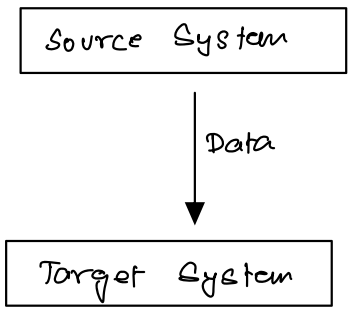
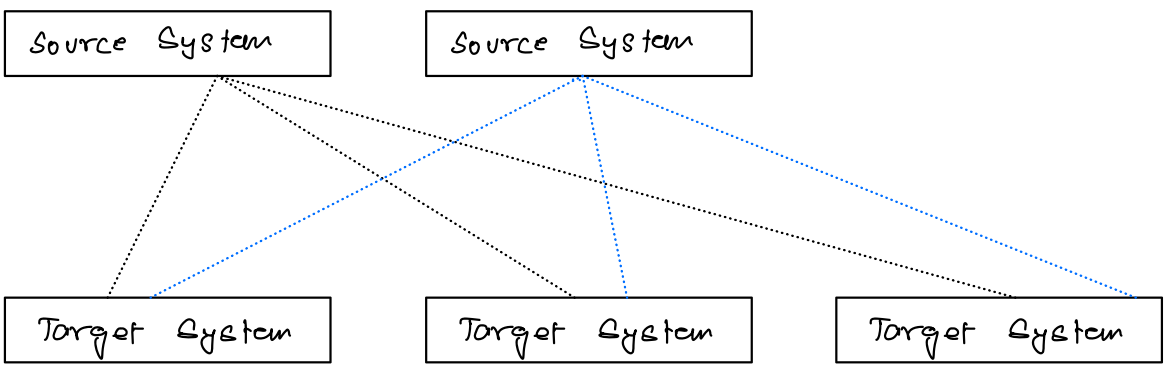
now maintained by Kafka → Created by Linkedin
Confluent, IBM ←
Cloudera.

At the start of the company we have just one source system and one target system



As the company grows in size the number of source system and the number of target systems increases.

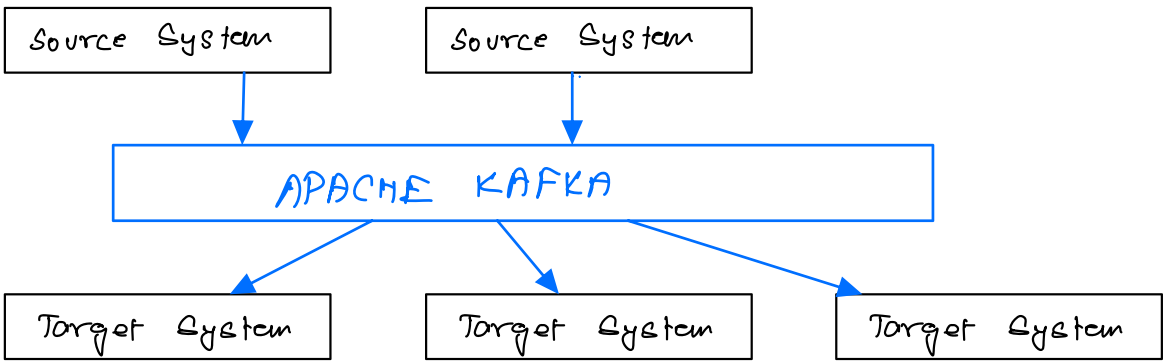


As the number of source systems and the target systems increases it becomes difficult as the source should now communicate with all the target systems. Also while sending

data to the target systems the source systems should take care of the target systems

1. Protocol (Eg: TCP, HTTP, REST)
2. Data Format (Eg: Binary, CSV, JSON)
3. Data Schema & Evolution

Now each source system will have increased load. This is where Apache Kafka comes.



Now the source systems and the target systems are decoupled

Kafka Use Cases

Sources: Website Events, Pricing Data, Financial Transactions, User Interactions

Targets : Database , Analytics - Email Systems - Audit

1) Messaging Systems
2) Activity Tracking
3) Gather metrics from different locations
4) Application logs Gathering
5) De-coupling of system dependencies
6) Micro services Pub / Sub.
7) Stream Processing → Ingesting a cont. stream of data to analyze, filter, transform data in real time

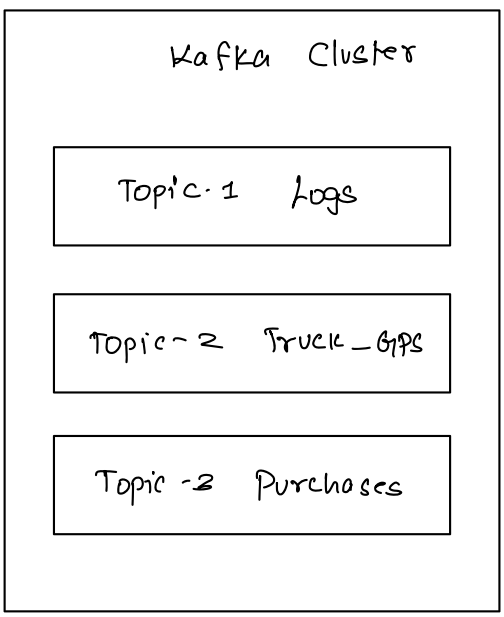
Advantages of Kafka :

* Kafka can be horizontally scaled to 100's of users
* can be scaled to millions of messages per second
* can be scaled to latency less than 10 MS

Thus they can be used in real time systems

Kafka Topics

Kafka Topics are a particular stream of data. We can have as many topics as we want in a cluster. A Kafka topic is similar to a table in a database



A topic is identified by its name. It can store any kind of data (JSON, CSV, Binary)

Sequence of messages is called data stream

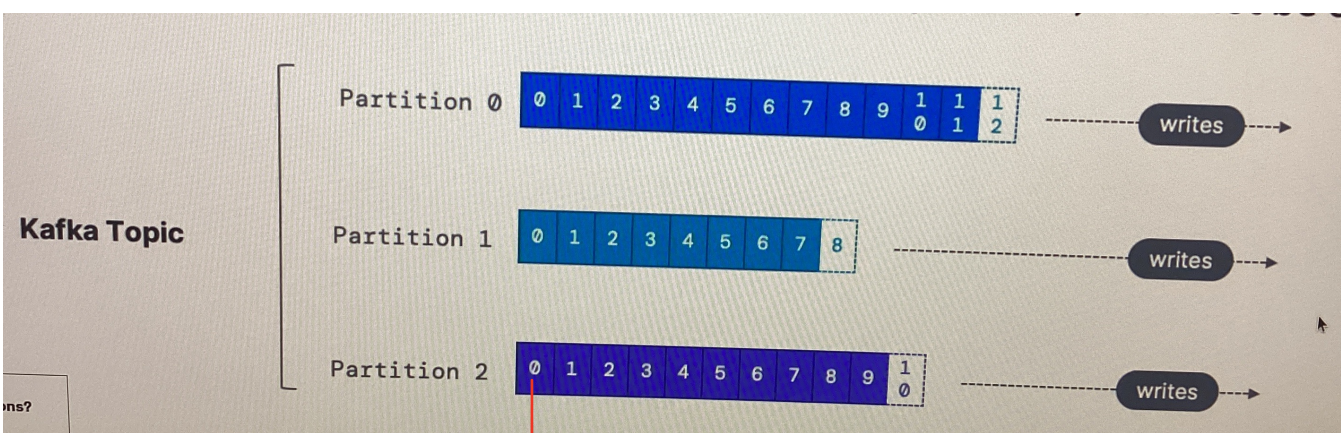
Topics cannot be queried and they are also immutable i.e data added to Kafka cannot be updated / deleted

Data in a Topic is stored for a limited time (default : One week- configurable). After that period the data will be deleted

Kafka Producers send messages to the topics and the Kafka Consumers receive them.

Partitions and Offsets :

Topics are divided into Partitions



As messages are added to the partitions they are assigned with an ID (starting from 0) known as the Kafka Partition Offset

Even if the messages are deleted the offsets are not reused ie if I have 100 messages then I delete them instead of the offsets starting

from 0 they start from 101

The messages sent to a topic are stored orderly within these partitions. Order is guaranteed only within a partition (not across partitions)

Producers & Message Keys

- The producers decide in which partition the messages are to be stored

Producers:

Producers know to which partition the messages have to be written and which Kafka Broker (Kafka Server) has it

In case of Kafka broker failures, Producers will automatically recover.

Message Keys:

The keys are hashed to the corresponding partitions using the Murmur 2 algorithm

Producers can send a key with the message (optional)

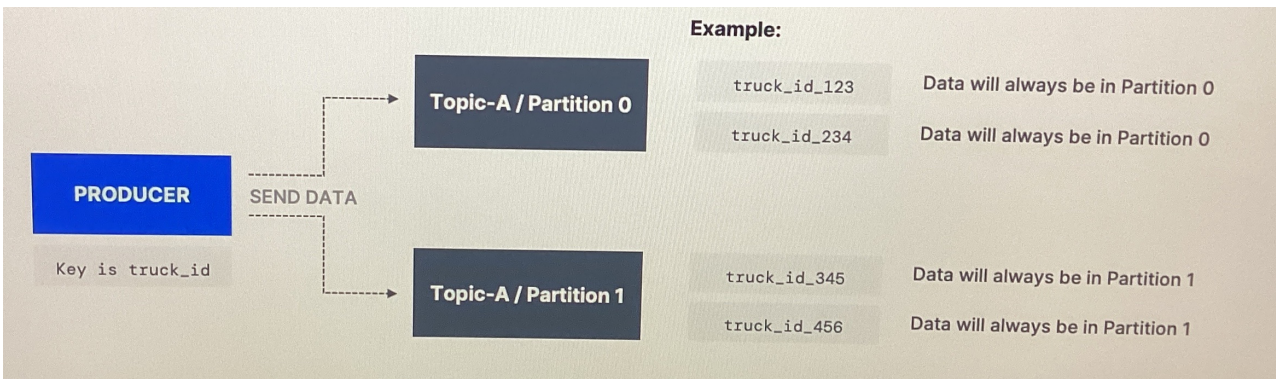
The key can be String, number, binary

When to send a key along with the message?

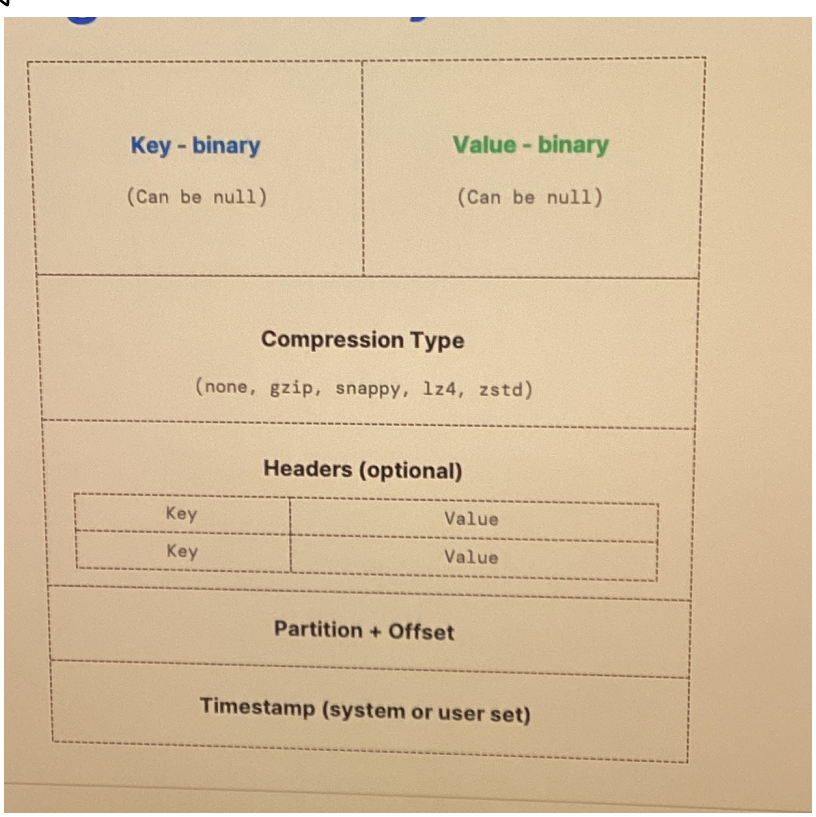
When a key is not sent by the producer then the messages sent will be distributed among the partitions in round robin.

But if a key is sent by the producers then it is guaranteed that the messages will end up in the same partition

Eg:



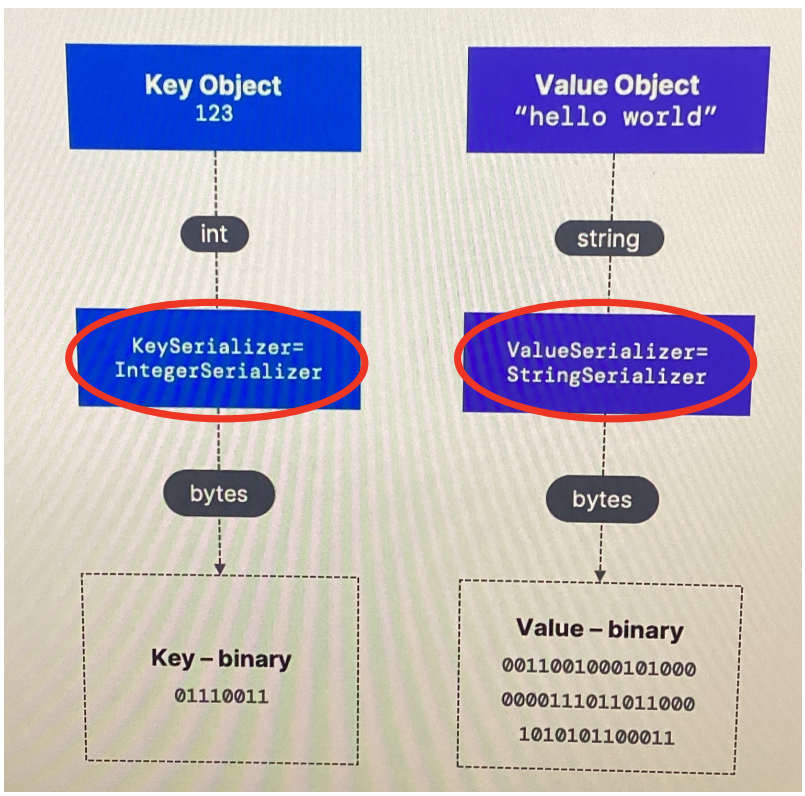
Message format:



Kafka only accepts bytes as an input from producers and sends bytes as output to consumers

Message Serialization

The process of transformation of objects / data into bytes



Common Serializers:

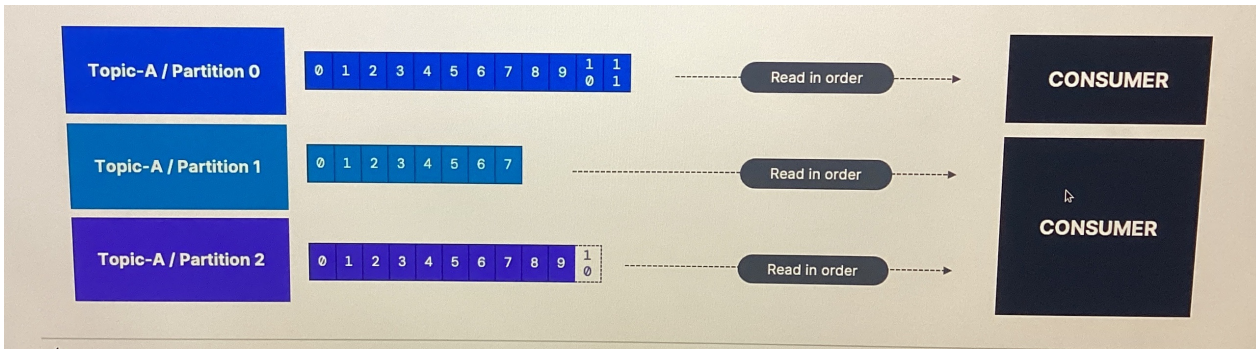
1. Strings (inc. JSON)
2. Int, Float
3. Avro
4. Protobuf

Consumers & Deserialization

Consumers should pull (request) data from the kafka topic. Consumers know to which broker and which partition the request should be sent.

In case of broker failures the consumers know how to recover.

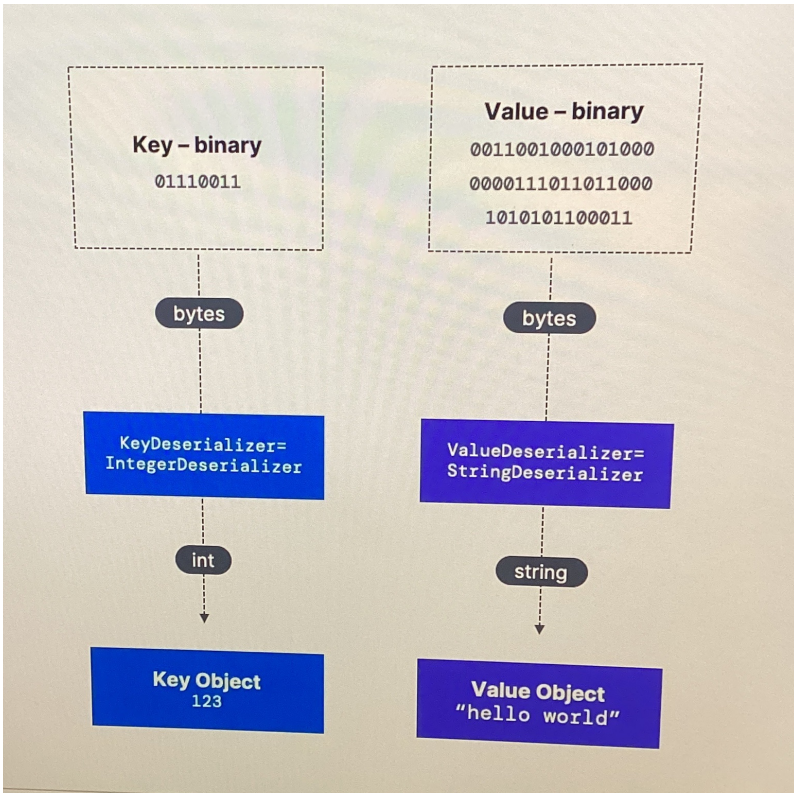
Data is read from low to high offset within each partitions



Consumer Deserialization

This a process of converting bytes into objects / data. The consumers should know in advance what is the format of the message. Eg key → int

value → string



Common Deserializers

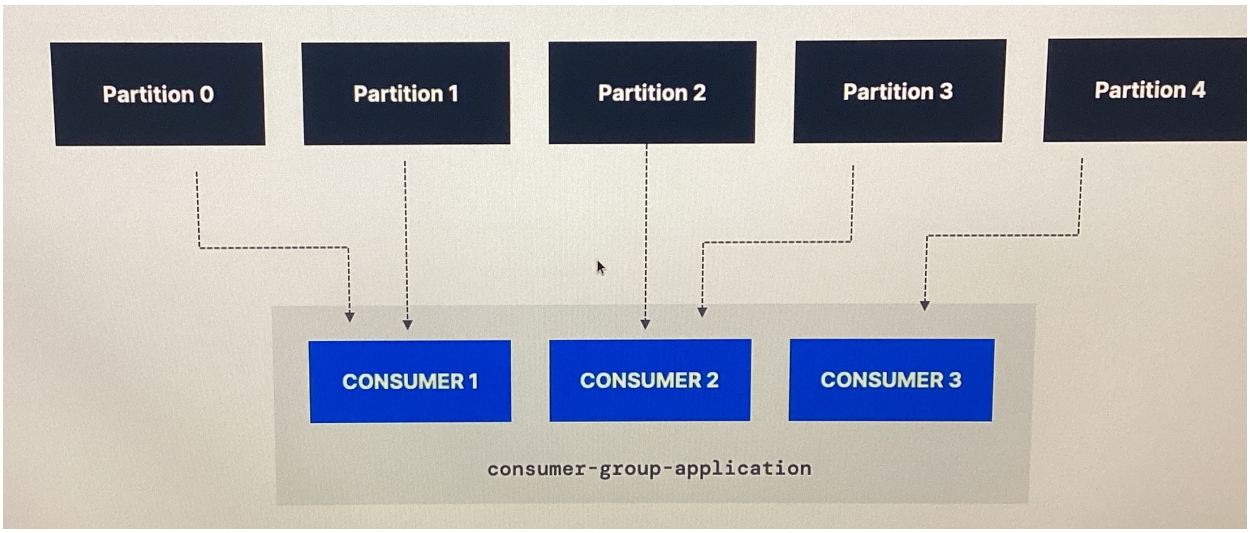
1. Strings (inc. JSON)
2. Int, Float
3. Avro
4. Protobuf

NOTE

The serialization / deserialization type should not change during a topic life cycle (create a new topic instead)

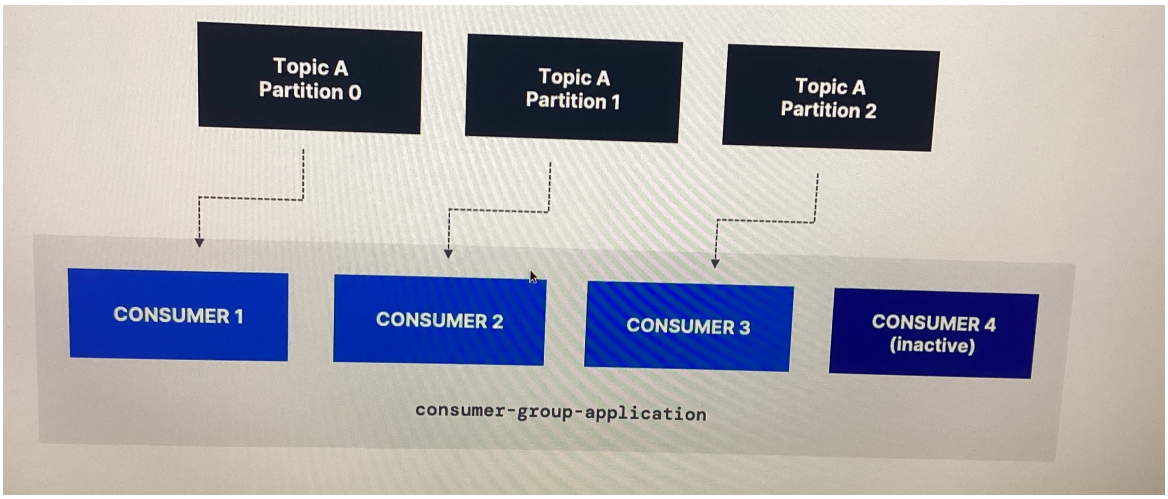
Consumer Group & Consumer Offsets

All the consumers in an application read data as consumer groups

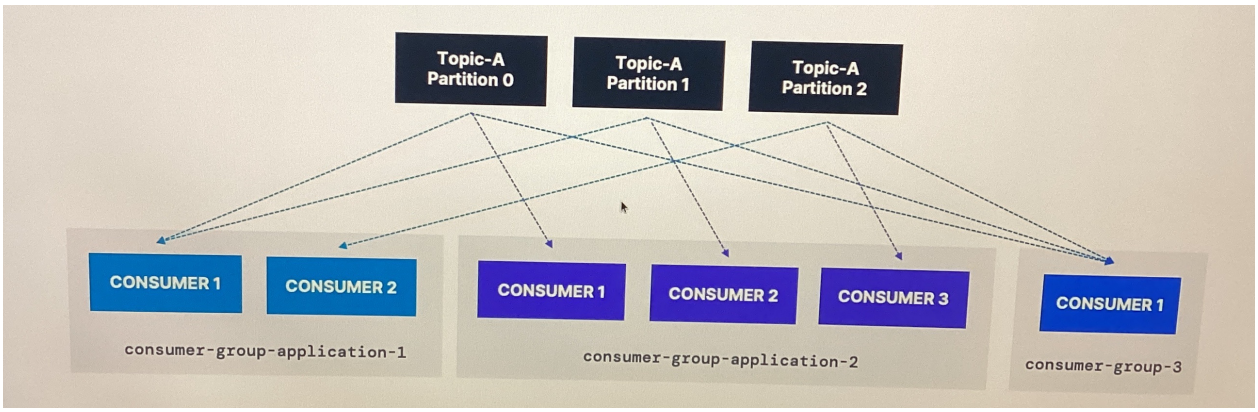


Each consumer within a group reads data from distinct partitions

Inactive Consumers



Multiple Consumer Groups



To create distinct consumer groups the consumer group property **group.id** is used.

Consumer Offsets

Kafka **stores** the **offsets** at which the consumer group has been reading.

Eg: If a consumer has read a message from 23rd offset in Partition 1 Topic-1 then 23 is stored.

These offsets are stored in a kafka topic called **__consumer_offsets**.

The offsets are committed to the __consumer_offsets only **once in a while**. This ensure that if a consumer **dies and comes back**, the consumer will be able to read back from **where it left**.

By default java consumers use automatically commit offsets (at least once)

If we choose to manually commit then there are 3 semmantics

- **At Least Once**

Offsets are committed after the message is processed. If the processing goes wrong then the messages will be read again. This results in message duplication. This should be used when processing of message again wont impact the system)

- **At most Once:**

Offsets are processed as soon as the messages are received. If the processing goes wrong then some messages will be lost.

- **Exactly Once**

Kafka Workflows ⇒ Use Transactional API (in the Kafka Streams API)

External System ⇒ use an idempotent workflow consumer.

Brokers & Topics

Kafka Cluster:

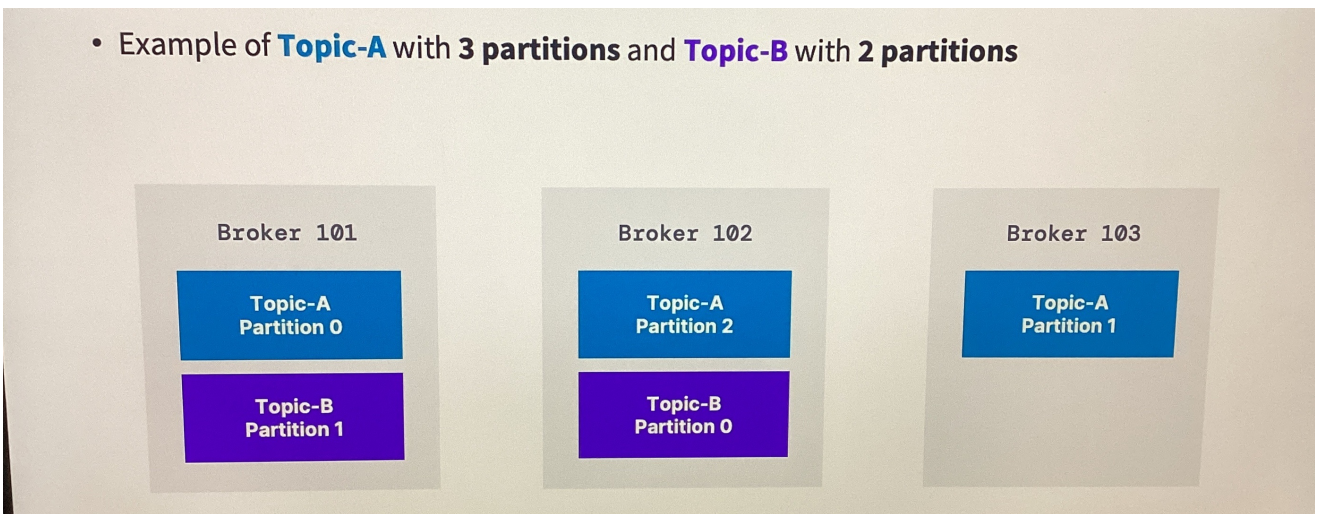
A kafka cluster is composed of multiple brokers (servers)

Brokers → A good number of brokers to start with is 3

A kafka broker is just a server but since they receive and send data they are known as brokers.

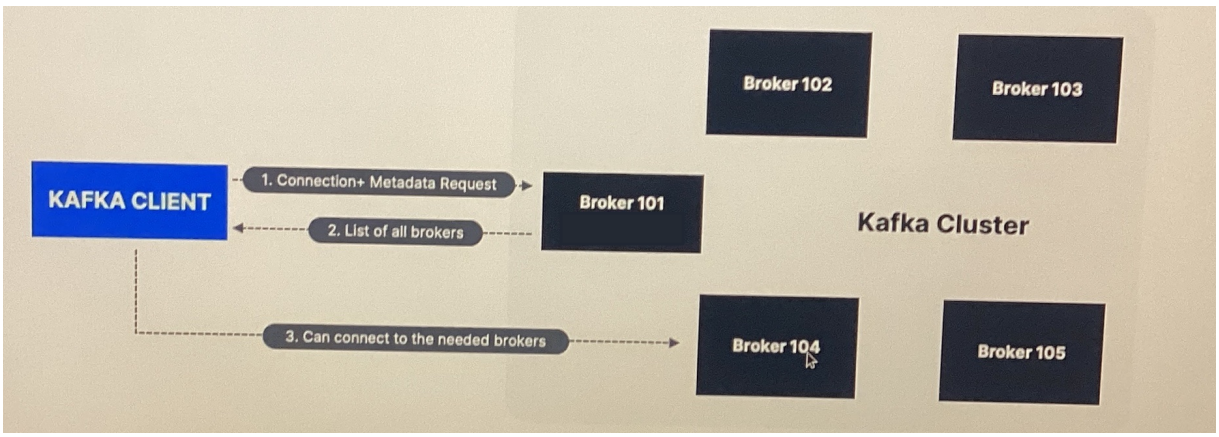
A broker is identified with an ID which is an integer

Each broker contains certain topic partitions ie the data is distributed among the brokers.



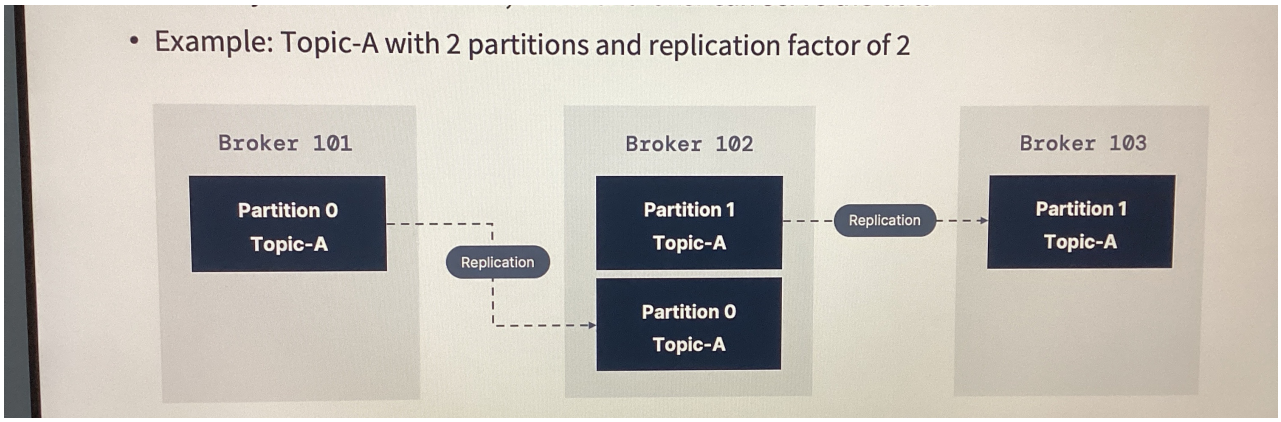
Kafka Broker Discovery

Every Kafka Broker is called a Bootstrap Server because if a consumer connects to just one broker then the consumer knows how to get connected to all the brokers in the cluster.



Topic Replication

Topics should have a replication factor > 1 (For production it should be 3). Thus if a broker is down another broker can serve the data.

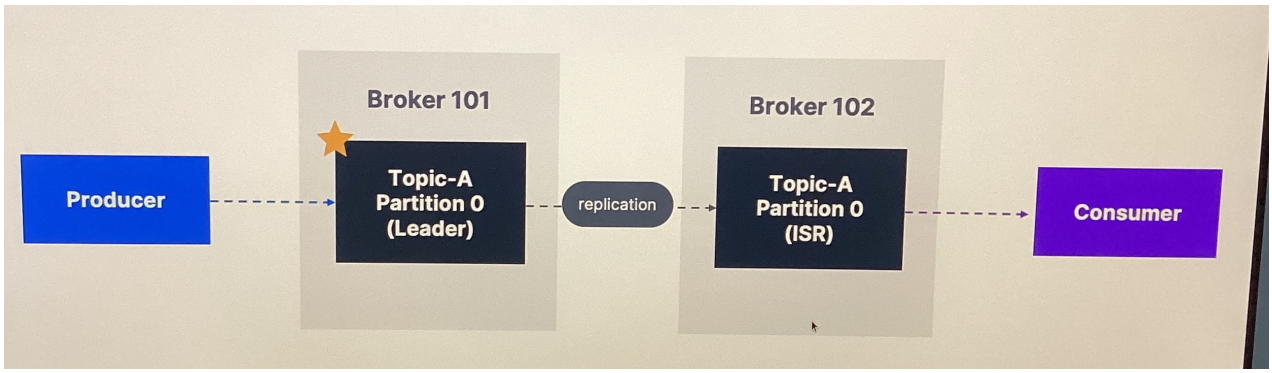


Leader for a Partition

At any time only one broker can be a leader for a given partition and the producers send data only to the broker that is the leader of the partition and the other brokers replicates the data from the leader.

Similarly the consumers read messages only from the leader but in the latest version (Kafka v2.4+) of Kafka there is a feature known as the

Kafka Consumer Replica Fetching which allows the consumers to read data from replicas



This might help to reduce latency (the consumer is close to the replica broker)

In Sync Replica (ISR)

The replica brokers send fetch requests to the leader to get the latest messages. If the leader doesn't receive a fetch request within the configured replica.lag.time.max.ms or the follower has not consumed the message within this period then the replica broker is removed from the ISR list

Producer Acknowledgement

The producers can choose to receive acknowledgement that the data has been stored successfully by the broker. There are 3 settings

acks = 0 => Producer won't wait for data acknowledgement (possible data loss)

acks = 1 => Producer will wait for the leader acknowledgement (limited data loss)

acks = all => Leader + Replicas acknowledgement (no data loss)

Zookeeper

Versions

4.x => Doesn't need Zookeeper

3.x => Can work without Zookeeper (KIP500) using Kafka Raft instead (KRAFT)

2.x => Cannot work without Zookeeper.

Zookeeper is slowly disappearing and is going to be replaced because Zookeeper is less secure than kafka and we need to configure zookeeper in such a way that it accepts connection only from Kafka. With Kafka > V 0.10 doesn't store any consumer offsets or any other consumer data.

Zookeeper is a software that has a list of all the kafka brokers and whenever a broker goes down Zookeeper performs the leader election

Zookeeper also sends notifications to the brokers when

① A topic is created / deleted

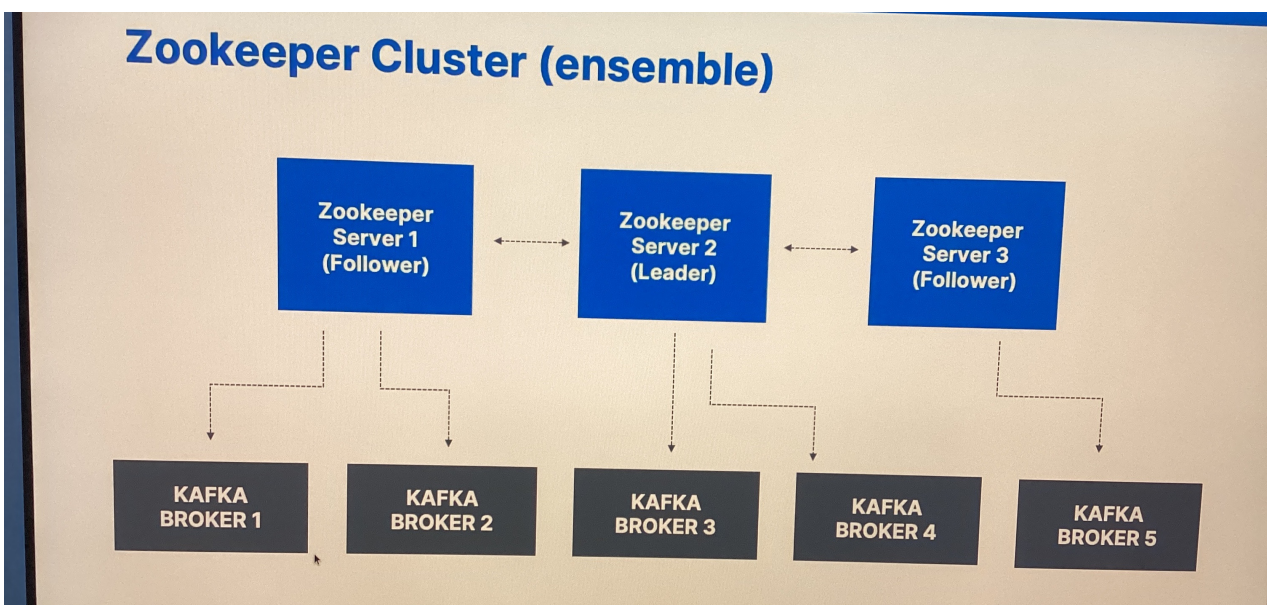
② A broker goes down / comes up.

Zookeeper

Zookeeper Cluster

Zookeeper by design works with an odd number of (Zookeeper) servers ie. 1, 3, 5, 7 (max)

Zookeeper has a leader which writes to the rest of the follower server



Should Zookeeper be used??

with Kafka brokers?

Yes, until Kafka 4.0 is out

with Kafka clients ?

Do we need Kafka clients to connect to Kafka brokers directly instead of using the Zookeeper's endpoint. So for Kafka clients NO

KAFKA RAFT (or) KRAFT

Zookeeper had issues when Kafka clusters had > 1,00,000 partitions.

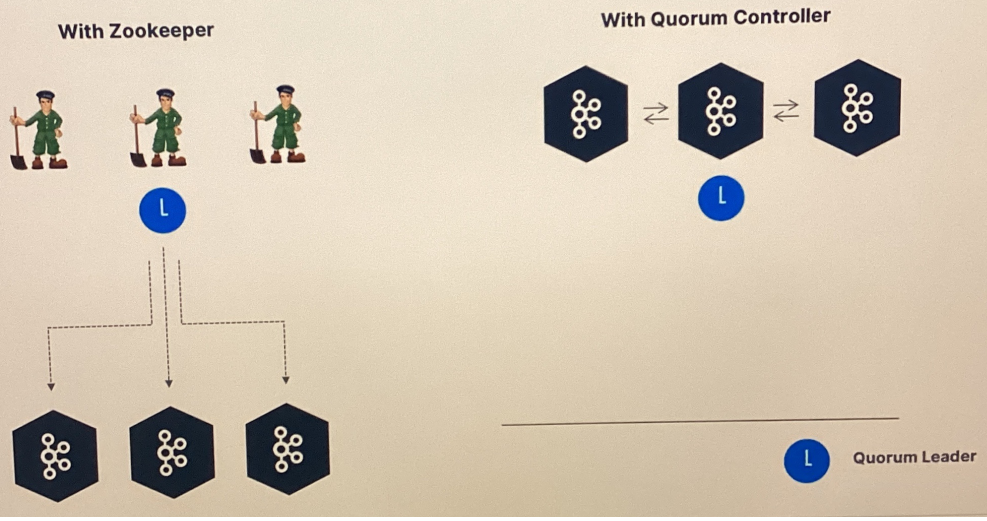
By removing Zookeeper, Kafka can

1. Scale to millions of partitions
2. Single security model for the whole system (Before we had to take care of zookeeper)
3. Single process to start and faster controller shutdown and recovery time

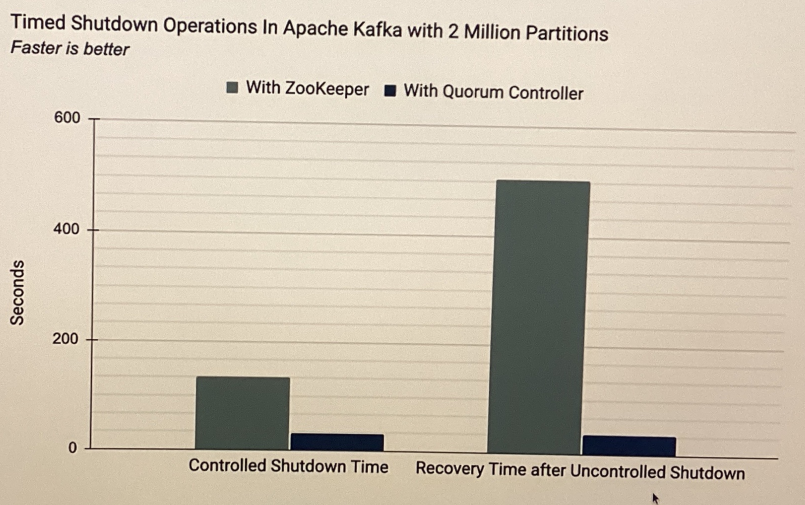
So Kafka 3.x implemented the KRAFT protocol to replace Zookeeper (production ready since 3.3.1) KIP 833

In Zookeeper we have the Zookeeper cluster in which the leader will control all the brokers but in KRAFT we have Quorum controller where we have just the Kafka brokers among which one will be the Quorum leader

Kafka KRaft Architecture



KRaft Performance Improvements



https://www.confluent.io/blog/kafka-without-zookeeper-a-sneak-peek/