Apache Kafka

Source: udemy.com/apache-kafka/learn/v4/overview.

① Apache Kafka helps in decouple data streams.

Why Kafka

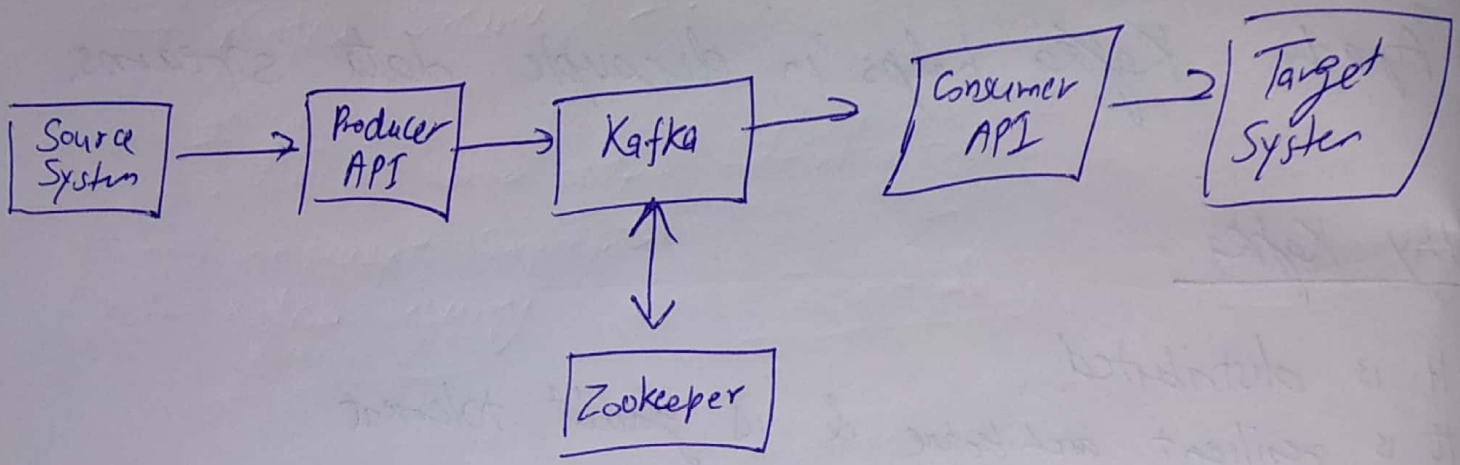
→ It is distributed
→ It is resilient architecture & is fault tolerant
→ offers Horizontal scalibility
→ High Performance (latency less than 10ms).

Uses of Kafka

→ Messaging system, like in place of JMS.
→ For activity tracking, [redacted]
→ Gather Metrics from different Locations.
→ Application Logs gathering.
→ Stream Procesing (Kafka Stream API)
→ Decoupling of system dependencies
→ Integration with spark, Flink, Storm, Hadoop & many other Big data technologies.

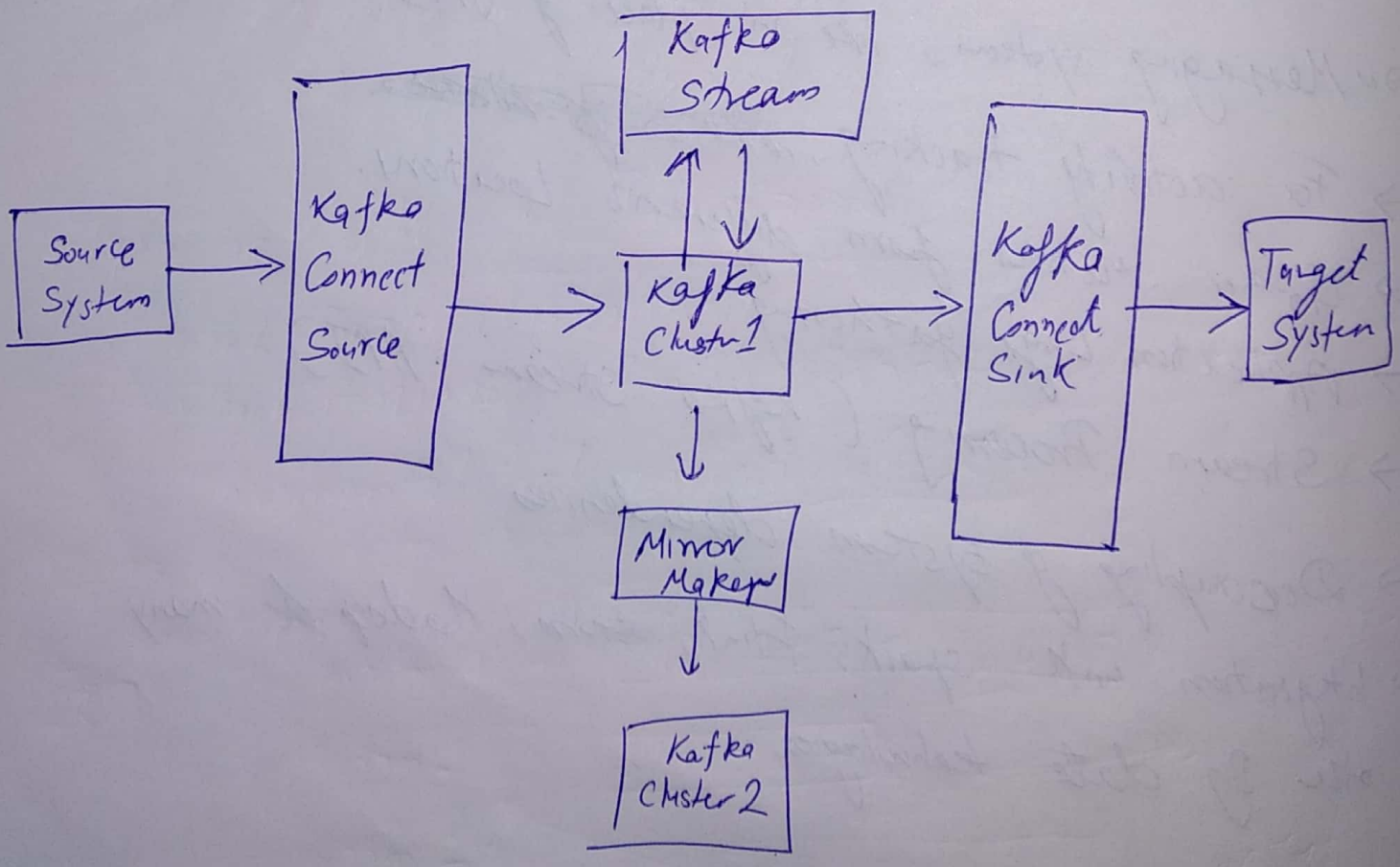
Kafka Ecosystem

Kafka Core (Imp)

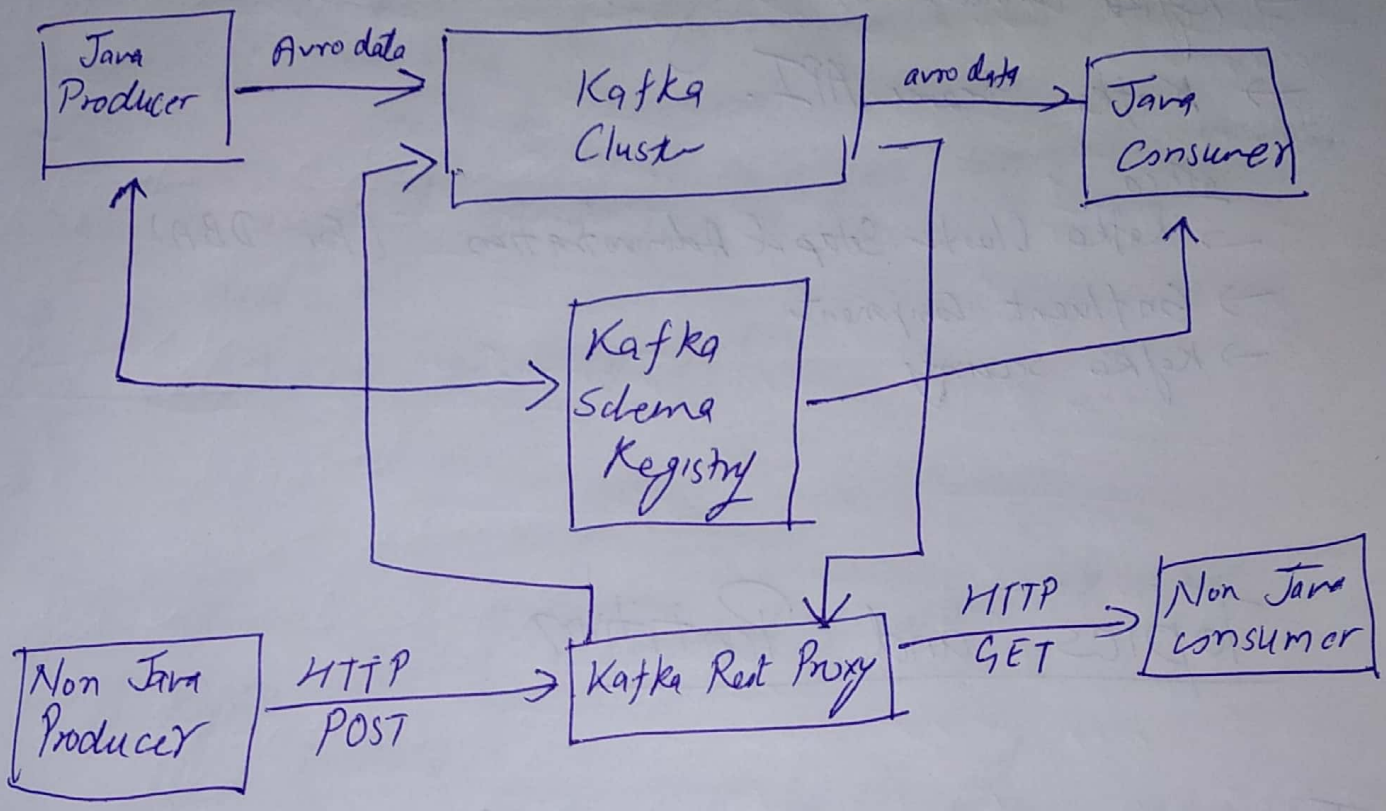


Kafka Extended Ecosystem (not required now)

→ with Kafka Connect, Kafka Streams, Mirror Maker



Kafka Confluent Schema Registry & Rest Proxy (not required now)



Kafka Administration & Monitoring Tools

→ Topics UI
→ Schema UI
→ Connect UI
→ Kafka Manager
→ ~~Kafka~~ Burrow
→ Exhibitor
→ Kafka Monitor
→ Kafka Tools
→ Kafkat
→ JMX Dump
→ Control Centre / Auto Data Balance / Replicator.

Kafka steps to study

① → Kafka for Beginner
→ Kafka Connect API
→ Kafka Stream API.

} Imp

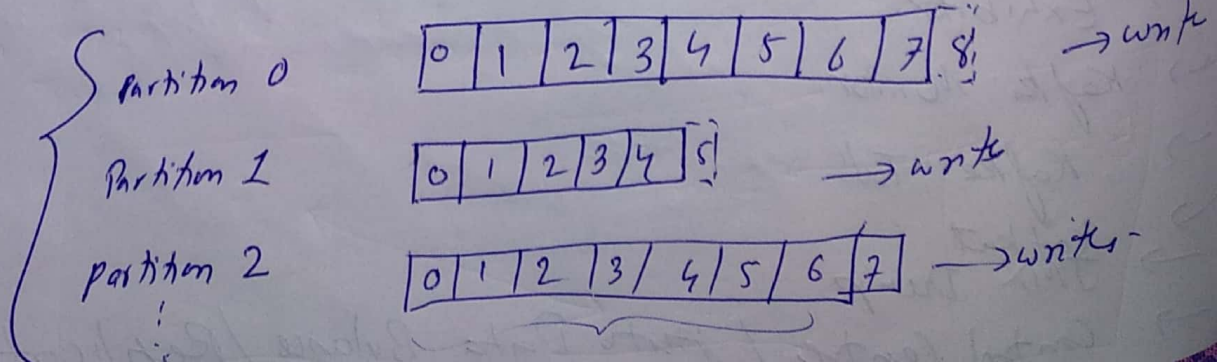
others
② → Kafka Cluster Setup & Administration (For DBA)
→ Confluent Component
→ Kafka Security

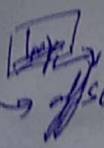
Topics and Partition

Topic → It is a stream of data similar to tables in a database
→ A topic is identified by its name

Topics are split in **Partitions**.
→ Each Partition is ordered
→ Each message within a partition gets an incremental id called **offset**.

Diagram



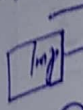


→ offset have a meaning only for a specific partition.

→ Order is guaranteed only within a partition

→ Data is kept only for a limited time. (default is one week)

→ Immutability → once data is written to a partition, it can't be changed



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

→ One can have as many partition per topic as you want.

Brokers

↳ A kafka cluster is composed of multiple servers called Brokers.

→ Each broker is identified by its id.

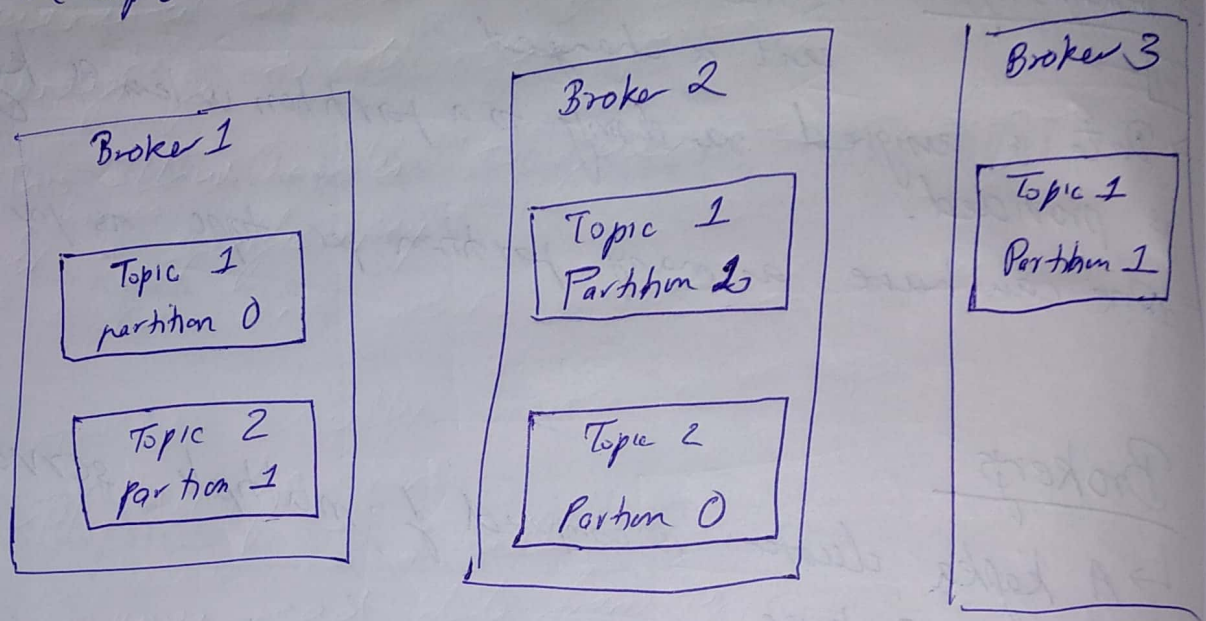
→ Each broker contains certain topic partitions.

→ After connecting to any broker (called bootstrap broker), you will be connected to entire cluster.

→ A good number to start is 3 brokers, but some big clusters have over 100 brokers.

Brokers and Topics (Example)

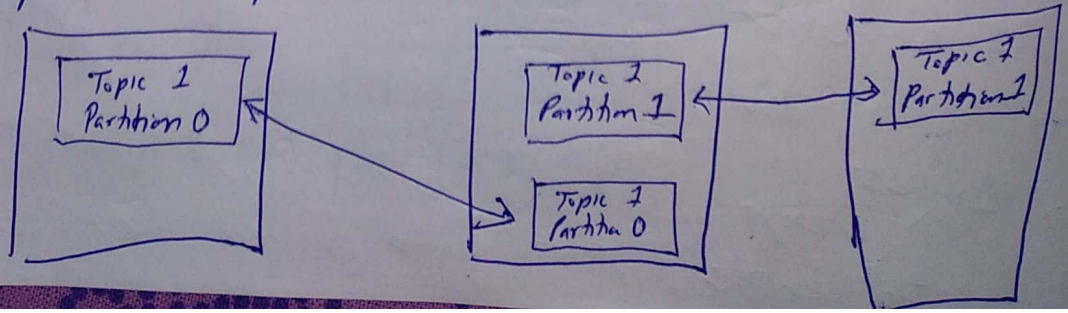
① → Example of 3 broker and 2 topics.
→ One topic has 2 & other has 3 partitions.



② → Data is distributed & broker 3 doesn't have Topic 2 data.

Topic Replication Factor

→ Topics should have a replication factor >1 (usually 2 & 3)
→ This way when a broker is down, another broken can serve the data.
→ Eg:- Topic with 2 partitions & 2 replication factor



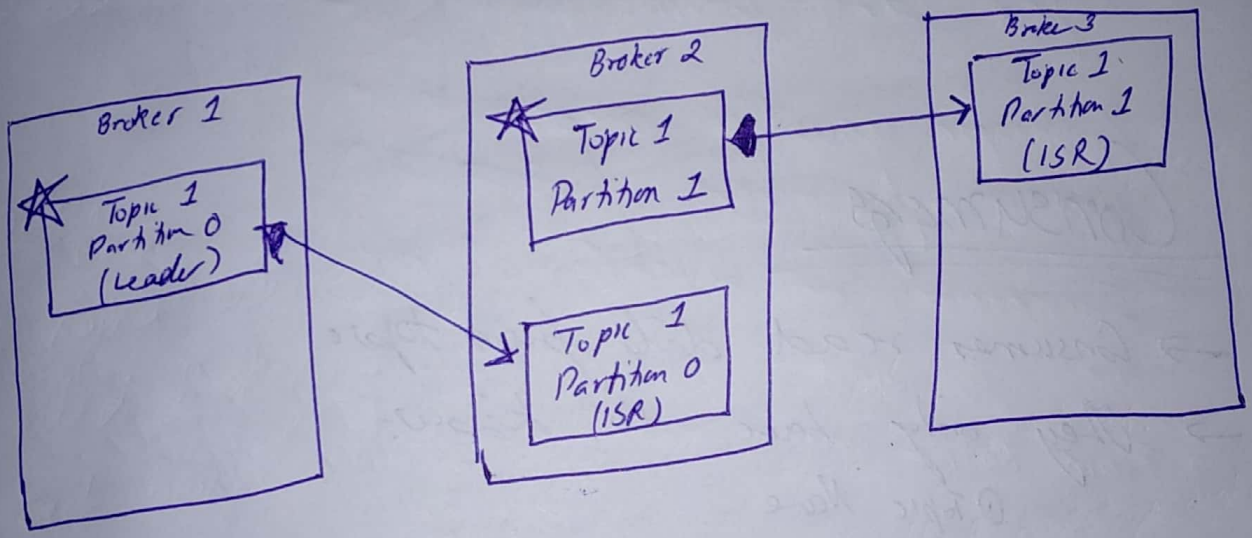
Concept of Leader for a Partition

→ At any time only 1 broker can be leader for a given partition.

→ Only that leader can receive & serve data for a partition.

→ The other brokers will synchronize the data

→ Thus each partition has one leader & multiple ISR (in-sync Replica)



Producers

→ Produces write data to topics

→ They only have to specify the ① topic name & ② one broker and Kafka will automatically take care of routing the data to right brokers.

→ Producers can choose to receive acknowledgement of data writes.

- Acks = 0 ; → no acknowledgement (possible data loss)
- Acks = 1 ; → wait for leader acknowledgment (limited data loss)
- Acks = All ; → Leader + ISR acknowledgment (no data loss)

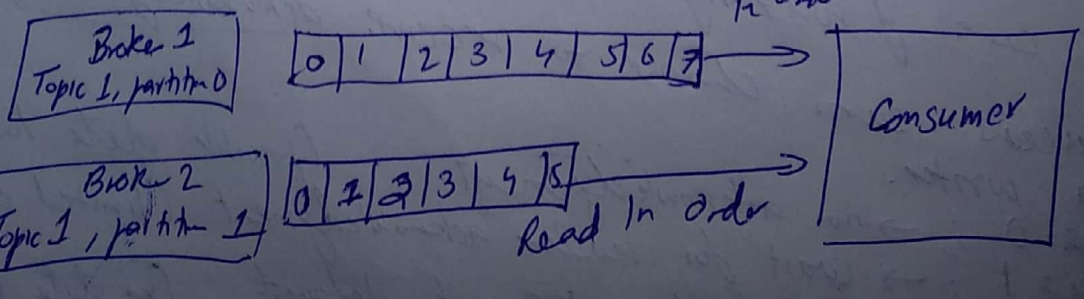
Message Keys

① → Producers can choose to send a key with message
→ If key is sent, then the producer has the guarantee that all message for that key will always go to the same partition.

② → This allows/enables to guarantee ordering for a specific key.

Consumers

→ Consumers read data from topic
→ They only have to know:
  ① Topic Name
  ② One broker name
  to connect to, & Kafka will automatically take care of pulling data from right brokers.
→ Data is read in order for each partitions.
→ Consumer will read data in order within a partition but order b/w partitions will not be maintained as they are parallel across partitions.



Consumer Groups

→ Consumers are organised into consumer groups to enhance parallelism.

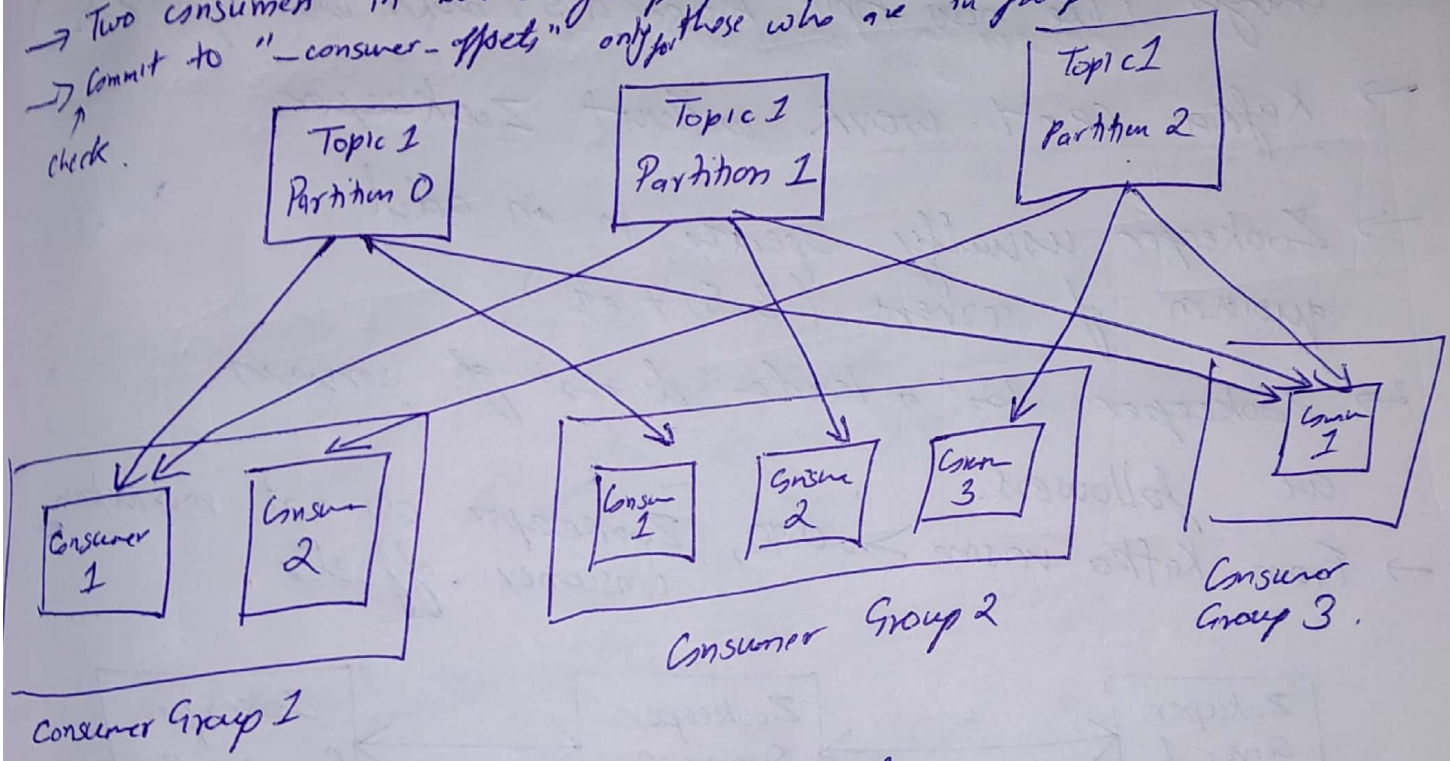
→ Each consumer within a group reads from exclusive partitions

→ You cannot have more consumers than partitions

→ Two consumers in same group will read from mutually exclusive brokers.

→ Commit to "_consumer_offsets" only for those who are in group

↑ check.



How Consumer know where to read from

Answer → Consumer offsets.

↳ Kafka stores the offset at which a consumer group has been reading.

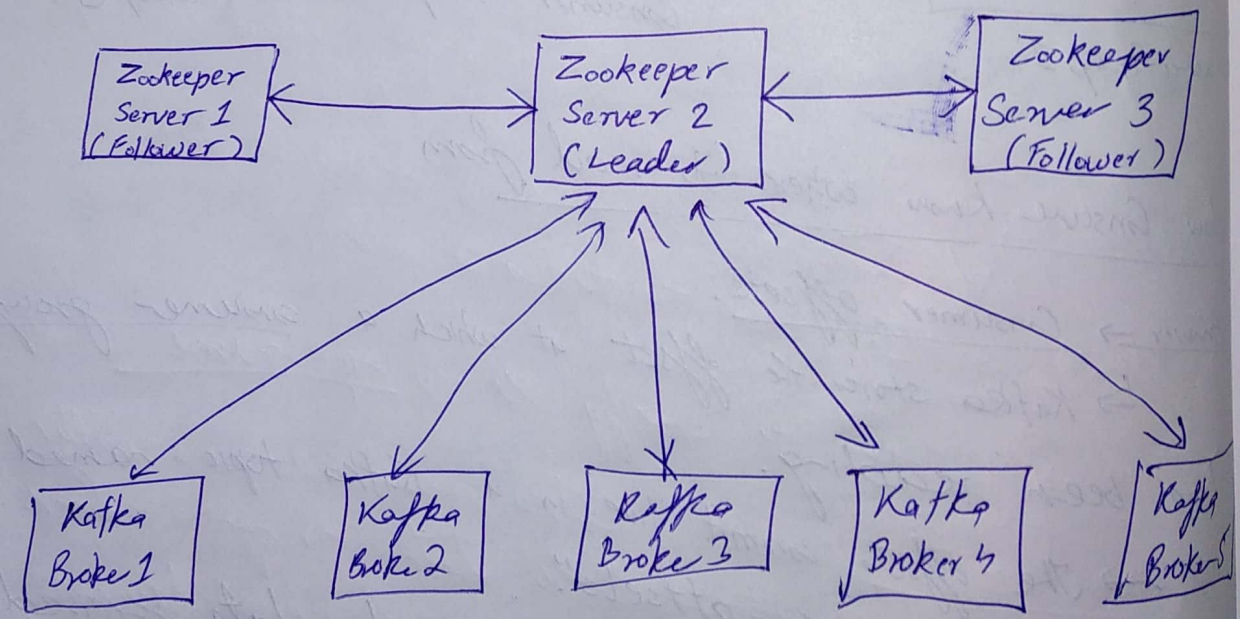
→ The offsets commit live in a Kafka topic named "_consumer_offsets"

→ When a consumer has procened data received some Kafka, it should be committing the offsets

→ Thus if a consumer process dies, it will be able to read back from where it left due to consumer offsets.

Zookeeper

→ Zookeeper manage brokers
→ It keeps a list of them.
→ It helps in performing leading election for partitions
→ It sends notification to kafka in case of changes (like new topic, broker dies, broker comes up etc.)
→ Kafka can't work without Zookeeper.
→ Zookeeper usually operate in an odd quorum of servers. (3, 5, 7 etc)
→ Zookeeper has a leader & rest of servers are followers.
→ From Kafka version >=0.90, Zookeeper do not maintain consumer offsets.



Kafka Guarantees

ordering guaranteed for partition not topic.

→ Messages are appended to a topic-partition in order they are sent

→ Messages are read in order stored in a topic partition.

→ With a replification factor of N, producers & consumers can tolerate upto N-1 brokers down.

→ This is why replication factor of 3 is a good idea.
- 1 broker down for maintenance/update
- allows 1 more to go down unexpectedly

→ Imp → As long as number of partitions remain constant for a topic, the same key will always go to same partition

Delivery Semantics for Consumers

→ Consumers choose when to commit offsets. This can be done the following ways:-

① At Once More :- offsets are commited by consumer as soon as message is received. If processing goes wrong, the message will be lost.

② At Least Once :- offset are committed after message is processed. If something goes wrong message is processed again, which can lead to duplicate processing of messages. Make sure your processing is idempotent, ie processing of duplicate message won't impact the system. (Eg:- use upsert not insert)

① → Exactly Once → Very difficult to achieve / mostly unachievable

Bottomline → Mostly we will use at least once & make sure your transformation processing is idempotent.

②