Kafka

Topic

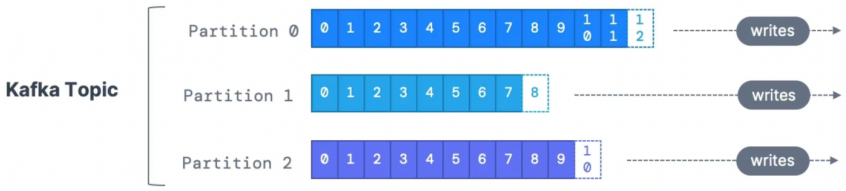
- a particular stream of data, log purchases twitter_tweet
- like a table in a database without all constraints
- as many topics as you want
- identified by its name

Partitions and offsets

- each partitions has order
- each partitions has its offset
- topics are immutable

- once data is written to a partition, it cannot be changed(immutability)
- data only kept for a limited time
- offset only have a meaning for a specific partition (e.g. offset 1 in partition 1 is not represent offset 1 in partition 0)
- order only within a partition not cross partitions
- data is assignment randomly to a partition unless a key is provided

Topics, partitions and offsets – important notes



- Once the data is written to a partition, it cannot be changed (immutability)
- Data is kept only for a limited time (default is one week - configurable)
- Offset only have a meaning for a specific partition.
  - E.g. offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1
  - Offsets are not re-used even if previous messages have been deleted
- Order is guaranteed only within a partition (not across partitions)
- Data is assigned randomly to a partition unless a key is provided (more on this later)
- You can have as many partitions per topic as you want

Producer

- producer write data to topics
- each topic contains partitions
- producer know to which partition to write to(which kafka broker it has)
- if broker failed, producer will recover automatically

Message key

- producer send a key with the message
- if key is null, data is sent round robin(1 -> 2 -> 3)
- if not null, all message for that key will always end up to the same partition
- key are typically sent if you need message ordering for a specific field

Key - binary, Value-binary
compression type

message serialization

accepts bytes as an input from producers and sen bytes out as an output to consumer
means transforming objects/data into bytes
they are used on the value and the key

Key hashing

it is the process of determining the mapping of a key to a partition

For the curious: Kafka Message Key Hashing

- A Kafka partitioner is a code logic that takes a record and determines to which partition to send it into.

- **Key Hashing** is the process of determining the mapping of a key to a partition
- In the default Kafka partitioner, the keys are hashed using the **murmur2 algorithm,** with the formula below for the curious:

```
targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)
```


Consumers

- Consumer read data form a topic from broker - pull model
- Consumer automatically know which broker to read from
- if broker fail, it knows how to recover - by using offset
- data is read in order from low to high offset within each partitions

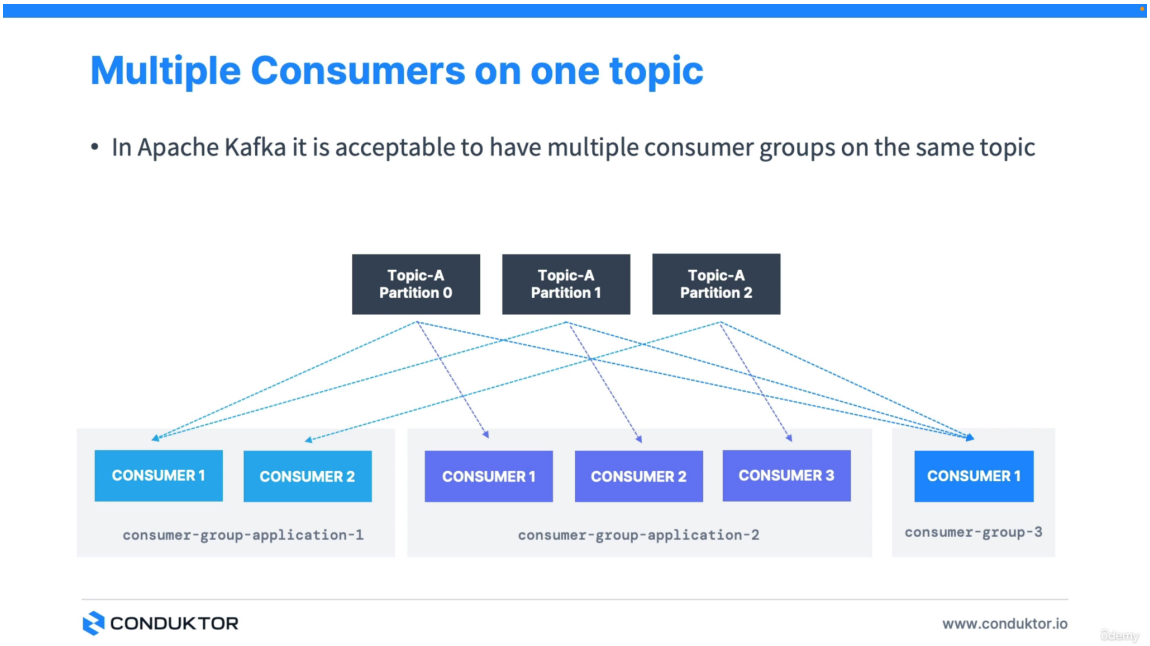
Deserializer

- how to transform bytes into objects/data
- used on the value and the key of the message
- common deserializer
    - string, int Float, Avro, Protobuf

Consumer Groups

- all the consumer in a app read data as a group

- if the number of partitions is larger than the number of consumers, then it is possible on consumer consume partitions
- more consumers than partitions in same group, which one to one, cannot be like 2 consumer consuming 1 partitions
- acceptable to have multiple consumer groups on the same topic
- if multiple group consuming partitions, it could be diff consumer from diff group consuming same partition



Consumer offsets

- stores the offset at which a consumer group has been reading

Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
- The offsets committed are in Kafka topic named `__consumer_offsets`
- When a consumer in a group has processed data received from Kafka, it should be **periodically** committing the offsets (the Kafka broker will write to `__consumer_offsets`, not the group itself)
- If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets!

Delivery semantics for consumers

- by default, consumers will auto commit offsets
- 2 semantics if you choose to commit manually
  - At least one (usually preferred)
  - at most one
  - Exactly once

Delivery semantics for consumers

- By default, Java Consumers will automatically commit offsets (at least once)
- There are 3 delivery semantics if you choose to commit manually
- At least once (usually preferred)
  - Offsets are committed after the message is processed
  - If the processing goes wrong, the message will be read again
  - This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won't impact your systems)
- At most once
  - Offsets are committed as soon as messages are received
  - If the processing goes wrong, some messages will be lost (they won't be read again)
- Exactly once
  - For Kafka => Kafka workflows: use the Transactional API (easy with Kafka Streams API)
  - For Kafka => External System workflows: use an idempotent consumer

Kafka Brokers

- a cluster is composed of multipole brokers(servers)
- each broker has its ID
- contains topic
- once connecting to one broker, it will connected to entire cluster
- good number started by 3

Kafka broker discovery

- broker also called a bootstrap server
- you only need to connect to one broker
- each broker knows about all brokers, topic, partitions(metadata)

Replication factor

- topic should have a replication factor >1
- broker is down, another broker can serve the data
- topic A with 2 partitions and replication factor of 2
-

Only One broker can be a leader
producer can only send data to the broker that is leader of partition
each partition has a leader and ISR (its sync replica)

Consumers Replica Fetching (v2.4+)

- allow consumer to read from the closest replica
- help improve latency, decrease network costs if using cloud

Producer Acknowledgements

- acks=0 producers can choose to receive acknowledgment of data writes
- acks=1 producer will wait for leader acknowledgment
- acks=all leader + replicas acknowledgment

Topic Durability

- the replication factor of N, lose up to N-1 brokers and till recover your data

Zookeeper

- manages brokers list of them
- helps performing leader
- send notification to kafka in case of changes, a lot of metadata
- 2.x cannot without zookeeper
- 3.x can without zookeeper
- 4.x will not have zookeeper
- it is designed operates with an odd number of servers(1, 3, 5, 7) never more than 7
- has leader the rest of the servers are followers
- not store consumer offset, store internal kafka topic since 0.9

Zookeeper cluster(ensemble)
should you use zookeeper

- with broker - Yes before 4.x version
- with clients - over time the kafka clients and CLI have been

Should you use Zookeeper?

- With Kafka Brokers?
  - Yes, until Kafka 4.0 is out while waiting for Kafka without Zookeeper to be production-ready
- With Kafka Clients?
  - Over time, the Kafka clients and CLI have been migrated to leverage the brokers as a connection endpoint instead of Zookeeper
  - Since Kafka 0.10, consumers store offset in Kafka and Zookeeper and must not connect to Zookeeper as it is deprecated
  - Since Kafka 2.2, the `kafka-topics.sh` CLI command references Kafka brokers and not Zookeeper for topic management (creation, deletion, etc...) and the Zookeeper CLI argument is deprecated.
  - All the APIs and commands that were previously leveraging Zookeeper are migrated to use Kafka instead, so that when clusters are migrated to be without Zookeeper, the change is invisible to clients.
  - Zookeeper is also less secure than Kafka, and therefore Zookeeper ports should only be opened to allow traffic from Kafka brokers, and not Kafka clients
  - **Therefore, to be a great modern-day Kafka developer, never ever use Zookeeper as a configuration in your Kafka clients, and other programs that connect to Kafka.**

Kafka KRaft

- 3.x KRaft to replace zookeeper
- KRaft by using Quotum leader