Kafka Fundamentals Brain Teasers: M5: Integrating Kafka into Your Environment

Overview

Key concepts from Module 5 are:

- Kafka Connect
- Connectors
- Worker groups
- Confluent REST Proxy
- Data compatibility
- Confluent Schema Registry
- Schema evolution
- ksqlDB
- Kafka Streams
- Comparing ksqlDB, Kafka Streams, and the Producer/Consumer APIs

Here's the quick quiz on Module 5 (https://forms.gle/hGP6uvcwh9m925SQ6) from the Online Talk Series.

Before we proceed to problem #4, let's dig into one of the quiz questions and the correct answer as shown below in bold text.

Suppose we have an Avro schema to represent a `song`. It consists of the following fields and types:

- `title`, `string`
- `artist`, `string`
- `year`, `int`
- `song_id`, `int`

Suppose, additionally, we are using Schema Registry, and this schema is totally new. Then:

1. Suppose a producer wants to produce a message whose value is of type `song` to the topic `songs-topic`. We are using the Avro serializer. How does the producer know what schema ID to use?
    - Producer assigns schema ID
    - Schema ID found in producer's cache
    - **Schema ID generated by Schema Registry and sent to producer**
    - Schema ID randomly generated
2. Oops, we realized we also wanted to include a field for genre in the `song` schema, so we make a new version of the schema. Will this version use the same schema ID or a different one? Explain.
    - Same
    - **Different**
    - It depends
3. consumer, configured to use the Avro deserializer for values, reads from the topic `songs-topic` for the first time. How does it know how to deserialize the value of the message correctly?
    - Consumer can pull off appended schema ID and look up schema in Schema Registry
    - **Consumer can pull off prepended schema ID and look up schema in Schema Registry**
    - Consumer can pull off appended schema ID and look up schema by polling Kafka again

- Consumer can pull off prepended schema ID and look up schema by polling Kafka again

Question A above is about Confluent Schema Registry (https://docs.confluent.io/platform/current/schema-registry/index.html). It's about populating a topic for songs that our jukeboxes might play. There's no built-in data type for a `song`, of course, we have to define one. We could use a JSON schema to represent the fields above and then use Avro, along with Schema Registry, to perform the heavy lifting. Producers must serialize messages to send them across the "wire" and for them to be stored in Kafka. Avro can do all of the serialization work, as long as it has the `song` schema. From there, the schema used to serialize songs has to become associated with each song. Imagine in our jukebox application, we have thousands of songs. If we sent the `song` schema along with every one of them, that's thousands of extra copies costing extra bandwidth of network traffic. The solution: Schema Registry.

When we have a producer wired up to use Schema Registry and Avro and we first try to send a message whose value uses the `song` schema, the producer sends that schema to the Schema Registry. The Schema Registry doesn't know the schema, so it registers the schema, and sends back a new schema ID to represent the `song` schema. Let's say that schema ID is 72. Avro serializes the value and prepends the schema ID 72 to the message.

For Question B above, when we add a new `genre` field to the `song` schema, we are effectively creating Version 2 of the `song` schema. When we use this schema for the first time, the Schema Registry will register this as a *new version* of the `song` schema, so it will get a different schema ID. Let's say it's 105. In short, the Schema Registry stores a versioned history of all schemas and a schema ID uniquely identifies a *version of* a schema.

Per Question C, when a consumer reads a message whose value had been prepended with schema ID 105, the consumer will pull off the prepended schema ID 105, ask the Schema Registry for the corresponding schema, and get Version 2 of the `song` schema (the one with `genre` too), and use Avro to deserialize.

This question is here because it provides some additional context to the problem we are about to cover. The problem below is designed to dive deeper into the digital jukebox application discussed earlier.

Problem #5A: Back to the Jukebox with ksqlDB

Suppose we are building a jukebox app and there is also a topic `plays-topic`. Many messages have been produced to it. For each message:

- The key is a `song_id`, formatted as a `string` again
- The value of each message is a JSON-formatted string `song_play`, containing a timestamp when the song was played and `jukebox_id`, identifying which jukebox played it

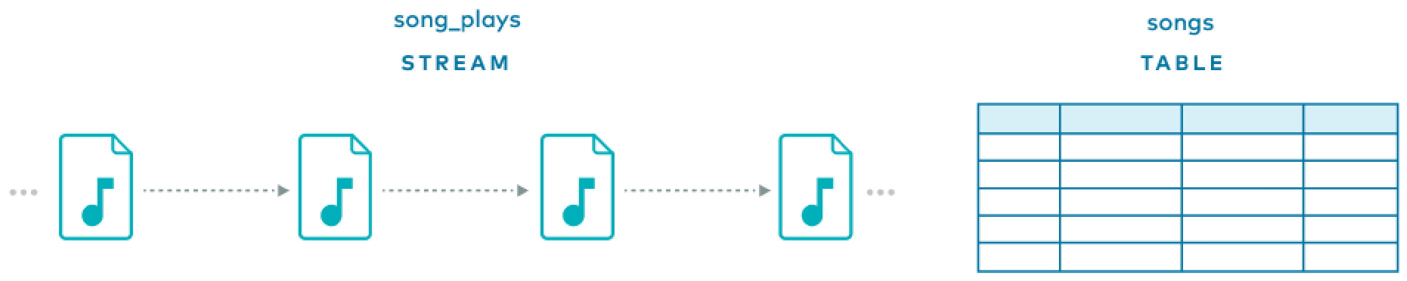
Suppose we have ksqlDB set up with a `TABLE` called songs, sourced from the topic `songs-topic`, and a `STREAM` called `song_plays`, sourced from the topic plays-topic. Write a ksqlDB statement that joins together each `song_play` with the corresponding title, artist, and year and creates a `STREAM` called `song_plays_enriched`.

Solution #5A: Back to the Jukebox with ksqlDB

Let's start by discussing these facts about our ksqlDB setup. We have:

- a `TABLE` called `songs`, sourced from the topic `songs-topic`
- a `STREAM` called `song_plays`, sourced from the topic `plays-topic`

Here's what that looks like:

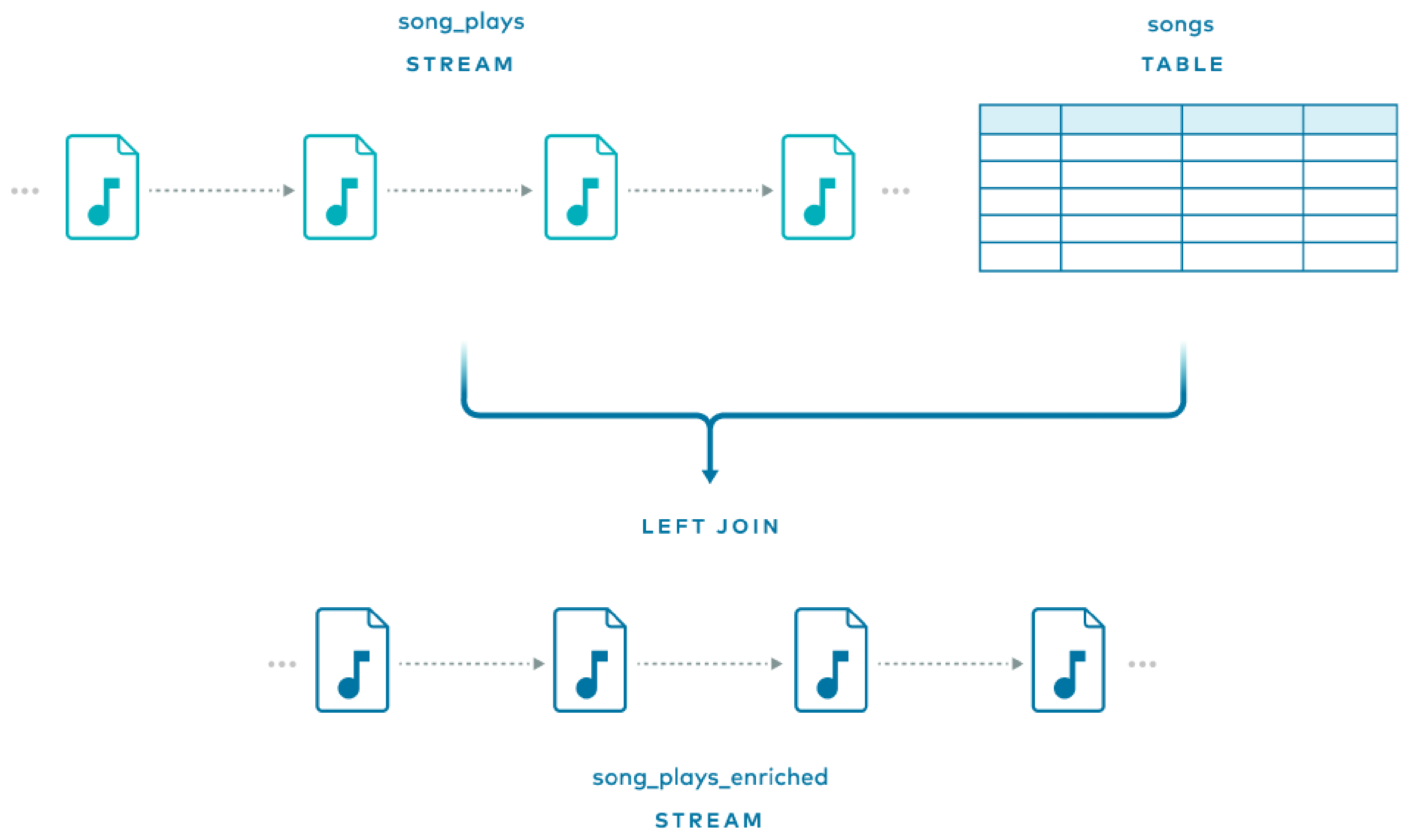


Our songs don't change very often, if at all. The topic `songs-topic` is what we might call a changelog topic, and we'd use it to source a `TABLE` in ksqlDB. Think of this table just like a table in most any relational database system you know.

The plays of the songs don't change either, but the point is that each song from `songs` will be played many times; each time the song is played is an event in the `STREAM` called `song_plays`. Each event is new; it is not an update of some previous event. It is a common pattern to have tables and streams like this, which we want to join. Each entry in the `STREAM` called `song_plays` will correspond to one entry in the `TABLE` called `songs`, or, put differently, each entry in the `TABLE` called `songs` will have many plays in the `STREAM` called `song_plays`. We want to left join our `song_plays` with our `songs`, as follows:

```
song_plays p LEFT JOIN songs s ON p.song_id = s.song_id
```

Here's what the result looks like:



This will go in the `FROM` clause. We want to know everything about the song play, along with the title, artist, and year. So, our `SELECT` clause will look something like this:

```
SELECT s.title, s.artist, s.year, p.jukebox_id, p.timestamp
```

We could simply put these parts together and run the query. But in this case, we'll do what is called a persistent query and save the result on the ksqlDB server. We've enriched the `STREAM song_plays` by joining it with a `TABLE`, but we still have a `STREAM`. The construct `CREATE STREAM AS SELECT` makes the stream persistent.

Putting it all together, we get this:

```
CREATE STREAM song_plays_enriched AS
   SELECT s.title, s.artist, s.year,
          p.jukebox_id, p.timestamp
     FROM song_plays p LEFT JOIN songs s
          ON p.song_id = s.song_id;
```

Ideally, we would add in an `EMIT CHANGES` line, too.

Problem #5B: Figure out the Mystery Query!

Suppose we are building a jukebox app and there is also a topic `plays-topic`. Many messages have been produced to it. For each message:

- The key is a `song_id`, formatted as a string again
- The value of each message is a JSON-formatted string `song_play`, containing a timestamp when the song was played and `jukebox_id`, identifying which jukebox played it

Suppose we have ksqlDB set up with a `TABLE` called `songs`, sourced from the topic `songs-topic`, and a `STREAM` called `song_plays`, sourced from the topic `plays-topic`. Here's a different ksqlDB statement that illustrates the power of ksqlDB:

```
CREATE TABLE mystery AS
   SELECT artist, count(*)
   FROM song_plays_enriched
   WINDOW TUMBLING (SIZE 1 HOUR)
   GROUP BY artist
   HAVING count(*) >= 5
   EMIT CHANGES;
```

What does this do?

Solution #5B: Figure out the Mystery Query!

Our `FROM` clause is `FROM song_plays_enriched`, indicating that our source of data is the stream that we created previously.

We add the following code:

```
GROUP BY artist
HAVING count(*) >= 5
```

This is just like most flavors of SQL. We're going to aggregate all of the plays by the same artist with the `GROUP BY`. The `HAVING` will filter to select only those artists who have at least five plays. You can decipher the `SELECT` too if you've worked with most any flavor of SQL: For each artist, we report the number of plays. All of this can be done with traditional SQL. Here's where ksqlDB (https://ksqldb.io/) comes in.

If we did only the lines we've discussed so far, we'd get a listing of all the artists who have ever had their songs played five or more times. That might be useful, but with lots of jukeboxes, it's going to morph into an unwieldy list of many, many artists. ksqlDB (and Kafka Streams) are based on time and therefore allow for windowing. The `WINDOW TUMBLING (SIZE 1 HOUR)` clause will provide a report every hour of artists who had been played five or more times out of the top five artists played across all jukeboxes in the last hour. We couple it with the `CREATE TABLE AS SELECT` syntax to make this table persistent and the `EMIT CHANGES` syntax to see these hourly reports.

Imagine implementing the jukeboxes: You could use these results to feature certain popular artists everywhere, given that we haven't filtered on location, on the jukebox or jukebox app home screens and potentially drive interest in more plays. This is just scratching the surface of stream processing. For a deeper dive, check out the Confluent Developer Skills for Building Apache Kafka (https://assets.confluent.io/m/a47e1a7a6ecd2fb/original/20200403_DS_Confluent-Developer-Skills-for-Apache-Kafka.pdf) course and full three-day Confluent Stream Processing with Kafka Streams and ksqlDB (https://assets.confluent.io/m/6fad0e105d5ba553/original/20200511-DS-Confluent_Stream-Processing-using-Apache-Kafka_Streams-and-ksqlDB.pdf) course.

Problem #5C - Extra Problem: Getting Song Data to the Jukebox

Continuing off problem #5B, all of the song data for all of the songs offered on this company's jukeboxes lives on a relational database. How might you get all of that data into the Kafka cluster in the first place?

Solution #5C - Extra Problem: Getting Song Data to the Jukebox

You might first think about using a producer to read from the database and produce messages to the `songs` topic. That would get the job done, but a faster, more fault-tolerant, well-tested, and robust way is to use Kafka Connect and have the relational database as your source. The popular JDBC source connector (https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc) would be perfect for this problem. You wouldn't need to write any code to do it; you'd just set configurations and deploy Kafka Connect.