Spain

Blog Post
Author: Lionel Ferder

Testing the performance of KSQL joins

SWORD
UPGRADE YOUR BUSINESS

KSQL (now named KsqlDB) is a query language built on top of Kafka Streams that allows you to write SQL like queries for stream processing. One of its nicest features is the possibility to perform joins between two different streams. At Lemonade we've been using this functionality for calculating business statistics for one of our clients (you can read more about it here).

Given that some of our joins will be long lasting, we decided to run some tests to see how the joins behave.

The test performed is quite simple, create two streams, join them and see what happens.

First, I created two topics:

kafka-topics –create –zookeeper 127.0.0.1:2181 –replication-factor 1 –partitions 12 –topic orders –config retention.ms=3600000
kafka-topics –create –zookeeper 127.0.0.1:2181 –replication-factor 1 –partitions 12 –topic shipments –config retention.ms=3600000
Both topics have a retention period of 1 hour, so records older than one hour will be removed from the topic.

Now it's time to create the streams and queries in KSQL:

```
CREATE STREAM orders WITH (KAFKA_TOPIC='orders',
VALUE_FORMAT='AVRO', PARTITIONS=12);
CREATE STREAM shipments WITH (KAFKA_TOPIC='shipments',
VALUE_FORMAT='AVRO', PARTITIONS=12);
```

And most importantly, the join stream:

```
CREATE STREAM shipment_info
WITH(KAFKA_TOPIC=shipment_info_query_topic',
PARTITIONS=12, VALUE_FORMAT='AVRO') AS
SELECT
orders.client_name,
orders.order_number,
shipment.address
FROM
orders
INNER JOIN shipments WITHIN 7 DAYS ON orders.order_id =
shipments.order_id;
```

Once the shipment_info stream is created, the new topics appear in kafka for storing the state:

_confluent-ksql-default_CSAS_SHIPMENT_INFO_1-KSTREAM-JOINTHIS-xxxx-store-changelog: This topic will contain the shipments. The retention configuration for this topic is "cleanup.policy:compact, delete" and "retention.ms:1296000000" which is 15 days.

_confluent-ksql-default_CSAS_SHIPMENT_INFO_1-KSTREAM-JOINOTHER-xxxx-store-changelog: same but for the orders.

I've created a program that does the following:

1.Send orders for a fixed amount of time.
2.Once the orders are sent, it starts sending the related shipments.
3.Consumes the messages from the shipment_info_query_topic until there is nothing more to consume.

First Execution

The first time I executed the program I configured it to send orders to kafka for one hour.

I could see in kafka that during that hour the orders and its changelog topics grew, but then passing the hour mark the orders topic started to decrease in size since the retention policy was set to one hour, but the changelog topic stayed the same, maintaining all the orders.

During that hour a total of 19,798,681 orders were sent. For the sending process I added a sleep of 100 ms every 1000 records sent to kafka.

When all the orders were sent, it was the turn of the shipments, those 19 million shipments were sent in 56 minutes (again, a sleep was included).

Nevertheless, at the end of the process, only 11 million records were consumed from the shipment_info_query_topic.

In the following graph, the number of orders sent are in green, the number of shipments sent are in blue and the number of the join results consumed are in red.

So what happened? Why were only 11 million join results stored in the shipment_info_query_topic? During the sending, KSQL was having problems keeping up and the lag between the shipments topic and what was stored in the changelog grew bigger and bigger. After one hour, the records stored in the shipments topic started to expire and 8 million of them were never added to the changelog and never processed by KSQL. They were gone. Forever.

A solution to this problem could be to simply change the retention policy of the shipments topic to allow KSQL enough time to process everything, or even better add more KSQL engines in the cluster so the processing goes faster. Given that the topics have 12 partitions I could have up to 12 KSQL engines sharing the workload.

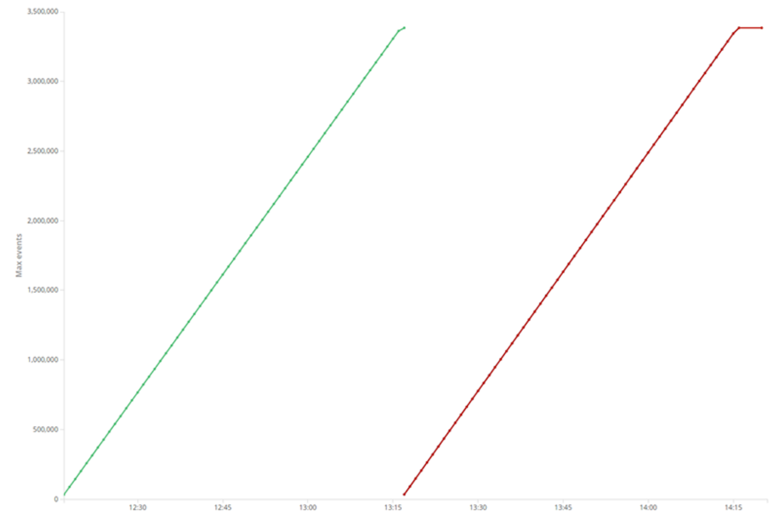
But. Unfortunately I only had one server with 8 GB of RAM running KSQL, Kafka, Zookeeper, Schema Registry and my java producer/consumer application, so there was not much margin to add more KSQL servers. So I decided to make the sending to kafka slower, changing the sleep from 100 ms to 1000 ms, after all, our application was not going to process 19 million records per hour (though yours might!).

I re-executed the test with these new parameters and got the following:

Second Execution

For this second execution where we are sending only about 1000 records per second I got the following results:

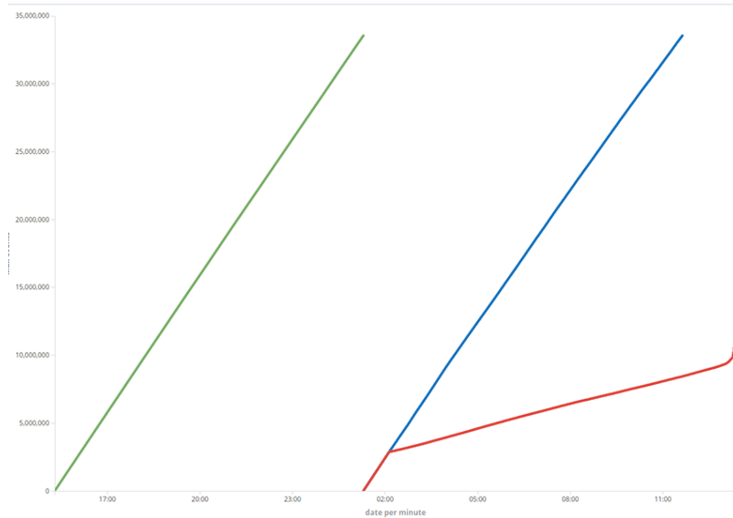
Number of orders sent: 3,385,000
Number of shipments sent: 3,385,000
Number of shipment info consumed: 3,385,000
This looks better, here KSQL was able to keep up, join all the orders and shipments and we were able to consume all the joined messages.



Again, here the green line represents the orders sent but the blue (sent shipments) and red (read joins) overlap since there was almost no lag in KSQL.
At the end of the processing, the changelog topics were consuming about 1.5 GB of disk.

Third Execution

So, if everything went smoothly with a 1000 records per second throughput in a two hours test, things should scale and go smoothly on a 20 hours test, right?



Unfortunately not. Up to around 3 million there was no lag in processing the join, but then things got sour and we were left with 23 million unprocessed joins.

The final numbers are:

•Number of orders sent: 33,558,000
•Number of shipments sent: 33,558,000
•Number of shipment info consumed: 10,516,176

As for disk space, the KSQL logs consumed about 8 GB and RockDB another 7GB.

The conclusion I get from this exercise is that it’s not only the high throughput affecting the lag but as well the number of records being processed. For our production environment we will probably have to add more KSQL nodes to the cluster and increase the retention time of the topics participating in the join operations.