Definitions

Definitions

1. **Kafka topics**

   Kafka topics organize related events. Kafka topics can contain any kind of message in any format, and the sequence of all these messages is called a data stream.
   For example, we may have a topic called **logs,** which contains logs from an application.

   Topics are roughly analogous to SQL tables. However, unlike SQL tables, Kafka topics are not queryable. Instead, we must create Kafka producers and consumers to utilize the data. The data in the topics are stored in the key-value form in binary format.

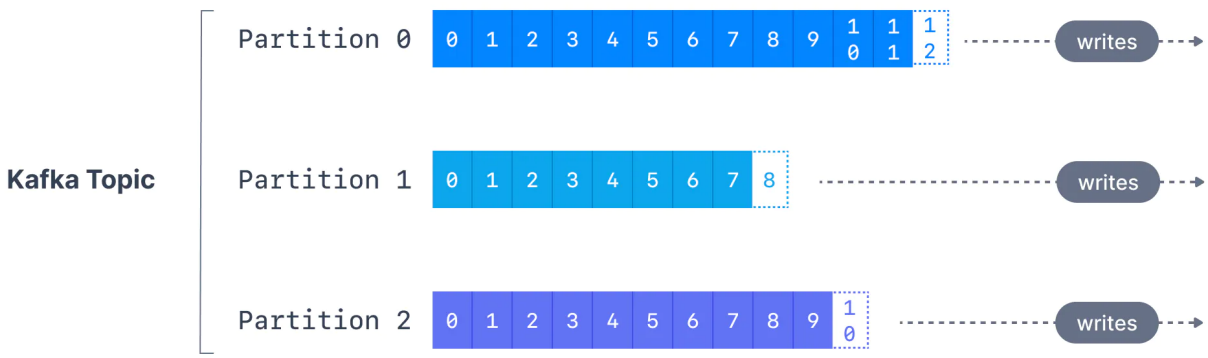
   Data in Kafka topics is deleted after one week by default (also called the default message retention period), and this value is configurable. This mechanism of deleting old data ensures a Kafka cluster does not run out of disk space by recycling topics over time.

   Kafka topics are **immutable**: once data is written to a partition, it cannot be changed

2. **Kafka Partitions**

   Topics are broken down into a number of partitions. A single topic may have more than one partition, it is common to see topics with 100 partitions.

   The number of partitions of a topic is specified at the time of topic creation. Partitions are numbered starting from `0` to `N-1`, where `N` is the number of partitions. The figure below shows a topic with three partitions, with messages being appended to the end of each one.



3. **Kafka offsets**

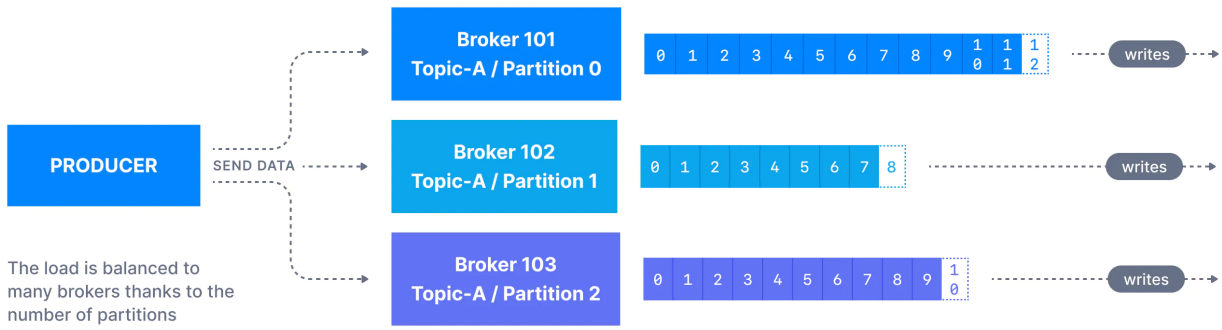
   Apache Kafka offsets represent the position of a message within a Kafka Partition. Offset numbering for every partition starts at `0` and is incremented for each message sent to a specific Kafka partition. This means that Kafka offsets only have a meaning for a specific partition, e.g., offset 3 in partition 0 doesn't represent the same data as offset 3 in partition1.

   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.

   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.

4. **Kafka 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 (more on it below).

   For a message to be successfully written into a Kafka topic, a producer must specify a level of acknowledgment (acks).

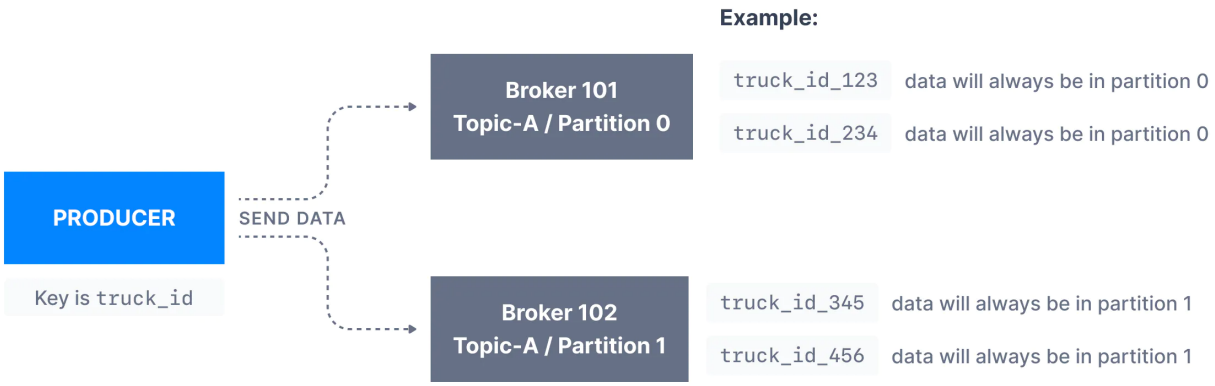
5. **Message Keys**

   Each event message contains an optional key and a value.

   In case the key (`key=null`) is not specified by the producer, messages are distributed evenly across partitions in a topic. This means messages are sent in a round-robin fashion (partition *p0* then *p1* then *p2*, etc... then back to *p0* and so on...).

   If a key is sent (`key != null`), then all messages that share the same key will always be sent and stored in the same Kafka partition. A key can be anything to identify a message - a string, numeric value, binary value, etc.

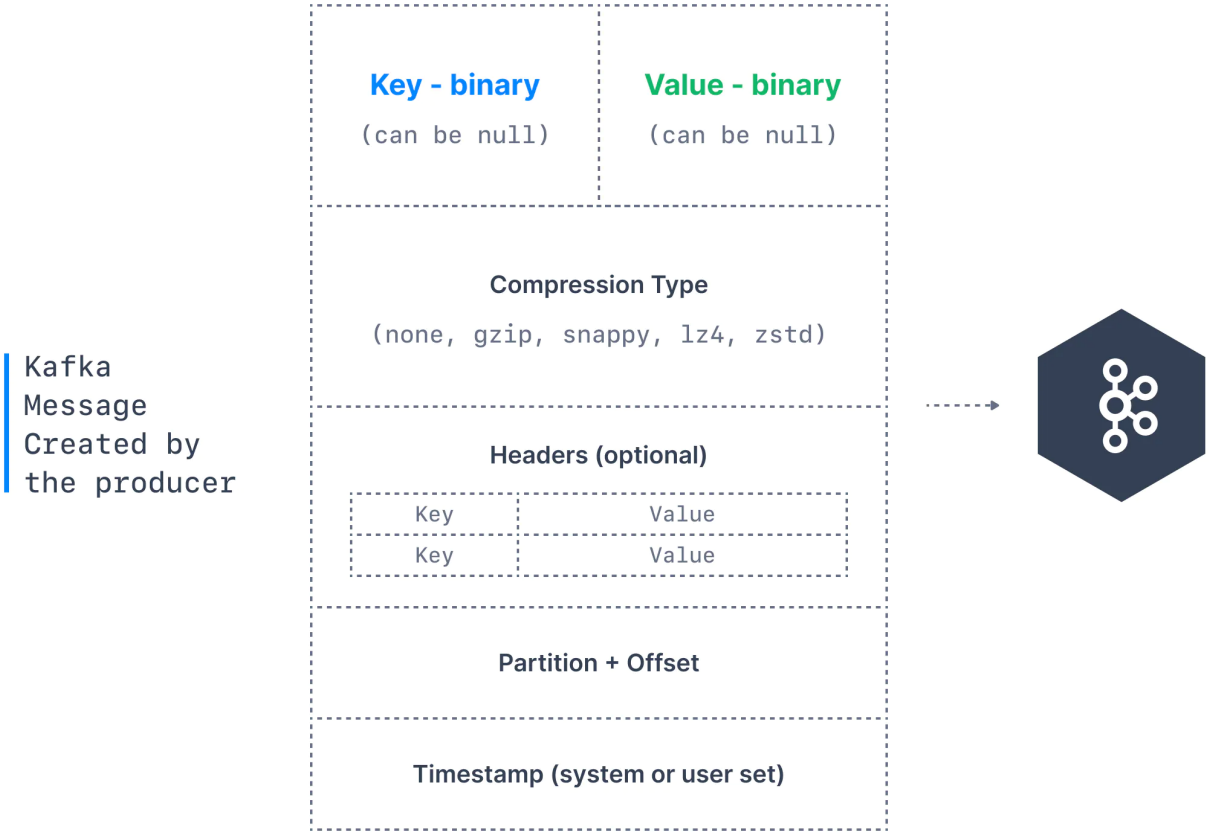
Kafka message keys are commonly used when there is a need for message ordering for all messages sharing the same field. For example, in the scenario of tracking trucks in a fleet, we want data from trucks to be in order at the individual truck level. In that case, we can choose the key to be `truck_id`. In the example shown below, the data from the truck with id *truck_id_123* will always go to partition *p0*.



A Kafka partitioner is a code logic that takes a record and determines to which partition to send it into. In that effect, it is common for partitioners to leverage the Kafka message keys to route a message into a specific topic-partition. As a reminder, all messages with the same key will go to the same 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)
```

6. **Kafka message format**



7. **Kafka message serializers**

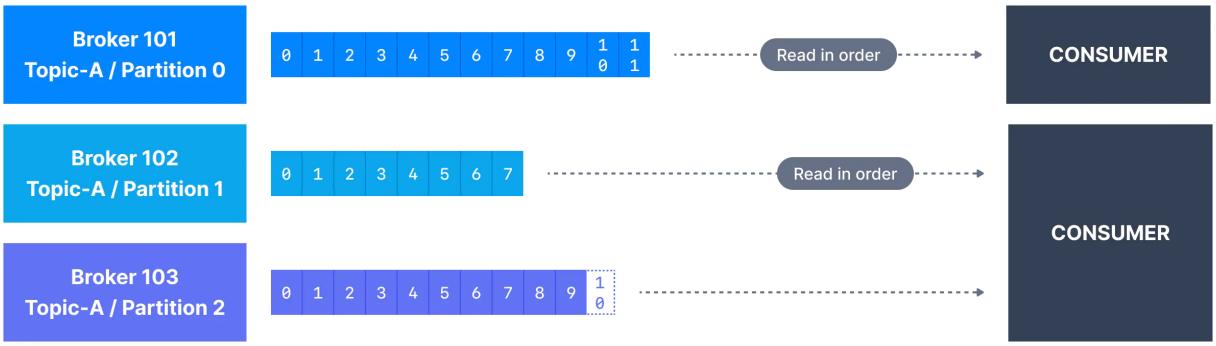
   In many programming languages, the key and value are represented as objects, which greatly increases the code readability. However, Kafka brokers expect byte arrays as keys and values of messages. The process of transforming the producer's programmatic representation of the object to binary is **called message serialization**.

   As part of the Java Client SDK for Apache Kafka, several serializers already exist, such as string (which supersedes JSON), integer, float. Other serializers may have to be written by the users, but commonly distributed Kafka serializers exist and are efficiently written for formats such as JSON-Schema, Apache Avro and Protobuf, thanks to the Confluent Schema Registry.

8. **Kafka consumers**

   Applications that read data from Kafka topics are known as consumers. Applications integrate a Kafka client library to read from Apache Kafka.

   Consumers can read from one or more partitions at a time in Apache Kafka, and data is read in order **within each partition** as shown below.



A consumer always reads data from a lower offset to a higher offset and cannot read data backwards (due to how Apache Kafka and clients are implemented).

If the consumer consumes data from more than one partition, the message order is not guaranteed across multiple partitions because they are consumed simultaneously, but the message read order is still guaranteed within each individual partition.

By default, Kafka consumers will only consume data that was produced after it first connected to Kafka. Which means that to read historical data in Kafka, one must specify it as an input to the command, as we will see in the practice section.

Kafka consumers are also known to implement a "pull model". This means that Kafka consumers must request data from Kafka brokers in order to get it (instead of having Kafka brokers continuously push data to consumers). This implementation was made so that consumers can control the speed at which the topics are being consumed.

9. **Kafka message deserializers**

As we have seen before, the data sent by the Kafka producers is serialized. This means that the data received by the Kafka consumers must be correctly deserialized in order to be useful within your application.Data being consumed must be deserialized in the same format it was serialized in. For example:

- if the producer serialized a `String` using `StringSerializer`, the consumer must deserialize it using `StringDeserializer`
- if the producer serialized an `Integer` using `IntegerSerializer`, the consumer must deserialize it using `IntegerDeserializer`

The serialization and deserialization format of a topic must not change during a topic lifecycle. If you intend to switch a topic data format (for example from JSON to Avro), it is considered best practice to create a new topic and migrate your applications to leverage that new topic.

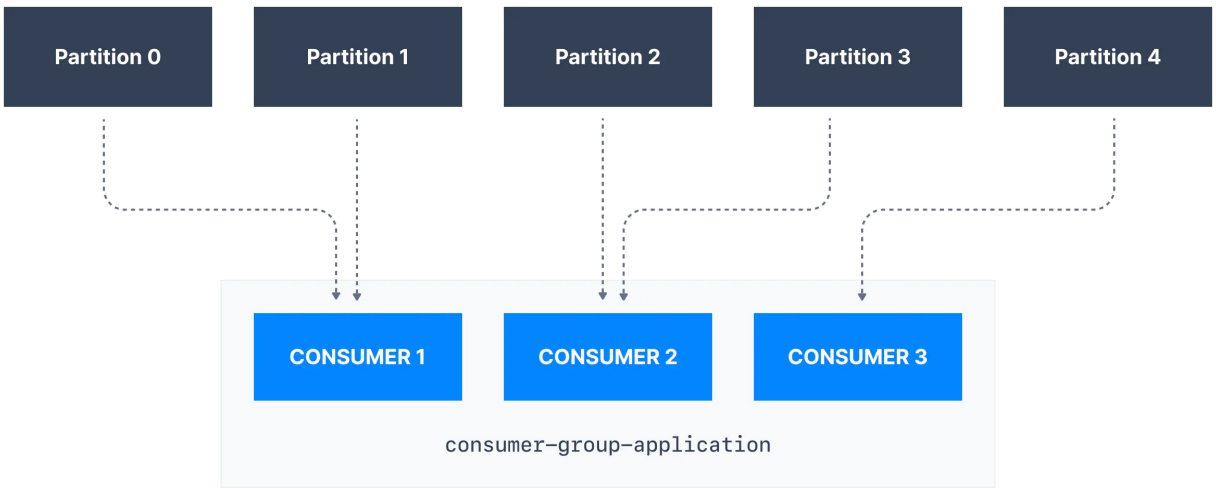
Failure to correctly deserialize may cause crashes or inconsistent data being fed to the downstream processing applications. This can be tough to debug, so it is best to think about it as you're writing your code the first time.

10. **Kafka consumer groups**

    Consumers that are part of the same application and therefore performing the same "logical job" can be grouped together as a Kafka consumer group.

    A topic usually consists of many partitions. These partitions are a unit of parallelism for Kafka consumers.

    The benefit of leveraging a Kafka consumer group is that the consumers within the group will coordinate to split the work of reading from different partitions.



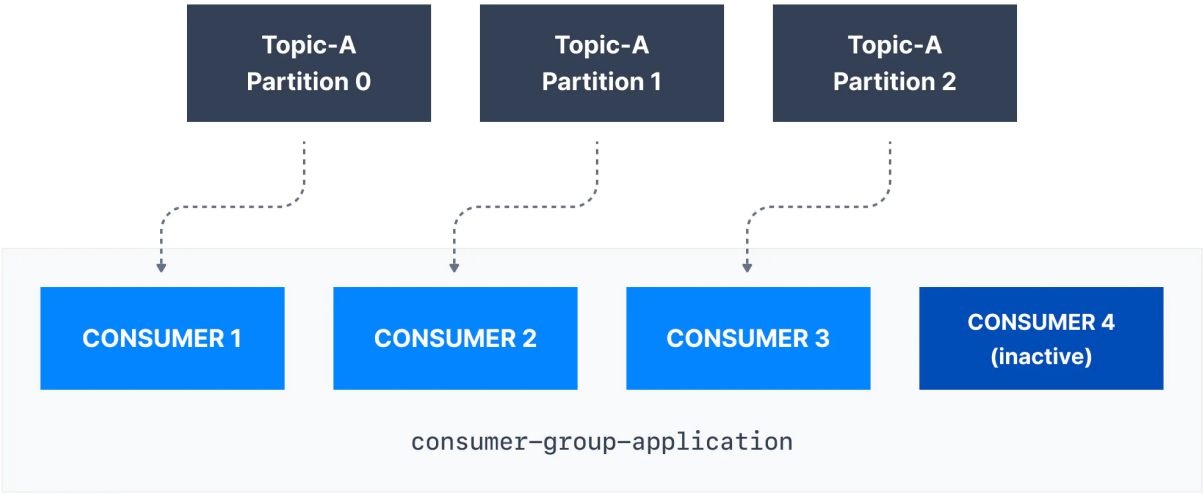
11. **Kafka consumer group id**

    In order for indicating to Kafka consumers that they are part of the same specific group , we must specify the consumer-side setting `group.id`.

    Kafka 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.

    It is important to note that 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.

    If there are more consumers than the number of partitions of a topic, then some consumers will remain inactive as shown below. Usually, we have as many consumers in a consumer group as the number of partitions. If we want more consumers for higher throughput, we should create more partitions while creating the topic. Otherwise, some of the consumers may remain inactive.



12. **Kafka consumer offsets**

    Kafka brokers use an internal topic named `__consumer_offsets` that keeps track of what messages a given **consumer group** last successfully processed.

    As we know, each message in a Kafka topic has a partition ID and an offset ID attached to it.

    Therefore, in order to "checkpoint" how far a consumer has been reading into a topic partition, the consumer will regularly **commit** the latest processed message, also known as **consumer offset**.

    In the figure below, a consumer from the consumer group has consumed messages up to offset `4262`, so the consumer offset is set to `4262`.

    Most client libraries automatically commit offsets to Kafka for you on a periodic basis, and the responsible Kafka broker will ensure writing to the `__consumer_offsets` topic (therefore consumers do not write to that topic directly).

    The process of committing offsets is not done for every message consumed (because this would be inefficient), and instead is a periodic process.

This also means that when a specific offset is committed, all previous messages that have a lower offset are also considered to be committed.

Why use consumer offsets?
Offsets are critical for many applications. If a Kafka client crashes, a rebalance occurs and the latest committed offset help the remaining Kafka consumers know where to restart reading and processing messages.

In case a new consumer is added to a group, another consumer group rebalance happens and consumer offsets are yet again leveraged to notify consumers where to start reading data from.

Therefore consumer offsets must be committed regularly.

13. **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 (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. 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.

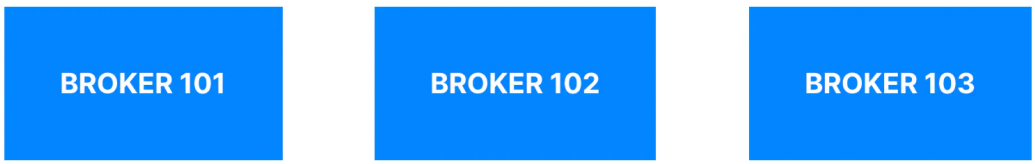
14. **Kafka brokers**

We know that a topic may have more than one partition. Each partition may live on different servers, also known as Kafka brokers.

A single Kafka server is called a Kafka Broker. That Kafka broker is a program that runs on the Java Virtual Machine (Java version 11+) and usually a server that is meant to be a Kafka broker will solely run the necessary program and nothing else.

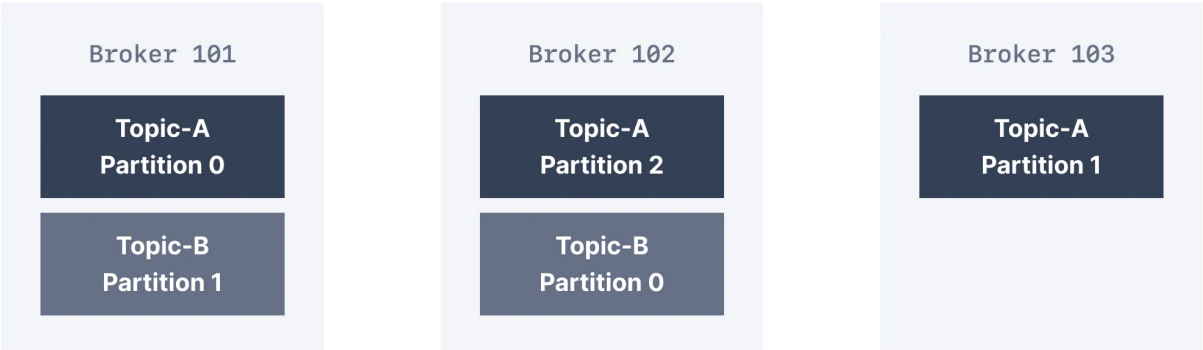
An ensemble of Kafka brokers working together is called a Kafka cluster. Some clusters may contain just one broker or others may contain three or potentially hundreds of brokers. Companies like Netflix and Uber run hundreds or thousands of Kafka brokers to handle their data.

A broker in a cluster is identified by a unique numeric ID. In the figure below, the Kafka cluster is made up of three Kafka brokers.



Kafka brokers store data in a directory on the server disk they run on. Each topic-partition receives its own sub-directory with the associated name of the topic.

To achieve high throughput and scalability on topics, Kafka topics are partitioned. If there are multiple Kafka brokers in a cluster, then partitions for a given topic will be distributed among the brokers evenly, to achieve load balancing and scalability.



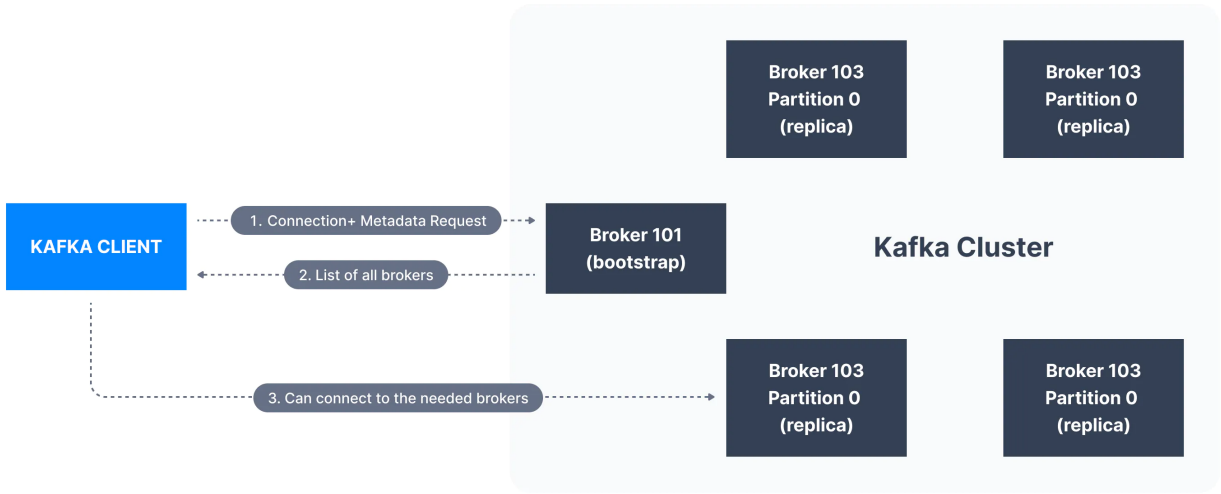
In the diagram above, there are two topics illustrated - *Topic-A* has three partitions. They are distributed evenly among the three available brokers in the cluster. Alternatively, there may be fewer (or more) partitions of a topic than the number of brokers in the cluster. *Topic-B*, in our case, has two partitions only. In this case, *Broker 103* does not contain any partition of *Topic-B*.

There is no relationship between the broker ID and the partition ID - Kafka does a good job of distributing partitions evenly among the available brokers. In case the cluster becomes unbalanced due to an overload of a specific broker, it is possible for Kafka administrators to rebalance the cluster and move partitions.

15. **Clients connnecting to kafka cluster**

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, in the case one of them not being available, the other one should still respond to the connection request. That means that Kafka clients (and developers / DevOps) do not need to be aware of every single hostname of every single broker in a Kafka cluster, but only to be aware and reference two or three in the connection string for clients.

16. **Kafka topic replication**

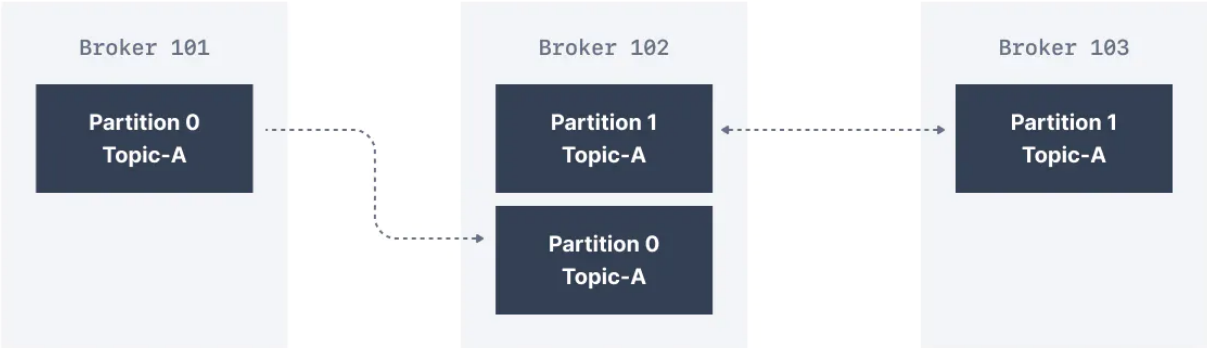
One of the main reasons for Kafka's popularity, is the resilience it offers in the face of broker failures. Machines fail, and often we cannot predict when that is going to happen or prevent it. Kafka is designed with replication as a core feature to withstand these failures while maintaining uptime and data accuracy.

In Kafka, 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.

- A replication factor of `1` means no replication. It is mostly used for development purposes and should be avoided in test and production Kafka clusters
- A replication factor of `3` is a commonly used replication factor as it provides the right balance between broker loss and replication overhead.

In the cluster below consisting of three brokers, the replication factor is `2`. When a message is written down into *Partition 0* of *Topic-A* in *Broker 101*, it is also written down into *Broker 102* because it has *Partition 0* as a replica.



Thanks to a replication factor of 2, we can withstand the failure of one broker. This means that if *Broker 102* failed, as you see below, *Broker 101 & 103* would still have the data.

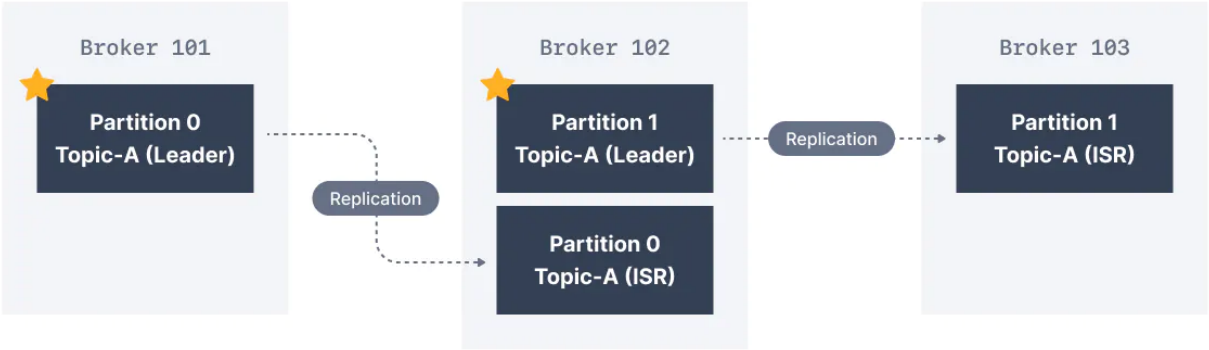
17. **Kafka partition leader & replicas**

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.

18. **In-sync 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.



Here we have *Broker 101* as *Partition 0* leader and *Broker 102* as the leader of *Partition 1*. *Broker 102* is a replica for *Partition 0* and *Broker 103* is a replica for *Partition 1*. If the leader broker were to fail, one of the replicas will be elected as the new partition leader by an election.

19. **Kafka producers ACKS setting**

Kafka producers only write data to the current leader broker for a partition.

Kafka producers 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.

Default value of acks=1 for kafka < v3.0 & acks=all for kafka >= v3.0

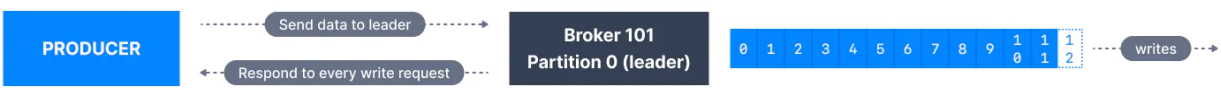
acks=0

When `acks=0` producers consider messages as "written successfully" the moment the message was sent without waiting for the broker to accept it at all.

If the broker goes offline or an exception happens, we won't know and will lose data. This is useful for data where it's okay to potentially lose messages, such as metrics collection, and produces the highest throughput setting because the network overhead is minimized.

acks=1

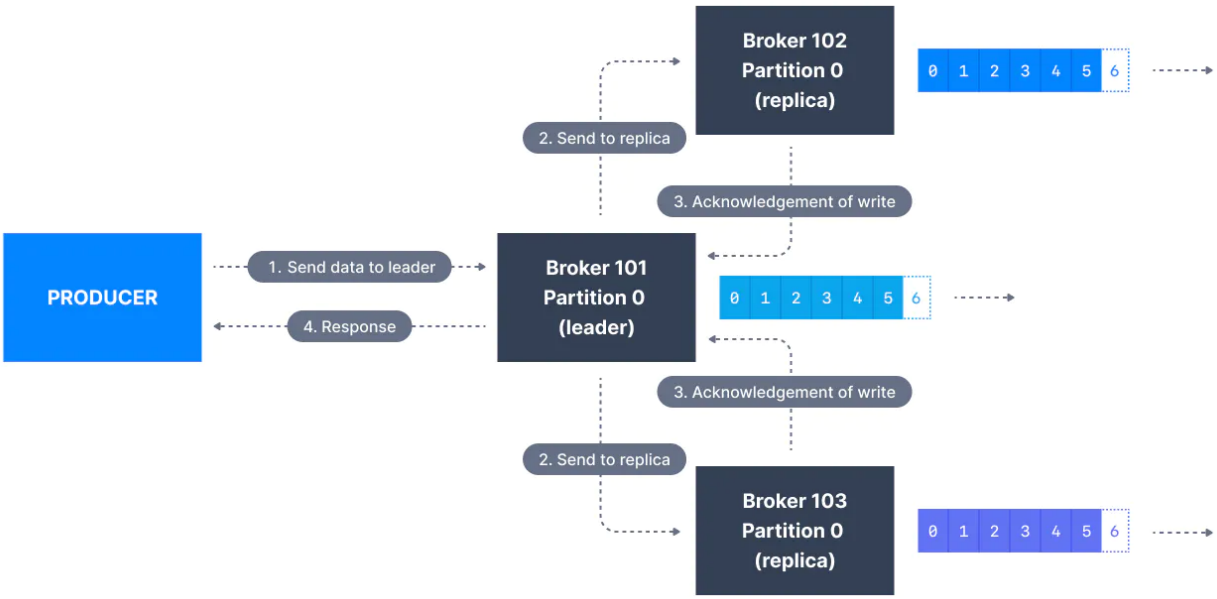
When `acks=1` , producers consider messages as "written successfully" when the message was acknowledged by only the leader.



Leader response is requested, but replication is not a guarantee as it happens in the background. If an ack is not received, the producer may retry the request. If the leader broker goes offline unexpectedly but replicas haven't replicated the data yet, we have a data loss.

acks=all

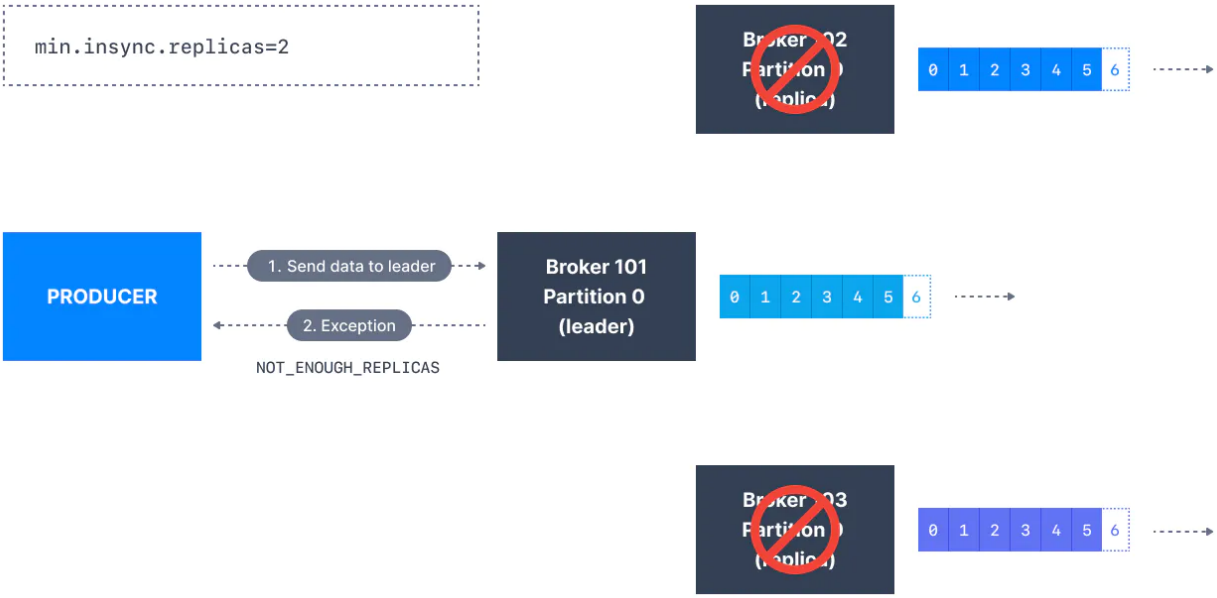
When `acks=all`, producers consider messages as "written successfully" when the message is accepted by all in-sync replicas (ISR).



The lead replica for a partition checks to see if there are enough in-sync replicas for safely writing the message (controlled by the broker setting `min.insync.replicas`). The request will be stored in a buffer until the leader observes that the follower replicas replicated the message, at which point a successful acknowledgement is sent back to the client.

The`min.insync.replicas` can be configured both at the topic and the broker-level. The data is considered committed when it is written to all in-sync replicas - `min.insync.replicas.` A value of 2 implies that at least 2 brokers that are ISR (including leader) must respond that they have the data.

If you would like to be sure that committed data is written to more than one replica, you need to set the minimum number of in-sync replicas to a higher value. If a topic has three replicas and you set `min.insync.replicas` to `2`, then you can only write to a partition in the topic if at least two out of the three replicas are in-sync. When all three replicas are in-sync, everything proceeds normally. This is also true if one of the replicas becomes unavailable. However, if two out of three replicas are not available, the brokers will no longer accept produce requests. Instead, producers that attempt to send data will receive `NotEnoughReplicasException`.



20. **Kafka topic durability & availability**

For a topic replication factor of 3, topic data durability can withstand the loss of 2 brokers. As a general rule, for a replication factor of `N`, you can permanently lose up to `N-1` brokers and still recover your data.

Regarding availability, it is a little bit more complicated... To illustrate, let's consider a replication factor of 3:

- Reads: As long as one partition is up and considered an ISR, the topic will be available for reads
- Writes:
  - `acks=0` & `acks=1` : as long as one partition is up and considered an ISR, the topic will be available for writes.
  - `acks=all`:
    - `min.insync.replicas=1` (default): the topic must have at least 1 partition up as an ISR (that includes the reader) and so we can tolerate two brokers being down
    - `min.insync.replicas=2`: the topic must have at least 2 ISR up, and therefore we can tolerate at most one broker being down (in the case of replication factor of 3), and we have the guarantee that for every write, the data will be at least written twice.
    - `min.insync.replicas=3`: this wouldn't make much sense for a corresponding replication factor of 3 and we couldn't tolerate any broker going down.
    - in summary, when `acks=all` with a `replication.factor=N` and `min.insync.replicas=M` we can tolerate `N-M` brokers going down for topic availability purposes

Note: `acks=all` and `min.insync.replicas=2` is the most popular option for data durability and availability and allows you to withstand at most the loss of **one** Kafka broker

21. **Kafka consumers replicas fetching**

    Kafka consumers read by default from the partition leader. But since Apache Kafka 2.4, it is possible to configure consumers to read from in-sync replicas instead (usually the closest).

    Reading from the closest in-sync replicas (ISR) may improve the request latency, and also decrease network costs, because in most cloud environments cross-data centers network requests incur charges.

22. **Preferred leader**

    The preferred leader is the designated leader broker for a partition at topic creation time (as opposed to being a replica).

    When the preferred leader goes down, any partition that is an ISR (in-sync replica) is eligible to become a new leader (but not a preferred leader). Upon recovering the preferred leader broker and having its partition data back in sync, the preferred leader will regain leadership for that partition.

23. **Zoopker with Kafka**

    Zookeeper is used to track cluster state, membership, and leadership

    How do the Kafka brokers and clients keep track of all the Kafka brokers if there is more than one? The Kafka team decided to use Zookeeper for this purpose.

    Zookeeper is used for metadata management in the Kafka world. For example:

    - Zookeeper keeps track of which brokers are part of the Kafka cluster
    - Zookeeper is used by Kafka brokers to determine which broker is the leader of a given partition and topic and perform leader elections
    - Zookeeper stores configurations for topics and permissions
    - Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc....)

    Important - Zookeeper Being Eliminated from Kafka v4.x:

    1. Kafka 0.x, 1.x & 2.x must use Zookeeper
    2. Kafka 3.x can work without Zookeeper (KIP-500) but is not production ready yet
    3. Kafka 4.x will not have Zookeeper