Apache Kafka

Introduction :

Created by LinkedIn nav open source project maintained by confluent.

Its distributed residence architecture with fault tolerant

Horizontal scalability

Cancel to hundreds of brokers

can scale to millions of messages per second.

High performance.

Mainly used as a transportation mechanism.

Topics :

Topic is a particular stream of data

Topics are split in two partitions

Each message within a partition gets an incremental ID called offset

Upset is generated within only a partition.

Data is kept for a limited time default is one way.

Once the data is written to a partition it can be changed. Immutability.

Data is assigned randomly to a partition unless a key is provided.

Brokers :

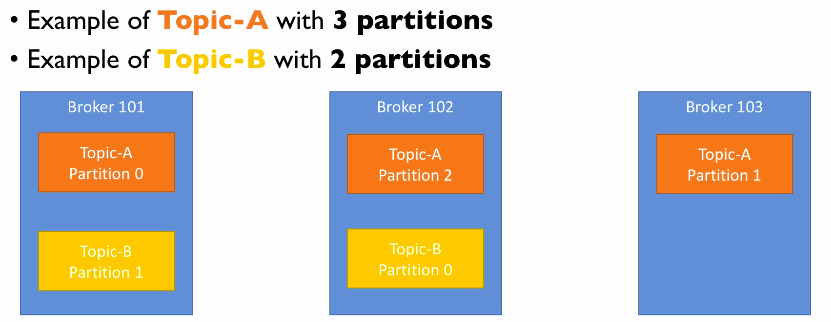
What actually does holds the topic and partition it's a broker.

Kafka cluster is composed of multiple brokers (servers).

Each broker is identified with its integer ID.

Each broker contains certain topic partitions, is it the distributed system each broker contain some data not all the data.

After connected to any bootstrap broker , ultimately you will be connected to the entire cluster.

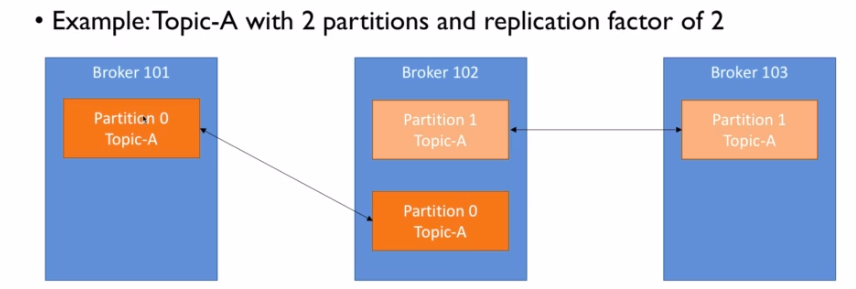


Topic replication factor :

Which broker contain some data but not all the data , so what if broker goes down right so that's why we do need to replicate the data to multiple brokers.

Topics should have a replication factor usually between 2 and 3, 2 is a bit risky but 3 is a gold standard.

So why this way if a broker goes down we still have another copy of data in another broker to serve.



There is a concept of leader for a partition.

At any given time only one broker can be a leader for a given partition.

Only that leader can receive and serve data for a partition, and other brokers will synchronise the data.

Therefore each partition has one leader and multiple ISR (in-sync replica).

So what if leader goes down, one of ISR will be a leader, and it will be determined by kafka and zookeeper.

Producers :

Producers write data to topic which is made of partitions.

Producers automatically knows to which broker and partition to write data to.

In case of that broker fails producer will automatically recover.

Is there are three partitions of a topic and a producer rights without a key then it will be kind of a round-robin load balancing are there.

If producer writes with key then data with same key will be guaranteed to be produced in the same partition.

User can choose to receive acknowledgement of a data rights.

acks=0

Producer will not wait for acknowledgement that can result in two possible data loss as there might be a possibility e of data was not returned properly so it bit risky.

acks=1

Producer will wait for leaders acknowledgement so once the leader gets the data leader will acknowledge to producer and it does not wait to data to be written in all replicas so it's better than the acks=0 , but still chance to limited data loss.

acks=all

Producer win wait for acknowledgement from leader + replicas, so no chance to data loss.

But it will be taking more time as leader broker will wait to get acks from all ISR.

min.insync.replicas :

It is recommanded to use min.insync.replicas with acks=all.

e.g. 3 brokers , replication factor = 3, acks=all along with min.insync.replicas=2

that means leader broker will wait for two writes including leader broker.

That means you can tolared 1 broker going down. Otherwise producer will receive an exception that NOT\_ENOUGH\_REPLICAS

For Kafka > 2.1

producer retries in case of no ack from broker / NOT\_ENOUGH\_REPLICAS

default producer retry setting is 2147483647

retry.backoff.ms = 100ms by default so producer will keep trying after every 100 ms untill successed.

That does not mean that producer will keep trying forever.

Its bounded by delivery.timeout.ms = 120000 ms (2 min)

in case of retryies there is a chance that messages will be sent out of order.

If you relay on key-base-ordering than it can be an serious issue.

Producer can send parallel request

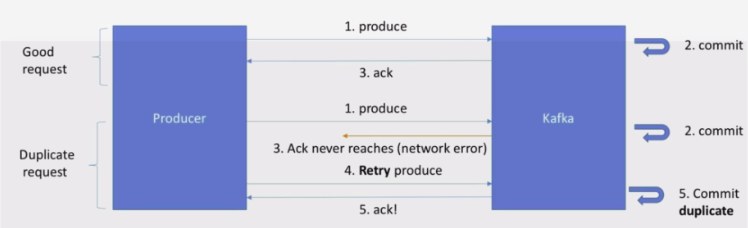
max.in.flight.requests.per.connection = 5 (default)

set it to 1 if you need to ensure ordering , that will come with cost of througput.

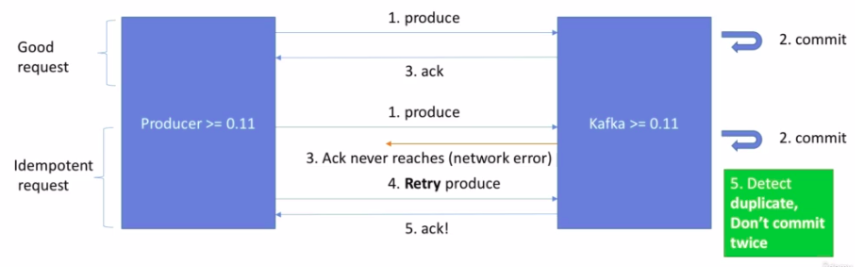
Idempotent Producer

What if kafka has already commited that record and while sending ack ther is network error.

If producer did not get ack from kafka , than producer will retry sending record.

So kafka will have duplicate record.

To overcome this problem we do have Idempotent Producer.



When producer do not get ack from kafka , it will resend record with id. Kakfa will check while commiting if id is duplicate than it will not commit but send the ack to producer.

ProducerProps.put( “enable.idempotence ”, true );

Consumers :

Consumers read data from topic.

Consumers know automatically to read data from which broker.

So in case of broker failure consumer know how to recover.

Data always be read in order within each partitions. One consumer can be reading from multiple partitions. Data will not be in order across the partitions.

Consumer Group:

Consumers read data in consumer groups.

So if there are three partitions in a topic, and to consumers in one consumer group, Partitions will be divided between two consumers. So one consumer will assigned with two partition and another consumer will be assigned with one partition.

If number of partition in a topic and number of consumers in a consumer group are same then every consumer will be reading from the one partition.

If there are only one consumer in one consumer group then three of the participants will be assigned to one consumer.

If if there are four consumers in consumer group then three consumers are assigned with one partition and one consumer will be inactive.

If there are two consumer groups listening to same topic then both the consumer will have the same data so it will not be divided between two consumer group.

Consumer Offsets :

Top ka stores the offset at consumer group level.

When consumer in a group has processed data received from kafka it should be committing the offset.

So if consumer group goes down when it comes back it knows from when to start reading.

Let's a topic having three partition and a consumer group having three consumer, so every consumer will be assigned with one partition.

Now what if one consumer goes down,

In that case partition will be reassigned to any of available consumer so now there will be one consumer with two partition and one consumer with one partition in that consumer group. Once consumer comes back the partition will be reassigned again.

Delivery semantics for consumers:

Consumer can choose when to commit the offset.

There are mainly three types of delivery semantics,

At most once-

Offsets are committed as soon as the message is received.

It's a bit risky as if the processing goes wrong the message will be lost.

At least once-

Offsets are committed after the messages processed so it's usually the preferred one.

If the processing goes wrong the message will be read again as its not been committed.

The problem with this is duplicate processing of message so make sure your processing is idempotent.

Means in case of processing again will not impact your system.

Exactly once-

Can be achieved for kafka to kafka wordpress using kafka streams API.

Kafka broker discovery :

Every broker in kafka cluster known as the boosted server.

Supply need to connect to anyone broker and it ultimately connected to entire cluster.

Each broker knows about all brokers topics and partitions (metadata).

So when client wants to connect to kafka cluster it first sends connection + metadata request to any broker , broker will respond with list of all brokers along with topics and partition details.

So now kafka client knows very well to connect with which broker.

Zookeeper :

Zookeeper manages the broker.

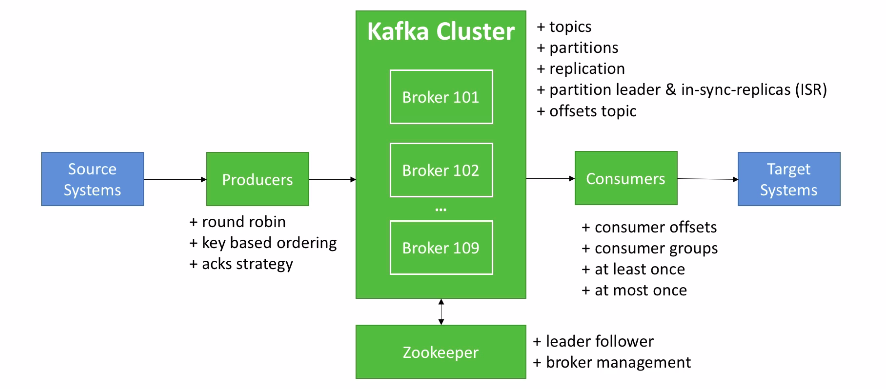
Zookeeper helps to do leader election for partition.

Zookeeper send notification to kafka in case of any change in cluster for example if broker goes down comes up.

Zookeeper by design operates with odd number of server (3, 5, 7).

Zookeeper has a leader that handles right and the rest of the follower servers handles read.

Summery :



Producer Message Compression

Producer can send records in batch. And before sending it to kafka producer can compress it

Compression will happen at producer level and it does not require any configuration changes at kafka or consumer level.

compression.type = none(default), gzip, lz4, snappy

Advantage :

Much smaller producer request size.

Less letencey – faster totransfer over network.

Better throughput

Best disc utilisation in kafka.

Disadvantage (minor):

Producer and Consumer must commit some CPU cycles to compression.

Linger.ms :

Number of milli seconds producer will wait before sedning a batch out.

Batch.size:

If the batch is full before the end of linger.ms period.it will be sent to kafka right away.

Buffer.memory

If the producer produces faster than the broker can take then the records will be buggered in producer memory.

buffer.memory = 33554432 (32 MB) by default.

Max.block.ms

if that buffer is full than the .send() method will start to block.

Max.block.ms=60000

producer has filled up its buffer.

Broker is not accepting new data.

60 seconds has lapsed.

CLI

1. Zookeeper start :

$ zookeeper-server-start.sh config/zookeeper.properties

Default port would be 2181

1. Kafka broker start :

$ kafka-server-start.sh config/server.properties

Default port would be 9092

1. Kafka Topic Create

$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic test\_topic --create --partitions 3 --replication-factor 1

1. Kafka Topic list

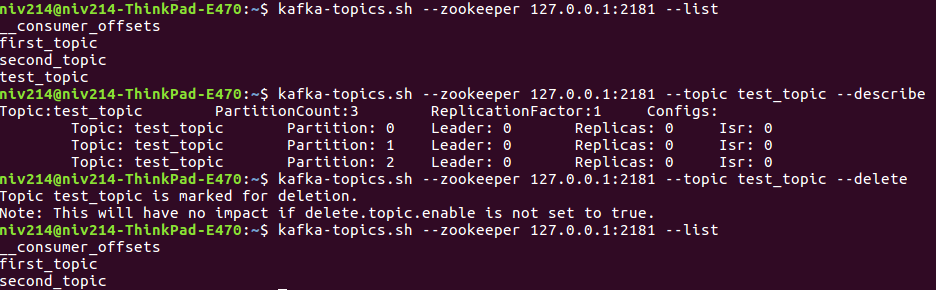
$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --list

1. Kafka topic describe

$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic test\_topic --describe

1. Kafka topic delete

$ kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic test\_topic --delete



1. Kafka Producer

$ kafka-console-producer.sh --broker-list 127.0.0.1:9092 --topic first\_topic --producer-property acks=all

If a producer produces messages with the same key then it’s guaranteed to be produced on the same partition.

kafka-console-producer --broker-list 127.0.0.1:9092 --topic first\_topic --property parse.key=true --property key.separator=,

> key,value

1. Kafka Consumer

$ kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic first\_topic

Above consumer will read new messages only as per default setup.

In case of read all the messages from beginning need to add below property

$ kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic first\_topic --from-beginning

1. Kafka Consumer Group

$ kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --topic first\_topic

--group first\_group

Consumer will be consuming under group now.

So if we start multiple consumers under same group than messages will be devided between all consumers.

If consumers are in separate group than all consumer group will be consuming all messages.

1. Consumer Group List

$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --list

1. Consumer Group describe

$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --group first\_group\_test --describe

Consumer group 'first\_group\_test' has no active members.

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST CLIENT-ID

first\_topic 0 61 61 0 - - -

first\_topic 2 76 76 0 - - -

first\_topic 1 51 51 0 - - -

1. Consumer group offset reset

$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --group first\_group\_test --reset-offsets --to-earliest --execute --topic first\_topic

It will reset all offsets of all partitions of given group to 0.

$ kafka-consumer-groups.sh --bootstrap-server 127.0.0.1:9092 --group first\_group\_test --reset-offsets --shift-by -2 --execute --topic first\_topic

It will reset all offsets of all partitions to left 2 for given group.

13) Kafka Config Describe

$ kafka-config.sh –zookeeper 127.0.0.1:2181 –entity-type topics –entity-name first\_topic --describe

14) Kafka Config add configuration

$ kafka-config.sh –zookeeper 127.0.0.1:2181 –entity-type topics –entity-name first\_topic –add-config min.insync-replicas=2 --alter

15) Kafka Config delete configuration

$ kafka-config.sh –zookeeper 127.0.0.1:2181 –entity-type topics –entity-name first\_topic –delete-config min.insync-replicas=2 --alter

Partitions and Segments :

Topics are made of partitions.

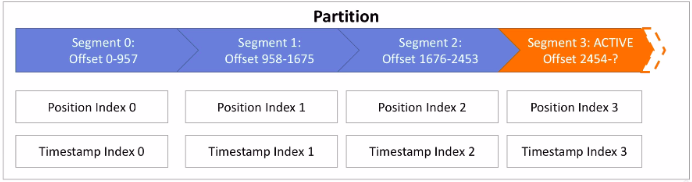
Partitions are made of segments. (Segments is nothing but a File , yeah its true :) )



Only one segment is ACTIVE, the one that is data being written to.

Log.segment.bytes - the max size of single segment in bytes. (1GB by default)

Log.segment.ms - Time that kafka will wait before committing the segment if not full. (1 week by default)

Segments comes with Two indexes.

Offset to Position index : Allows kafka where to read to find a message.

Timestamp to Offset index : Allows kafka to find message with a timestamp.

Log Cleanup Policies :

log.cleanup.policy=delete (default for all user topic)

Delete based on age of data (default 1 week)

Delete based on max size of log (-1 default means infinite)

log.cleanup.policy=compact (default for topic \_\_consumer\_offset)

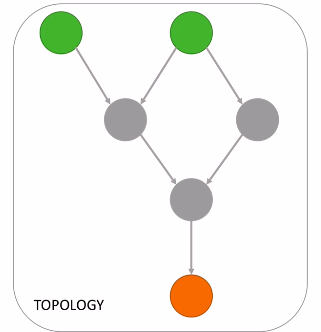
Delete based on keys.

Delete old duplicate keys after active segment committed .

Log clean up happens on partition segments.

Smaller segments means log cleanup will happen more often . And it will consume cpu usage.

**Kafka Stream and Ktable**

****

**Stream** is a sequence of immutable data records, that is fully ordered and can be replayed, and also a fault tolerent.

**Stream Processor** is a node in processor topology. It transforms incoming streams record by record and may create new stream out of it.

**Source Processor** is a special processor that takes data directly from the kafka topic.

It has not predecessors and it does not transform the data.

**Sink Processor** is a processor that does not having any children. It sends stream data directly to the kafka topic.

In between of this two processor (source and sink) all other streams and data does not need to be persisted to kafka and it stays on the streams only.

**High Level DSL**

It is simple.

It has all the operations that we do need to perform almost all the transactions.

**Low Level Processor API**

Can be used to implement most complex logic. So rarely needed.

Most of the properties will remain unchanged when it comes to producer and consumer properties. Only few changes as below.

**application.id**

Specific to stream api.

Will be used as group id. (applicatoin id == group id)

Dont change it as its being used as prefix of internal change log topics.

**default.key.serede** and **default.value.serede**

For serialization and de-serialization of data.