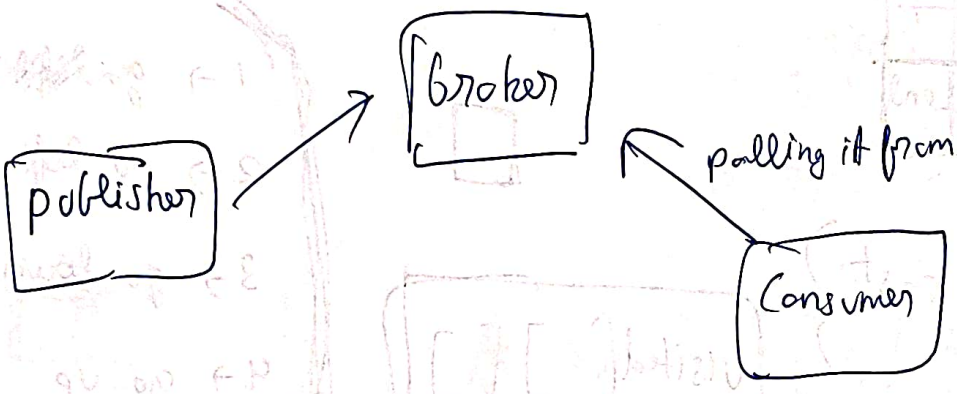
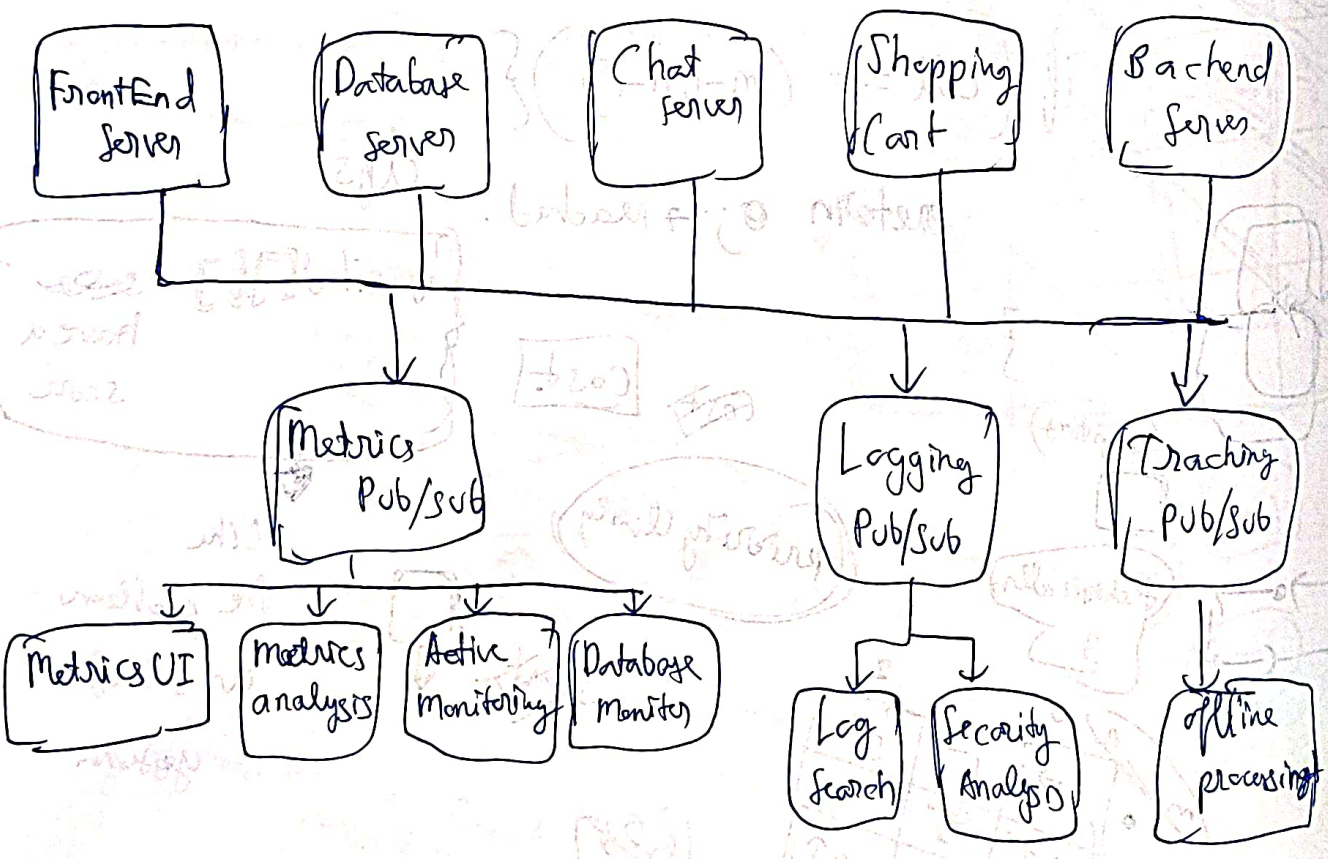
Chapter 1

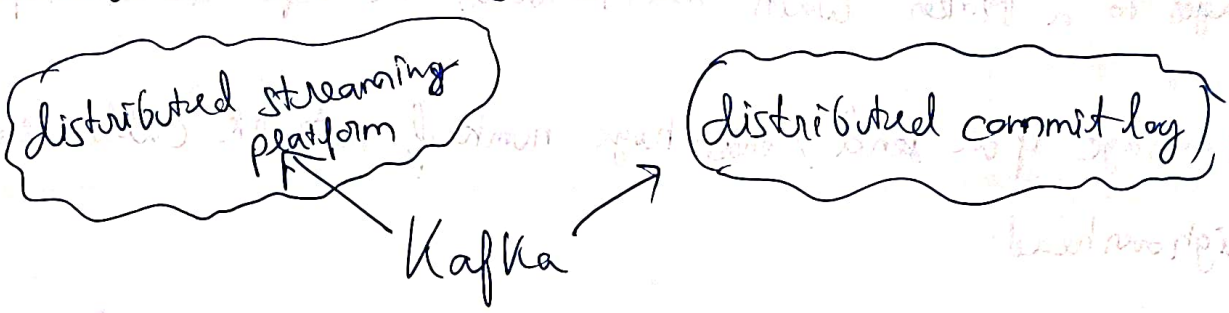


where Kafka wins over (Pub-Sub Queues)



Same duplicate data which has to be sent across to different queues for different consumption processes and requirements

Kafka solves this problem by becoming a centralised "distributed commit log".



Similar to a file system, or a database commit log. Kafka makes this durable and data can be read in the way we want. (+) there is added replication to handle failures

(Messages & Batches)

From a database background think of message as a (row/record).

Message has a optional day Key. (Key, message)

Key is also a byte array

No use of Key, unless Key is used to generate which partition this message is destined for.

(hash(Key) % no of Partitions) → deciding where to write

given that # Partitions are not changing, ensures messages with the same key are written to the same partition

Batching of messages is important. Producer before sending messages to a broker waits till it sends the info to a topic

Every message if we send, then huge number of network calls leading to high overhead.

So batch messages and send together to broker. We will learn more about these settings in chapter 2.

(+) Compression is applied on these batches to compress it and reduce payload size to send over network.

(Schemas)

In Kafka the messages are stored in (Array of bytes). Chunked Data. Kafka doesn't store messages as

key = "k1"
value = "{
name = "---",
topic = "---"
}"

Kafka needs bytes, same as network, over network we ~~don't~~ need to to send bytes and not the message as it is.

But at the producer and consumer side, we need to serializer and deserialiser respectively to process the message accordingly

JSON → JavaScript Object Notation

XML → ~~Ex~~Extensible Markup Language

these can be used for schema definitions, but have some problems or rather are not compatible among versions.

[JSON 1.0 (pub) & JSON 2.0 (sub)] → Problem.

Apache Avro is used for this Schema formating. Avro is a serialization framework providing both backward & forward compatibility for both.

Topics & Partitions

In terms of database, Topics = {database table / File in a FileSystem}

Partition is same as how we partition a table in database

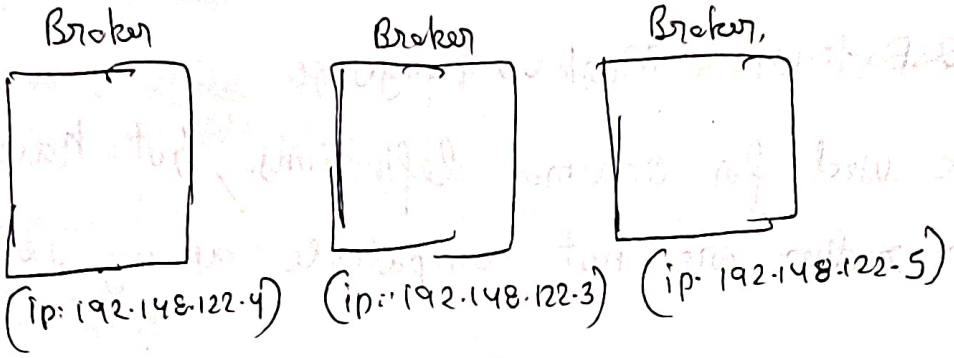
Kafka ⇒ "distributed commit log".

Messages are written to it in append only fashion.

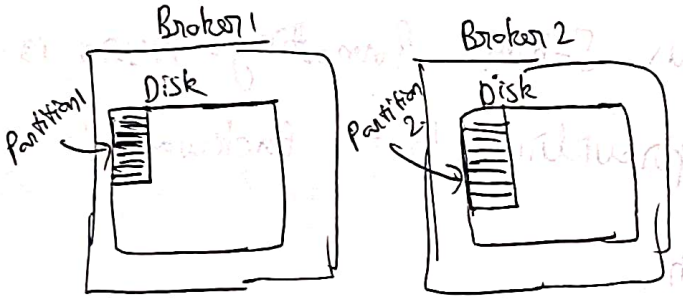
Ordering present within single partition,

Ordering not present within topic. (if multiple partitions).

Here I had a concept clearing enlightenment.



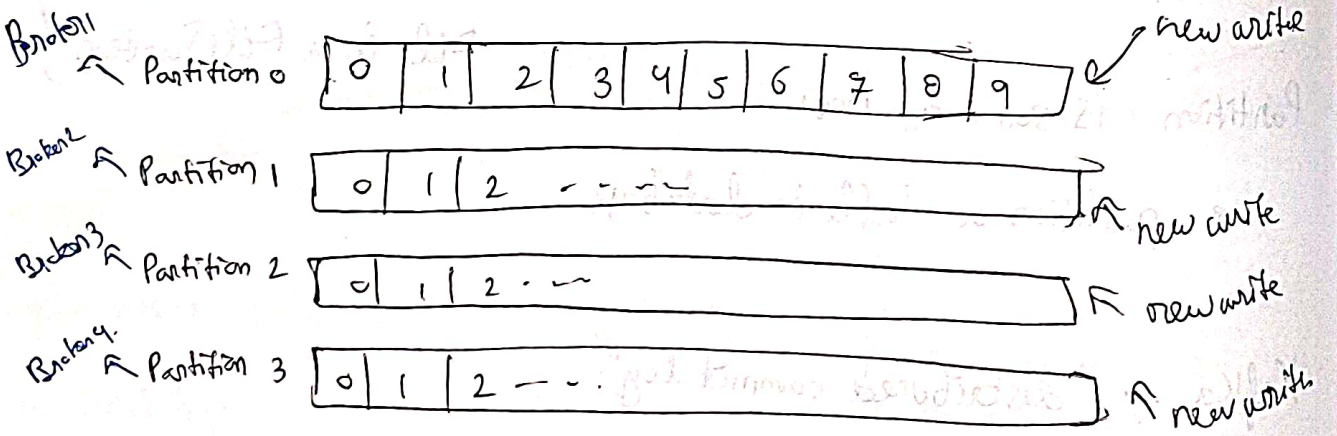
These are the actual servers in Kafka. Real machines with Ram and disk.



Inside each broker the File System is divided into multiple ~~partitions~~ segments, which act as partitions.

And group of partitions together is called a topic.

TopicName = "topicName".



Each of these ~~factors~~ partitions can be hosted on different brokers/ servers. (Actual servers). which can be individually scaled differently.

Scale up brokers as per load on partition.

(Producers & Consumers)

(Producer)

By default, when a message is produced to a topic. Kafka will try to balance messages across a partition evenly.

But we can write logic to direct message to a particular partition.

(key, message) → hash(key) → to direct message to a partition

(Consumer)

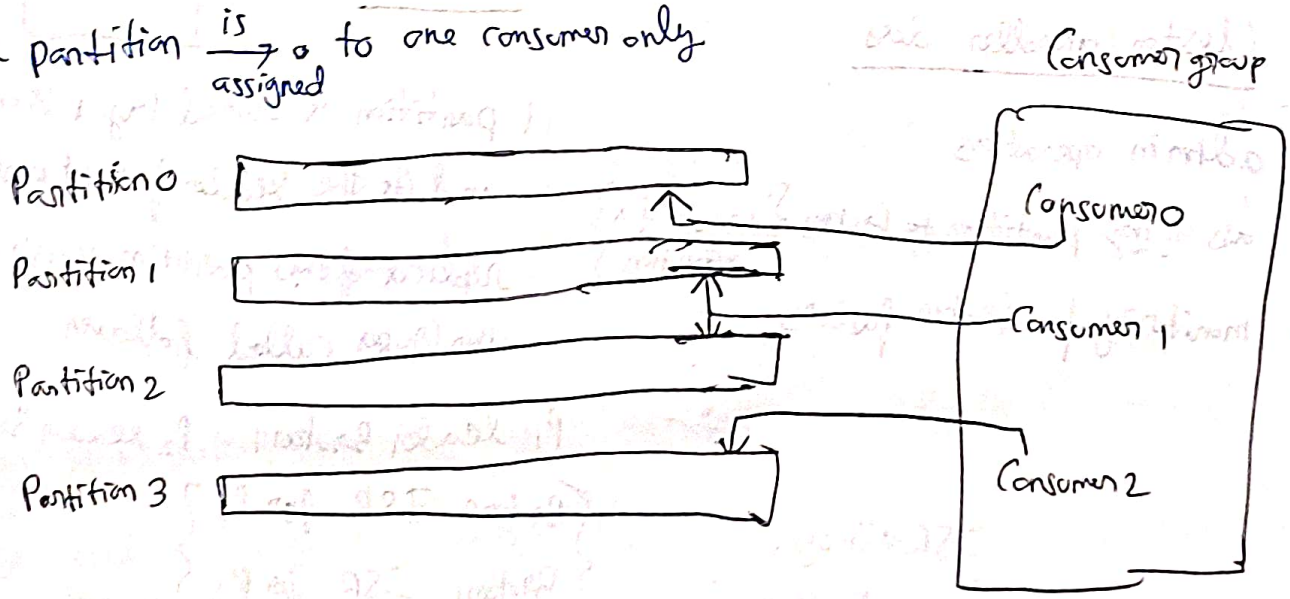
When producer produces a message to a partition, it gets automatically added to the Kafka.

And Kafka adds an offset metadata → to each message before adding it to the partitions.

Each message within a partition is given an offset, unique and is greater than the previous offset, though its not monotically increasing

Consumer works as a part of consumer group.

One partition is assigned to one consumer only

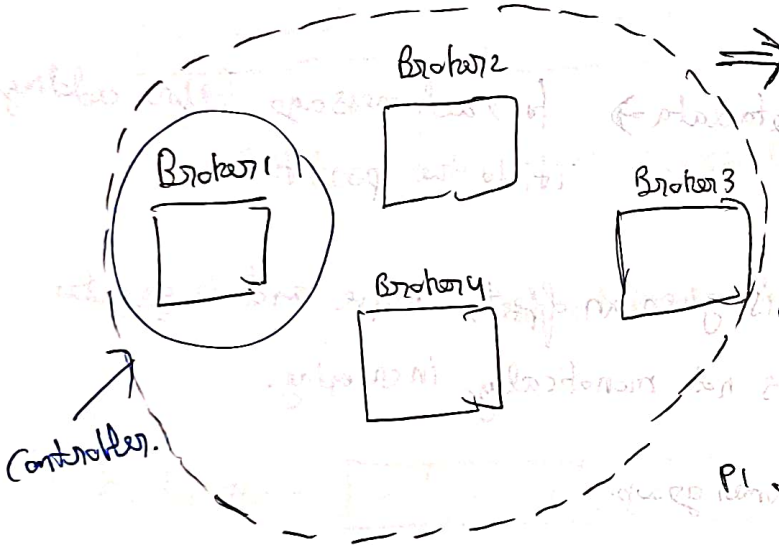


(Brokers and Clusters)

A single Kafka server is called a Broker, the Broker receives messages from producers and assigns offsets to them and write messages to store on disk.

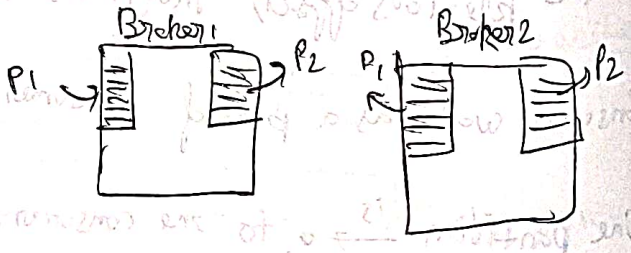
Broker → [Single instance = Broker (ip)]

Single Broker → Scale, ~~only~~ on vertical scaling can handle [1K partitions & 1M msgs/s].



⇒ (Cluster.)

In this cluster of brokers, 1 Broker is considered as the cluster controller.



1 partition is owned by 1 broker and its the leader of that partition, replica of this partition exists on other broker called follower.

P1 leader Broker1 P2 leader Broker2

{Broker2 ISR for P1
Broker1 ISR for P2}

ISR → In sync Replica

Cluster Controller does

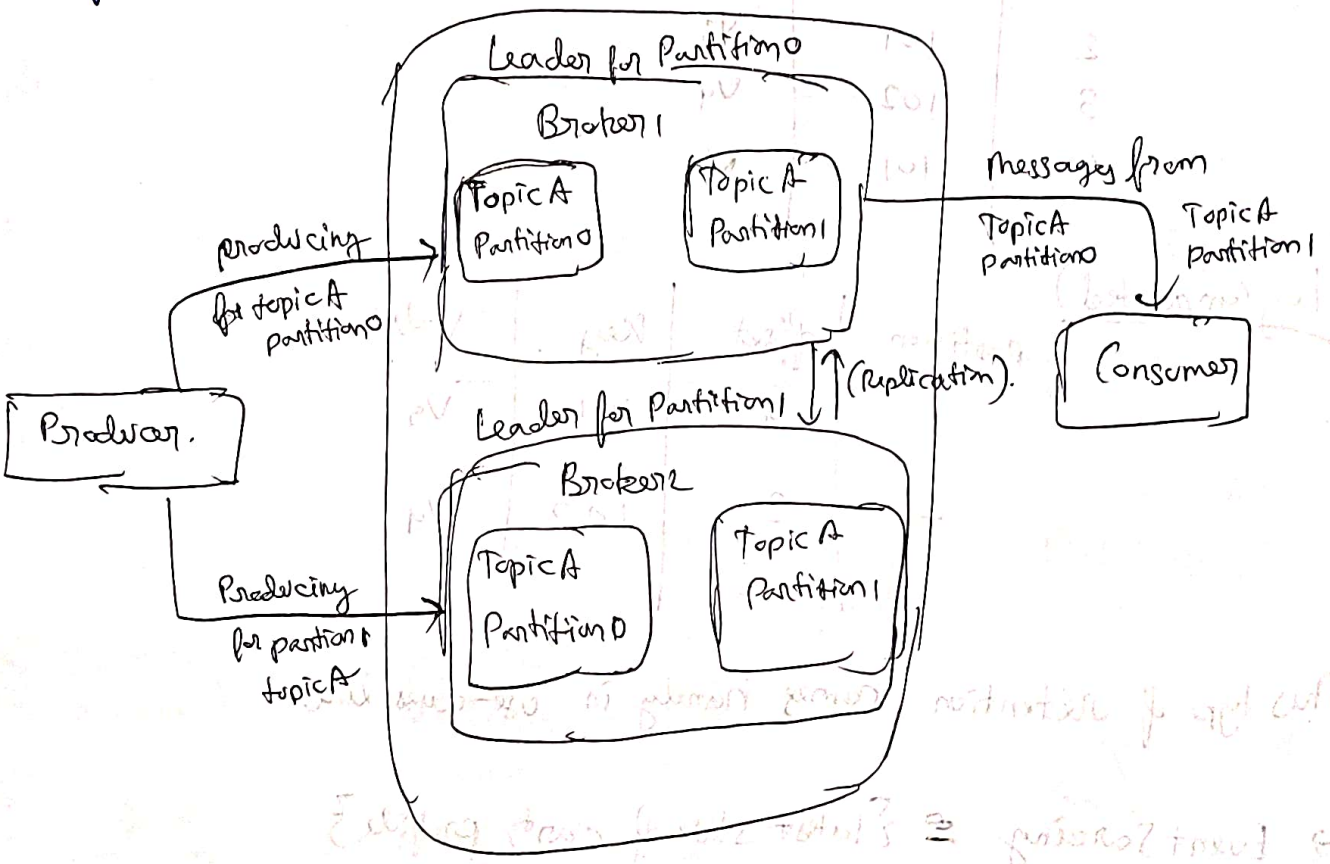
* admin operations
* assigning partitions to brokers } leader of a partition
* monitoring for broker failures

Producer produces messages to only leader broker of that partition.

Remember → Producer message goes to 1 Partition, and whichever broker is the leader of this partition.

Partition connects to that broker..

Consumer on the other hand can connect to leader or follower node of that partition.



Retention → A key feature of Kafka is that retention which makes it a durable storage of messages for some period of time

topic level retention → (7 days) Remove msgs older than 7 days

Size level retention → (1 GiB). Partition grows bigger than 1 GiB, keep removing oldest messages
partition

Another new kind of retention policy I learnt about.

log compacted) → Kafka only keeps retains the last message produced with a specific key

Example)

offset	key	value
0	101	V_1
1	102	V_2
2	101	V_3
3	102	V_4
4	101	V_5

key(101) → Partition1
key(102) → Partition2.

(Log Compacted)

partition	offset	key	value
1	4	101	V_5
2	3	102	V_4

This type of retention comes handy in use-cases like

→ Event Sourcing ≙ {latest state of cart, profile}

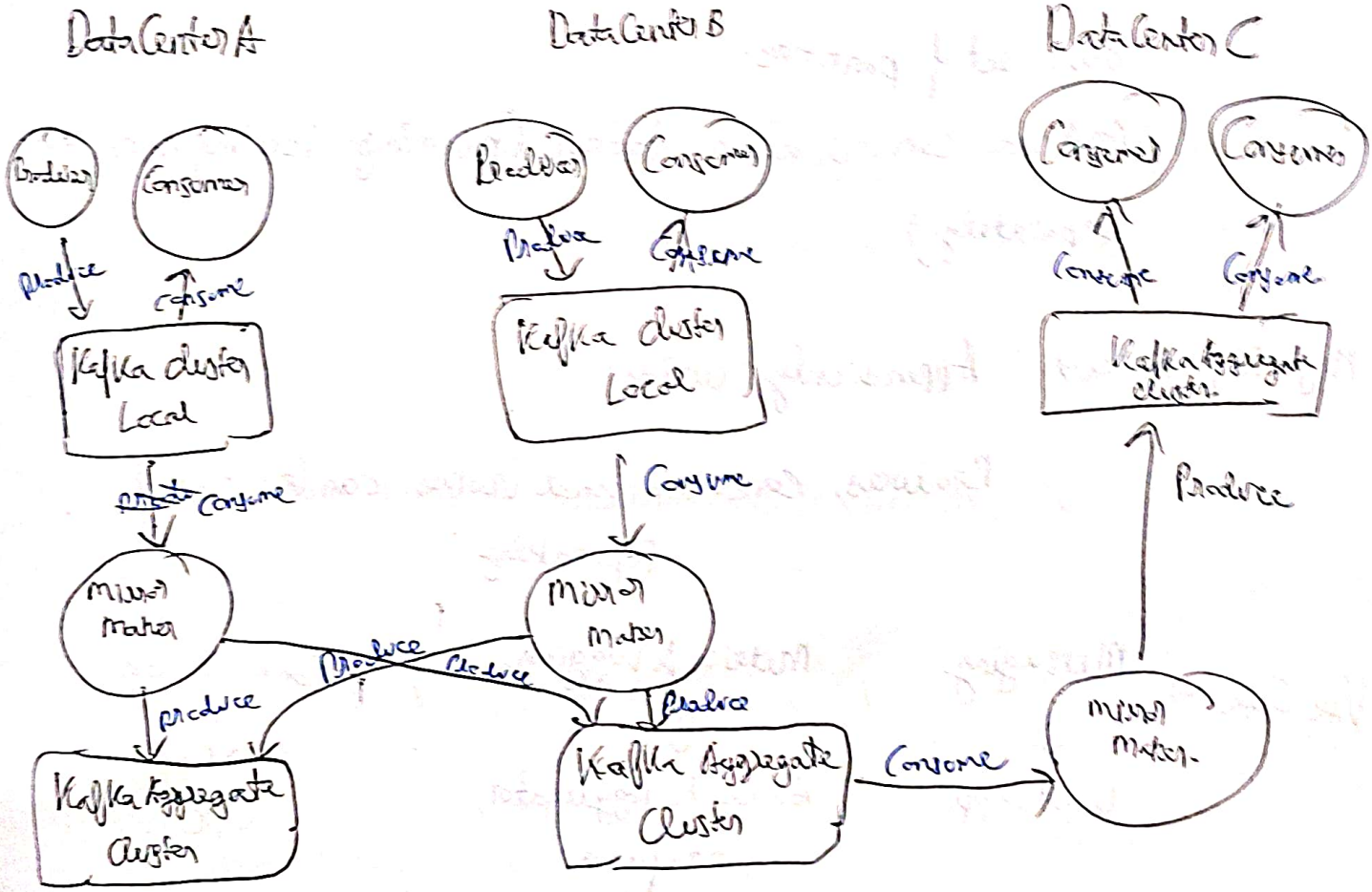
→ CDC ≙ {last change log which arrived for a user} for conflict resolution.

(Multiple Clusters)

The replication mechanisms work within the Kafka cluster are only designed to work within the Kafka clusters and not b/w multiple clusters.

Kafka project tool, called MirrorMaker used for replicating data to other clusters. MirrorMaker is responsible for copying data from one Kafka cluster into another.

At the core MirrorMaker is a queue pub/sub only. One cluster produces to MirrorMaker and another cluster consumes from it



(Features of Kafka)

Multiple Producers → No need for multiple queues

Multiple Consumers → Each consumer has its own offset, unlike other queues, messages consumed by 1 consumer, does not make it unavailable for other consumers.

Other consumers also gets that message from a different partition, having its own offsets.

Disk-based-Retention → Messages are retained in the Kafka broker, and no fear of loss of messages.

Scalable → multiple brokers on a cluster, Each broker has its own set of partitions.

And we can scale a broker (meaning scaling a partition separately).

High Performance → Append only writes,

Producers, Consumers and brokers can be scaled separately

Use Cases → Messaging ⇓ Whatsapp , Metrics & Logging ⇓ Ad Click Aggregator, Grafana , Commit Log ⇓ CDC.

Birth of Kafka → Kafka developed by LinkedIn's principal Engineer. Who previously developed Voldemort → Distributed K-V Store

Then Kafka → Kafka scaled so high that it was responsible for streaming 5PB data/daily

With horizontal scaling → Create more partitions, scale consumers, producers, brokers separately

Apache Avro for serialisation.

Open Source → 2010 Kafka was made open source, then it became an Apache Incubator Project and in 2012 came out as an open source.

Apache Incubator Project → Apache Incubator is the official entry path for new projects to become part of Apache Software Foundation. (ASF). From there these projects graduate to become top-level projects.