operates on a single instance of a stream or table at a time. The syntax for describing a stream or table is:

```
DESCRIBE <identifier> ;
```

<identifier> is the name of a stream or table. For example:

```
ksql> DESCRIBE titles ;
```

Inserting Values

The syntax for inserting values into a collection is:

```
INSERT INTO <collection_name> [ ( column_name [, ...]] ) ]
VALUES (
   value [,...]
);
```

4) Insert data into both collections, **titles** and **production_changes**.

Queries

In this section, we will explore some basic methods for filtering and transforming data in ksqlDB. Remember that at this stage, our change-tracking application consumes data from two different Kafka topics: **titles** and **production_changes**. We have already completed Step 1 of our application (see Figure 1) by creating collections for each of these topics.

Next, we will address Step 2 of our application, which requires us to filter **production_changes** for season length changes only, transform the data into a simpler format, and write the filtered and transformed stream to a new topic called **season_length_changes**.

Note: To read data from the beginning of our topics, set the '**auto.offset.reset**' property to '**earliest**':

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

5) Write a Push Query to return all production changes created before April 14, 2023, at 12:00:00.
6) Write a Push Query to return all records from **production_changes** where the column **"change_type"** starts with the word **"season"**.
7) Create a derived stream named **season_length_changes** from **production_changes**, which contains only production changes of type **season_length**.
   - Assume that **season_length_changes** writes to a topic with the same name.
   - The topic has 4 partitions, 1 replica, and its messages are encoded in JSON.
   - The **season_length_changes** stream includes the following columns:
     - **ROWKEY**, **title_id**, **created_at**
     - **season_id:** This column takes the value of **after->season_id**.

       If **after->season_id** is **null**, **season_id** takes the value of **before->season_id**.

     - **old_episode_count:** This column takes the value of **before->episode_count.**
     - **new_episode_count:** This column takes the value of **after->episode_count**.
8) Write a Push Query to return the titles of all videos contained in **season_length_changes.**
9) Create the stream **season_length_changes_enriched**, which contains all attributes of **season_length_changes**, plus the **title** attribute from the **titles** collection.
   - The **created_at** column in **season_length_changes_enriched** contains the timestamp that ksqlDB must use for temporal operations.

10) Create the derived table **season_length_change_counts** from **season_length_changes_enriched.**
    - The **season_length_change_counts** table represents the number of changes made for each title.
    - Additionally, it contains the **episode_count** attribute, which represents the latest value of **new_episode_count**.
    - The **season_length_change_counts** table is created using a 1-hour Tumbling Window.

Exercise 2:

Present an example of using Spark Streaming (Structured Streaming) with Apache Kafka.

Note: The Python code must include: (1) the creation of a Streaming DataFrame to read data from Kafka, (2) an example of running a query illustrating how the result is updated after each insert.

Tip: To avoid blocking situations, use a **.py** file instead of a notebook and execute it in a command prompt.