Internal KSQL test system, June 2017

Created by Nick Dearden, last modified by Chris Egerton on Jun 21, 2017

Introduction

Since the last KSQL internal test, where we offered the chance to kick the tires on an embryonic all-in-one JAR file which ran in a terminal window and combined the functions of a basic CLI with in-process query execution, things have moved forward. And moved a lot! Thanks to all the hard work put in over the last couple of months we now have the chance to play with a considerably more advanced and, hopefully, interesting test system!

Now it's time for us to ask for your feedback and hands-on experimentation once again, as the big reveal in late August draws closer, in order that we can tune up anything that's still confusing and gain some operational experience of the server-side daemons under multi-user load. I'm sure we can flush out a few more bugs 🙂.

Architecture

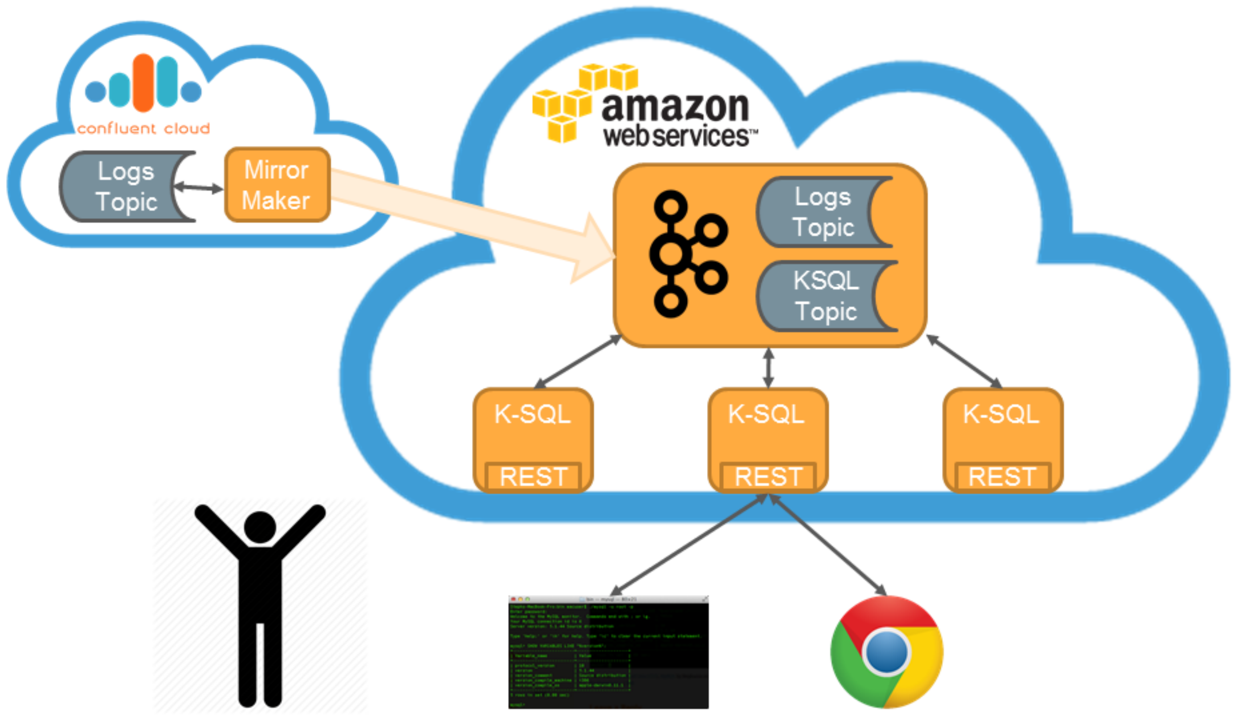
As alluded to above, we've evolved the software significantly since last time around. Now there's a choice of how to run KSQL:

1. in an entirely self-contained way, locally on your laptop, which is intended for developers to use while experimenting and iterating. If you participated in the first 'quickstart demo' earlier in the year, this is still a similar experience.

2. in a client-server fashion, where one or more pools of remote worker processes handle and share the work of actual query execution. A remote CLI can be connected to these workers via a REST interface to interact with running queries. More details of the technical design can be found in KQL Distributed Service Design but, for the impatient, the TL;DR; is that each worker process embeds a REST interface over which queries and commands can be sent. A worker receiving such an instruction will share it with all the workers in the same pool via the 'command topic pattern' (writing to a shared compacted topic, similar to blueway). Every worker in the pool listens on the command topic and executes the statements found there. Each new query read from the command topic currently causes the worker reading it to start a new Streams job (yes, this has the potential to use up lots of threads on the worker). The worker pool forms a single consumer group and so Kafka will take care of distributing input topic partitions across these Streams jobs, and rebalancing them in the event that workers are added to or removed from the pool.

Deployment

For the purposes of this test we've deployed several KSQL worker nodes in AWS, sharing a local broker, and are continuously mirroring log data from the CCloud 'Staging' environment into the broker to use as live input data to query against. The following diagram should supply the necessary 1k words:



(Yes that's you, the ecstatic K-SQL user, in the bottom corner there)

To interact with the KSQL workers you can either download a pre-built version of the CLI or, just for this test, we whipped up a very rough-and-ready HTML page to get you started with the fewest possible dependencies.

The CLI is the preferred route for best experience, and the one we intend to ship to customers in the upcoming 'Developer Preview'.

What we want from you

Download the CLI (or jump straight to the web interface) and try out the test cases described in detail below. Make your own variations of them and experiment a little. There are no limits imposed on creating new derived streams and tables so feel free to do so, just remember you are playing in the same environment shared by the other users.

Your candid feedback, in the form either of comments at the foot of this page or in Slack (#ksql), is very welcome. Especially interesting are bug reports and things that confused you or seemed jarring, along with suggestions for improvement - we can use these as we build out the documentation to stress the necessary concepts for our future users. Do bear in mind that it's far from finished software yet.

Also, we'll be closely watching the server-side performance and logs as this test phase unfolds so we can start to learn the operational quirks.

System Status
System status updates - e.g. if we manage to crash the whole thing - will be posted to #ksql in slack.

Getting started

You have an initial choice to make. **Either**:

1. download the CLI attached here. This will give you the best UX and is closest to how we want our eventual users to interact with KSQL.
   Run it like this:
   ```
   > java -jar ksql-cli-1.0-SNAPSHOT-standalone.jar remote http://ec2-54-149-235-246.us-west-2.compute.amazonaws.com:8080
   ```
2. just click through to the internal demo HTML page **here**. Please note that we don't currently intend to ship this page to anyone outside the company, it's just to (a) ease testing; and (b) prove that the REST interface is working as intended. So don't bother filing any tickets to tell us how beautiful it is, your flattery will be wasted 🙂.

Important: Network Caveat
Note: especially in light of the recent Jenkins server incident, the test servers are deployed in a secured network within our AWS account, meaning they are **not accessible from anywhere outside the office(s)**. If you're in either the Palo Alto or London office, you should be able to connect through just fine without taking any special steps. If you're elsewhere you will need to **use VPN first** to connect into Palo Alto and be tunneled through from there. VPN setup instructions were posted in #engineering last week, and you are kindly requested to use responsibly so as to reserve some concurrent connections for emergency access by the CCloud team.

Use-cases and Available Data

We wanted to setup a test which allows us all to explore 2 prime use-cases for KSQL, namely streaming-transformation (the 'T' of real-time ETL, if you like) and event monitoring. As mentioned earlier, the CaaS monitoring log is the stream of log records from the CaaS cluster. Using MirrorMaker tool we bring the CaaS log into a Kafka topic in KSQL Kafka cluster. The messages in the topic are in JSON format and have the following schema:

```
 Field         | Type
------------------------
 ROWTIME       | int64
 ROWKEY        | string
 SOURCE        | string
 KEY           | string
 TIMESTAMPMS   | int64
 OFFSET        | int64
 LOGGER        | string
 LOGLEVEL      | string
 CLOUD         | string
 REGION        | string
 NETWORKID     | string
 HOST          | string
 K8SID         | string
 K8SNAME       | string
 CLUSTERID     | string
 CLUSTERNAME   | string
 TENANTID      | string
 TENANTNAME    | string
 CPCOMPONENTID | string
 SKUID         | string
 CAASVERSION   | string
 ZONE          | string
 SERVERID      | string
 MESSAGE       | string
```

Here is the content of a sample message from the CaaS monitoring log, just to give you an idea of how it looks:

```
1497640110862 | 1497640108568 | log | zookeeper-1-fjgwc | 1497640108568 | 8432087 | org.apache.zookeeper.server.NIOServerCnxn | INFO | aws
```

In the demo environment, we've already registered the input topic in the catalog and defined the LOGSPROCESSED stream over it. As the first simple example of streaming transformation, we also created a CAASLOGS stream from the LOGSPROCESSED one. This is a simple re-partitioning and re-keying task to make subsequent queries run faster, and we defined it like this:

```
> CREATE STREAM CAASLOGS WITH (partitions = 4) AS SELECT * FROM LOGSPROCESSED PARTITION BY timestampMs;
```

Looking Around

Let's first see what topics, streams and tables exist in the system. You can get this information using the following statements:

```
SHOW TOPICS;
SHOW STREAMS;
SHOW TABLES;
```

Syntax
A mostly-complete list of available statements, CLI commands, and SQL syntax can be found at https://confluentinc.atlassian.net/wiki/display/Engineering/KSQL+Syntax+Guide

If you run a SHOW STREAMS statement you will see the list of available streams similar to the following:

```
Stream Name     | Ksql Topic
--------------------------------------
 COMMANDS       | __COMMANDS_TOPIC
 LOGSPROCESSED  | LOGS_PROCESSED_TOPIC
 CAASLOGS       | CAASLOGS
```

You can use the DESCRIBE command to see the schema of a stream or table.

Here is the result of DESCRIBE command for CAASLOGS stream:

```
Field         | Type
------------------------
 ROWTIME       | int64
 ROWKEY        | string
 SOURCE        | string
 KEY           | string
 TIMESTAMPMS   | int64
 OFFSET        | int64
 LOGGER        | string
 LOGLEVEL      | string
 CLOUD         | string
 REGION        | string
 NETWORKID     | string
 HOST          | string
 K8SID         | string
 K8SNAME       | string
 CLUSTERID     | string
 CLUSTERNAME   | string
 TENANTID      | string
 TENANTNAME    | string
 CPCOMPONENTID | string
 SKUID         | string
 CAASVERSION   | string
 ZONE          | string
 SERVERID      | string
 MESSAGE       | string
```

Query 101 - Data Investigation

Now that we know the schema of CAASLOGS stream, let's run some exploratory KSQL queries.

Let's first see all of the columns in the stream, including the automatically-supplied 'virtual columns' for the message timestamp and message key. You can do this by running the following query:

```
SELECT * FROM CAASLOGS;
```

After a second or two you will see the latest rows belonging to the CAASLOGS topic start populating. To interrupt this query (it will never end otherwise!), press CTRL-C in the CLI or, in the web UI, hit the 'Cancel Query' button.

Now let's run a simple projection and filtering query where we want to see TENANTNAME, LOGLEVEL and MESSAGE when LOGLEVEL is "ERROR". Here is the query that will produce this result:

```
SELECT TENANTNAME, LOGLEVEL, MESSAGE FROM CAASLOGS WHERE LOGLEVEL = 'ERROR';
```

Case Sensitivity
All statement and command keywords, as well as topic names and column names, are case-insensitive in KSQL. We tend to write the examples in upper-case as that's a common convention but you can enter them however you wish.

You should be seeing the result of the query streaming to the screen similar to the following:

```
perf-test | ERROR | Closing socket for 100.96.52.103:9072-100.96.73.4:39476 because of error
perf-test | ERROR | Closing socket for 100.96.52.103:9072-100.96.73.4:39488 because of error
perf-test | ERROR | Closing socket for 100.96.52.103:9072-100.96.73.4:39484 because of error
perf-test | ERROR | Closing socket for 100.96.52.103:9072-100.96.73.4:39480 because of error
soak-test | ERROR | Found invalid messages during fetch for partition [__consumer_offsets,39] offset 0 error Record is corrupt (stored crc = 2216632250, computed crc = 3318419633)
soak-test | ERROR | Found invalid messages during fetch for partition [__consumer_offsets,32] offset 328533 error Record is corrupt (stored crc = 2913927020, computed crc = 381511739)
```

Note that if the current error rate is very low (CCloud is very stable!), you may have to wait a little while to see any result data show up! TIP: if that happens, try switching 'ERROR' to 'INFO'.

Aggregations

Now let's try a more interesting query where we want to see the number of warning messages for each tenant in 10 second tumbling windows. Here is the query:

```
SELECT TENANTID, COUNT(*) FROM CAASLOGS WINDOW TUMBLING(SIZE 10 SECOND) WHERE LOGLEVEL = 'WARN' GROUP BY TENANTID;
```

The following is a sample of results for the above query:

```
 t46 | 20
 t46 | 23
 t47 | 33
 t46 | 46
 t46 | 4
 t46 | 25
 t46 | 28
 t47 | 39
```

Note that what we're seeing here is the *changelog* output - every new input record to the aggregation will trigger a new output record to be displayed. You can think of this as always showing "the latest result so far" for each window, updated every time new data is received.

Let's make the above query a bit more complicated by adding a HAVING clause and show the tenant warning counts in the 10 second window only if the count is greater than 30. Here is the revised query:

```
SELECT TENANTID, COUNT(*) FROM CAASLOGS WINDOW TUMBLING(SIZE 10 SECOND) WHERE LOGLEVEL = 'WARN' GROUP BY TENANTID HAVING COUNT(*) > 30;
```

Here is a sample results that you would see:

```
t46 | 44
t47 | 51
t46 | 68
t46 | 31
t47 | 41
t46 | 55
t46 | 39
```

Functions

KSQL provides a set of functions and string manipulation features that can be used in your queries. Let's try some of them to inspect the MESSAGE column in the CAASLOGS stream.

The following query will count the number of accepted socket connection by each tenant in 20 second time windows:

```
SELECT TENANTID, TENANTNAME, COUNT(*) FROM CAASLOGS WINDOW TUMBLING(SIZE 20 SECOND) WHERE MESSAGE LIKE 'Accepted socket connection%' GROUP
```

Here is an sample output for this query:

```
t57 | saturation-test | 2
t47 | perf-test | 2
t46 | soak-test | 1
t39 | jon | 2
t47 | perf-test | 5
t39 | jon | 3
```

Finally, the following query will show the ever-increasing total number of messages for tenant with name 'caas-monitor'. Notice how we didn't specify any windowing criteria, so the results are computed for all time going forwards from 'now' (when you submit the query).

```
SELECT TENANTID, TENANTNAME, COUNT(*) FROM CAASLOGS WHERE TENANTNAME = 'caas-monitor' GROUP BY TENANTID, TENANTNAME;
```

Note that the above query does not specify any aggregate window so the aggregation will keep happening until we stop the query. Here is a sample output for the above query:

```
t39 | caas-monitor | 27
t39 | caas-monitor | 29
t39 | caas-monitor | 61
t39 | caas-monitor | 66
t39 | caas-monitor | 98
t39 | caas-monitor | 116
t39 | caas-monitor | 142
t39 | caas-monitor | 180
t39 | caas-monitor | 209
t39 | caas-monitor | 239
t39 | caas-monitor | 255
t39 | caas-monitor | 281
t39 | caas-monitor | 313
t39 | caas-monitor | 350
t39 | caas-monitor | 392
t39 | caas-monitor | 428
t39 | caas-monitor | 429
t39 | caas-monitor | 435
```

Persisting Your Queries

So far all we've done is select some results to be interactively returned to our client. But what if we've now refined our queries and want them to run in a continuous, lights-out mode and be always writing to an output stream for consumption by some downstream system ? Easy! We simply prefix our 'SELECT ....' statement with either a 'CREATE STREAM' or a 'CREATE TABLE' clause, depending on whether we want to create a changelog output or a simple stream of records. Check the earlier example of how we defined CAASLOGS for a simple example.

Next Steps

You've made it to the end of the pre-canned portion of the test, now time for some fun! Using what we've learned so far, and the information about available functions and syntax found in KSQL Syntax Guide (there are many more functions than we've used so far) go ahead and create your own stream or table and then try interactively querying from it. Have fun with it, and remember to let us know your feedback!