

Apache Kafka Overview

Date published: 2019-08-22

Date modified: 2024-04-03

CLOUdera

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Kafka Introduction.....	4
Kafka Architecture.....	4
Brokers.....	5
Topics.....	6
Records.....	7
Partitions.....	7
Record order and assignment.....	8
Logs and log segments.....	9
Kafka brokers and Zookeeper.....	10
Leader positions and in-sync replicas.....	11
Kafka stretch clusters.....	14
Kafka stretch cluster requirements.....	14
Kafka stretch cluster architectures.....	15
Cluster configuration for Kafka stretch clusters.....	17
Kafka disaster recovery.....	20
Kafka rack awareness.....	21
Rack awareness for Kafka brokers.....	21
Rack awareness for Kafka consumers.....	25
Rack awareness for Kafka producers.....	26
Kafka KRaft [Technical Preview].....	28
The Raft algorithm.....	31
KRaft metadata management.....	31
KRaft metadata replication and the HWM.....	32
KRaft leader changes.....	34
KRaft log reconciliation.....	38
KRaft broker state machines.....	43
Kafka FAQ.....	43
Basics.....	43
Use cases.....	45

Kafka Introduction

Apache Kafka is a high performance, highly available, and redundant streaming message platform.

Kafka functions much like a publish/subscribe messaging system, but with better throughput, built-in partitioning, replication, and fault tolerance. Kafka is a good solution for large scale message processing applications. It is often used in tandem with Apache Hadoop, and Spark Streaming.

You might think of a log as a time-sorted file or data table. Newer entries are appended to the log over time, from left to right. The log entry number is a convenient replacement for a timestamp.

Kafka integrates this unique abstraction with traditional publish/subscribe messaging concepts (such as producers, consumers, and brokers), parallelism, and enterprise features for improved performance and fault tolerance.

The original use case for Kafka was to track user behavior on websites. Site activity (page views, searches, or other actions users might take) is published to central topics, with one topic per activity type.

Kafka can be used to monitor operational data, aggregating statistics from distributed applications to produce centralized data feeds. It also works well for log aggregation, with low latency and convenient support for multiple data sources.

Kafka provides the following:

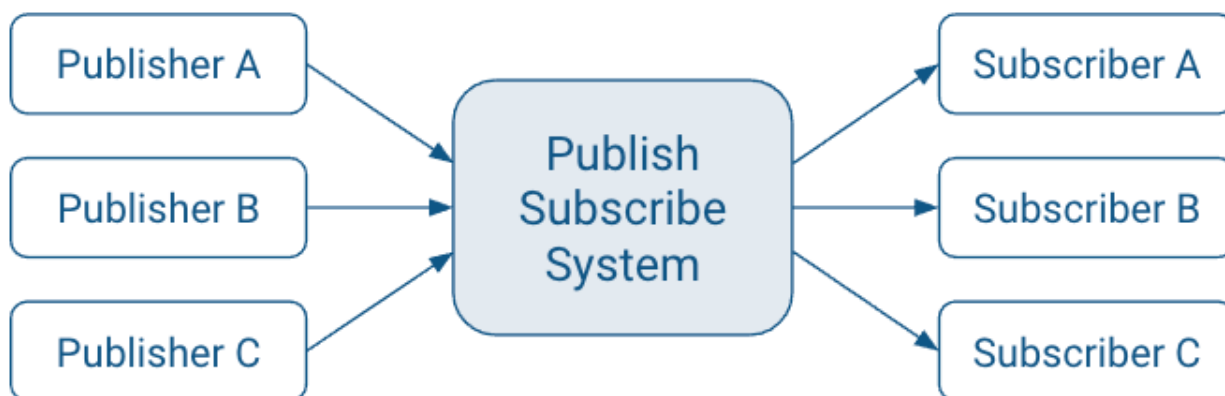
- Persistent messaging with $O(1)$ disk structures, meaning that the execution time of Kafka's algorithms is independent of the size of the input. Execution time is constant, even with terabytes of stored messages.
- High throughput, supporting hundreds of thousands of messages per second, even with modest hardware.
- Explicit support for partitioning messages over Kafka servers. It distributes consumption over a cluster of consumer machines while maintaining the order of the message stream.
- Support for parallel data load into Hadoop.

Kafka Architecture

Learn about Kafka's architecture and how it compares to an ideal publish-subscribe system.

The ideal publish-subscribe system is straightforward: Publisher A's messages must make their way to Subscriber A, Publisher B's messages must make their way to Subscriber B, and so on.

Figure 1: Ideal Publish-Subscribe System



An ideal system has the benefit of:

- Unlimited Lookback. A new Subscriber A1 can read Publisher A's stream at any point in time.

- **Message Retention.** No messages are lost.
- **Unlimited Storage.** The publish-subscribe system has unlimited storage of messages.
- **No Downtime.** The publish-subscribe system is never down.
- **Unlimited Scaling.** The publish-subscribe system can handle any number of publishers and/or subscribers with constant message delivery latency.

Kafka's architecture however deviates from this ideal system. Some of the key differences are:

- Messaging is implemented on top of a replicated, distributed commit log.
- The client has more functionality and, therefore, more responsibility.
- Messaging is optimized for batches instead of individual messages.
- Messages are retained even after they are consumed; they can be consumed again.

The results of these design decisions are:

- Extreme horizontal scalability
- Very high throughput
- High availability
- Different semantics and message delivery guarantees

Kafka Terminology

Kafka uses its own terminology when it comes to its basic building blocks and key concepts. The usage of these terms might vary from other technologies. The following provides a list and definition of the most important concepts of Kafka:

Broker

A broker is a server that stores messages sent to the topics and serves consumer requests.

Topic

A topic is a queue of messages written by one or more producers and read by one or more consumers.

Producer

A producer is an external process that sends records to a Kafka topic.

Consumer

A consumer is an external process that receives topic streams from a Kafka cluster.

Client

Client is a term used to refer to either producers and consumers.

Record

A record is a publish-subscribe message. A record consists of a key/value pair and metadata including a timestamp.

Partition

Kafka divides records into partitions. Partitions can be thought of as a subset of all the records for a topic.

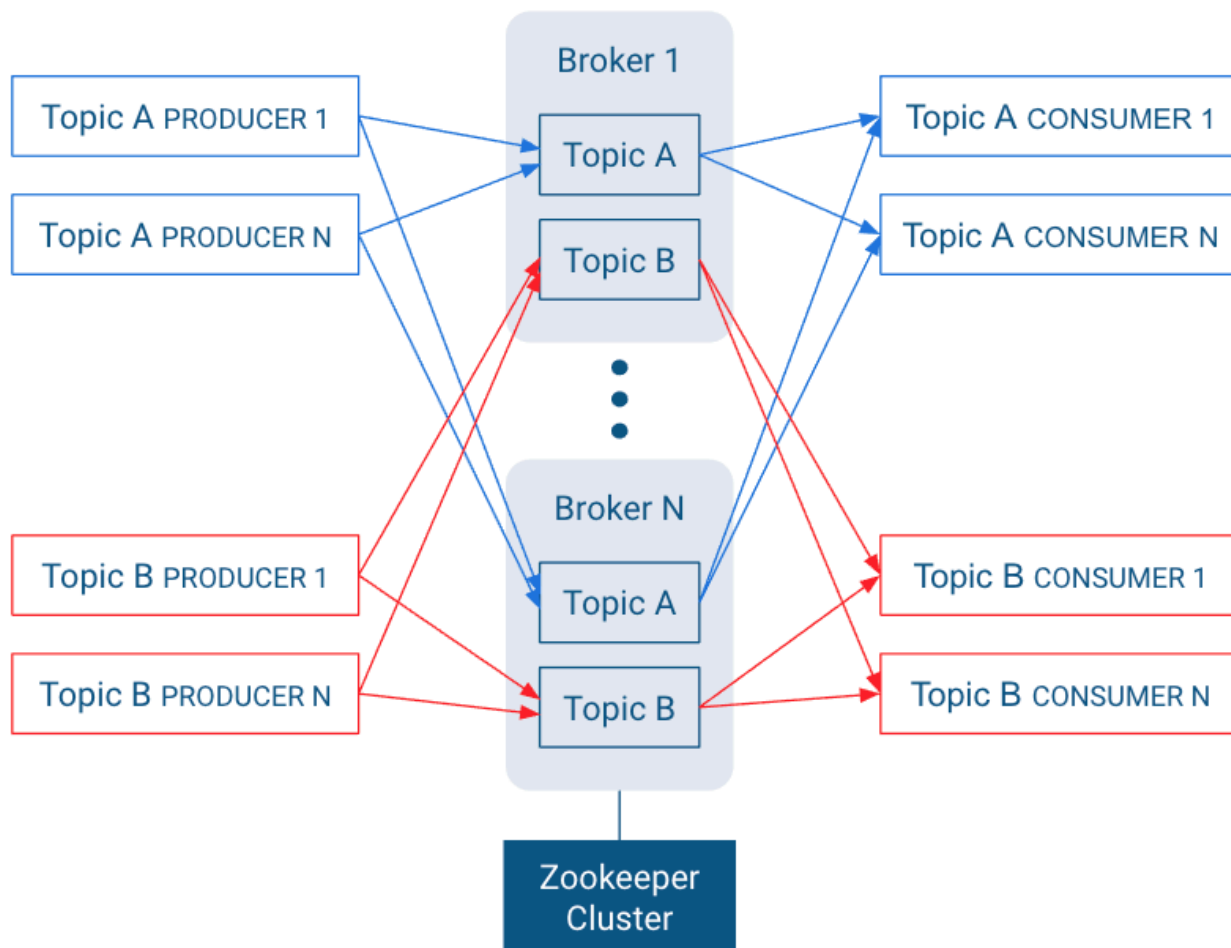
Continue reading to learn more about each key concept.

Brokers

Learn more about Brokers.

Kafka is a distributed system that implements the basic features of an ideal publish-subscribe system. Each host in the Kafka cluster runs a server called a broker that stores messages sent to the topics and serves consumer requests.

Figure 2: Brokers in a Publish-Subscribe System



Kafka is designed to run on multiple hosts, with one broker per host. If a host goes offline, Kafka does its best to ensure that the other hosts continue running. This solves part of the “No Downtime” and “Unlimited Scaling” goals of the ideal publish-subscribe system.

Kafka brokers all talk to Zookeeper for distributed coordination, which also plays a key role in achieving the “Unlimited Scaling” goal from the ideal system.

Topics are replicated across brokers. Replication is an important part of “No Downtime”, “Unlimited Scaling,” and “Message Retention” goals.

There is one broker that is responsible for coordinating the cluster. That broker is called the controller.

Topics

Learn more about Kafka topics.

In any publish-subscribe system, messages from one publisher, called producers in Kafka, have to find their way to the subscribers, called consumers in Kafka. To achieve this, Kafka introduces the concept of topics, which allow for easy matching between producers and consumers.

A topic is a queue of messages that share similar characteristics. For example, a topic might consist of instant messages from social media or navigation information for users on a web site. Topics are written by one or more

producers and read by one or more consumers. A topic is identified by its name. This name is part of a global namespace of that Kafka cluster.

As each producer or consumer connects to the publish-subscribe system, it can read from or write to a specific topic.

Figure 3: Topics in a Publish-Subscribe System



Records

Learn more about Kafka records.

In Kafka, a publish-subscribe message is called a record. A record consists of a key/value pair and metadata including a timestamp. The key is not required, but can be used to identify messages from the same data source. Kafka stores keys and values as arrays of bytes. It does not otherwise care about the format.

The metadata of each record can include headers. Headers may store application-specific metadata as key-value pairs. In the context of the header, keys are strings and values are byte arrays.

For specific details of the record format, see Apache Kafka documentation.

Related Information

[Record Format](#)

Partitions

Learn more about Kafka partitions.

Instead of all records handled by the system being stored in a single log, Kafka divides records into partitions. Partitions can be thought of as a subset of all the records for a topic. Partitions help with the ideal of “Unlimited Scaling”.

Records in the same partition are stored in order of arrival.

When a topic is created, it is configured with two properties:

partition count

The number of partitions that records for this topic will be spread among.

replication factor

The number of copies of a partition that are maintained to ensure consumers always have access to the queue of records for a given topic.

Each topic has one leader partition. If the replication factor is greater than one, there will be additional follower partitions. (For the replication factor = M , there will be $M-1$ follower partitions.)

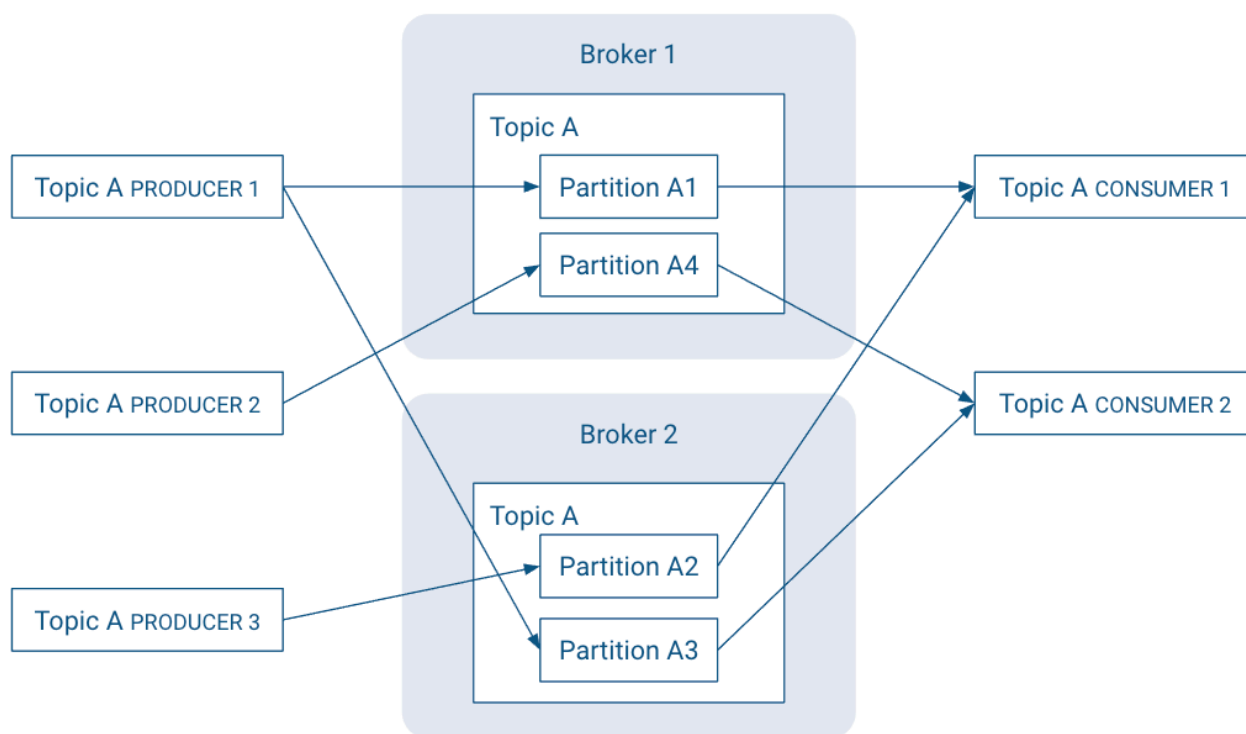
Any Kafka client (a producer or consumer) communicates only with the leader partition for data. All other partitions exist for redundancy and failover. Follower partitions are responsible for copying new records from their leader partitions. Ideally, the follower partitions have an exact copy of the contents of the leader. Such partitions are called in-sync replicas (ISR).

With N brokers and topic replication factor M , then

- If $M < N$, each broker will have a subset of all the partitions
- If $M = N$, each broker will have a complete copy of the partitions

In the following illustration, there are $N = 2$ brokers and $M = 2$ replication factor. Each producer may generate records that are assigned across multiple partitions.

Figure 4: Records in a Topic are Stored in Partitions, Partitions are Replicated across Brokers



Partitions are the key to keeping good record throughput. Choosing the correct number of partitions and partition replications for a topic:

- Spreads leader partitions evenly on brokers throughout the cluster
- Makes partitions within the same topic are roughly the same size
- Balances the load on brokers.

Record order and assignment

Learn about how Kafka assigns records to partitions.

By default, Kafka assigns records to a partitions round-robin. There is no guarantee that records sent to multiple partitions will retain the order in which they were produced. Within a single consumer, your program will only have record ordering within the records belonging to the same partition. This tends to be sufficient for many use cases, but does add some complexity to the stream processing logic.



Tip: Kafka guarantees that records in the same partition will be in the same order in all replicas of that partition.

If the order of records is important, the producer can ensure that records are sent to the same partition. The producer can include metadata in the record to override the default assignment in one of two ways:

- The record can indicate a specific partition.
- The record can include an assignment key.

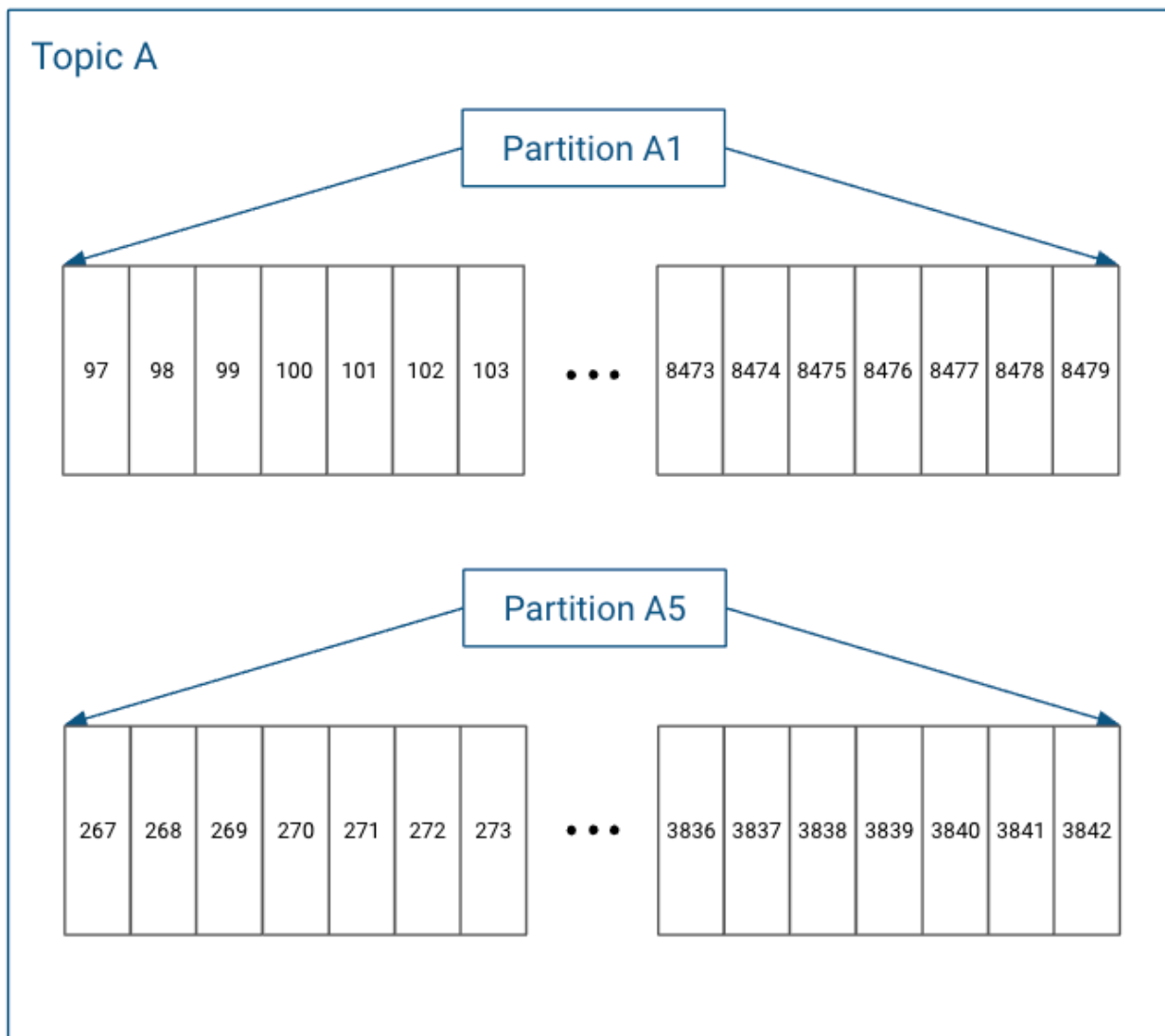
The hash of the key and the number of partitions in the topic determines which partition the record is assigned to. Including the same key in multiple records ensures all the records are appended to the same partition.

Logs and log segments

Learn more about logs and log segments.

Within each topic, each partition in Kafka stores records in a [log structured format](#). Conceptually, each record is stored sequentially in this type of “log”.

Figure 5: Partitions in Log Structured Format



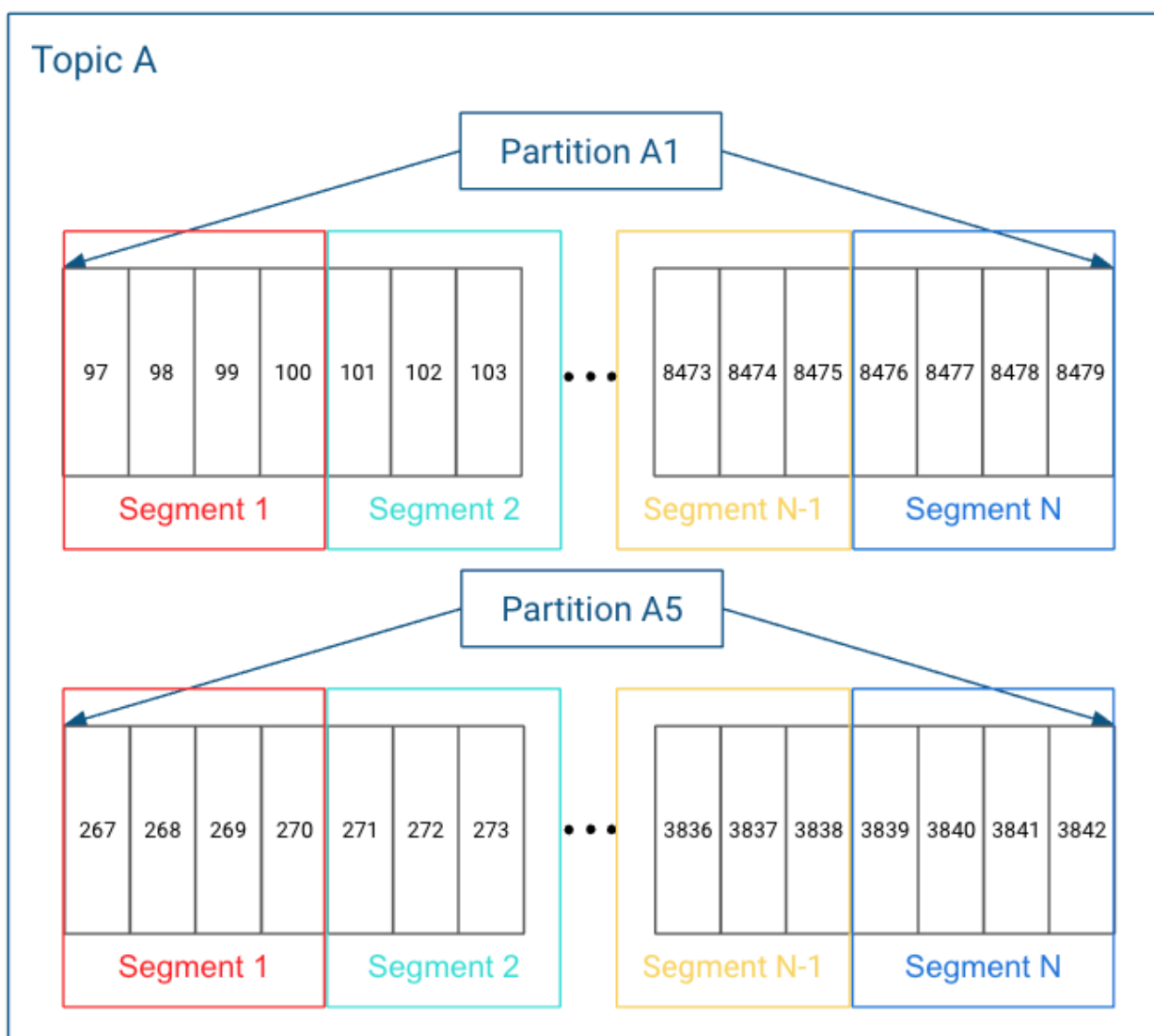


Note: These references to “log” should not be confused with where the Kafka broker stores their operational logs.

In actuality, each partition does not keep all the records sequentially in a single file. Instead, it breaks each log into log segments. Log segments can be defined using a size limit (for example, 1 GB), as a time limit (for example, 1 day), or both. Administration around Kafka records often occurs at the log segment level.

Each of the partitions is broken into segments, with Segment N containing the most recent records and Segment 1 containing the oldest retained records. This is configurable on a per-topic basis.

Figure 6: Partition Log Segments



Related Information

[Log-structured file system](#)

Kafka brokers and Zookeeper

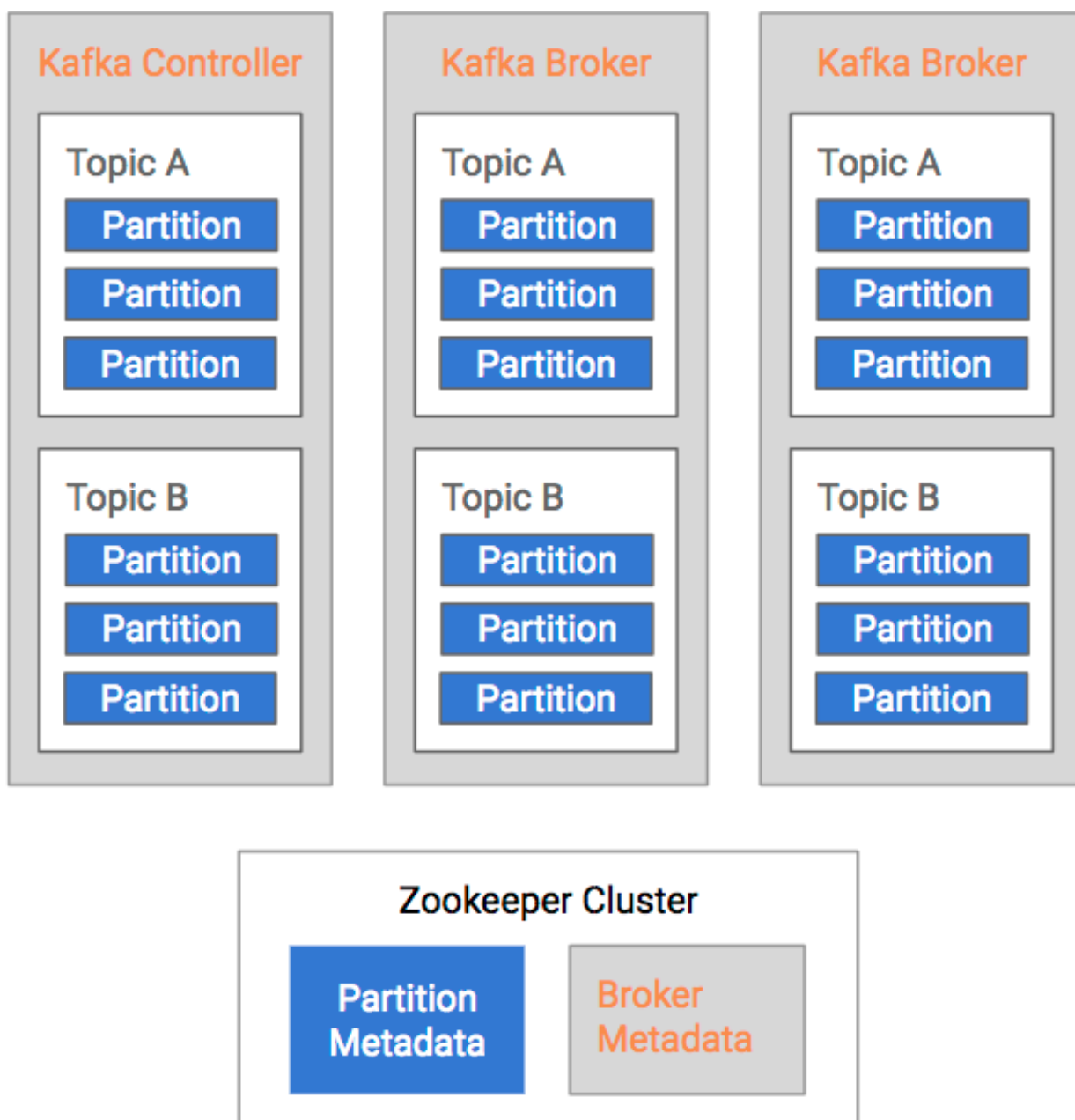
Learn about the types of data maintained in Zookeeper by the brokers.

The broker, topic, and partition information are maintained in Zookeeper. In particular, the partition information, including partition and replica locations, updates fairly frequently. Because of frequent metadata refreshes, the

connection between the brokers and the Zookeeper cluster needs to be reliable. Similarly, if the Zookeeper cluster has other intensive processes running on it, that can add sufficient latency to the broker/Zookeeper interactions to cause issues.

- Kafka Controller maintains leadership through Zookeeper (shown in orange)
- Kafka Brokers also store other relevant metadata in Zookeeper (also in orange)
- Kafka Partitions maintain replica information in Zookeeper (shown in blue)

Figure 7: Broker/ZooKeeper Dependencies

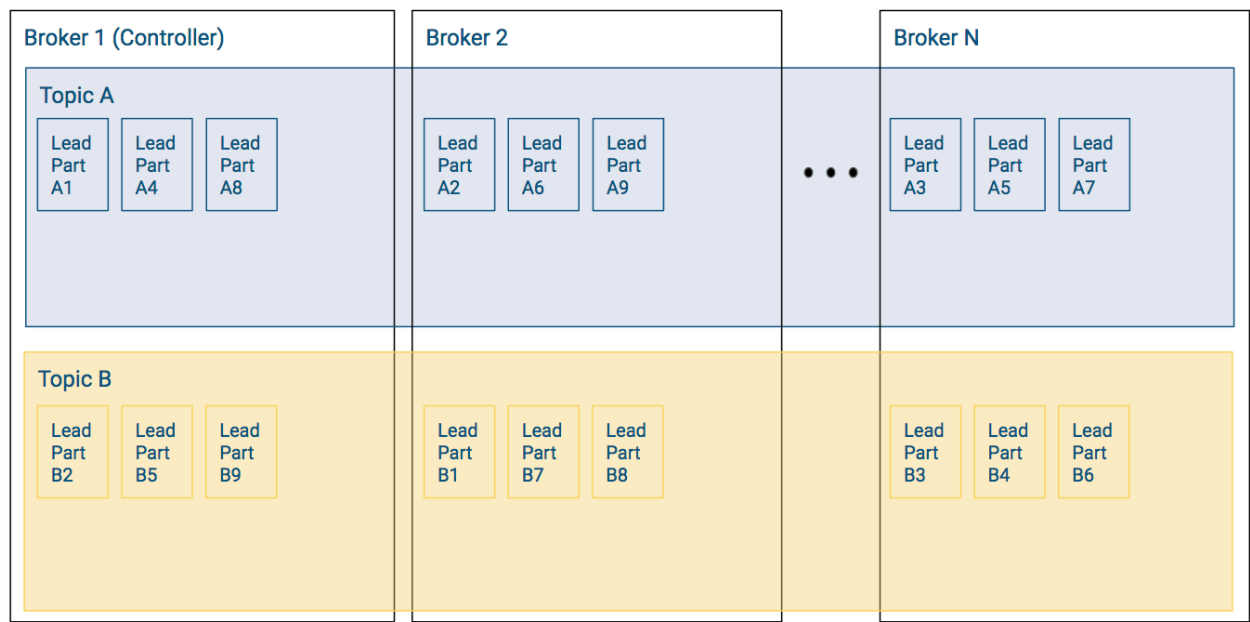


Leader positions and in-sync replicas

An overview of how leader positions and in-sync replicas can affect Kafka performance.

Consider the following example which shows a simplified version of a Kafka cluster in steady state. There are N brokers, two topics with nine partitions each. Replicated partitions are not shown for simplicity.

Figure 8: Kafka Cluster in Steady State



In this example, each broker shown has three partitions per topic and the Kafka cluster has well balanced leader partitions. Recall the following:

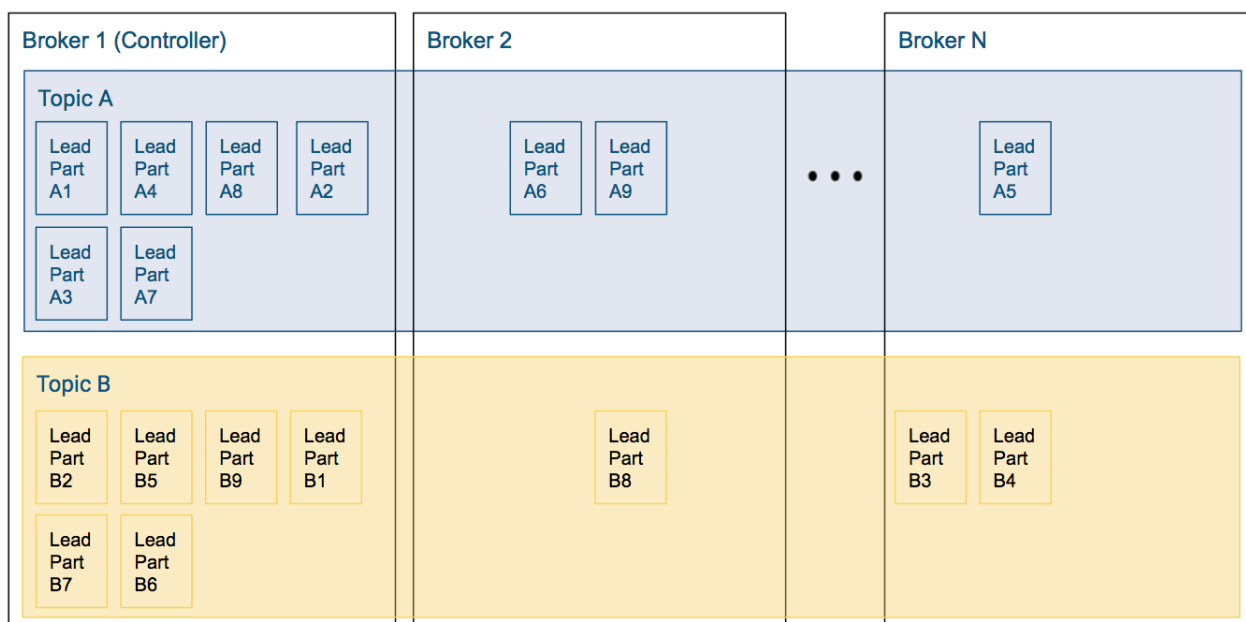
- Producer writes and consumer reads occur at the partition level.
- Leader partitions are responsible for ensuring that the follower partitions keep their records in sync.

Since the leader partitions are evenly distributed, most of the time the load to the overall Kafka cluster is relatively balanced.

Leader Positions

Now lets look at an example where a large chunk of the leaders for Topic A and Topic B are on Broker 1.

Figure 9: Kafka Cluster with Leader Partition Imbalance



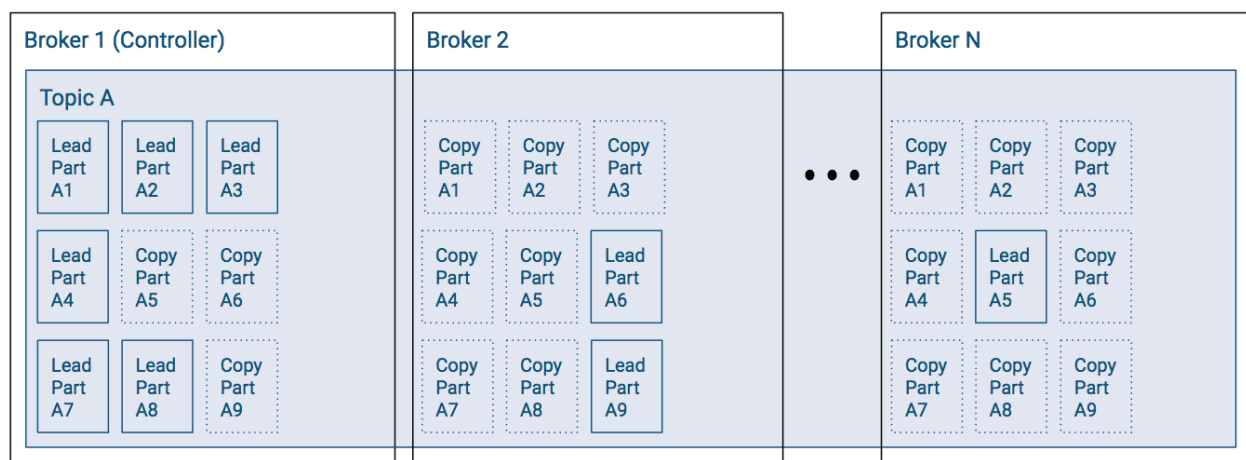
In a scenario like this a lot more of the overall Kafka workload occurs on Broker 1. Consequently this also causes a backlog of work, which slows down the cluster throughput, which will worsen the backlog. Even if a cluster starts with perfectly balanced topics, failures of brokers can cause these imbalances: if the leader of a partition goes down one of the replicas will become the leader. When the original (preferred) leader comes back, it will get back leadership only if automatic leader rebalancing is enabled; otherwise the node will become a replica and the cluster gets imbalanced.

In-Sync Replicas

Let's take a closer look at Topic A from the previous example that had imbalanced leader partitions. However, this time let's visualize follower partitions as well:

- Broker 1 has six leader partitions, broker 2 has two leader partitions, and broker 3 has one leader partition.
- Assuming a replication factor of 3.

Figure 10: Kafka Topic with Leader and Follower Partitions



Assuming all replicas are in-sync, then any leader partition can be moved from Broker 1 to another broker without issue. However, in the case where some of the follower partitions have not caught up, then the ability to change leaders or have a leader election will be hampered.

Kafka stretch clusters

Stretch clusters are highly resilient and cost-effective Kafka deployments. Learn the requirements, possible deployment architectures, and configuration needed to deploy Kafka stretch clusters.

A stretch cluster is a single logical Kafka cluster deployed across multiple Data Centers (DC) or other independent physical infrastructures such as cloud availability zones. By utilizing a single Kafka cluster over multiple DCs, you can achieve the following:

- Strong data durability guarantees as a result of synchronous replication between DCs.
- Automatic failover of clients when a DC goes down.

Stretch clusters have specific requirements for latency between DCs and require specific configurations for achieving strong guarantees on data durability. Additionally, in some use cases, complimentary features can be used to minimize cross-DC traffic. It is important to note that the extra latency introduced by this setup can have a significant impact on the throughput of Kafka.

Kafka stretch cluster requirements

Learn about the requirements of a stretch cluster deployment.

Number of Data Centers and the ZooKeeper quorum

A stretch cluster requires at least three DCs to function properly. While Kafka can support a setup where brokers are hosted in two DCs, Kafka is dependent on ZooKeeper for metadata storage. Theoretically, it is possible to use more than three DCs in the stretch cluster architecture. However, in the majority of cases, the cost of the additional cross-DC data traffic will outweigh the benefits of having the data copied to more than three DCs.

Since ZooKeeper is a quorum-based system, a majority is required to keep the metadata storage and Kafka running. To support service continuity in case of a DC failure, sufficient ZooKeeper nodes must remain so that a majority vote can happen. For example, in a two DC setup, one of the two DCs must host more ZooKeeper nodes than the other. If the DC that hosts the higher number of ZooKeeper nodes goes down, ZooKeeper majority is lost. This results in the Kafka cluster also going down.



Note: KIP-500 introduced a Raft based quorum system (KRaft) that acts as an alternative to ZooKeeper.

While this document discusses the use of Zookeeper, the requirements and architecture of a stretch cluster are the similar for deployments that use Kraft. This is because Kraft is also a quorum based system.

Latency between Data Centers

In order to achieve high throughput, Kafka is designed from the ground up with the assumption that latency is low between the cluster nodes. Because of this, a stretch cluster deployment requires low latency between the DCs. There are two major issues with high latency in a stretch cluster: Kafka replication throughput and metadata operation latency.

While the replication stays functionally correct even with high latency, the throughput of Kafka suffers greatly from increased latency.

ZooKeeper is also sensitive to latency. ZooKeeper changes are synchronized in the cluster, making metadata changes run slow. These changes involve controller election, topic metadata related operations (create/delete/update config), and ISR changes. In an ideal deployment with low latency, these are infrequent operations and do not affect the throughput of data production and consumption. However, the higher the latency, the more frequent these operation become. With increased latency, replicas lag behind the ISR more easily, which generates more ISR change operations. There is also a phenomenon called ISR thrashing. This occurs when a replica frequently joins but then lags behind the ISR, which is also tied to high latency scenarios.

Because of these issues, Cloudera recommends using infrastructure where the maximum latency between the DCs is 50 ms. In general, the latency should be minimized. Any increase in latency greatly affects the throughput of a Kafka stretch cluster. While it is possible for a Kafka stretch cluster to function correctly in some use cases with higher latency (for example, light duty clusters), Cloudera does not recommend using the stretch cluster architecture if you have high latency.

Related Information

[KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum](#)

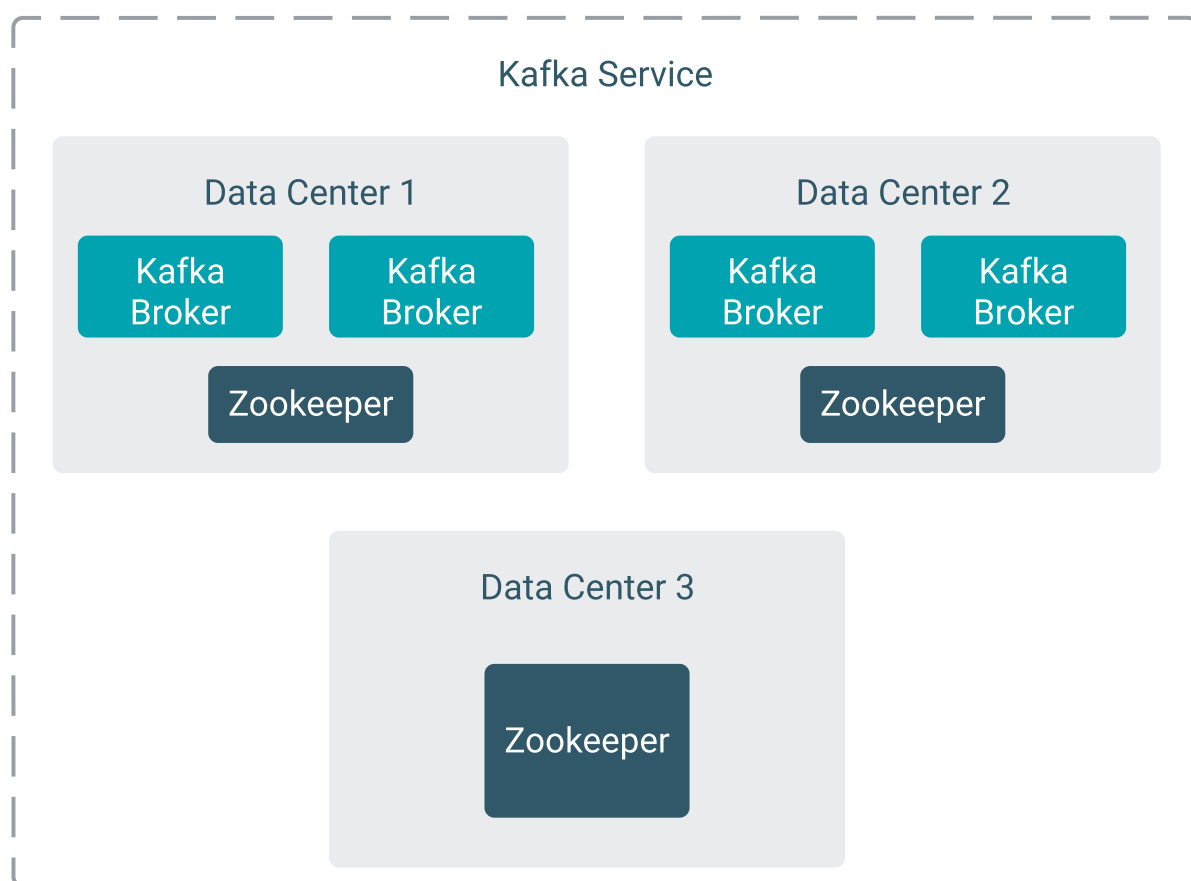
Kafka stretch cluster architectures

A Kafka stretch cluster can be operated in a 3 or 2.5 DC configuration.

A stretch cluster requires at least three DCs to function correctly. Based on the use case, it is possible to span the Kafka cluster over two DCs, and only use the third cluster as a tie-breaker ZooKeeper node. This architecture is called the 2.5 DC setup. Typical use cases would either use the 2.5 DC setup, or a 3 DC setup. It is possible to use multiple DCs (given that the latency requirements are met), but in general, cross-DC traffic should be minimized.

2.5 DC stretch cluster

Figure 11: 2.5 DC Kafka stretch cluster



Pros:

- Cost efficient (both in terms of nodes used and cross-DC data traffic).
- Can ensure synchronized writes into two DCs (data durability, RPO=0).

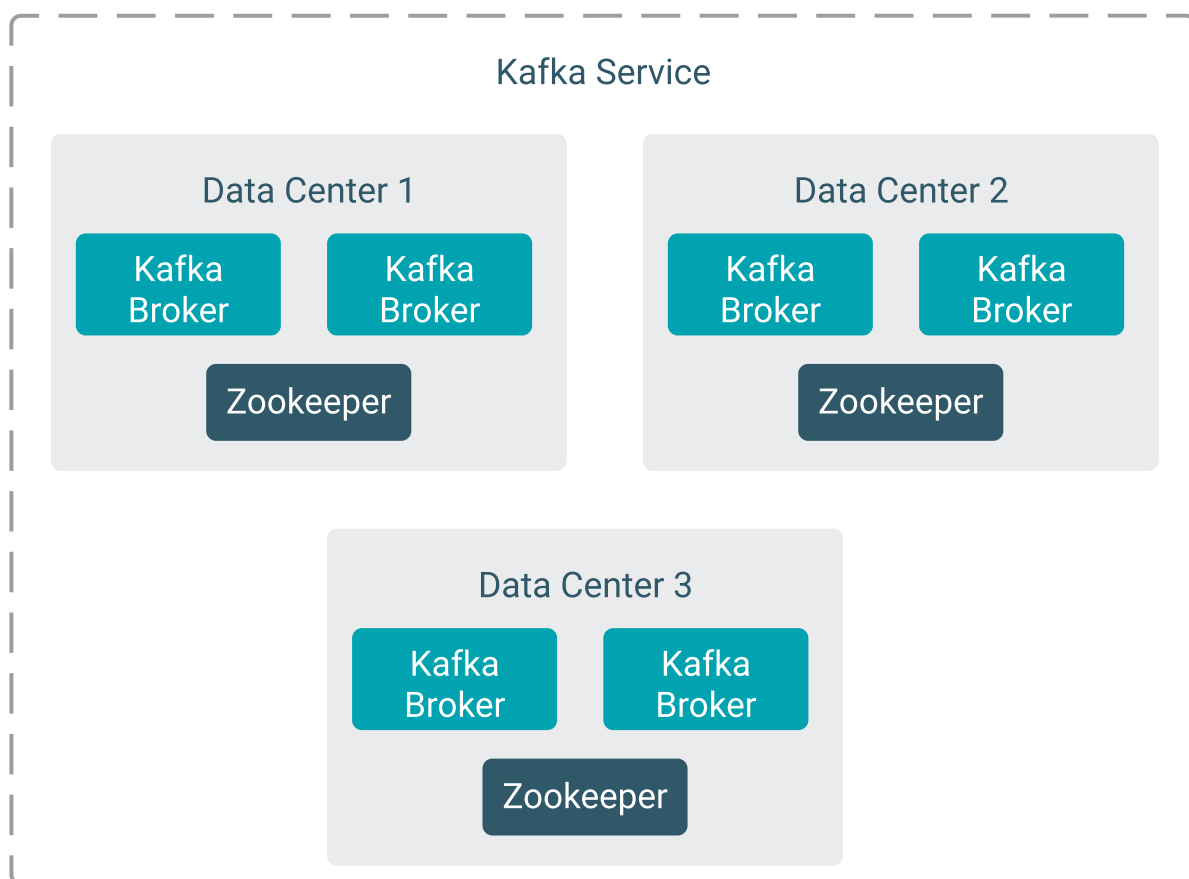
- Can tolerate single DC failure, reads supported.
- Less sensitive to the tie-breaker DC failure.

Con:

- Writes for durable topics are not supported after a DC that hosts brokers goes down.

3 DC stretch cluster

Figure 12: 3 DC stretch cluster



Pros:

- Can ensure synchronized writes into at least two DCs (data durability, RPO=0).
 - Depending on the configuration, can guarantee writes over three DCs, but write availability is reduced.
- Can tolerate a single DC going down. This is true for both reads and writes.
 - Write availability depends on the configuration (whether two or three DC guarantee was configured).

Con:

- As a result of more nodes in all DCs and more cross-DC data traffic, this architecture is more expensive than a 2.5 DC setup.

Cluster configuration for Kafka stretch clusters

Kafka stretch clusters require the brokers, topics, and clients of the deployment to be configured. Configuration is needed to achieve the desired guarantees on data durability.

To achieve the high data durability in a stretch cluster, configuring the brokers, topics and clients in the deployment is required. Specifically you have to:

- Configure brokers and topics so that topic partition replicas are distributed evenly among the DCs.
- Configure producers to write messages with the highest durability guarantee (acks=all).
- Configure topics to have the required minimum in-sync replicas when accepting writes.

Even distribution of topic replicas among DCs

To ensure that replicas are distributed evenly among the DCs, stretch clusters use Kafka's rack awareness feature (*KIP-36*). Kafka brokers have a `broker.rack` configuration. This property specifies the physical location of the broker in the form of a rack ID. The rack ID is an arbitrary, user defined string that can represent any type of physical infrastructure. In stretch clusters, the property and the ID can be used to let the broker know which DC it is running in. For example, the ID can be set to `DC1` for the brokers inside `DC1`, `DC2` for `DC2`, and so on.

If a topic is created with a replication factor of two or more in a Kafka instance that has its rack IDs specified, Kafka assigns the replicas in a way that not only ensures that the replicas of the same partition are located in different brokers, but will also try to distribute them evenly among the DCs (using round-robin).

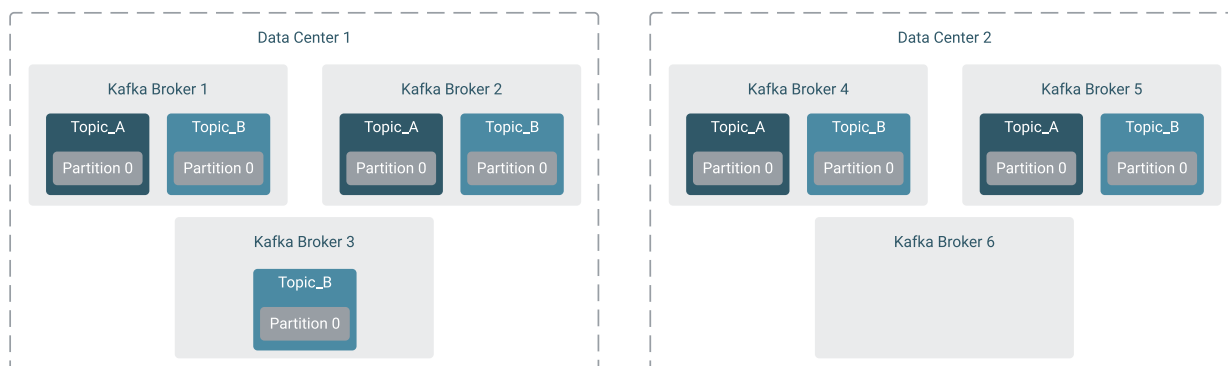
For example, assume you have two DCs ("racks") with six brokers and two topics. The replication factor of the topics is not uniform. `Topic_A` has a replication factor of 4, `Topic_B` has a replication factor of 5. In this scenario, Kafka will create the following replica assignments:

```
Topic_A, RF=4 partition 0: [1, 4, 2, 5]
Topic_B, RF=5 partition 0: [1, 4, 2, 5, 3]
```

For `Topic_A` there are two replicas in `DC1` and two replicas in `DC2`. `Topic_A` is evenly distributed. This happens if $[***RF***] \% [***DC\ COUNT***] == 0$.

For `Topic_B` there are three replicas in `DC1` and two replicas in `DC2`. Since $[***RF***] \% [***DC\ COUNT***] != 0$, distribution is not perfectly even, but round-robin ensures that the maximum difference between replica counts per rack is one.

Figure 13: Kafka stretch cluster replica assignment example



To utilize this behavior to your advantage, durable topics should be configured to have a replication factor which is a multiple of the number of DCs. For example, if you have three DCs, set the replication factor to 3, 6, 9, and so on.

In the majority of cases, having a single replica inside a DC is sufficient. That is, setting the replication factor higher than the number of DCs is usually not necessary. Doing so increases cross-DC traffic and storage capacity requirements. However, having more replicas does have benefits, which are as follows:

1. Increased availability. More broker failures can be tolerated without the topic being unavailable.
2. Follower fetching is supported even in case of single broker failures.
3. Increased data durability in case of disk failures.



Note: Increased data durability can also be achieved with fault tolerant disk technologies or services.

Producer acknowledgments (acks)

Producers have a configuration property called acks. The acks property affects when a leader should send an acknowledgement to the producer after a write operation. The property can be set to three different values, each corresponding to a level of acknowledgement. Each level represents a tradeoff between throughput and write durability. The configuration values are as follows:

0

No acknowledgement. The producer handles the write as finished right after it is sent to the leader. This setting provides the highest throughput, but also has the highest risk of losing writes. Writes might not even make it to the leader log (and disk) before the leader goes down.

1

Leader acknowledgement. The leader sends an acknowledgement right after the message was appended to its log. This setting provides lower throughput compared to acks=0, but ensures that at least a single log has the message appended.

all

ISR acknowledgement. The leader appends the message to its log, waits for the in-sync replicas (ISR) to replicate the message, and then sends the acknowledgment. This setting provides the lowest throughput. This is because the leader might need to wait for multiple followers to fetch and write the message to their log before the acknowledgment can be sent out. However, it has the highest durability as it ensures that multiple brokers have appended the message to their log. With this setting, the decision on when a write should be considered done is handed over from the producer to the partition leader.

For high durability, producers must be configured to use acks=all. To understand and fully utilize the ISR concept in the stretch cluster, you need to also correctly configure the topics.

ISR and minimum in-sync replicas of a topic

For durable writes, besides the replication factor, it is also important to correctly configure the minimum in-sync replica count (min.insync.replicas) of a topic.

The ISR is tracked by Kafka based on broker configuration. The leader always keeps track which of its followers are in sync (caught up), and which are lagging behind. The replicas that are caught up are part of the ISR.

When a producer sends a request with acks=all, the leader will wait for **all members of the ISR** to replicate the message before sending an acknowledgment. If there are nine replicas (one leader and eight followers), and all are part of the ISR, then the acknowledgment will be sent after all eight followers have replicated the message. By default, if only the leader is part of the ISR, then the acknowledgment is sent right after the leader appends the message to its log.

To change this behavior, topics have a min.insync.replicas property. This property represents a minimum size requirement for the ISR when a producer writes into a topic with acks=all. This means that if min.insync.replicas=2, then the leader and at least a single follower must be part of the ISR and that the ISR must replicate the message before the acknowledgment is sent.

In a stretch cluster setup, if you want to ensure that writes are replicated to multiple DCs before acknowledgment happens, the min.insync.replicas property must be correctly configured. The following example demonstrates how you can calculate the correct min.insync.replicas value for your deployment.

Assume that you have three DCs. You want to replicate writes into at least two DCs before a write is acknowledged. Replication factor is six. That is:

```
[***DC COUNT***]= 3
[***MINIMUM DC REPLICAS***]= 2
[***RF***]= 6
```

First, you need to calculate how many replicas reside in each DC, *[***REPLICA PER DC***]*. This can be done by dividing the replication factor by the number of DCs:

```
[***RF***]/[***DC COUNT***]=[***REPLICA PER DC***]
```

```
6 / 3 = 2
```

Afterwards, you can calculate `min.insync.replicas` using the following formula:

```
[***REPLICA PER DC***] * ([***MINIMUM DC REPLICAS***] - 1) + 1 = min.insync.replicas
```

```
2 * (2 - 1) + 1 = 3
```

This formula ensures that whichever replicas are in sync for the topic, there will always be at least a *[***MINIMUM DC REPLICAS***]* number of DCs hosting the active replicas. However, whenever you have fewer replicas in the ISR, writes will start to fail because the `min.insync.replica` requirement is not met.

With `min.insync.replicas=3` you can ensure that even in the worst case scenario (most of the replicas in the ISR are located in the same DC), at least one replica will be located in a different DC.

To look at another example, assume you have the same setup, but change *[***MINIMUM DC REPLICAS***]* to three, `min.insync.replicas` would change to five.

```
2 * (3 - 1) + 1 = 5
```

With `min.insync.replicas=5` you can ensure that even in a worst case scenario of ISR members, all three DCs are replicating the write before it is acknowledged. However, at the same time, this means that any DC going down reduces the ISR size to four, which will cause the cluster to fail durable produce requests.

Partition leadership

Everything described so far about the ISR and durable writes depends on the fact that partition leadership changes depend on the ISR. When the leader is not available, Kafka transfers the leadership to one of the ISR members. This ensures that all writes acknowledged by the cluster will be present on the next leader. Because of this, all durable topics must have `unclean.leader.election.disabled`. Otherwise, accepted writes might get lost in an unclean leader election.

Conclusion: Durable writes (RPO=0)

With the three essential configurations done (`broker.rack`, `acks=all`, `min.insync.replicas`), you ensure that:

- Replicas are evenly distributed among DCs.
- Producers write with the highest durability.
- The level of durability configured with `min.insync.replicas` ensures that writes are synchronized to the required number of DCs.
- Topics only allow clean leader elections (based on the ISR).

Related Information

[KIP-36 Rack aware replica assignment](#)

Kafka disaster recovery

Learn about the cluster architectures you can use when designing highly available and resilient Kafka clusters.

Kafka has built-in replication for its topics and allows users to carefully tweak the data durability configurations to achieve desired redundancy guarantees. When designing a deployment where data is replicated over multiple Data Centers (DC) and has disaster recovery (DR) options, you need to carefully analyse your data durability and business continuity requirements. The following sections introduce the architectural options that you can choose from when building a resilient multi-DC Kafka deployment. Additionally, guidance is provided that help you in choosing the right architecture for your use case.

There are two major architectural groups for achieving multi-DC DR with Kafka. These are as follows:

Stretch clusters

A stretch cluster is a single logical Kafka cluster deployed across multiple DCs or other independent physical infrastructures such as cloud availability zones. For more information, see *Stretch clusters* or *Kafka Stretch cluster reference architecture*.

Replication using SRM

Streams Replication Manager (SRM) is an enterprise-grade replication solution that enables fault tolerant, scalable, and robust cross-cluster Kafka topic replication. A deployment that utilizes SRM for disaster recovery uses multiple Kafka clusters. SRM acts as a bridge between the clusters and replicates the data. For more information, see *Streams Replication Manager Overview*.

The major difference between stretch clusters and an SRM based architecture is how data is replicated between DCs. With stretch clusters, synchronous replication can be achieved. This enables strict guarantees on data durability. With SRM based replication, only asynchronous replication is available. This provides weaker guarantees but higher performance.



Note: In addition to using either a stretch cluster or SRM, you can also choose to have multiple stretch cluster deployments that are replicated by SRM. This can, under the right circumstances, provide the benefits of both approaches.

The DCs located close to each other (low latency) can run as a stretch cluster. SRM can be used to replicate the data to remote DCs (high latency). In this setup, using the 2.5 DC architecture for the stretch clusters can also lower the cost, while the usage of SRM compensates for the lower availability guarantees provided by the 2.5 DC architecture. Additionally, when requirements allow it, if a DC goes down, clients can fail over to a backup cluster and keep the service functional with a temporarily lowered durability guarantee.

This type of architecture is not covered in detail in this document.

Use a stretch cluster if

You have a zero Recovery Point Objective (RPO=0) requirement

RPO=0 means that data should be replicated to multiple DCs before a write is deemed successful (acknowledged by the broker). This can only be achieved by using synchronous replication between DCs, which can be achieved by using a stretch cluster.

However, consider the following. If you can replay data from upstream sources, such as databases, then implementing that recovery function once may be easier than operating and maintaining a stretch cluster.

You need strict message ordering guarantees across DCs

Strict message ordering per partition can only be achieved with a single topic spanning multiple DCs. The SRM architecture involves multiple topics (one original and many replicas), which cannot guarantee the strict ordering during failover or failback operations.

However, consider the following. If your data has an attribute that can be reliably used for ordering, then implementing a reordering step in your downstream processing during application development

might be an easier and more cost-effective solution compared to operating and maintaining a stretch cluster.

You need automatic failover for your clients when a DC goes down

The Kafka protocol has built-in cluster discovery and leadership migration on failures. Therefore, fully automatic failover operations can be achieved using a stretch cluster. The SRM based architecture requires a manual step in the failover process. This makes SRM unsuitable for this use case.

You need exactly once transactional processing

Exactly once processing in Kafka is currently only supported within a single cluster.

Use SRM with multiple clusters if**You need high availability during cluster maintenance**

When a Kafka cluster needs to be stopped for maintenance, clients can fail over to a backup cluster. With the stretch cluster solution, this is not supported. This is due to the fact there is a single Kafka cluster in the architecture.

You need replication between clusters that have high latency (replication across multiple regions)

The stretch cluster architecture is sensitive to high latency, making it unsuitable for multi-region deployments. The asynchronous replication provided by SRM works well even in high latency environments.

You need high throughput replication between DCs

The throughput of the stretch cluster architecture degrades rapidly with increasing latency. SRM, on the other hand, can provide better replication throughput even in high latency environments.

Related Information

[Kafka stretch clusters](#)

[Apache Kafka stretch cluster reference architecture](#)

[Streams Replication Manager Overview](#)

Kafka rack awareness

Learn about Kafka rack awareness and multi-level rack awareness.

Racks provide information about the physical location of a broker or a client. A Kafka deployment can be made rack aware by configuring rack awareness for the Kafka brokers and clients respectively. Enabling rack awareness can help in hardening your deployment, it provides durability guarantees for your Kafka service, and significantly decreases the chances of data loss



Note: Although the feature is called rack awareness, the term rack does not necessarily mean an actual physical server rack. Instead, a rack from Kafka's perspective represents a physical location or independent physical infrastructure. For example, in many production deployments, the feature is used to specify the individual Data Centers that the brokers and clients are running in.

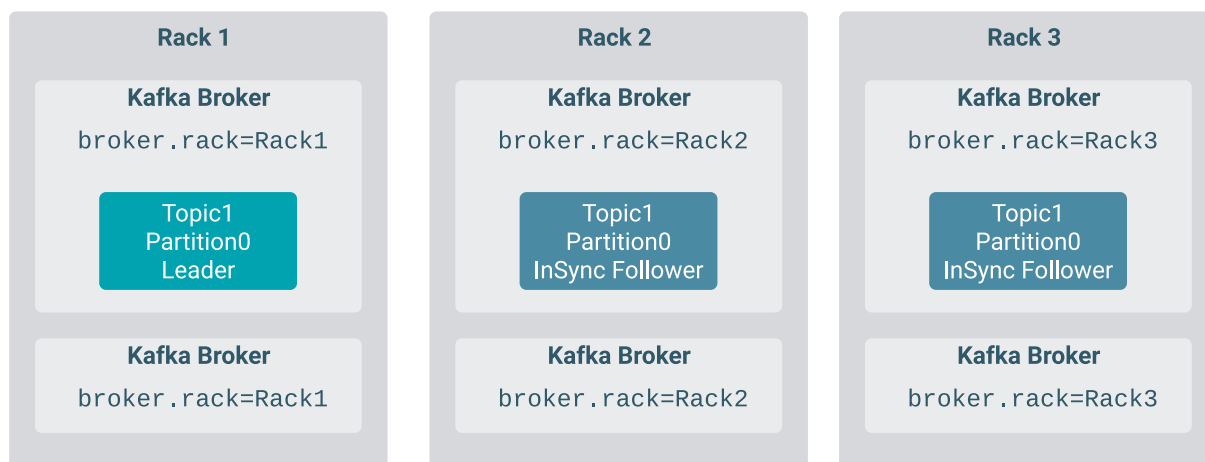
Rack awareness for Kafka brokers

Learn about Kafka broker rack awareness and how rack-aware Kafka brokers behave.

To avoid a single point of failure, instead of putting all brokers into the same rack, it is considered a best practice to spread your Kafka brokers among racks. In cloud environments Kafka brokers located in different availability zones or data centers are usually deployed in different racks. Kafka brokers have built in support for this type of cluster topology and can be configured to be aware of the racks they are in.

If you create, modify, or redistribute a topic in a rack-aware Kafka deployment, rack awareness ensures that replicas of the same partition are spread across as many racks as possible. This limits the risk of data loss if a complete rack fails. Replica assignment will try to assign an equal number of leaders for each broker, therefore, it is advised to configure an equal number of brokers for each rack to avoid uneven load of racks.

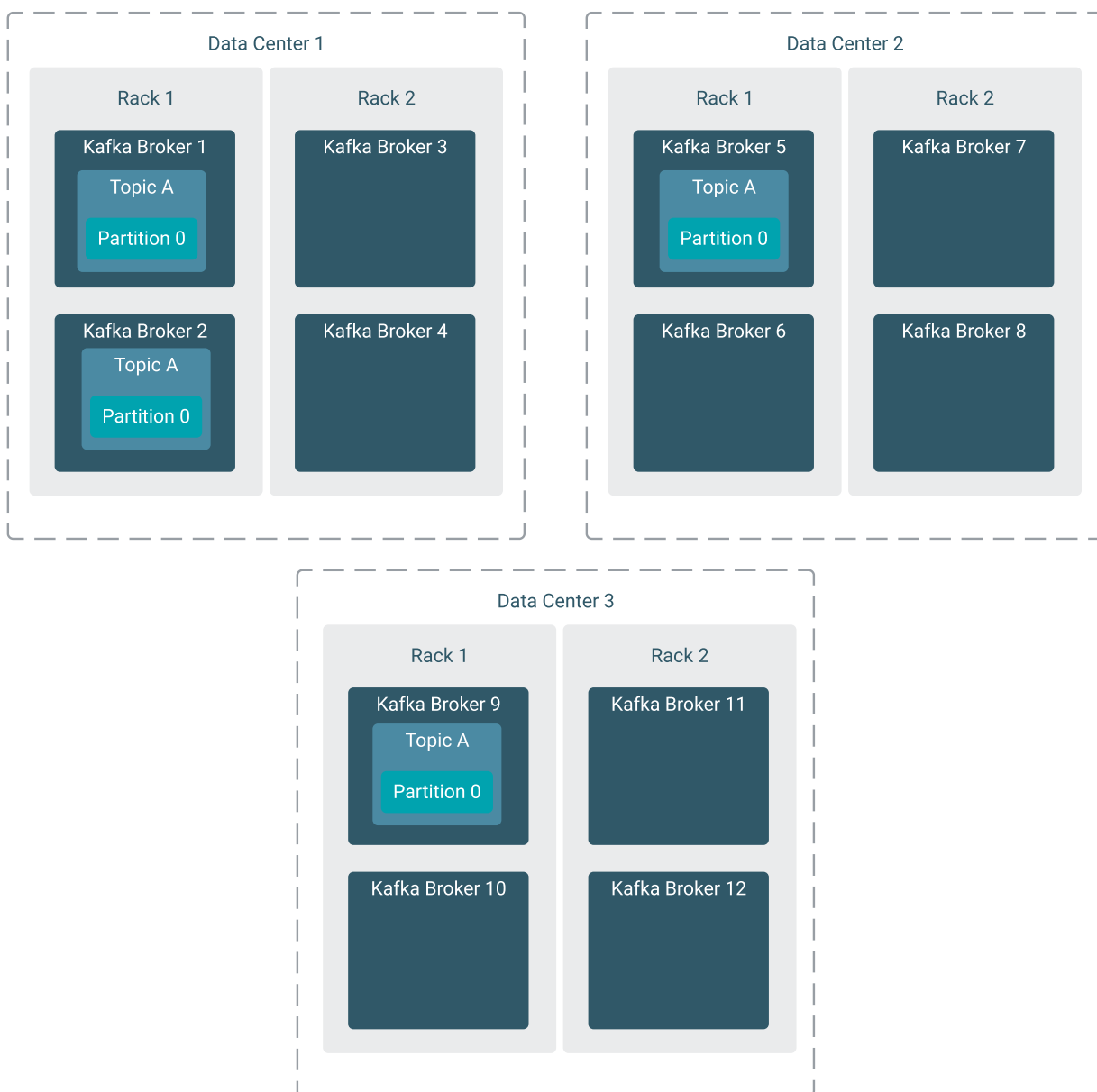
For example, assume you have a topic partition with 3 replicas and have the brokers configured in 3 different racks. If rack awareness is enabled, Kafka will try to distribute the replicas among the racks evenly in a round-robin fashion. In the case of this example, this means that Kafka will ensure to spread all replicas among the 3 different racks, significantly decreasing the chances of data loss in case of a rack failure.



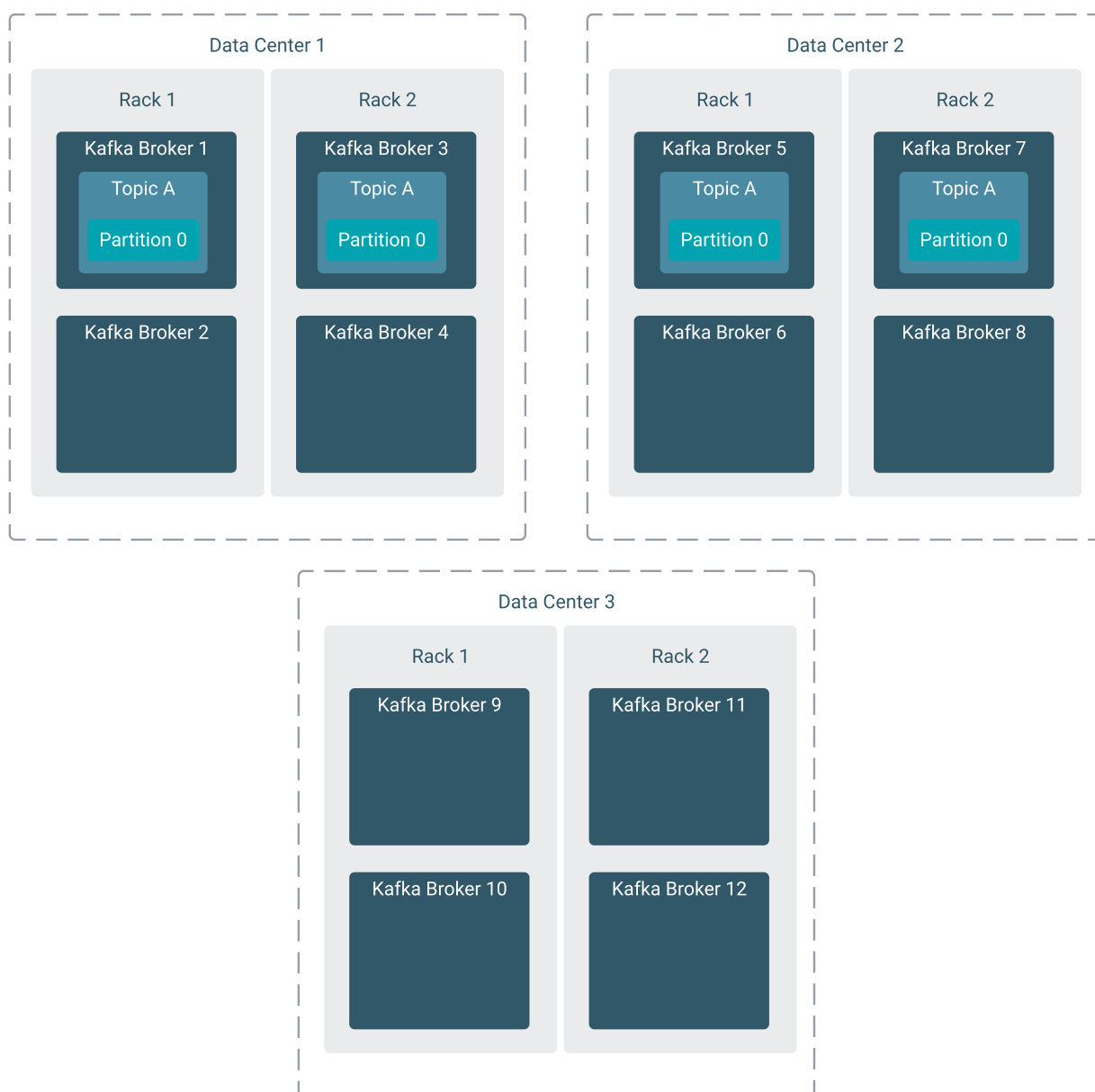
Multi-level rack awareness

Standard rack awareness handles all racks as unique physical locations (for example, Data Centers) that have identical importance. In some use cases, physical locations follow a hierarchical system. For example, besides having multiple DCs, there can be (physical) racks located inside those DCs. In a use case like this, the aim is not only to distribute the replicas among the topmost racks (DCs), but among the second level racks as well (physical racks).

With standard rack awareness, this goal cannot be met. If you use the topmost level as the broker rack IDs (for example, `/DC1`, `/DC2`, `/DC3`), you lose the subsequent levels of the infrastructure. This means that no guarantees are provided for even replica distribution on the second level. Notice how in the following example Rack 2 in each DC is unpopulated.



You can use the full physical location as the broker rack IDs (for example, `/DC1/R2`, `/DC2/R5`), but then the standard rack awareness feature handles all of the IDs as unique locations that are on the same level. As a result, no guarantee is provided that Kafka evenly distributes replicas on the top (DC) level. Notice how in the following example Data Center 3 is unpopulated.



To ensure that multi-level rack guarantees can be met, in addition to standard rack awareness, Cloudera Kafka supports a multi-level rack aware mode. This mode of rack awareness can be configured by specifying multi-level rack IDs and selecting a feature toggle in Cloudera Manager. When enabled, Kafka takes into consideration all levels of the hierarchy and ensures that replicas are spread evenly in the deployment.

Cruise Control optimizations with multi-level rack awareness

If Cruise Control is present in the cluster, and the Kafka brokers run with multi-level rack awareness enabled, Cruise Control will replace all the standard rack aware goals in its configuration with a multi-level rack-aware goal. This ensures that Cruise Control optimizations do not violate the multi-level rack awareness guarantees. For more information on how the goal works, see *Setting capacity estimations and goals* in the Cruise Control documentation.

Related Information

[Setting capacity estimations and goals](#)

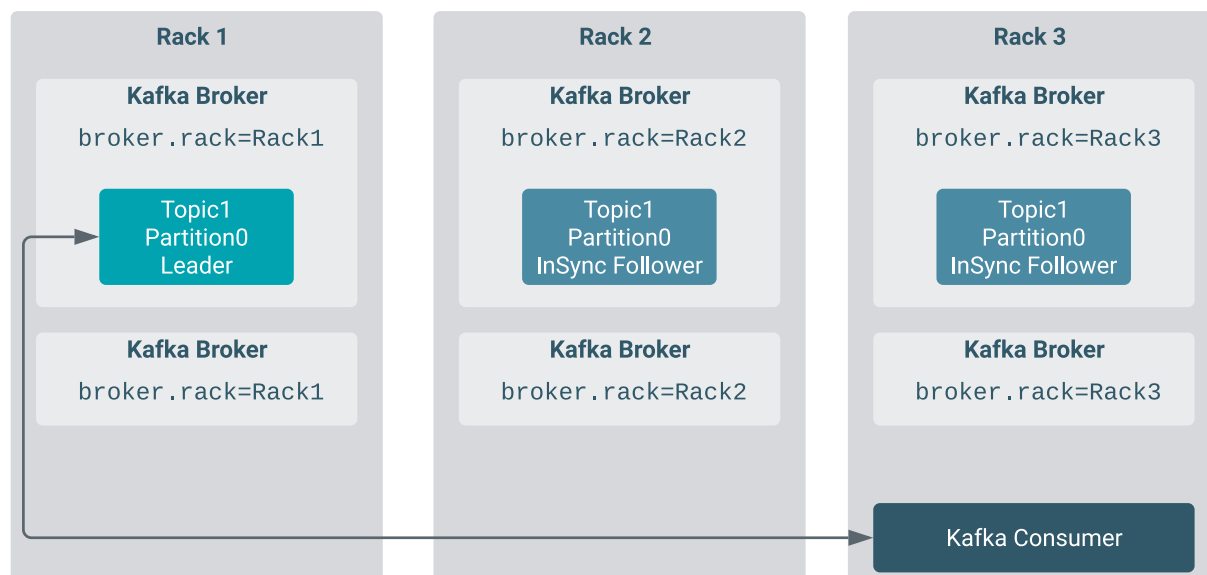
[Configuring rack awareness for Kafka brokers](#)

[Configuring multi-level rack awareness for brokers](#)

Rack awareness for Kafka consumers

Learn about follower fetching, which can be used to make Kafka consumers rack aware

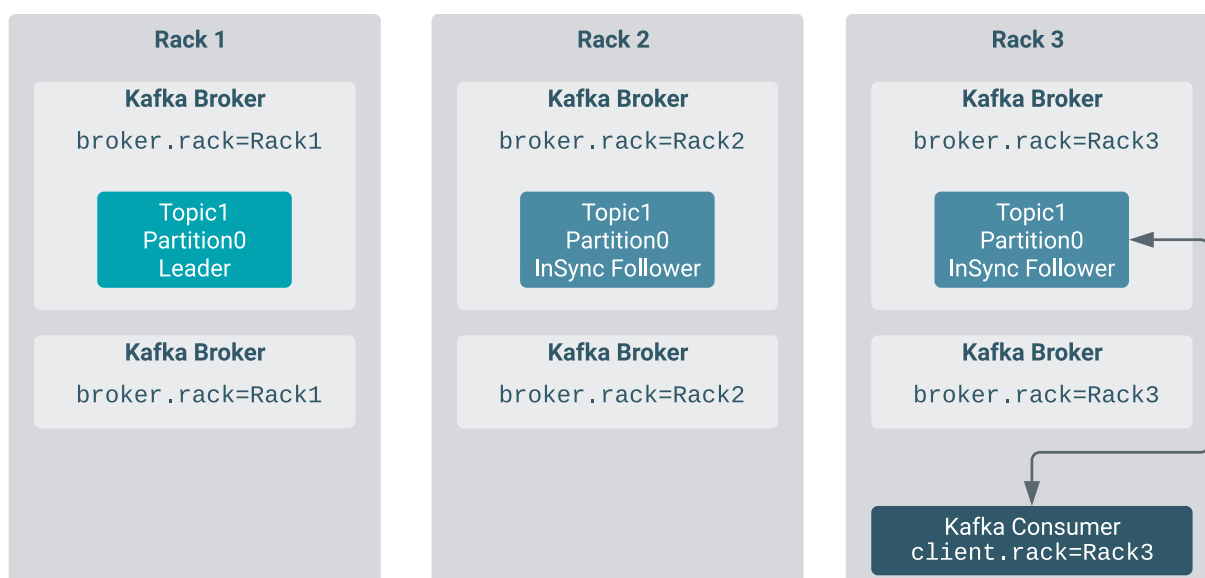
When a Kafka consumer tries to consume a topic partition, it fetches from the partition leader by default. If the partition leader and the consumer are not in the same rack, fetching generates significant cross-rack traffic, which has a number of disadvantages. For example, it can generate high costs and lead to lower consumer bandwidth and throughput.



For this reason, it is possible to provide the client with rack information so that the client fetches from the closest replica instead of the leader. If the configured closest replica does not exist (there is no replica for the needed partition in the configured closest rack), it uses the partition leader. This feature is called follower fetching and it can be used to mitigate the costs generated by cross-rack traffic or increase consumer throughput.



Note: Due to the nature of the Kafka protocol and high watermark propagation, consumers might experience increased message latency when fetching from a replica compared to when they are fetching from the leader.



Follower fetching in multi-level deployments

Follower fetching is also supported if multi-level rack awareness is enabled for the brokers. When Kafka brokers are running in multi-level rack-aware mode, a multi-level rack-aware ReplicaSelector is automatically installed. This selector ensures that if a consumer that has a multi-level rack ID, the closest replica is selected from the multi-level hierarchy.

Related Information

[Configuring rack awareness for Kafka consumers](#)

[Configuring multi-level rack awareness for consumers](#)

Rack awareness for Kafka producers

Learn about rack awareness for Kafka producers.

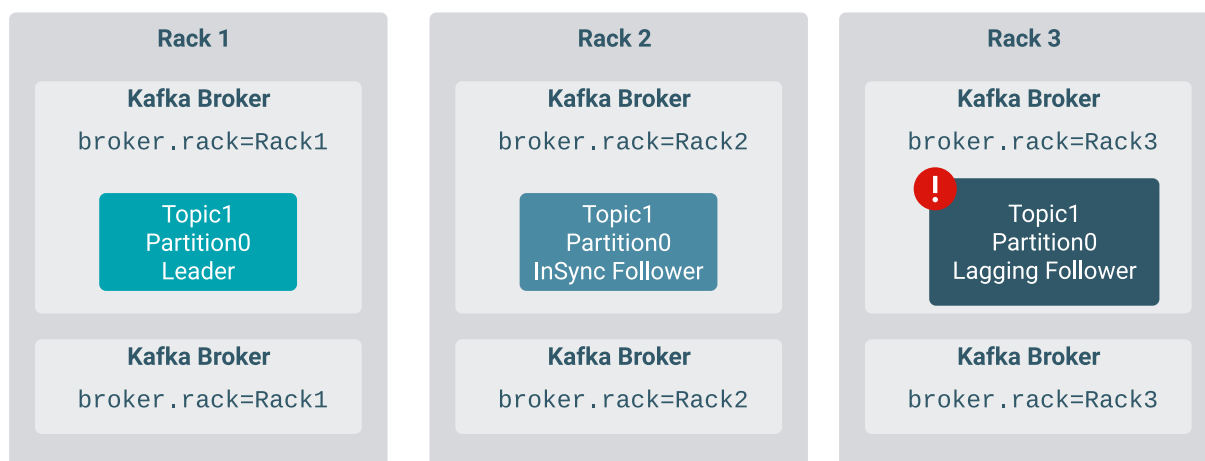
Compared to brokers or consumers, there are no producer specific rack-awareness features or toggles that you can enable. However, in a deployment where rack awareness is an important factor, you can make configuration changes so that producers make use of rack awareness and have messages replicated to multiple racks.

Specifically, Cloudera recommends a configuration that ensures that the produced messages are replicated to at least two different racks before the messages are considered to be successful. This involves configuring acks to all in the producer configuration and setting up `min.insync.replicas` for the topics in a way that ensures a minimum of two racks get the message before the produce request is considered successful.

The configuration of the acks property is fixed. If you want to make your producers rack aware, the property must be set to all no matter the cluster topology or deployment.

The exact value you set for `min.insync.replicas` on the other hand depends on your cluster deployment. Specifically, the `min.insync.replicas` value you must set will depend on the number of racks, brokers, and the replication factor of your topics. Cloudera recommends that you exercise caution and review the following examples to better understand configuration.

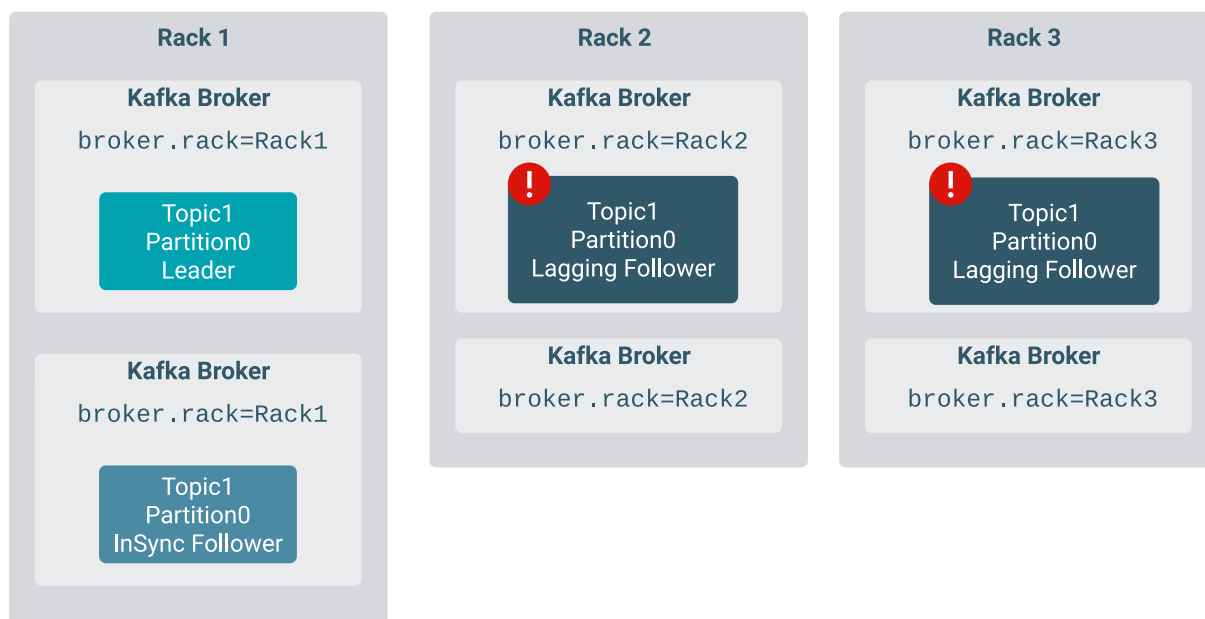
For example, consider a Cloudera recommended deployment that has three racks with topic replication set to 3. In a case like this, a `min.insync.replicas` setting of 2 ensures that you always have data written to at least two different racks even if one replica is lagging.



Understand however, that setting `min.insync.replicas` to 2 does not universally work for all deployments and may not guarantee that you always have your produced message in at least two racks. Configuration depends on the number of replicas, as well as the number of racks and brokers.

If you have more replicas and brokers than racks, you will have at least two replicas in the same rack. In a case like this, setting `min.insync.replicas` to 2 is not sufficient, a partition might become unavailable under certain circumstances.

For example, assume you have three racks with topic replication factor set to 4, meaning that there are a total of four replicas. Additionally, assume that only two of the replicas are in the in-sync replica set (ISR), the leader and one of the followers, and both are located in the same rack. The other two replicas are lagging. Unclean leader election is disabled to avoid data loss.



When the leader and the in-sync follower (located in the same rack) successfully append a produced message to the log, message production is considered successful. The leader does not wait for acknowledgement from the lagging replicas. This is because `acks=all` only guarantees that the leader waits for the replicas that are in the ISR (including itself). This means that while the latest messages are available on two brokers, both are located on the same rack. If the rack goes down at the same time or shortly after production is successful, the partition will become unavailable as only the two lagging replicas remain, which cannot become leaders.

In cases like this, a correct value for `min.insync.replicas` would be 3 instead of 2 as three ISRs would guarantee that messages are produced to at least two different racks.

Related Information

[Configuring rack awareness for Kafka producers](#)

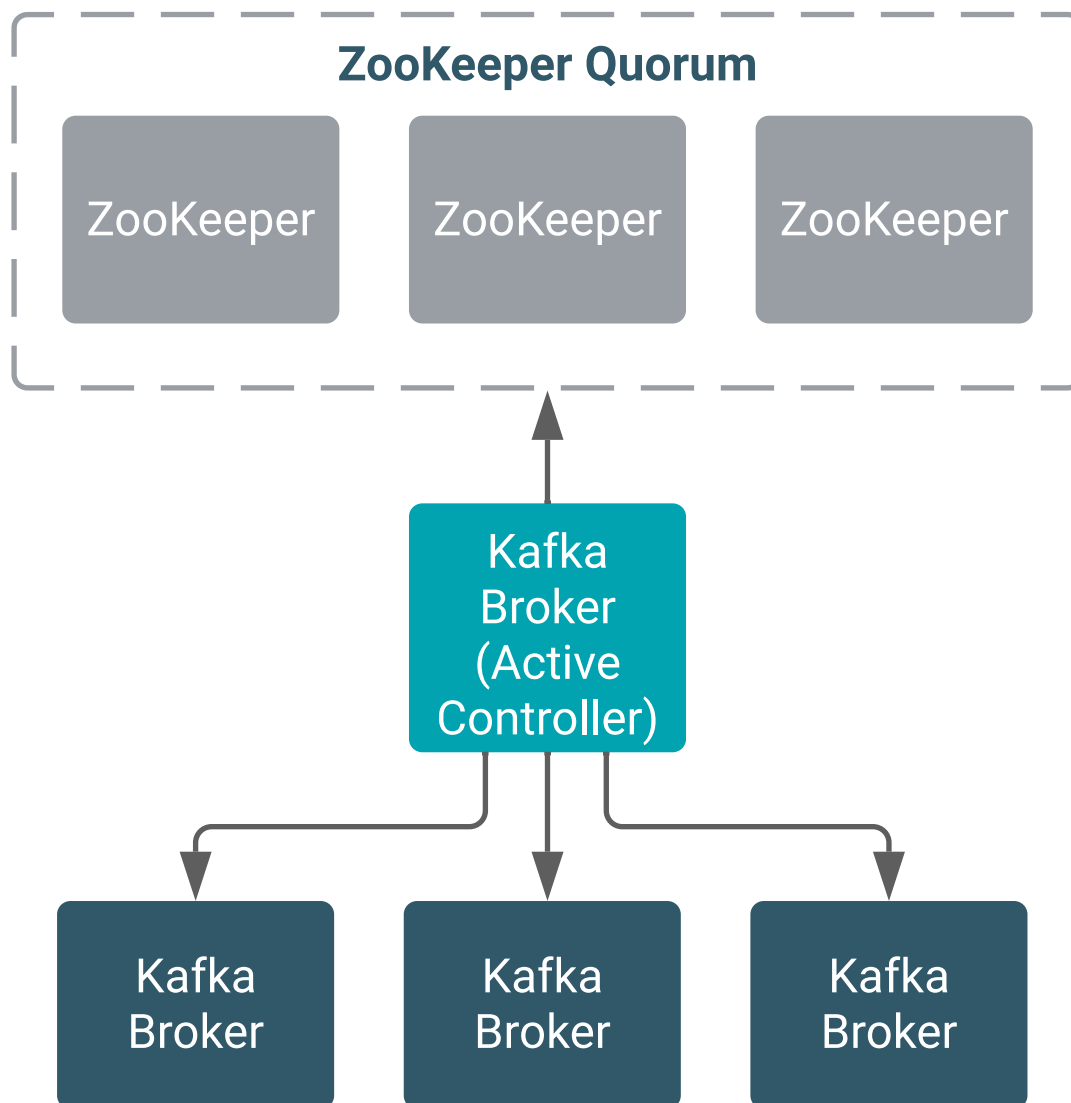
Kafka KRaft [Technical Preview]

Apache Kafka Raft (KRaft) is a consensus protocol used for metadata management that was developed as a replacement for Apache ZooKeeper. Using KRaft for managing Kafka metadata instead of ZooKeeper offers various benefits including a simplified architecture and a reduced operational footprint. Learn more about what KRaft is, how it works, and how it compares to ZooKeeper deployments.

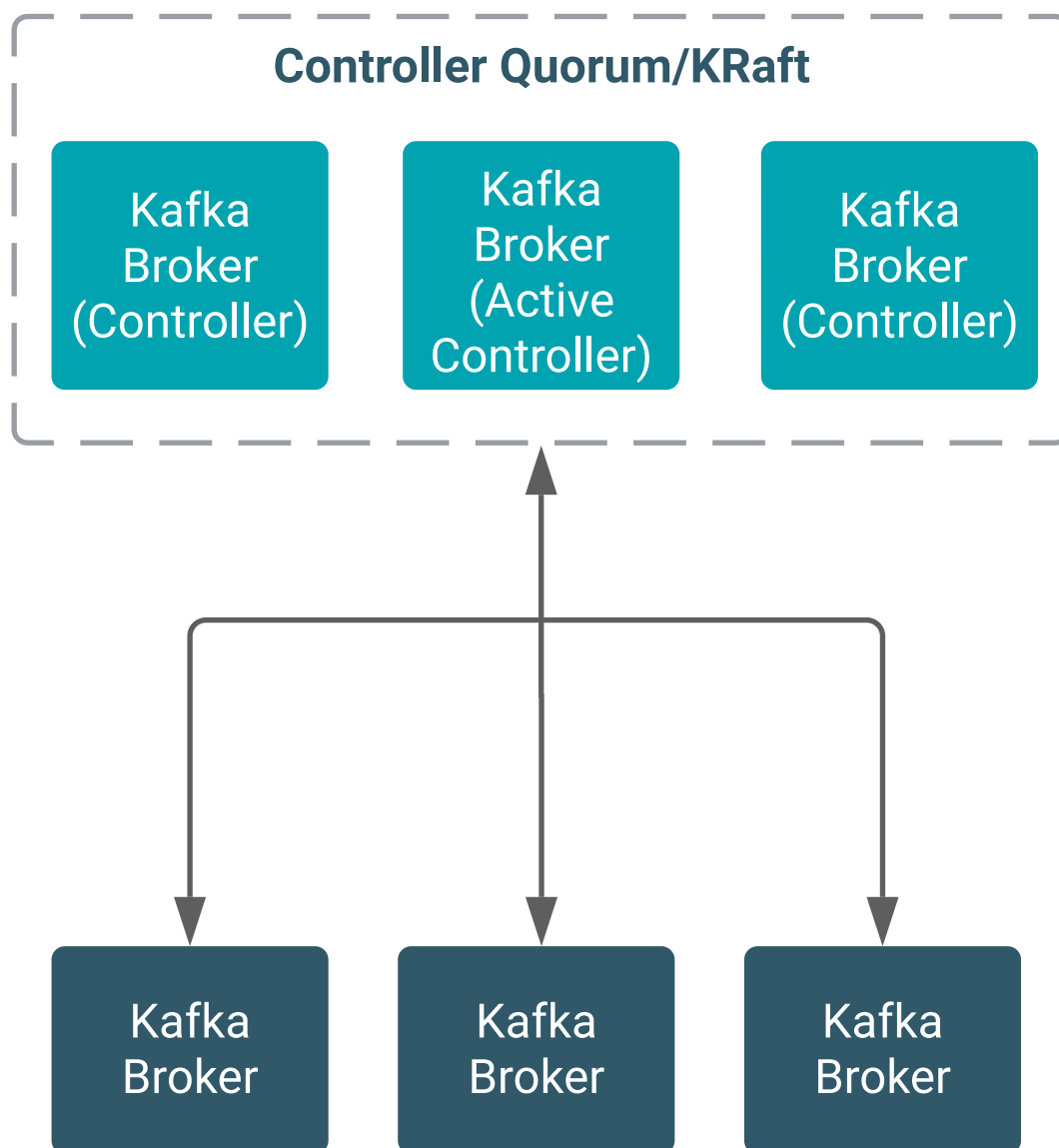


Note: Kafka KRaft is available in this version of CDP but is not ready for production deployment. Cloudera encourages you to explore this technical preview feature in non-production environments and provide feedback on your experiences through the [Cloudera Community Forums](#). For more information regarding KRaft limitations and unsupported features, see [Known Issues in Apache Kafka](#).

In traditional Kafka clusters, Kafka's metadata (including broker metadata, configurations, and client quotas) is stored in ZooKeeper. The brokers access this data in ZooKeeper through the broker that is designated as the active controller. This is the traditional way of how Kafka operates.



In the controller quorum mode, ZooKeeper is replaced with Kafka's own consensus implementation called Kafka Raft, or KRaft for short. KRaft is based on the Raft algorithm with some changes that are otherwise native in Kafka.



In this mode, a set of specialized brokers, called controllers, are deployed. These controllers form a cluster and are responsible for storing and maintaining Kafka metadata. The quorum of controllers ensures that the metadata is available at all times and that it is consistent. Additionally, one of the controllers is designated as the active controller (also known as the leader), which coordinates metadata changes with the brokers. The controllers store the metadata in a Kafka topic that can be found on each node.

Compared to the ZooKeeper, KRaft has a number of advantages including:

- **Fast and efficient metadata storage and propagation**

With ZooKeeper, cluster updates are committed to ZooKeeper through the active controller. In KRaft, updates are sent directly to the controller quorum, eliminating one hop in the chain. In addition, because metadata is stored in a Kafka topic, both brokers and controllers can easily replicate and store the data locally. Both of these aspects of KRaft can help in substantially improving Kafka's performance.

- **Simplified, deployment, setup, and management**

KRaft eliminates the need for a ZooKeeper service. This means that you only need to manage and maintain a single service, which is Kafka.

The Raft algorithm

Learn the basics of the Raft consensus algorithm, which KRaft is based on.

In many distributed systems there is a need for consensus. For example, consensus is needed to decide what is a distributed configuration's value, which servers are active, or which ones are passive. This type of information is critical for a distributed actor. Therefore, the information must be stored in a way that it is accessible reliably at all times.

There are many iterations to a safe consensus algorithm that ensure that all of the requirements above are met. A more modern version of such an algorithm is the Raft algorithm.



Note: The most prominent component to implement a consensus system is ZooKeeper, which has been used in many projects. Even though both provide highly reliable distributed coordination, ZooKeeper and Raft are somewhat different.

Raft uses timers and periodic heartbeats to keep track of alive nodes. As long as the majority of nodes are functioning, clients can publish and access the data. Raft replicates data on all of the Raft quorum nodes. Additionally a leader is elected randomly on first startup. When data is produced, it is pushed to peer nodes with the periodic heartbeats. Follower nodes acknowledge this data. A request is committed only when the majority of nodes confirm it.

Raft can ensure that partitioned controller quorums operate correctly. That is, if a partition does not have the majority, then data sent to those nodes is not committed. When partitions merge, the majority wins.

In case of a leader failure, remaining nodes hold a vote. A new leader is chosen from the remaining pool of nodes. After the new leader is elected, it becomes the origin of the data. Other nodes switch over and replicate from the new leader.

Voting is timeout based. If a node doesn't get a heartbeat within the session timeout, it sends out vote requests. Other nodes do the same, but at the same time accept incoming votes. Eventually, the first node who sent out requests wins and is elected the new leader.

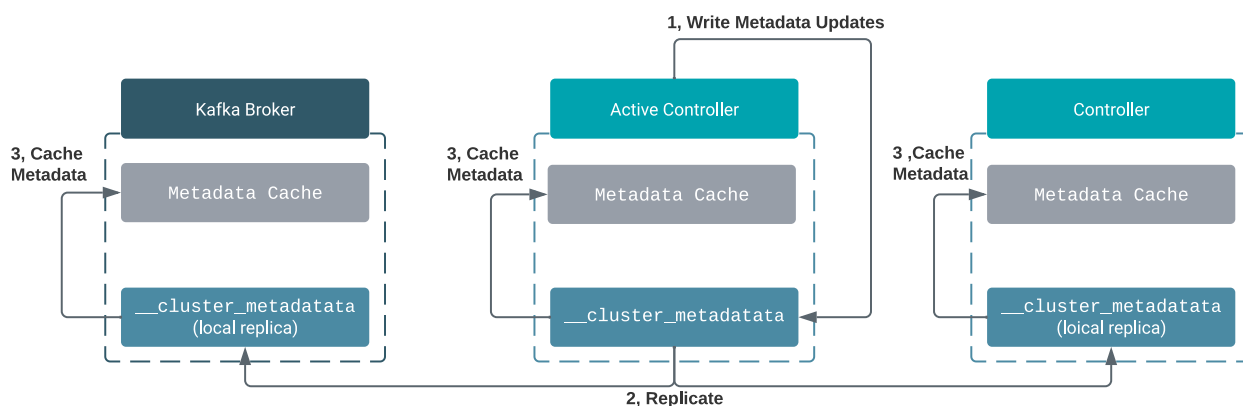
For more information on Raft, see <https://raft.github.io/>.

KRaft metadata management

Learn how Kafka metadata is stored and managed in KRaft.

In KRaft mode, metadata is stored in a dedicated internal Kafka topic called `__cluster_metadata`. This topic has a single partition that contains all information related to the current state of the Kafka service. The leader of the single partition is the active controller. The active controller is responsible for writing metadata changes to the `__cluster_metadata` topic.

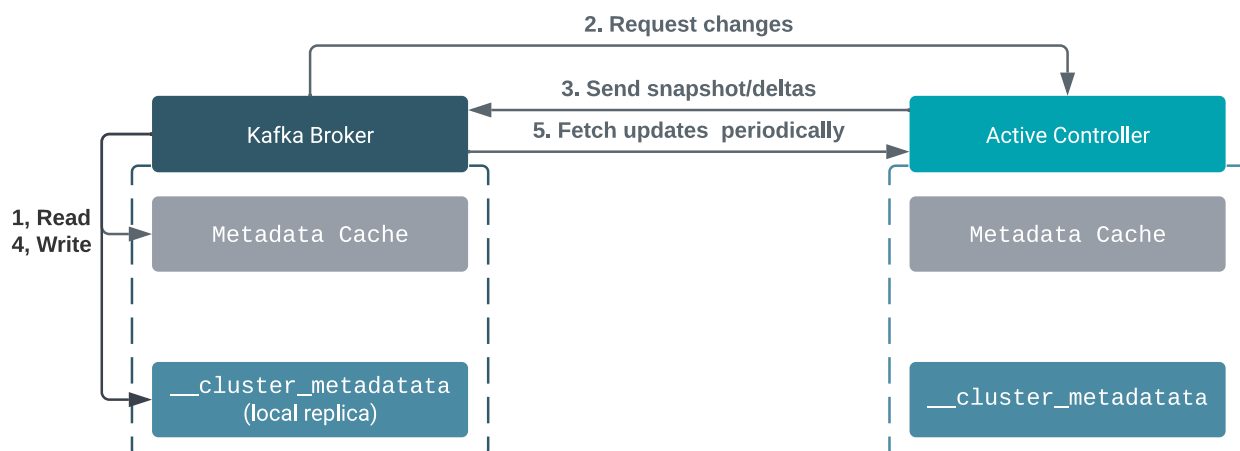
Both brokers and controllers fetch and replicate this topic to store it locally. Controllers are replica followers, meaning that they are part of the in-sync replica set (ISR) for the topic. Brokers are observers only. This means that while they replicate the topic, they are not part of the ISR. Both brokers and controllers periodically commit metadata from `__cluster_metadata` to their metadata cache.



When a broker starts up, it reads the most recent changes available in its metadata cache. Afterward, it requests the changes from the controller that happened since the last update.

Next, the active controller responds with a series of deltas or the full state of the cluster. The full state of the cluster is only sent if the broker has no cached metadata (it's an empty broker starting for the first time) or if it is severely lagging behind. The broker then commits the updates it receives to its cache and the topic replica.

Going forward, the broker periodically asks for updates from the active controller. These periodic fetch requests that the broker sends also act as heartbeats.



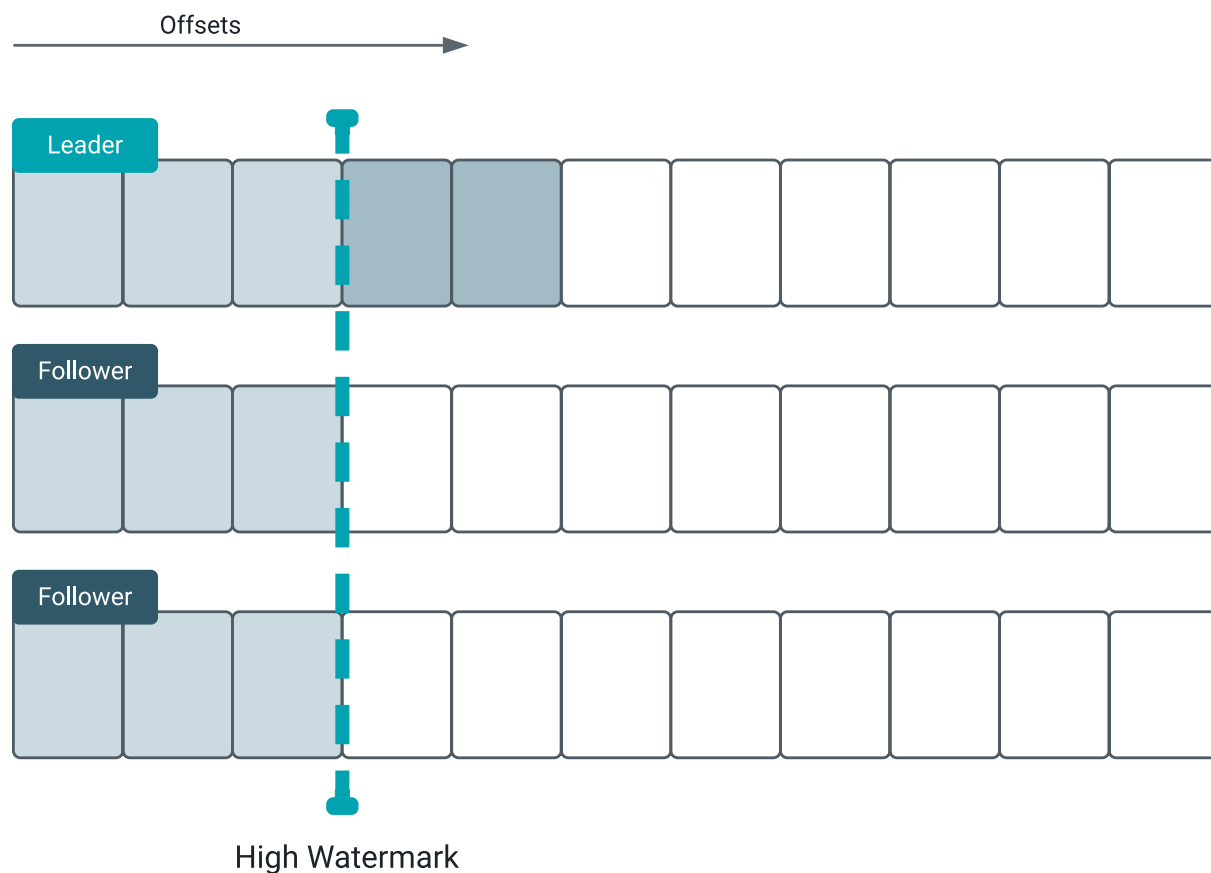
KRaft metadata replication and the HWM

Learn about the high watermark and how Kafka metadata is replicated in KRaft.

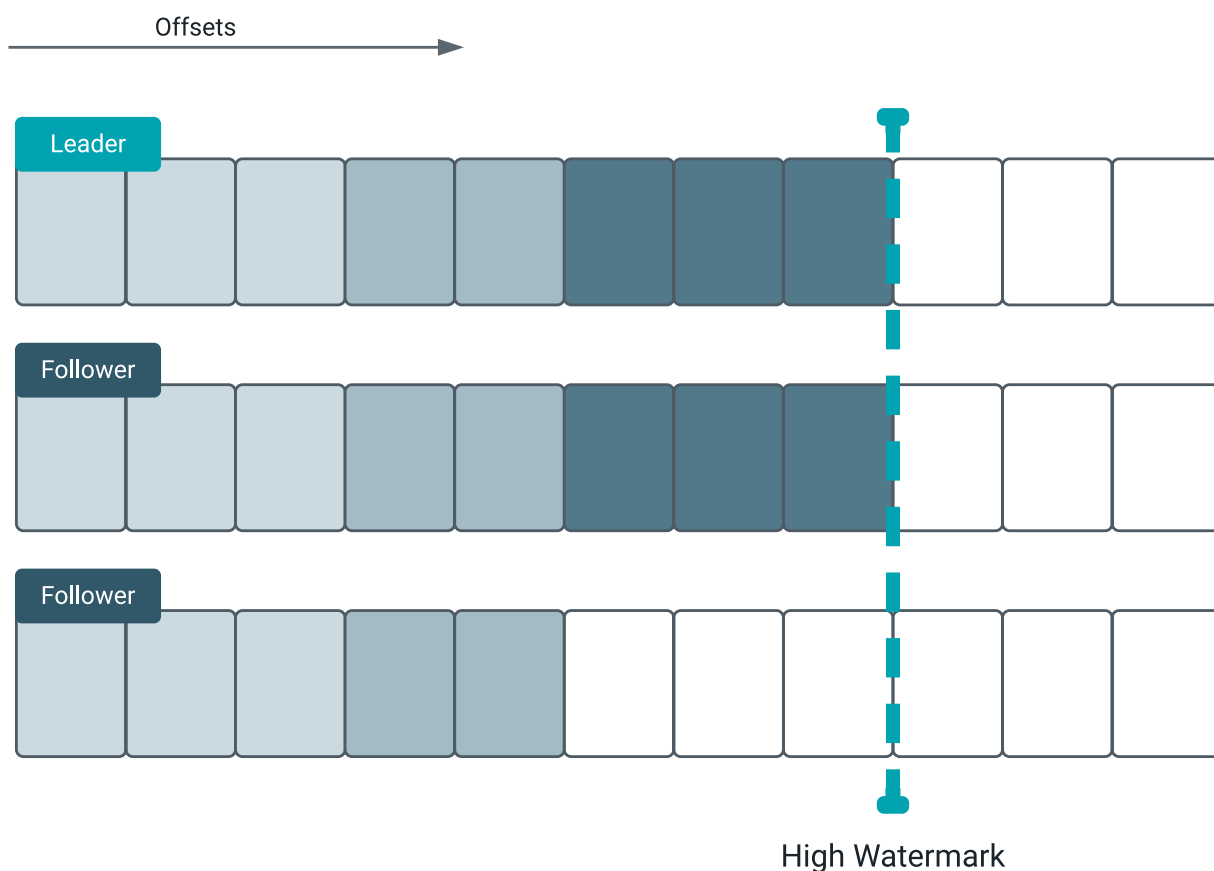
In KRaft mode, Kafka uses its own implementation of the Raft protocol. This implementation is based on Raft, but is heavily influenced by Kafka's log replication protocol.

One of the cornerstones of Kafka replication is the high watermark (HWM). The HWM defines the highest offset that has been replicated across all ISR replicas. It is maintained by the leader, but followers cache it and update it with each offset fetch so a consistent state is maintained across the cluster.

On leader change, all partitions truncate their log to the HWM (messages above are considered unsuccessful and the client retries). In general, the HWM must be larger than the log start offset and smaller than the log end offset.



In KRaft, the concept of the HWM is slightly different. Instead of being the **highest offset** that is replicated across **all ISR** replicas, it is the highest offset that is replicated by the **majority** of replicas. This way Kafka's replication closely mimics the Raft algorithm's behavior.



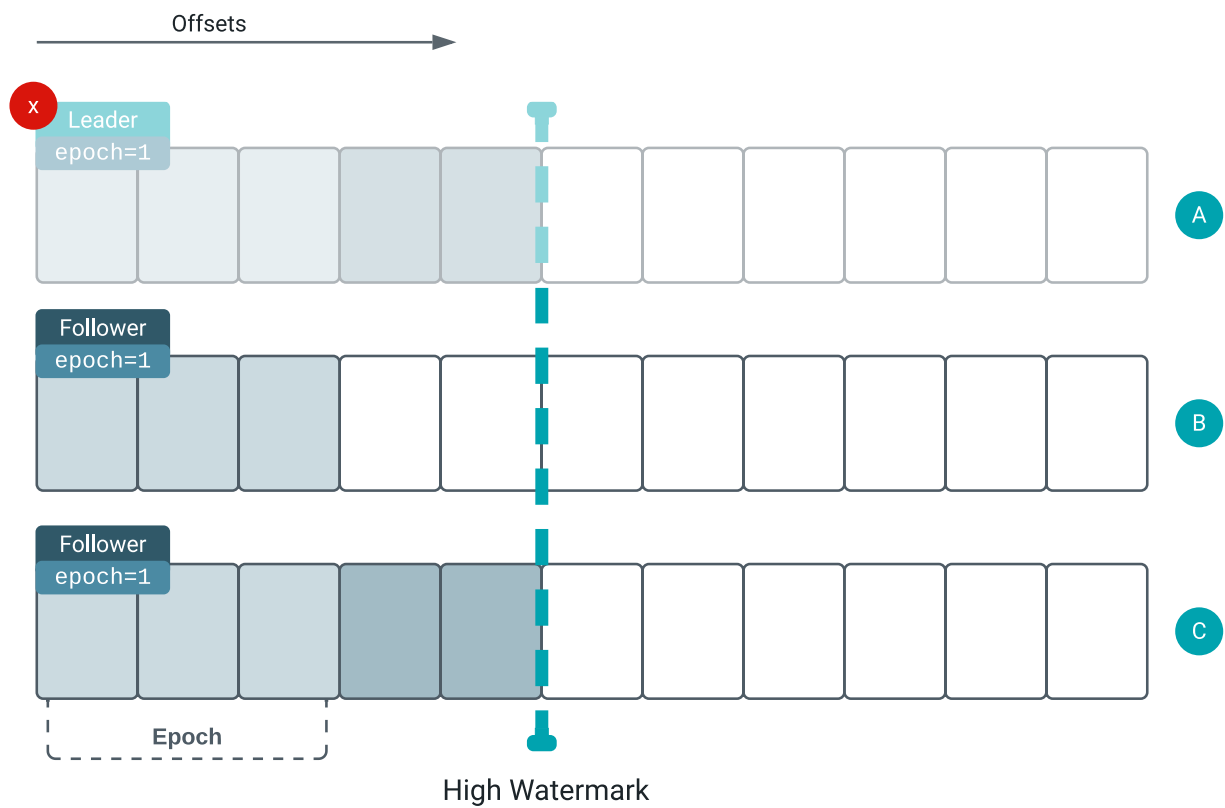
KRaft leader changes

Learn how leader changes are handled in KRaft.

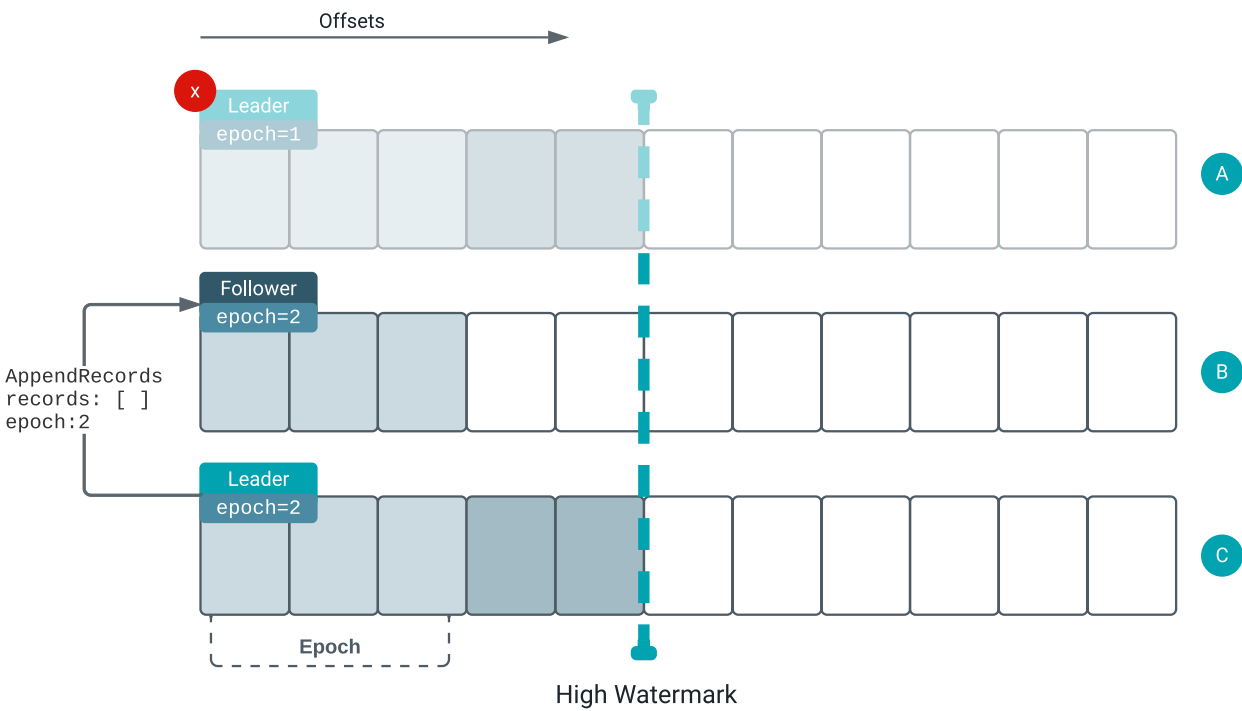
When the active controller (the leader for the controllers and the `__cluster_metadata` topic) fails, a new leader must be elected. Otherwise, Kafka can not continue to function. Leader changes in KRaft work similarly to the original protocol.

On leader change, as according to the protocol, each node gets to vote. In KRaft each node can cast a single vote per epoch. Nodes always vote for the node that has the “longest” log. Longest in this case means highest epoch, or, if the epoch is identical, larger offset. The new leader is decided by a simple majority. That is, the node with the most votes is elected leader.

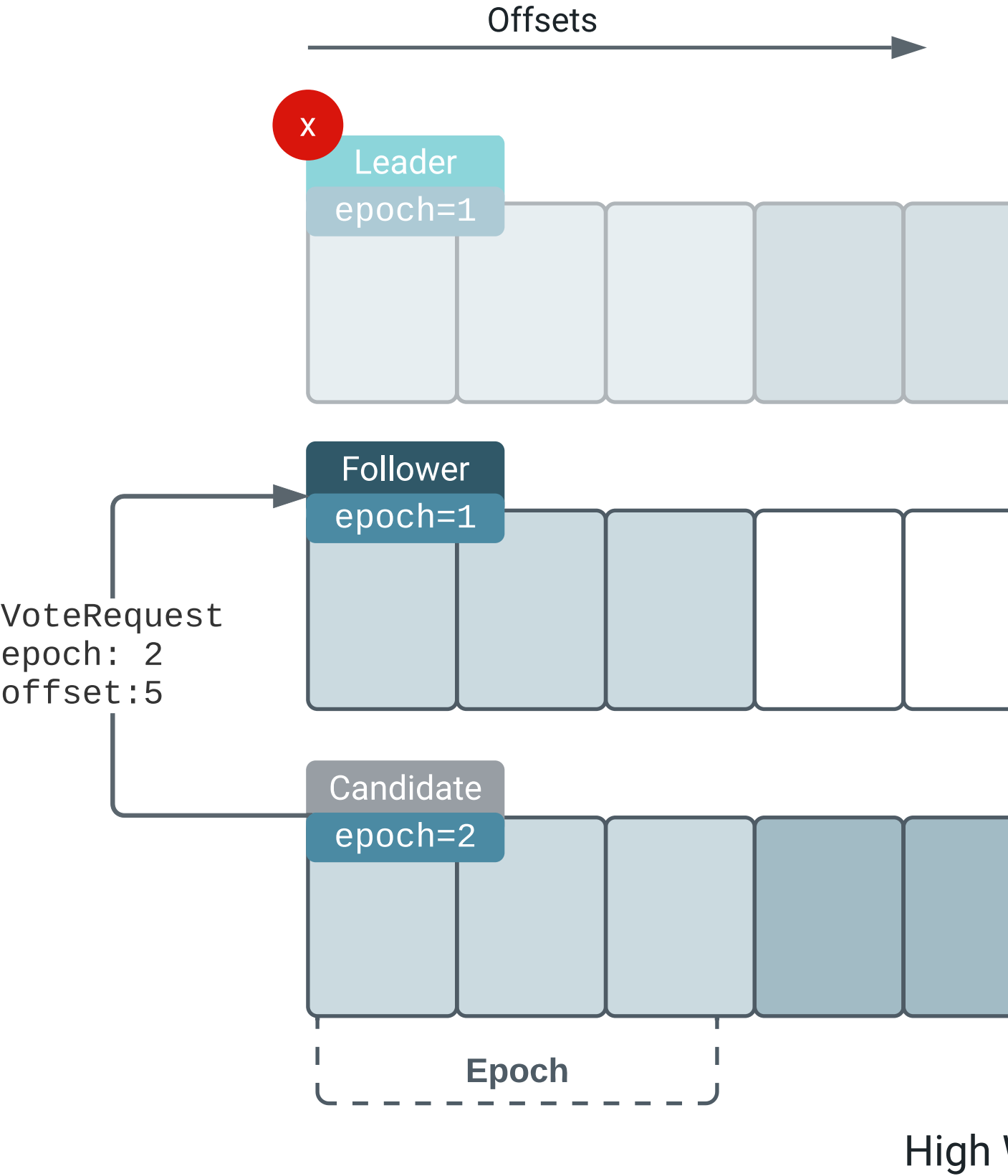
For example, assume that there are three controllers. Each of them has a replica of `__cluster_metadata`. Replica A was the leader, but due to a node failure it goes offline and falls out of the in-sync replica set (ISR).



A new leader is required. Each candidate sends a vote request which contains the epoch and the offset. If the sender has a longer log than the recipient of the request, the recipient votes for the sender.



In this example the candidate (replica C) has an epoch count of 2, while the follower (replica B) only has an epoch count of 1. The candidate has the “longest” log. The follower, therefore, votes for the candidate and the candidate becomes the leader. If leadership is decided, the followers truncate back to the HWM and fetch the new leader’s data.

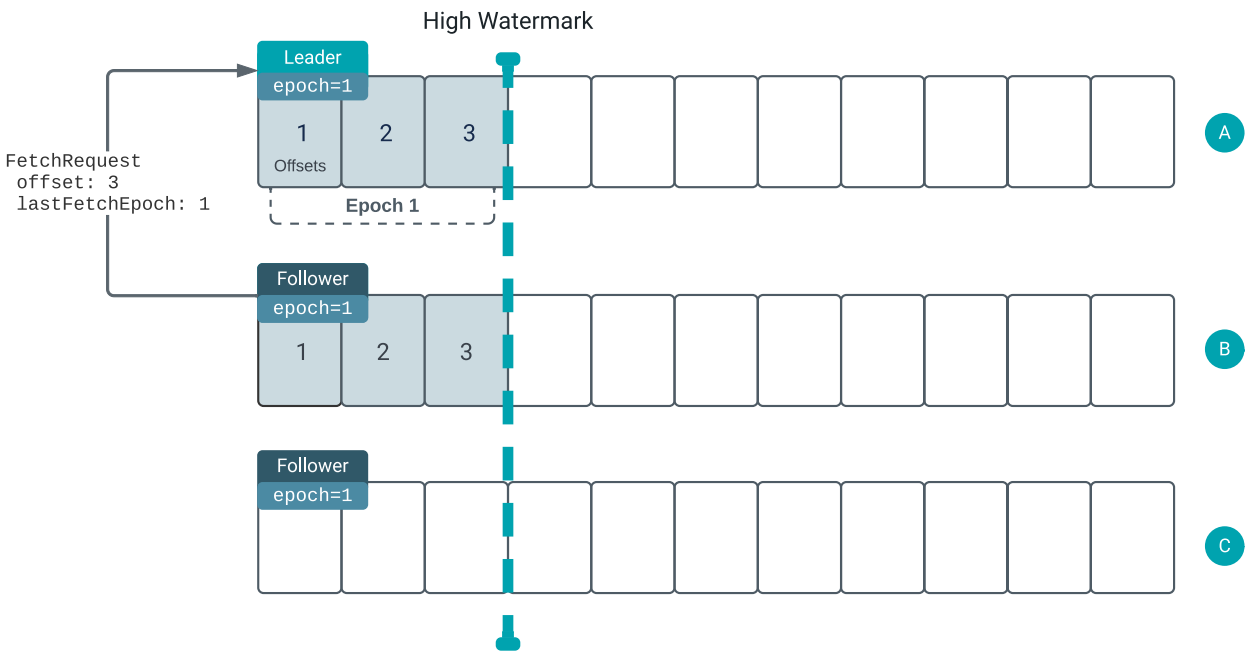


KRaft log reconciliation

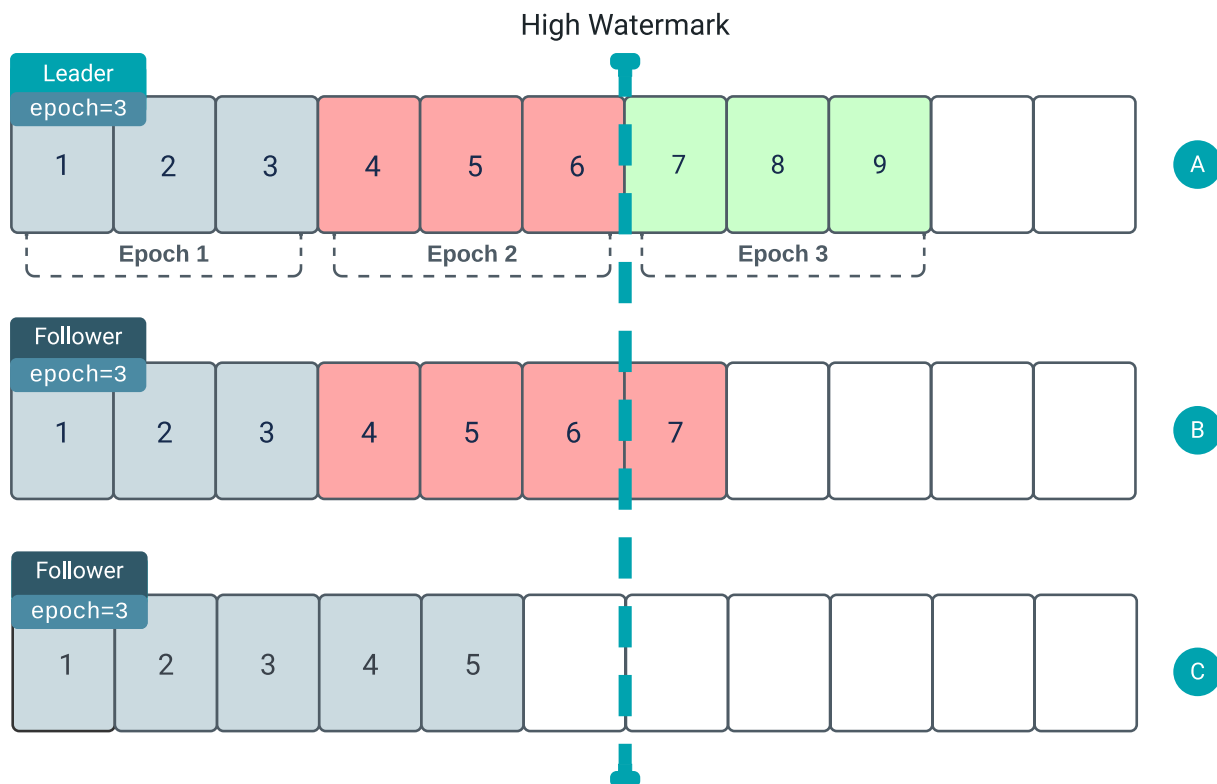
Learn how logs are reconciled and how the high watermark (HWM) is incremented in KRaft.

When a leader change happens, the log must be reconciled. After a new leader is elected, followers send fetch requests to the leader. The fetch request contains the last fetched epoch and an offset. The epoch is initially 1, but increments with each leader change. Data is in fact stored with the epoch numbers. Therefore, the high watermark (HWM) is incremented only if the majority of nodes replicated the data with the highest epoch.

For example, assume that you have three controllers. Each of them has a replica of `__cluster_metadata`. Replica A, which is the leader, is in epoch 1, offset 3. Replica B, a follower, is caught up with the leader. The majority of the nodes replicated the data with the highest epoch. As a result, the HWM is moved to offset 3.

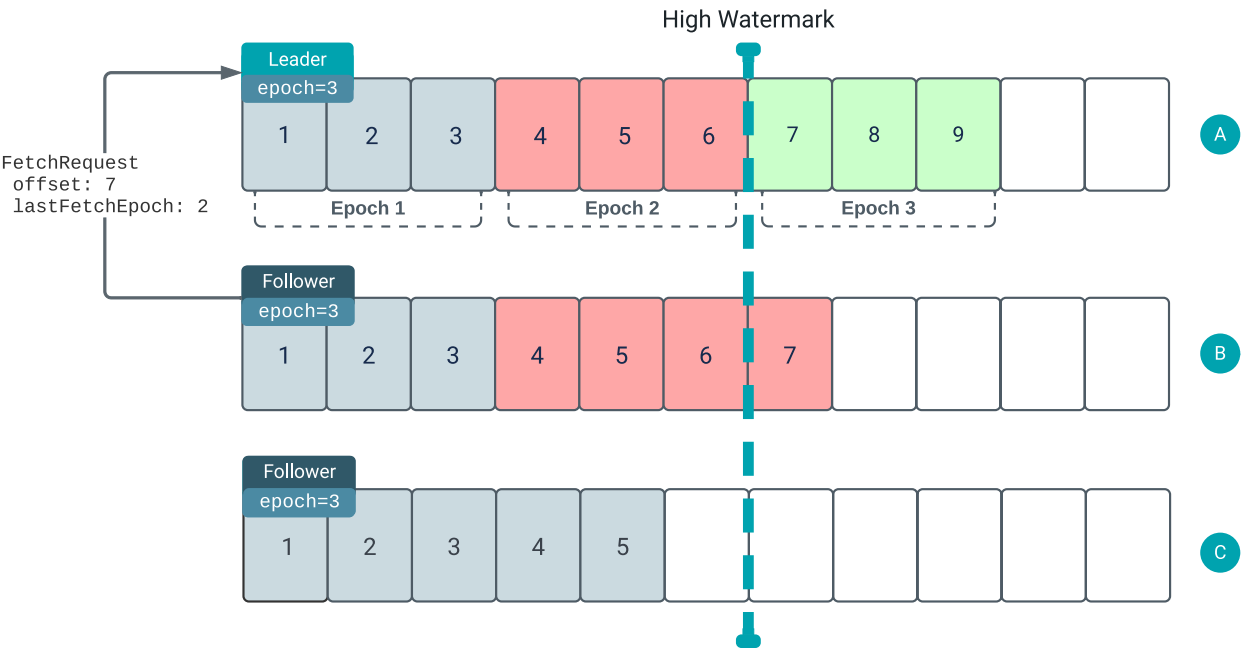


Let's look at a more complex example. Assume that you have the same setup, but more messages were committed and a number of leader changes happened. The cluster is in the following state.

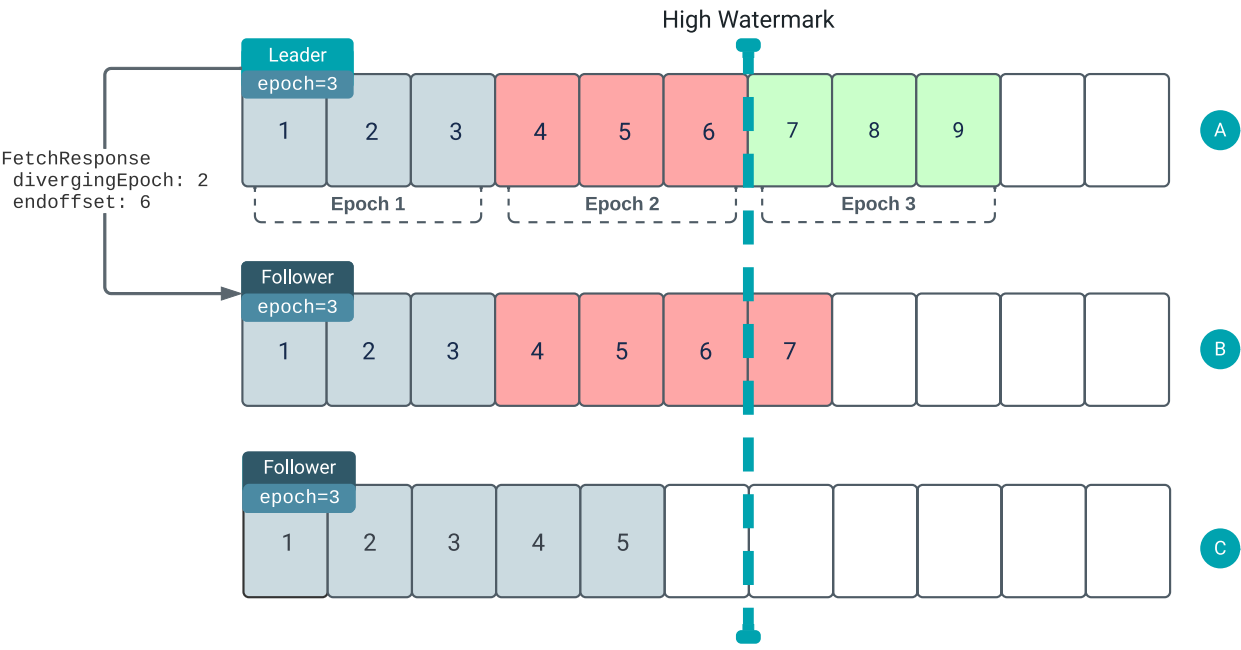


Notice how both followers (B and C) already accepted the leader's epoch, and that replica A and B have additional messages above the HWM. However, not all messages were replicated to the majority of nodes and the HWM is not incremented. In this scenario, the log must be reconciled before the HWM can be incremented. The reconciliation process is as follows.

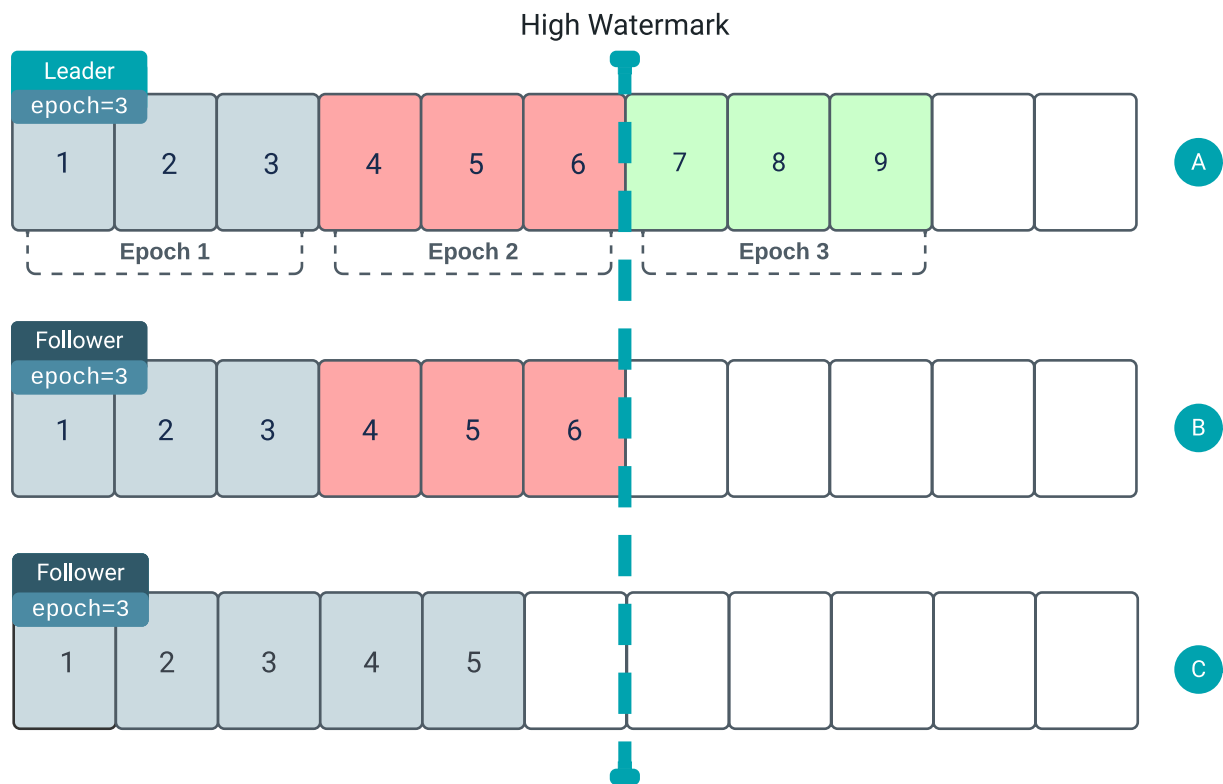
First, followers send a fetch request to the leader. The fetch request contains the offset as well as the last fetched epoch. For replica B, the offset is 7, the last fetched epoch is 2.



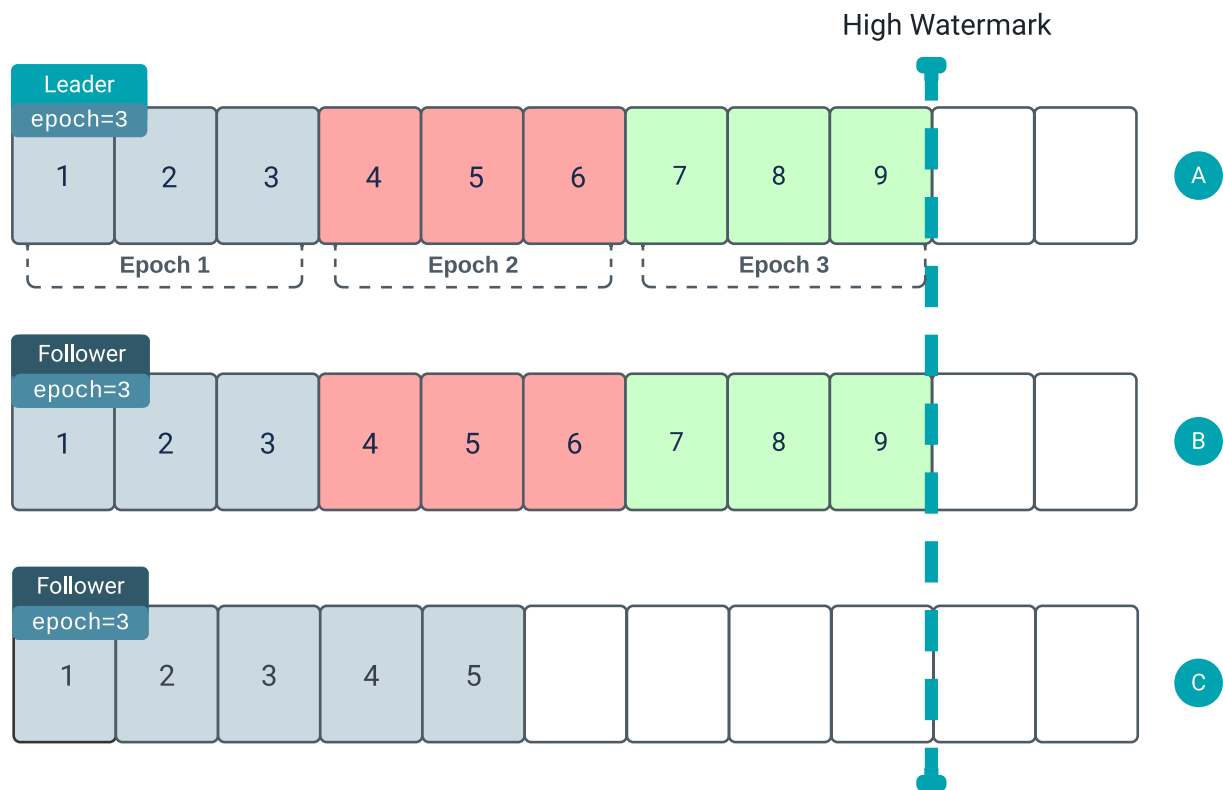
Based on the fetch request, the leader knows that the logs have diverged and sends a response that includes the diverging epoch and the end offset of that epoch. In this case, the diverging epoch is 2, the end offset is 6.



The information included in the fetch responses enables the follower to truncate its log to the correct position.



After truncating the log, the follower can continue to replicate any new messages from the leader. Once all new messages are replicated to the majority of nodes the HWM is incremented.



The same process also happens for replica C. Ultimately, it catches up to the leader as well.

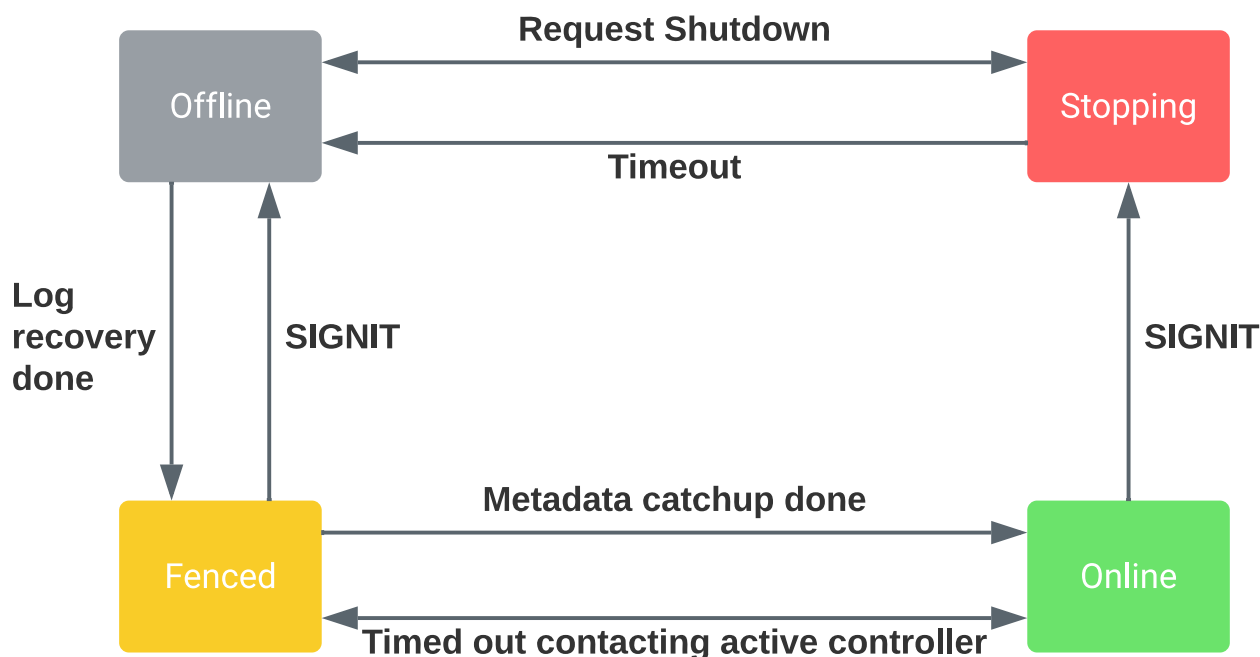


KRaft broker state machines

Brokers in KRaft mode have four states. Learn what these states are and when brokers enter each state.

The states are as follows:

- **Offline**: The broker hasn't started. The broker remains in this state until the local logs from the file system are loaded.
- **Fenced**: The broker has started, but can't receive requests from clients. Replicas aren't in sync with the cluster. The broker stays in this state until it catches up with the current metadata of the cluster.
- **Running**: The broker has started and is ready to receive requests from clients.
- **Stopping**: The broker is still running, but it is migrating the partition leaders to other brokers.



Kafka FAQ

A collection of frequently asked questions on the topic of Kafka.

Basics

A collection of frequently asked questions on the topic of Kafka aimed for beginners.

What is Kafka?

Kafka is a streaming message platform. Breaking it down a bit further:

“Streaming”: Lots of messages (think tens or hundreds of thousands) being sent frequently by publishers ("producers"). Message polling occurring frequently by lots of subscribers ("consumers").

“Message”: From a technical standpoint, a key value pair. From a non-technical standpoint, a relatively small number of bytes (think hundreds to a few thousand bytes).

If this isn't your planned use case, Kafka may not be the solution you are looking for. Contact your favorite Cloudera representative to discuss and find out. It is better to understand what you can and cannot do upfront than to go ahead based on some enthusiastic arbitrary vendor message with a solution that will not meet your expectations in the end.

What is Kafka designed for?

Kafka was designed at LinkedIn to be a horizontally scaling publish-subscribe system. It offers a great deal of configurability at the system- and message-level to achieve these performance goals. There are well documented cases that showcase how well Kafka can scale when everything is done right. One such example is [LinkedIn](#).

What is Kafka not well fitted for (or what are the tradeoffs)?

It's very easy to get caught up in all the things that Kafka can be used for without considering the tradeoffs. Kafka configuration is also not automatic. You need to understand each of your use cases to determine which configuration properties can be used to tune (and retune!) Kafka for each use case.

Some more specific examples where you need to be deeply knowledgeable and careful when configuring are:

- Using Kafka as your microservices communication hub
Kafka can replace both the message queue and the services discovery part of your software infrastructure. However, this is generally at the cost of some added latency as well as the need to monitor a new complex system (i.e. your Kafka cluster).
- Using Kafka as long-term storage
While Kafka does have a way to configure message retention, it's primarily designed for low latency message delivery. Kafka does not have any support for the features that are usually associated with filesystems (such as metadata or backups). As such, using some form of long-term ingestion, such as HDFS, is recommended instead.
- Using Kafka as an end-to-end solution
Kafka is only part of a solution. There are a lot of best practices to follow and support tools to build before you can get the most out of it (see this wise [LinkedIn post](#)).
- Deploying Kafka without the right support
Uber has given some numbers for their engineering organization. These numbers could help give you an idea what it takes to reach that kind of scale: [1300 microservers](#), [2000 engineers](#).

Where can I get a general Kafka overview?

Read [Kafka Introduction](#) and [Kafka Architecture](#), which cover the basics and design of Kafka. This should serve as a good starting point. If you have any remaining questions, come to this FAQ or talk to your favorite Cloudera representative about training or a best practices deep dive.

Where does Kafka fit well into an Analytic Database solution?

Analytic Database deployments benefit from Kafka by utilizing it for data ingest. Data can then populate tables for various analytics workloads. For ad hoc BI the real-time aspect is less critical, but the ability to utilize the same data used in real time applications, in BI and analytics as well, is a benefit that Cloudera's platform provides, as you will have Kafka for both purposes, already integrated, secured, governed and centrally managed.

Where does Kafka fit well into an Operational Database solution?

Kafka is commonly used in the real-time, mission-critical world of Operational Database deployments. It is used to ingest data and allow immediate serving to other applications and services through Kudu or HBase. The benefit of utilizing Kafka in the Cloudera platform for Operational Database is the integration, security, governance and central management. You avoid the risks and costs of siloed architecture and "yet another solution" to support.

What is a Kafka consumer?

If Kafka is the system that stores messages, then a consumer is the part of your system that reads those messages from Kafka.

While Kafka does come with a command line tool that can act as a consumer, practically speaking, you will most likely write Java code using the `KafkaConsumer` API for your production system.

What is a Kafka producer?

While consumers read from a Kafka cluster, producers write to a Kafka cluster.

Similar to the consumer (see previous question), your producer is also custom Java code for your particular use case.

Your producer may need some tuning for write performance and SLA guarantees, but will generally be simpler (fewer error cases) to tune than your consumer.

What functionality can I call in my Kafka Java code?

The best way to get more information on what functionality you can call in your Kafka Java code is to look at the Java documents. And read very carefully!

What's a good size of a Kafka record if I care about performance and stability?

There is an older blog post from 2014 from LinkedIn titled: [Benchmarking Apache Kafka: 2 Million Writes Per Second \(On Three Cheap Machines\)](#). In the “Effect of Message Size” section, you can see two charts which indicate that Kafka throughput starts being affected at a record size of 100 bytes through 1000 bytes and bottoming out around 10000 bytes. In general, keeping topics specific and keeping message sizes deliberately small helps you get the most out of Kafka.

Excerpting from [Deploying Apache Kafka: A Practical FAQ](#):

How to send large messages or payloads through Kafka?

Cloudera benchmarks indicate that Kafka reaches maximum throughput with message sizes of around 10 KB. Larger messages show decreased throughput. However, in certain cases, users need to send messages much larger than 10 KB.

If the message payload sizes are in the order of 100s of MB, consider exploring the following alternatives:

- If shared storage is available (HDFS, S3, NAS), place the large payload on shared storage and use Kafka just to send a message with the payload location.
- Handle large messages by chopping them into smaller parts before writing into Kafka, using a message key to make sure all the parts are written to the same partition so that they are consumed by the same Consumer, and re-assembling the large message from its parts when consuming.

Where can I get Kafka training?

Cloudera Educational Services offers various courses on Kafka. For a basic training that gives an overview of Kafka, its architecture, messages, producers and consumers (clients), as well as command line tools, see [Apache Kafka Basics](#). Additionally, if you are looking for more in depth training on Kafka, Kafka security, Kafka Connect, Streams Messaging Manager, and so on, search for *Kafka* in the course catalog on <http://education.cloudera.com>.

Use cases

A collection of frequently asked questions on the topic of Kafka aimed for advanced users.

Like most Open Source projects, Kafka provides a lot of configuration options to maximize performance. In some cases, it is not obvious how best to map your specific use case to those configuration options. We attempt to address some of those situations.

What can I do to ensure that I never lose a Kafka event?

This is a simple question which has lots of far-reaching implications for your entire Kafka setup. A complete answer includes the next few related FAQs and their answers.

What is the recommended node hardware for best reliability?

Operationally, you need to make sure your Kafka cluster meets the following hardware setup:

- Have a 3 or 5 node cluster only running Zookeeper (higher only necessary at largest scales).
- Have at least a 3 node cluster only running Kafka.
- Have the disks on the Kafka cluster running in RAID 10. (Required for resiliency against disk failure.)
- Have sufficient memory for both the Kafka and Zookeeper roles in the cluster. (Recommended: 4GB for the broker, the rest of memory automatically used by the kernel as file cache.)
- Have sufficient disk space on the Kafka cluster.
- Have a sufficient number of disks to handle the bandwidth requirements for Kafka and Zookeeper.
- You need a number of nodes greater than or equal to the highest replication factor you expect to use.

What are the network requirements for best reliability?

Kafka expects a reliable, low-latency connection between the brokers and the Zookeeper nodes:

- The number of network hops between the Kafka cluster and the Zookeeper cluster is relatively low.
- Have highly reliable network services (such as DNS).

What are the system software requirements for best reliability?

Assuming you're following the recommendations of the previous two questions, the actual system outside of Kafka must be configured properly.

1. The kernel must be configured for maximum I/O usage that Kafka requires.
 - a. Large page cache
 - b. Maximum file descriptions
 - c. Maximum file memory map limits
2. Kafka JVM configuration settings:
 - a. Brokers generally don't need more than 4GB-8GB of heap space.
 - b. Run with the +G1GC garbage collection using Java 8 or later.

How can I configure Kafka to ensure that events are stored reliably?

The following recommendations for Kafka configuration settings make it extremely difficult for data loss to occur.

- Producer
 - `block.on.buffer.full=true`
 - `retries=Long.MAX_VALUE`
 - `acks=all`
 - `max.in.flight.requests.per.connections=1`
 - Remember to close the producer when it is finished or when there is a long pause.
- Broker
 - `Topic replication.factor >= 3`
 - `Min.insync.replicas = 2`
 - Disable unclean leader election
- Consumer
 - Disable `enable.auto.commit`
 - Commit offsets after messages are processed by your consumer client(s).

If you have more than 3 hosts, you can increase the broker settings appropriately on topics that need more protection against data loss.

Once I've followed all the previous recommendations, my cluster should never lose data, right?

Kafka does not ensure that data loss never occurs. There are the following tradeoffs:

- Throughput vs. reliability. For example, the higher the replication factor, the more resilient your setup will be against data loss. However, to make those extra copies takes time and can affect throughput.
- Reliability vs. free disk space. Extra copies due to replication use up disk space that would otherwise be used for storing events.

Beyond the above design tradeoffs, there are also the following issues:

- To ensure events are consumed you need to monitor your Kafka brokers and topics to verify sufficient consumption rates are sustained to meet your ingestion requirements.
- Ensure that replication is enabled on any topic that requires consumption guarantees. This protects against Kafka broker failure and host failure.
- Kafka is designed to store events for a defined duration after which the events are deleted. You can increase the duration that events are retained up to the amount of supporting storage space.
- You will always run out of disk space unless you add more nodes to the cluster.

My Kafka events must be processed in order. How can I accomplish this?

After your topic is configured with partitions, Kafka sends each record (based on key/value pair) to a particular partition based on key. So, any given key, the corresponding records are “in order” within a partition.

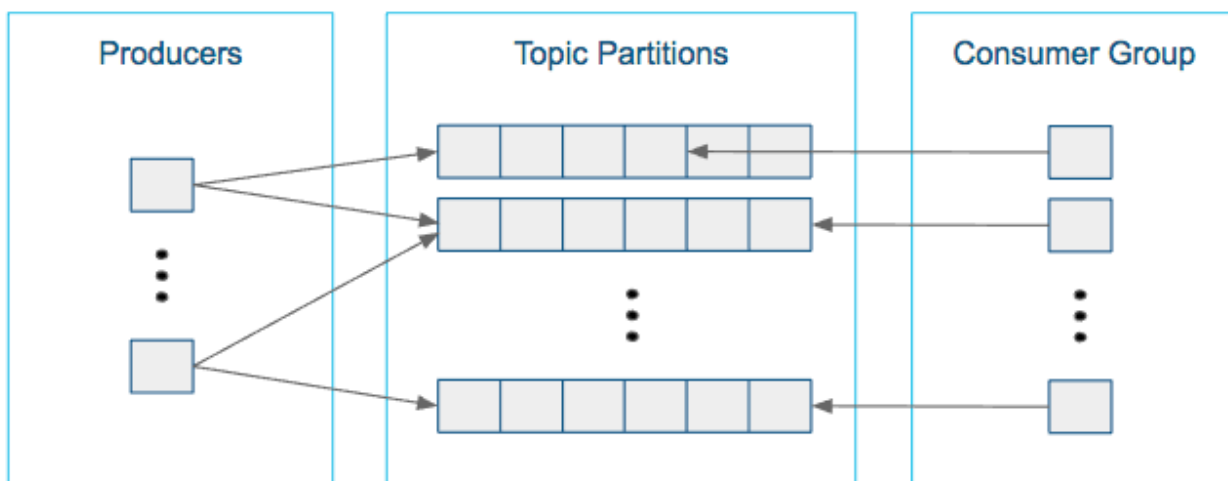
For global ordering, you have two options:

- Your topic must consist of one partition (but a higher replication factor could be useful for redundancy and failover). However, this will result in very limited message throughput.
- You configure your topic with a small number of partitions and perform the ordering after the consumer has pulled data. This does not result in guaranteed ordering, but, given a large enough time window, will likely be equivalent.

Conversely, it is best to take Kafka's partitioning design into consideration when designing your Kafka setup rather than rely on global ordering of events.

How do I size my topic? Alternatively: What is the “right” number of partitions for a topic?

Choosing the proper number of partitions for a topic is the key to achieve a high degree of parallelism with respect to writes and reads and to distribute load. Evenly distributed load over partitions is a key factor to have good throughput (avoid hot spots). Making a good decision requires estimation based on the desired throughput of producers and consumers per partition.



For example, if you want to be able to read 1 GB/sec, but your consumer is only able to process 50 MB/sec, then you need at least 20 partitions and 20 consumers in the consumer group. Similarly, if you want to achieve the same for producers, and 1 producer can only write at 100 MB/sec, you need 10 partitions. In this case, if you have 20 partitions, you can maintain 1 GB/sec for producing and consuming messages. You should adjust the exact number of partitions to number of consumers or producers, so that each consumer and producer achieve their target throughput.

So a simple formula could be:

```
#Partitions = max(NP, NC)
```

where:

- NP is the number of required producers determined by calculating: TT/TP
- NC is the number of required consumers determined by calculating: TT/TC
- TT is the total expected throughput for our system
- TP is the max throughput of a single producer to a single partition
- TC is the max throughput of a single consumer from a single partition

This calculation gives you a rough indication of the number of partitions. It's a good place to start. Keep in mind the following considerations for improving the number of partitions after you have your system in place:

- The number of partitions can be specified at topic creation time or later.
- Increasing the number of partitions also affects the number of open file descriptors. So make sure you set file descriptor limit properly.
- Reassigning partitions can be very expensive, and therefore it's better to over- than under-provision.
- Changing the number of partitions that are based on keys is challenging and involves manual copying.
- Reducing the number of partitions is not currently supported. Instead, create a new a topic with a lower number of partitions and copy over existing data.
- Metadata about partitions are stored in ZooKeeper in the form of znodes. Having a large number of partitions has effects on ZooKeeper and on client resources:
 - Unneeded partitions put extra pressure on ZooKeeper (more network requests), and might introduce delay in controller and/or partition leader election if a broker goes down.
 - Producer and consumer clients need more memory, because they need to keep track of more partitions and also buffer data for all partitions.
- As guideline for optimal performance, you should not have more than 4000 partitions per broker and not more than 200,000 partitions in a cluster.

Make sure consumers don't lag behind producers by monitoring consumer lag. To check consumers' position in a consumer group (that is, how far behind the end of the log they are), use the following command:

```
$ kafka-consumer-groups --bootstrap-server BROKER_ADDRESS --describe --group CONSUMER_GROUP --new-consumer
```

How can I scale a topic that's already deployed in production?

Recall the following facts about Kafka:

- When you create a topic, you set the number of partitions. The higher the partition count, the better the parallelism and the better the events are spread somewhat evenly through the cluster.
- In most cases, as events go to the Kafka cluster, events with the same key go to the same partition. This is a consequence of using a hash function to determine which key goes to which partition.

Now, you might assume that scaling means increasing the number of partitions in a topic. However, due to the way hashing works, simply increasing the number of partitions means that you will lose the "events with the same key go to the same partition" fact.

Given that, there are two options:

1. Your cluster may not be scaling well because the partition loads are not balanced properly (for example, one broker has four very active partitions, while another has none). In those cases, you can use the `kafka-reassign-partitions` script to manually balance partitions.
2. Create a new topic with more partitions, pause the producers, copy data over from the old topic, and then move the producers and consumers over to the new topic. This can be a bit tricky operationally.

How do I rebalance my Kafka cluster?

This one comes up when new nodes or disks are added to existing nodes. Partitions are not automatically balanced. If a topic already has a number of nodes equal to the replication factor (typically 3), then adding disks does not help with rebalancing.

Using the `kafka-reassign-partitions` command after adding new hosts is the recommended method.

Caveats

There are several caveats to using this command:

- It is highly recommended that you minimize the volume of replica changes to make sure the cluster remains healthy. Say, instead of moving ten replicas with a single command, move two at a time.
- It is not possible to use this command to make an out-of-sync replica into the leader partition.
- If too many replicas are moved, then there could be serious performance impact on the cluster. When using the `kafka-reassign-partitions` command, look at the partition counts and sizes. From there, you can test various partition sizes along with the `--throttle` flag to determine what volume of data can be copied without affecting broker performance significantly.
- Given the earlier restrictions, it is best to use this command only when all brokers and topics are healthy.

How do I monitor my Kafka cluster?

Kafka in CDP can be monitored and managed using Streams Messaging Manager (SMM). SMM is an operations monitoring and management tool that provides end-to-end visibility in an enterprise Apache Kafka environment. For more information, see [Introduction to Streams Messaging Manager](#) or the various SMM publications available in [How to>Streams Messaging](#).

What are the best practices concerning consumer group.id?

The `group.id` is just a string that helps Kafka track which consumers are related (by having the same group id).

- In general, timestamps as part of `group.id` are not useful. Because each `group.id` corresponds to multiple consumers, you cannot have a unique timestamp for each consumer.

- Add any helpful identifiers. This could be related to a group (for example, transactions, marketing), purpose (fraud, alerts), or technology (Spark).

How do I monitor consumer group lag?

This is typically done using the `kafka-consumer-groups` command line tool. Copying directly from the [upstream documentation](#), we have this example output (reformatted for readability):

```
$ bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe
--group my-group
TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID     HOST
CLIENT-ID
my-topic 0          2          4          2 consumer-1-69d6 /127.0.0.1
consumer-1
my-topic 1          2          3          1 consumer-1-69d6 /127.0.0.1
consumer-1
my-topic 2          2          3          1 consumer-2-9bb2 /127.0.
0.1 consumer-2
```

In general, if everything is going well with a particular topic, each consumer's `CURRENT-OFFSET` should be up-to-date or nearly up-to-date with the `LOG-END-OFFSET`. From this command, you can determine whether a particular host or a particular partition is having issues keeping up with the data rate.

How do I reset the consumer offset to an arbitrary value?

This is also done using the `kafka-consumer-groups` command line tool. This is generally an administration feature used to get around corrupted records, data loss, or recovering from failure of the broker or host. Aside from those special cases, using the command line tool for this purpose is not recommended.

By using the `--execute --reset-offsets` flags, you can change the consumer offsets for a consumer group (or even all groups) to a specific setting based on each partitions log's beginning/end or a fixed timestamp. Typing the `kafka-consumer-groups` command with no arguments will give you the complete help output.

How do I configure MirrorMaker for bidirectional replication across DCs?

Mirror Maker is a one way copy of one or more topics from a Source Kafka Cluster to a Destination Kafka Cluster. Given this restriction on Mirror Maker, you need to run two instances, one to copy from A to B and another to copy from B to A.

In addition, consider the following:

- Cloudera recommends using the "pull" model for Mirror Maker, meaning that the Mirror Maker instance that is writing to the destination is running on a host "near" the destination cluster.
- The topics must be unique across the two clusters being copied.
- On secure clusters, the source cluster and destination cluster must be in the same Kerberos realm.

How does the consumer max retries vs timeout work?

With the newer versions of Kafka, consumers have two ways they communicate with brokers.

- Retries: This is generally related to reading data. When a consumer reads from a brokers, it's possible for that attempt to fail due to problems such as intermittent network outages or I/O issues on the broker. To improve reliability, the consumer retries (up to the configured `max.retries` value) before actually failing to read a log offset.
- Timeout. This term is a bit vague because there are two timeouts related to consumers:
 - Poll Timeout: This is the timeout between calls to `KafkaConsumer.poll()`. This timeout is set based on whatever read latency requirements your particular use case needs.
 - Heartbeat Timeout: The newer consumer has a "heartbeat thread" which give a heartbeat to the broker (actually the Group Coordinator within a broker) to let the broker know that the consumer is still alive. This

happens on a regular basis and if the broker doesn't receive at least one heartbeat within the timeout period, it assumes the consumer is dead and disconnects it.

How do I size my Kafka cluster?

There are several considerations for sizing your Kafka cluster.

- Disk space

Disk space will primarily consist of your Kafka data and broker logs. When in debug mode, the broker logs can get quite large (10s to 100s of GB), so reserving a significant amount of space could save you some future headaches.

For Kafka data, you need to perform estimates on message size, number of topics, and redundancy. Also remember that you will be using RAID10 for Kafka's data, so half your hard drives will go towards redundancy. From there, you can calculate how many drives will be needed.

In general, you will want to have more hosts than the minimum suggested by the number of drives. This leaves room for growth and some scalability headroom.

- Zookeeper nodes

One node is fine for a test cluster. Three is standard for most Kafka clusters. At large scale, five nodes is fairly common for reliability.

- Looking at leader partition count/bandwidth usage

This is likely the metric with the highest variability. Any Kafka broker will be overloaded if it has too many leader partitions. In the worst cases, each leader partition requires high bandwidth, high message rates, or both. For other topics, leader partitions will be a tiny fraction of what a broker can handle (limited by software and hardware). To estimate an average that works on a per-host basis, try grouping topics by partition data throughput requirements, such as 2 high bandwidth data partitions, 4 medium bandwidth data partitions, 20 small bandwidth data partitions. From there, you can determine how many hosts are needed.

How can I build a Spark streaming application that consumes data from Kafka?

You will need to set up your development environment to use both Spark libraries and Kafka libraries:

- [Building Spark Applications](#)
- The [kafka-examples](#) directory on Cloudera's public GitHub has an example pom.xml.

From there, you should be able to read data using the `KafkaConsumer` class and using Spark libraries for real-time data processing. The blog post [Reading data securely from Apache Kafka to Apache Spark](#) has a pointer to a GitHub repository that contains a word count example.

For further background, read the blog post [Architectural Patterns for Near Real-Time Data Processing with Apache Hadoop](#).