Introduction to Apacha Kafka - Lab Report

Mathilde Da Cruz - 21801531

October 6, 2022

Send to : he.ciritoglu@criteo.com

Contents

1 Lab Report

2 Kafka use cases

2.1 Question 0

Can or cannot ?

1. Yes (correct)
2. No (correct)
3. Yes (correct)
4. Yes (correct)
5. Yes (correct)
6. Yes (correct)

2.2 Question 1

A topic is an abstraction of a meaningful information. It is the equivalent of a table in a relational database. Each topic is distributed over the partitions. A topic contains messages (which are equivalent of the rows of the table).

2.3 Question 2

We can't modify an event. Indeed, an event is an atomic action that describes a change within the system, and it works only in an append fashion. Modifying an event would be like trying to modify the past. You can only append new event (messages) eventually with the same key, so that it would some kind of "replace" the old one. But it is really a new message appended at the end of the partition, not a modification.

3 The lab environment : a fully configurated remote desktop

4 Why Zookeeper ?

4.1 Question 3

I tried to kill zookeeper (sudo systemctl stop zookeeper). Then I checked if my brokers were still working (with status) and they're indicated as running as before. Then I conclude that a Kafka cluster can run even without zookeeper.

5 Produce and consume some events through Kafka

5.1 Question 4

I have this information :

kafka_server_topic_messagesinpersecbroker$_id$ = "3", $instance$ = "$localhost$: 9194", job = "$prometheus$", $topic$ = "$my - topic - rf1 - p1$"

Then I can deduce that it is broker number 3. But it could be another one.

5.2 Question 5

A kafka offset is an integer (a bit) incrementally attributed to each message within a partition : It can be seen as a primary key for the messages of a partition because each message within a partition as a different offset. Also, as we increment it at each new message of the partition, the message with the lower offset is the older. Nevertheless, offsets are only meaningfull within a partition, offset of messages in different partitions canno't be compared.

6 Experiment failure

6.1 Question 6

We killed the broker that was handling the load, but because we only had 1 partition and 1 replica, we lost all our data because we only had one copy of it, which was on broker 3, and all the data was in the same partition. Then our application is not fault tolerant. To solve the issue we could change the number of replicas to more than one.

Another metric reporting this change could be the bytes in per second.

The leader of my new topic with replication factor 2 is broker 2.

We can see in prometheus that the messages are now sent to broker 1. The application is more fault tolerant than the previous one because it can handle one failure (but no more). Nothing stopped, we didn't loose data and the application is still running.

Now the leader is broker number 1.

6.2 Question 7

The kafka replication model is a master-follower. Indeed there's a leader, and if we want to push data, we talk to the leader. Data is first written to the leader and then replicated to others. But the master and followers keep the same data. If the leader is down, another one is chosen.

6.3 Question 8

This configuration is not scalable, because we have only one partition and 2 replicas, and 3 brokers. At most, we can have as many replications as brokers. With only 2 replicas, we can use only 2 of our brokers. If we have a huge amount of messages it will be difficult to scale (with 2 replicas and only one partition we can only scale vertically, but there will be physical limits at some point). Also if our 2 brokers fail, we will lose all the data. We could try to scale horizontally by adding another replication using the other broker.

7 Scaling -broker side

The partitions leaders and replicas are well balanced (it was not at the beginning because i forgot to start again the broker I killed before)

7.1 Question 9

The producer could choose the partition in a cyclic loop (0 then 1 then 2 then 3 then 0 then 1 etc), but here it is not the case. It seems random but we can suppose that the producer is using a partitioning function (hash function) depending on the key of the message. It could be modulo function.

7.2 Question 10

By checkin on prometheus, we can see that broker 1 is receiving more messages than the others. As there are 4 partitions and 3 brokers, one of the brokers has to handle more partitions than the others. Indeed with –describe, we note that broker 1 is the leader for 2 partitions.

7.3 Question 11

We have 4 partitions and 2 replicas. So if one broker is dead, we won't loose any data (but if 2 brokers fail we could loose one partition). Then it's fault tolerant (to one failure at a time). Also, we could scale horizontally by adding more replicas (we could have a replication factor 3 and use all the brokers), or vertically but only until a physical limit once again. But as it is now, it is not scalable. Here we have 2 consumers but in different consumer groups, so they do not share the workload and end up reading the same data so it is not really scalable. To scale in a better way, we could have 4 consumers in 1 group, one for each partition.

8 Scaling - consumer side

We observe that with 2 consumers one consumer is assigned with partitions 0 and 1, and the other one with partitions 2 and 3. With 4 consumers, each one has a partition.

8.1 Question 12

This application is fault tolerent because we have enough replicas to not lose data in cas of a failure. Also the topics are split into 4 partitions so the load can be well balanced within the different brokers. Also we can easily add consumers within the same consumer group and share the workload between them.

8.2 Question 13

If we have more consumers than we have partitions, one consumer won't be assigned to a partition. Then we don't have all instances receiving records.

The last tuple read by the last consumer is (1939, 0)
The last tuple for the producer is (2019, 1)
The first tuple read by the new consumer is (1940, 1).

8.3 Question 14

I suppose that Kafka is storing consumer offset in the order they arrive. The consumer application is at least once.

9 Data consistency inside Kafka

9.1 Question 15

We don't have an order record, the consumer could read a message after another that was send after. Indeed, the consumer is following the offset, but we can't compare offset within different partitions. Nevertheless we have the guarantee that within each partition the messages are read in the order they arrived, ie. in the order of their offset : from the oldest to the youngest.

9.2 Question 16

- ack = 0 : at most once (false, at least)
- ack = 1 : at least once (false, at most)
- ack = all : exactly once

10 Performance of kafka producer

10.1 Question 17

We can set ack = 0.

If we use batches we can increase the batch size to increase the throughput. We can also compress messages by batches.

We can add more threads and then more producers.