Workshop: Real-time Stream Processing with KSQL

Confluent ATG

Table of Contents

1. Introduction

KSQL is the streaming SQL engine for Apache Kafka. This workshop will step through some practical examples of how to use KSQL to build powerful stream-processing applications:

- Filtering streams of data
- Joining live streams of events with reference data (e.g. from a database)
- Continuous, stateful aggregations

2. Pre-reqs / testing the setup

First things first, let's get connected to the lab environment and make sure we have access to everything we need.

2.1. Open up a terminal session

You will be prompted for a password, which is `ksqlr0ck$`

http://console.lchawathe.west2020.gcp.confluent-demo.io (http://console.lchawathe.west2020.gcp.confluent-demo.io)

2.2. Control Center

Test that you can open the Control Center web page by opening up a browser window and navigating to http://lchawathe:ksqlr0ck$@controlcenter.lchawathe.west2020.gcp.confluent-demo.io (http://lchawathe:ksqlr0ck$@controlcenter.lchawathe.west2020.gcp.confluent-demo.io)

2.3. Syntax Reference

You will find it helpful to keep a copy of the KSQL syntax guide open in another browser tab:
`https://docs.ksqldb.io/en/0.7.1-ksqldb/developer-guide/syntax-reference/`

3. KSQL

KSQL can be accessed via either the command line interface (CLI), a graphical UI built into Confluent Control Center, or the documented REST API (https://docs.ksqldb.io/en/latest/developer-guide/api/).

In this workshop we will be using the CLI. If you have used tools for MySQL, Postgres or Oracle's sql*plus before this should feel very familiar. Let's switch back to our terminal session and fire it up!

BASH
```
ksql
```

This will connect to your personal KSQL Server for the lab. You can open as any many of these, in separate terminal sessions, as you need.

Make sure that you get a successful splash screen and admire the ASCII-art :-)

BASH
```
                  ===============================================
                  =       _              _ ____  ____           =
                  =      | | _____  __ _| |  _ \| __ )          =
                  =      | |/ / __|/ _` | | | | |  _ \          =
                  =      |   <\__ \ (_| | | |_| | |_) |         =
                  =      |_|\_\___/\__, |_|____/|____/          =
                  =                   |_|                       =
                  =  Event Streaming Database purpose-built     =
                  =        for stream processing apps           =
                  ===============================================

Copyright 2017-2020 Confluent Inc.

CLI v5.5.0, Server v5.5.0 located at http://ksql-0-internal.test1.svc.cluster.local:8088
[....]
```

3.1. Looking around

Let's quickly get familiar with this environment by taking a look around:

You can navigate your KSQL command history much like a BASH shell:

- type `history` to see a list of previous commands
- `!123` will retrieve a previous command
- `ctrl-r` invokes a 'backward search' for commands matching whatever you type next, use arrow keys to navigate matches

3.2. See available Kafka topics and data

KSQL can be used to view the topic metadata on a Kafka cluster - try `show topics;` and `show streams`:

- `show topics;`
- `show streams;`

We can also investigate some data:

- `print 'xxxx' limit 3` or
- `print 'xxxx' from beginning limit 3;`

The topics we will use today are `ratings` and `mysql.customer.customers-raw`

The event stream driving this example is a simulated stream of events representing the ratings left by users on a mobile app or website, with fields including the device type that they used, the star rating (a score from 1 to 5), and an optional comment associated with the rating.

Notice that we don't need to know the format of the data when `print`ing a topic; KSQL introspects the data and understands how to deserialize it.

Because kafka topic names are case-sensitive ("Ratings" and "ratings" are two different topics on a Kafka broker) we take care to single-quote the topic names and correctly case them whenever we have to reference them. All the KSQL constructs though, like Streams and Tables and everything else, are case-insensitive as you would expect from most database-like systems.

SQL

```
ksql> PRINT 'ratings';
Format:AVRO
9/12/19 12:55:04 GMT, 5312, {"rating_id": 5312, "user_id": 4, "stars": 4, "route_id": 2440, "rating_time":
1519304104965, "channel": "web", "message": "Surprisingly good, maybe you are getting your mojo back at long
last!"}
9/12/19 12:55:05 GMT, 5313, {"rating_id": 5313, "user_id": 3, "stars": 4, "route_id": 6975, "rating_time":
1519304105213, "channel": "web", "message": "why is it so difficult to keep the bathrooms clean ?"}
```

Press Ctrl-C to cancel and return to the KSQL prompt.

3.2.1. Session properties

Investigate session properties with `show properties;`. Although we won't be adjusting these today, the session properties mechanism is how you can temporarily adjust various performance settings for any subsequent queries you issue.

4. Getting started with DDL

To make use of our ratings and customers topics in KSQL we first need to define some Streams and/or Tables over them.

4.1. Create the Ratings data stream

Register the RATINGS data as a KSQL stream, sourced from the 'ratings' topic

```
create stream ratings with (kafka_topic='ratings', value_format='avro');
```
SQL

Notice that here we are using the Schema Registry with our Avro-formatted data to pull in the schema of this stream automatically. If our data were in some other format which can't be described in the Schema Registry, such as CSV messages, then we would also need to specify each column and it's datatype in the `create` statement.

Check your creation with `describe ratings;` and a couple of `select` queries. (HINT: in recent KSQL versions your `select` query will need to have `emit changes` added at the end!)

What happens ? Why ?

Try `describe extended ratings;`

4.2. Bring in Customer lookup data from MySQL

Defining a lookup table for Customer data from our MySQL CDC data-feed is a multi-step process:

```
create stream customers_cdc with(kafka_topic='mysql.customer.customers-raw', value_format='AVRO');
```
SQL

Quickly query a couple of records to check it (remember you can `describe` the stream to see the column names!).

What happens when you query from this new stream ? Why is that the case ?

> If we aren't pushing new records into this stream (technically into it's backing topic) by changing data in MySQL then we won't see any query output.

What we are seeing - or rather, not seeing! - here in KSQL is actually the normal behavior of any Kafka client application (and remember, that's exactly what a KSQL query is!). By default, all Kafka client applications when they start up will consume messages which arrive in their input topics *from that moment forwards*. Older records in the topic are not consumed. We can control this behavior though by setting a configuration property, called 'auto.offset.reset'. In KSQL, the various configuration settings for our SQL apps can be inspected by typing `show properties` in the KSQL CLI. if you try that you can see a whole long list of technical defaults, almost all of which we can safely ignore :-) We adjust the property which controls where new queries start reading from like this:

```
set 'auto.offset.reset' = 'earliest';
```
SQL

Now all the subsequent queries we issue will pick up this setting and consume *all* the records in a topic, including those produced in the past. This setting will stay in effect for the rest of your KSQL session.

4.2.1. Re-format the CDC data

OK, back to our CDC customer data. Now that we have changed the value of `auto.offset.reset` we can run a new query against the `customers_cdc` stream and now we should get back some data. We should receive 20 records back, representing the original inserts into MySQL that ran when we populated the workshop environment.

This is also a chance to practice the art of "struct-dereferencing" with the "→" operator.

```
select after->first_name as first_name, after->last_name as last_name from customers_cdc emit changes;
```
SQL

Observe that the message structure here is actually quite complex and nested. Remember you can get a view of that by `describe customers_cdc;`. You should see that there's a bunch of metadata about the actual change in MySQL (timestamp, transaction id, etc) and then a 'Before' and 'After' image of the changed row. For our use-case, we want to extract just the changed record values from the CDC structures, re-partition on the ID column, and set the target topic to have the same number of partitions as the source `ratings` topic:

```
create stream customers_flat with (partitions=1) as select
after->id as id,
after->first_name as first_name,
after->last_name as last_name,
after->email as email,
after->club_status as club_status,
after->comments as comments
from customers_cdc partition by after->id;
```
SQL

Register the CUSTOMER data as a KSQL table, sourced from this new, re-partitioned, topic

```
create table customers (rowkey int key) with (kafka_topic='CUSTOMERS_FLAT', value_format='AVRO');
```
SQL

So now we have a "pipeline" of queries to read the CDC data, reformat it, and push it into a KSQL table we can use to do lookups against. Let's check our table at this point:

```
select * from customers emit changes;
```
SQL

4.2.2. Changing data in MySQL

In a new terminal window, side-by-side with the one you are using already, connect to the server again (just like we did right at the beginning), and this time instead of running KSQL we want to launch the MySQL client:

```
mysql
```
BASH

You should be able to see your source CUSTOMERS table here, and inspect it's original 20 records with `select * from customers`.

Try inserting a new record or updating an existing one Example: update name of a record to be your own name

```
> update customers set first_name = 'Jay', last_name='Kreps' where id = 1;
```
SQL

If you leave your KSQL `select…from customers;` query running in the first window, watch what happens as you change data in the MySQL source.

We can further check our work with

```
describe extended customers;
```
SQL

check the "total messages" value and see how it changes over time if you re-issue this same instruction after making some more changes in MySQL.

4.3. Identify the unhappy customers

Now that we have both our ratings and our continuously-updating table of customer data, we can join them together to find out details about the customers who are posting negative reviews, and see if any of them are our valued elite customers.

- Back in KSQL, we start by finding just the low-scoring ratings

```
select * from ratings where stars < 3 and channel like 'iOS%' emit changes limit 5;
```

(play around with the `where` clause conditions to experiment.)

- Now convert this test query into a persistent one (a persistent query is one which starts with `create` and continuously writes its' output into a topic in Kafka):

```
create stream poor_ratings as select * from ratings where stars < 3 and channel like 'iOS%';
```

- Which of these low-score ratings was posted by an elite customer ? To answer this we need to join our customers table:

```
create stream vip_poor_ratings as
select r.user_id, c.first_name, c.last_name, c.club_status, r.stars
from poor_ratings r
left join customers c
on r.user_id = c.rowkey
where lcase(c.club_status) = 'platinum';
```

- What do you think would happen if you went and changed the `club_status` of a customer while this join query is running ?

Let's try that!

4.4. Monitoring our Queries

So what's actually happening under the covers here ? Let's see all our running queries:

```
show queries;
explain <query_id>;  (case sensitive!)
```
SQL

4.4.1. View Consumer Lag for a Query

Over in the Control Center browser window, navigate to 'Consumers' and, in the table of consumer groups, try to find the one for our join query and click on it.

All the names are prefixed with '*confluent_ksql*' plus the ID of the query, as shown in the output of `explain queries`.

What do we see ?

It's also possible (although not setup in this lab environment) to monitor a series of JMX metrics for each running query.

5. Application Overview

Also in the Control Center browser window, now select 'ksqlDB' in the left navigation bar. The summary table which gets displayed on the right will give us an overview of how many queries are continuously running within this application.

Now click into the application itself (here called just 'KSQL') and we can see the browser-based version of the CLI tool but the thing we want to focus on next is the 'Flow' tab where we can see the overview of how our new application is composed and even sample records from each stage by clicking it.

6. Extra Credit

Time permitting, let's explore the following ideas:

- which customers are so upset that they post multiple bad ratings in quick succession ? Perhaps we want to route those complaints direct to our Customer Care team to do some outreach…

```sql
select first_name, last_name, count(*) as rating_count
from vip_poor_ratings
window tumbling (size 5 minutes)
group by first_name, last_name
having count(*) > 1 emit changes;
```

SQL

This may take a minute or two to return any data as we are now waiting for the random data generator which populates the orginal 'ratings' to produce the needed set of output.

And of course we could prefix this query with `create table very_unhappy_vips as …` to continuously record the output.

7. Follow-on Discussion Points

- UDFs
- Testing tools
- mask the actual user names in the ouput
- explore and describe the available functions
- create a new stream over a topic that doesn't exist yet
- use `insert…values` to write a couple of test records into this new topic
- join it to one of our existing streams or tables

8. Further resources

Don't forget to check out the #ksql channel on our Community Slack group (https://slackpass.io/confluentcommunity)

Last updated 2020-09-16 17:01:58 -0700