As soon as a company has real-time data streaming needs, a streaming platform must be put in place. Apache Kafka is one of the most popular data streaming processing platforms in the industry today.

What are the use cases of Apache Kafka?

The use cases of Apache Kafka are many. These include stream processing for different business applications. Apache Kafka makes up the storage mechanism for some of the prominent stream processing frameworks, e.g., Apache Flink, Samza.

- Messaging systems
- Activity Tracking
- Gather metrics from many different locations, i.e., IoT devices
- Application logs analysis
- De-coupling of system dependencies
- Integration with Big Data technologies like Spark, Flink, Storm, Hadoop.
- Event-sourcing store

How is Kafka concretely being used within the industry?

UBER : The speed and flexibility of Kafka allows Uber to adjust their pricing models to the constantly evolving events in the real world (number of available drivers and their position, users and their position, weather event, other events), and bill users the right amount to manage offer and demand.
Netflix : It has integrated Kafka as the core component of its data platform. They refer to it internally as their Keystone data pipeline. As part of Netflix's Keystone, Kafka handles billions of events a day. Just to give an idea about the huge amount of data that Kafka can handle, Netflix sends about 5 hundred billion events and 1.3 petabytes of data per day into Kafka.

Kafka is at the core of lots of the services we enjoy on a daily basis from some of the world's largest tech companies such as Uber, Netflix, Airbnb, LinkedIn, Apple & Walmart.

Kafka Setup

- **Download**

  https://kafka.apache.org/downloads

  Download scala latest version in binary downloads.
- **Setup Kafka in Path**

  We are doing this to access Kafka Binaries with ease.
  - nano .zshrc
  - PATH="$PATH:/Users/shivamjaiswal/kafka_2.13-3.0.0/bin"
  - (We will not be required to provide full path for accessing Kafka binaries)

See complete setup guidance here :
https://www.conduktor.io/kafka/how-to-install-apache-kafka-on-mac

Start Zookeeper

```
zookeeper-server-start.sh   ~/kafka_2.13-3.4.0/config/zookeeper.properties
```

Start Kafka

```
kafka-server-start.sh  ~/kafka_2.13-3.4.0/config/server.properties
```

Changing the Kafka and Zookeeper data storage directory

- edit the zookeeper.properties file at ~/kafka_2.13-3.0.0/config/zookeeper.properties and set the following to your heart's desire dataDir=/your/path/to/data/zookeeper.
- edit the server.properties file at ~/kafka_2.13-3.0.0/config server.properties and set the following to your heart's desire log.dirs=/your/path/to/data/kafka.

Note:- Zookeeper is going to be completely removed from Kafka 4.0.

Install Kafka with KRaft (without Zookeeper) on Mac OS X

1. Install Java JDK version 11
2. Download Apache Kafka from https://kafka.apache.org/downloads under 'Binary Downloads'
3. Extract the contents on your Mac
4. Generate a cluster ID and format the storage using kafka-storage.sh
5. Start Kafka using the binaries
6. Setup the $PATH environment variables for easy access to the Kafka binaries

1. generate a new ID for your cluster.

```
~/kafka_2.13-3.4.0/bin/kafka-storage.sh random-uuid
```

this returns a UUID, for example bs8tpjusSZuLirvAj700Wg

2. Format your storage directory (replace <uuid> by your UUID obtained above)

```
~/kafka_2.13-3.4.0/bin/kafka-storage.sh format -t <uuid> -c ~/kafka_2.13-3.0.0/config/kraft/server.properties
```

This will format the directory that is in the `log.dirs` in the `config/kraft/server.properties` file (by default `/tmp/kraft-combined-logs`)

3. Now you can launch the broker itself in daemon mode by running this command

```
~/kafka_2.13-3.0.0/bin/kafka-server-start.sh ~/kafka_2.13-3.4.0/config/Kraft/server.properties
```

Kafka Terms :

Kafka Broker

- A single Kafka server is called a Kafka Broker.
- Kafka brokers store data in a directory on the server disk they run on.

Kafka Cluster

An ensemble of Kafka brokers working together is called a Kafka cluster.

Bootstrap Server

- A client that wants to send or receive messages from the Kafka cluster **may connect to any broker in the cluster.** Every broker in the cluster has metadata about all the other brokers and will help the client connect to them as well, and **therefore any broker in the cluster is also called a bootstrap server.**
- The bootstrap server will return metadata to the client that consists of a list of all the brokers in the cluster. Then, when required, the client will know which exact broker to connect to to send or receive data, and accurately find which brokers contain the relevant topic-partition.
- In practice, it is common for the Kafka client to reference at least two bootstrap servers in its connection URL

Topic

Similar to how databases have tables to organize and segment datasets, Kafka uses the concept of topics to organize related messages.

- A topic is identified by its name.
- Kafka topics can contain any kind of message in any format, and the sequence of all these messages is called a **data stream.**
- Data in Kafka topics is deleted after one week by default (also called the default message retention period), and this value is configurable. This mechanism of deleting old data ensures a Kafka cluster does not run out of disk space by recycling topics over time.
- Kafka topics are **immutable**: once data is written to a partition, it cannot be changed

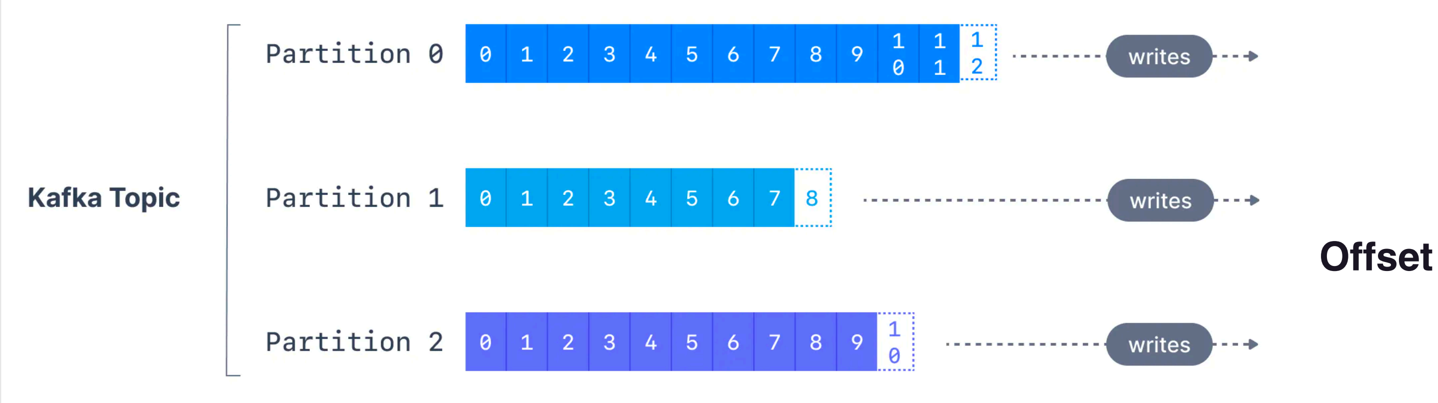
Unlike database tables, Kafka topics are not query-able. Instead, we have to create Kafka producers to send data to the topic and Kafka consumers to read the data from the topic in order.

Partitions

- Topics are broken down into a number of partitions.
- Partitions are 0 indexed.
- partitions for a given topic will be distributed among the brokers evenly, to achieve load balancing and scalability.

Offset

- The offset is an integer value that Kafka adds to each message as it is written into a partition.
- Offsets are 0 indexed.
- Each message in a given partition has a unique offset.
- Even though we know that messages in Kafka topics are deleted over time (as seen above), the offsets are not re-used. They continually are incremented in a never-ending sequence

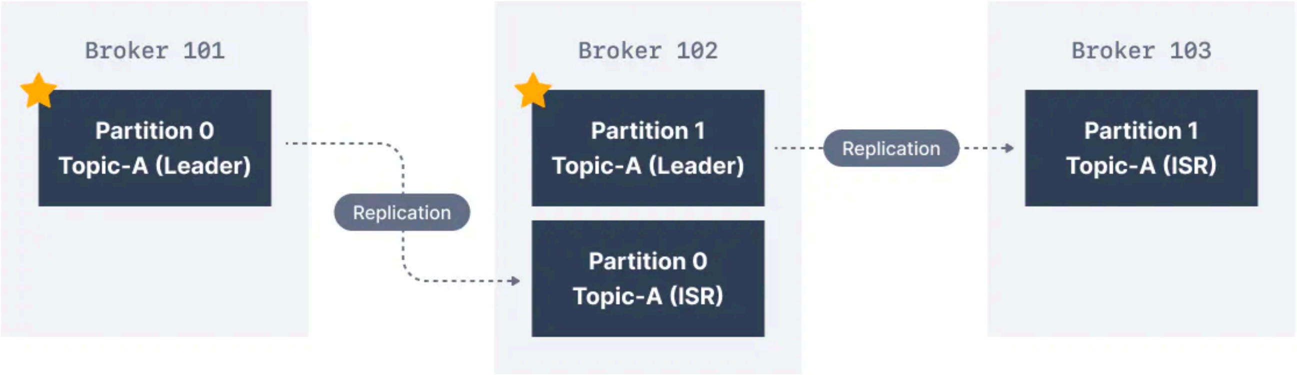


Replication Factor

- Replication means that data is written down not just to one broker, but many.
- The replication factor is a topic setting and is specified at topic creation time.
- replication factor of 1 means no replication, replication factor of 3 is a commonly used .

Kafka Partitions Leader and Replicas

- For a given topic-partition, one Kafka broker is designated by the cluster to be responsible for sending and receiving data to clients. That broker is known as the leader broker of that topic partition. Any other broker that is storing replicated data for that partition is referred to as a replica.
- Therefore, each partition has one leader and multiple replicas.
- An ISR is a replica that is up to date with the leader broker for a partition. Any replica that is not up to date with the leader is out of sync.



Ordering

If a topic has more than one partition, Kafka guarantees the order of messages within a partition, but there is no ordering of messages across partitions while reading.

Producers

- Applications that send data into topics are known as Kafka producers. Applications typically integrate a Kafka client library to write to Apache Kafka.
- A Kafka producer sends messages to a topic, and messages are distributed to partitions according to a mechanism such as key hashing.In case the key (`key=null`) is not specified by the producer, messages are distributed evenly across partitions in a topic(Round Robin Fashion).
- For a message to be successfully written into a Kafka topic, a producer must specify a level of acknowledgment (acks).
- We need to use appropriate serialiser to send message to Kafka.

Consumers

- Applications that read data from Kafka topics are known as consumers.
- Reads data from lower to higher offset.
- Consumers can read from one or more partitions at a time in Apache Kafka, and data is read in order **within each partition** .
- By default, Kafka consumers will only consume data that was produced after it first connected to Kafka.
- Data being consumed must be deserialized in the same format it was serialized in.

Kafka Partioner

A Kafka partitioner is a code logic that takes a record and determines to which partition to send it into.In the default Kafka partitioner, the keys are hashed using the **murmur2 algorithm.**

Messages sent to a Kafka topic that do not respect the agreed-upon serialization format are called poison pills.

Consumer group

- Kafka Consumers that are part of the same application and therefore performing the same "logical job" can be grouped together as a Kafka consumer group.
- Consumers automatically use a `GroupCoordinator` and a `ConsumerCoordinator` to assign consumers to a partition and ensure the load balancing is achieved across all consumers in the same group.
- Each topic partition is only assigned to one consumer within a consumer group, but a consumer from a consumer group can be assigned multiple partitions.

Consumer Offsets

- Kafka brokers use an internal topic named `__consumer_offsets` that keeps track of what messages a given **consumer group** last successfully processed.
- the consumer will regularly **commit** the latest processed message, also known as **consumer offset**.

- The process of committing offsets is not done for every message consumed (because this would be inefficient), and instead is a periodic process.
- If rebalancing happens ,assigned consumer starts reading from consumer offset.
- A consumer will not be able to consume messages from beginning if consumer offset is already set for respective consumer group.
- Reset offset cannot happen when the consumer is running.

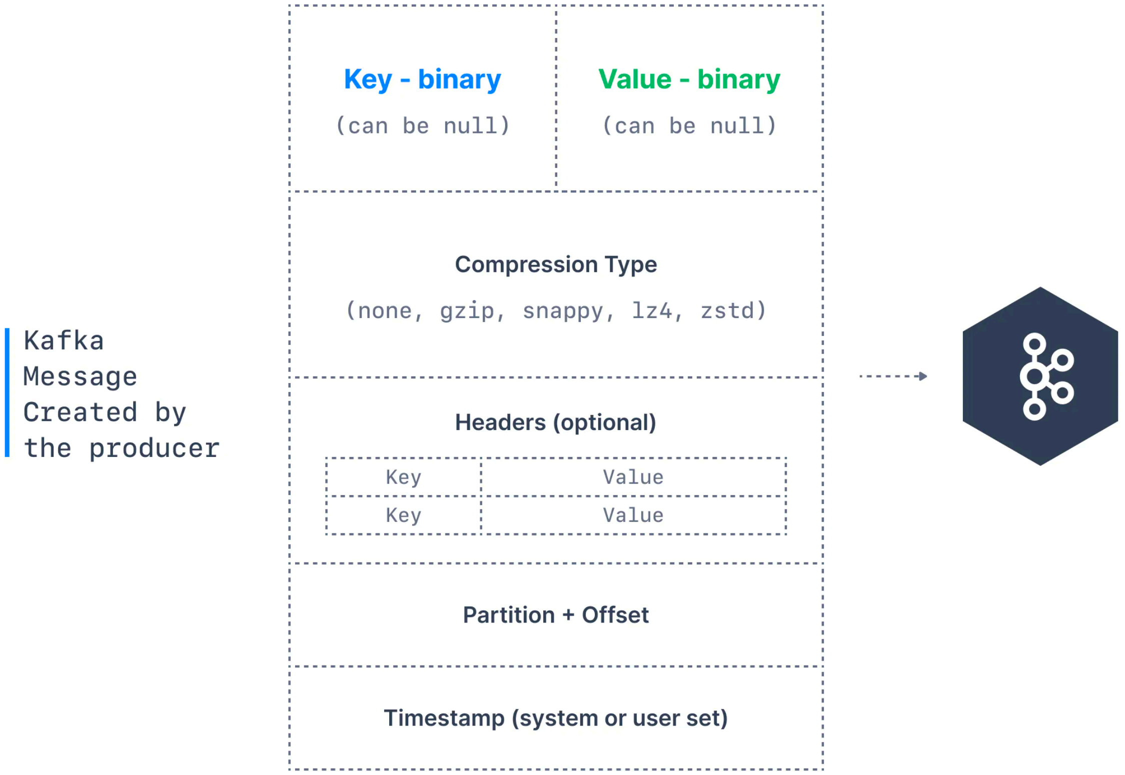
Delivery semantics for consumers

- By default, Java consumers automatically commit offsets (controlled by the `enable.auto.commit=true` property)
  every `auto.commit.interval.ms` (5 seconds by default) when `.poll()` is called.
- A consumer may opt to commit offsets by itself (`enable.auto.commit=false`).
- Depending on when it chooses to commit offsets, there are delivery semantics available to the consumer. The three delivery semantics are explained below.
  - ➡ **At most once**
    Offsets are committed as soon as the message is received.
    If the processing goes wrong, the message will be lost (it won't be read again).
  - ➡ **At Least once**
    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. Therefore, it is best practice to make sure data processing is idempotent (i.e. processing the same message twice won't produce any undesirable effects
  - ➡ **Exactly once**
    This can only be achieved for Kafka topic to Kafka topic workflows using the transactions API. The Kafka Streams API simplifies the usage of that API and enables exactly once using the setting `processing.guarantee=exactly_once_v2` (`exactly_once` on Kafka < 2.5).
    For Kafka topic to External System workflows,

to *effectively* achieve exactly once, you must use an idempotent consumer.In practice, at least once with idempotent processing is the most desirable and widely implemented mechanism for Kafka consumers.

Kafka Message Anatomy :

Kafka messages are created by the producer. A Kafka message consists of the following elements:



Key : Key is optional in the Kafka message and it can be null. A key may be a string, number, or any object and then the key is serialized into binary format.

Value : The value represents the content of the message and can also be null. The value format is arbitrary and is then also serialized into binary format.

Compression Type : Kafka messages may be compressed. The compression type can be specified as part of the message. Options are none, gzip, lz4, snappy, and zstd

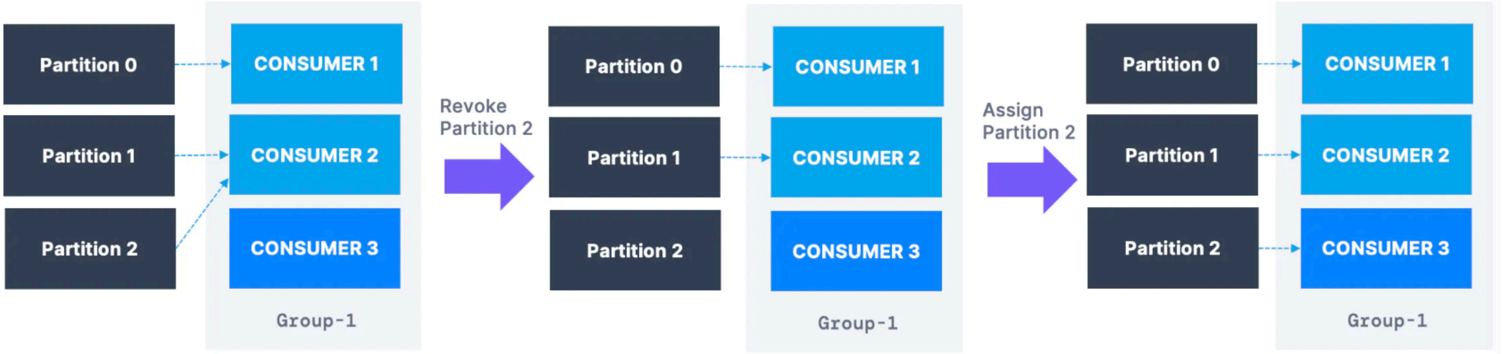
Headers : There can be a list of optional Kafka message headers in the form of key-value pairs. It is common to add headers to specify metadata about the message, especially for tracing.

Partition + Offset : Once a message is sent into a Kafka topic, it receives a partition number and an offset id. The combination of topic+partition+offset uniquely identifies the message.
Timestamp : A timestamp is added either by the user or the system in the message.

Consumer Rebalance

Consumer rebalances happen for the following events:

- Number of partitions change for any of the subscribed topics
- A subscribed topic is created or deleted
- An existing member of the consumer group is shutdown or fails
- A new member is added to the consumer group
- Rebalancing techniques
  1. **Eager rebalance** : all partitions get revoked from consumer and get reassigned.
  2. **Co-operative rebalancing** : only subset of partition(which are required to rebalance) get revoked hence other consumers keeps on reading the data from unrequited partitions.



Producer Configurations

1. min.insync.replicas
   it is used together with acks=all , and it assures acks from minimum x brokers.i.e., min.insync.replicas=x{1,2,3,4….}.
2. acks
   Kafka producers only write data to the current leader broker for a partition.It must also specify a level of acknowledgment `acks` to specify if the message must be written to a minimum number of replicas before being considered a successful write.producers consider messages as "written successfully" for

➡acks = 0 : the moment the message was sent without waiting for the broker to accept it at all.

➡acks =1 : when the message was acknowledged by only the leader.

➡acks=all(-1) : when the message is accepted by all in-sync replicas (ISR).
Leader partition ask for acks to other ISR partitions and responds to producer together with its acks.
default values is acks = all (-1)

3. retries
The retries setting determines how many times the producer will attempt to send a message before marking it as failed. The default values is MAX_INT, i.e., 2147483647 for Kafka >= 2.1.
4. retry.backoff.ms
time to wait before next retry
default values is 100ms
5. delivery.timeout.ms
retries are bounded by a timeout
default values is 120000ms == 2 min
6. max.in.flight.requests.per.connection
how many produce request(message batches) can be made in parallel
default values is 5, set it to 1 to ensure ordering of messages.
7. enable.idempotence =true
Note : For **idempotent producer** ordering will be same even for higher values and duplicates are not introduced due to network retries.This producers are default from Kafka 3.0.
if we enable idempotence enable=idempotence=true, then it is required for max.in.flight.requests.per.connection to be less than or equal to 5 with message ordering preserved for any allowable value!!
8. compression.type
can be at broker or producer level , producer level is recommended.
Note : at consumer end we don't need to decompress the batches , consumers do it by itself.
default values is none.

9. linger.ms
   how long to wait until we send a batch. Adding a small number for example 5ms helps add more messages in a batch at the expense of latency.
   default values is 0.

10. batch.size
    if a batch is filled before linger.ms, increase the batch size. smaller batches leads to more request and higher latency so prefer to have larger batch size.
    default values is 16kb
11. buffer.memory
    if the producer produces faster than the broker can take, the records will be buffered in memory
    default size is 33554432 (32 MB):the size of send buffer
    this buffer will fill up over time and empty back down when the throughput to the broker increases
12. max.block.ms
    the time the .send() will block until throwing the exception.
    Exception occurs when:
    if the buffer is full , then the .send() method will start to block(won't return right away).
    if broker is down or overloaded
    if 60 sec has elapsed

Consumer Configurations

1. partition.assignment.strategy
   RangeAssignor: assign partitions on a per-topic basis (can lead to imbalance)
   RoundRobin: assign partitions across all topics in a round-robin fashion, optimal balance.
   StickyAssignor: balanced like RoundRobin, and then minimises partition movements when consumers join/leave the group in order to minimise movements

CooperativeStickyAssignor: rebalance strategy is identical to StickyAssignor but supports cooperative rebalances and therefore consumers can keep on consuming from the topic.

2. group.instance.id
   it makes the consumer a static member. I.e., upon leaving consumer has session.timeout.ms to join back and get back its partition, without triggering a rebalance.

Kafka Java Programming

- Create gradle project.
- Add dependencies

```
implementation 'org.apache.kafka:kafka-clients:3.0.0'
implementation 'org.slf4j:slf4j-api:1.7.32'
implementation ‘org.slf4j:slf4j-simple:1.7.32'
```

- `flush()` and `close()` are required to ensure the producer is shut down after the message is sent to Kafka.

Sticky Partitioner(used for performance improvement)

which means the producer that receives messages sent in time close to each other will try to fill a batch into ONE partition before switching to creating a batch for another partition.
Keys become useful when a user wants to introduce ordering and ensure the messages that share the same key end up in the same partition.

Seek and Assign

to read specific messages from specific partitions, the .seek() and .assign() API may help you. These APIs are also helpful to replay data from a specific offset.

Left to traverse
Kafka connect
Kafka streams
KSql HandsOn
Kafka setup and administration
Confluent Schema Registry and Kafka rest proxy
Kafka security
Kafka monitoring and operations
Advanced configuration

It is tough to achieve fault tolerance, exactly once, and distribution ordering