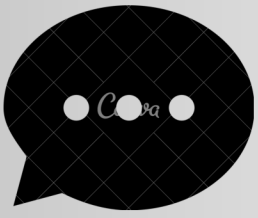
Kafka Overview

Like



Comment

Repost

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:

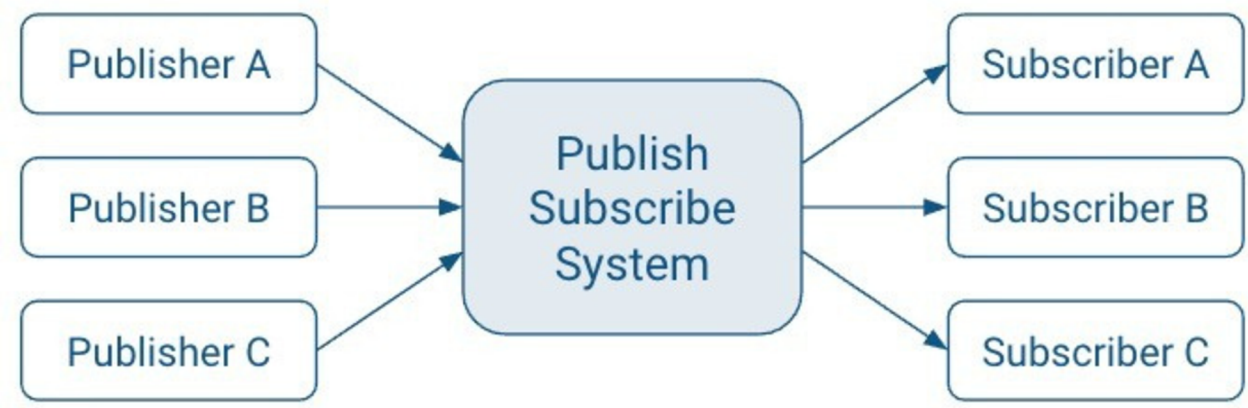
- Persistent messaging with O(1) disk structures, meaning that the execution time of Kafka's algorithms is independent of the size of the input. Execution time is constant, even with terabytes of stored messages.
- High throughput, supporting hundreds of thousands of messages per second, even with modest 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.

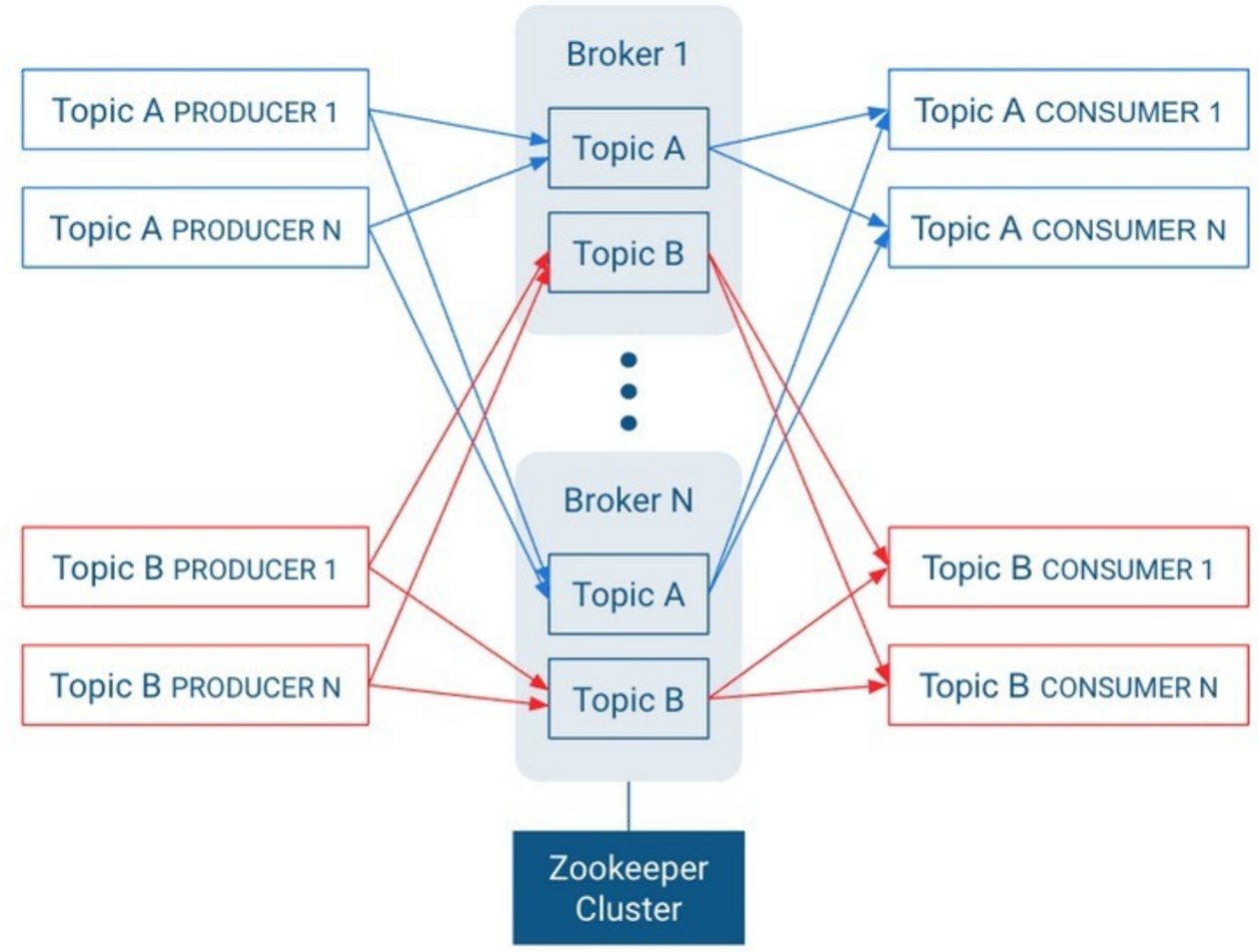
Continue reading to learn more about each key concept.

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.

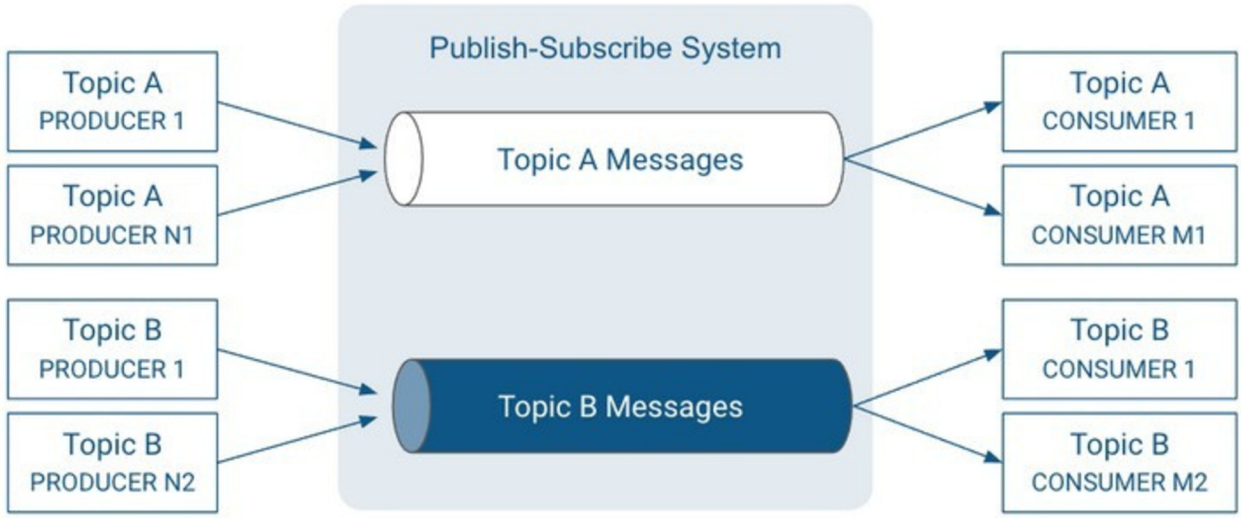
Topics

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.)

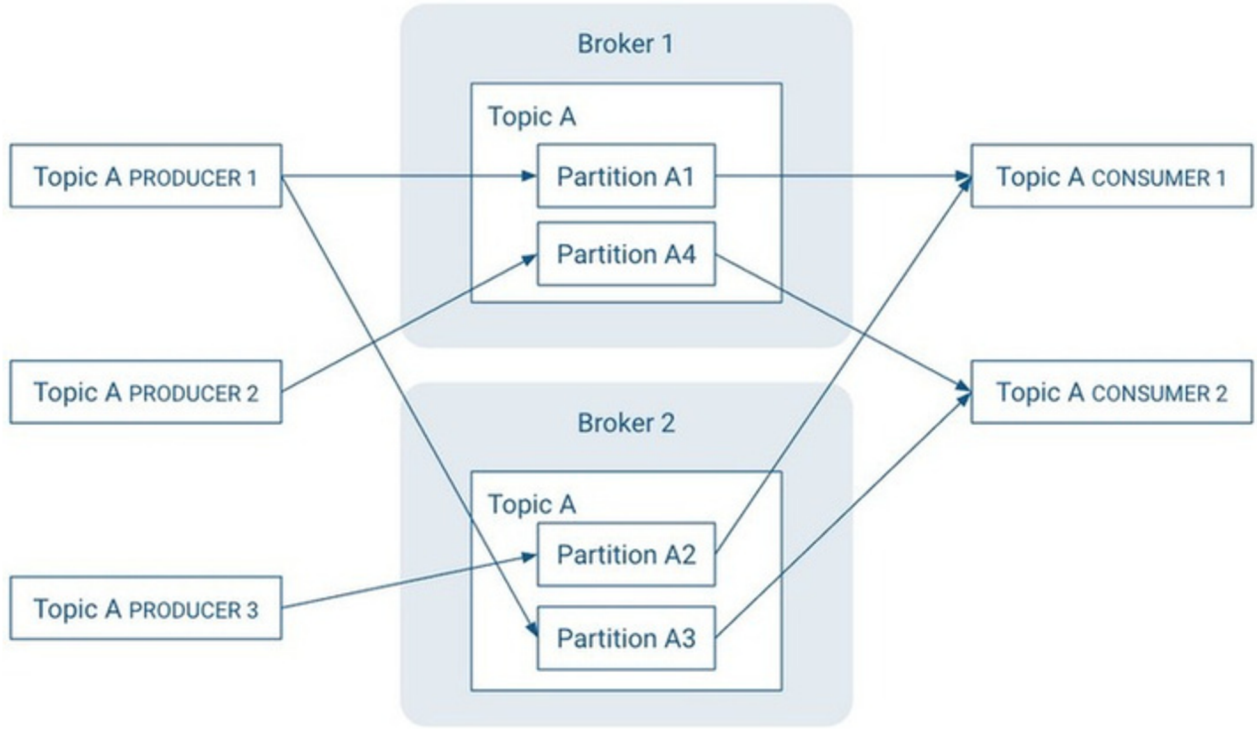
Any Kafka client (a producer or consumer) communicates only with the leader partition for data. All other partitions exist for redundancy and failover. Follower partitions are responsible for copying new records from their leader partitions. Ideally, the follower partitions have an exact copy of the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

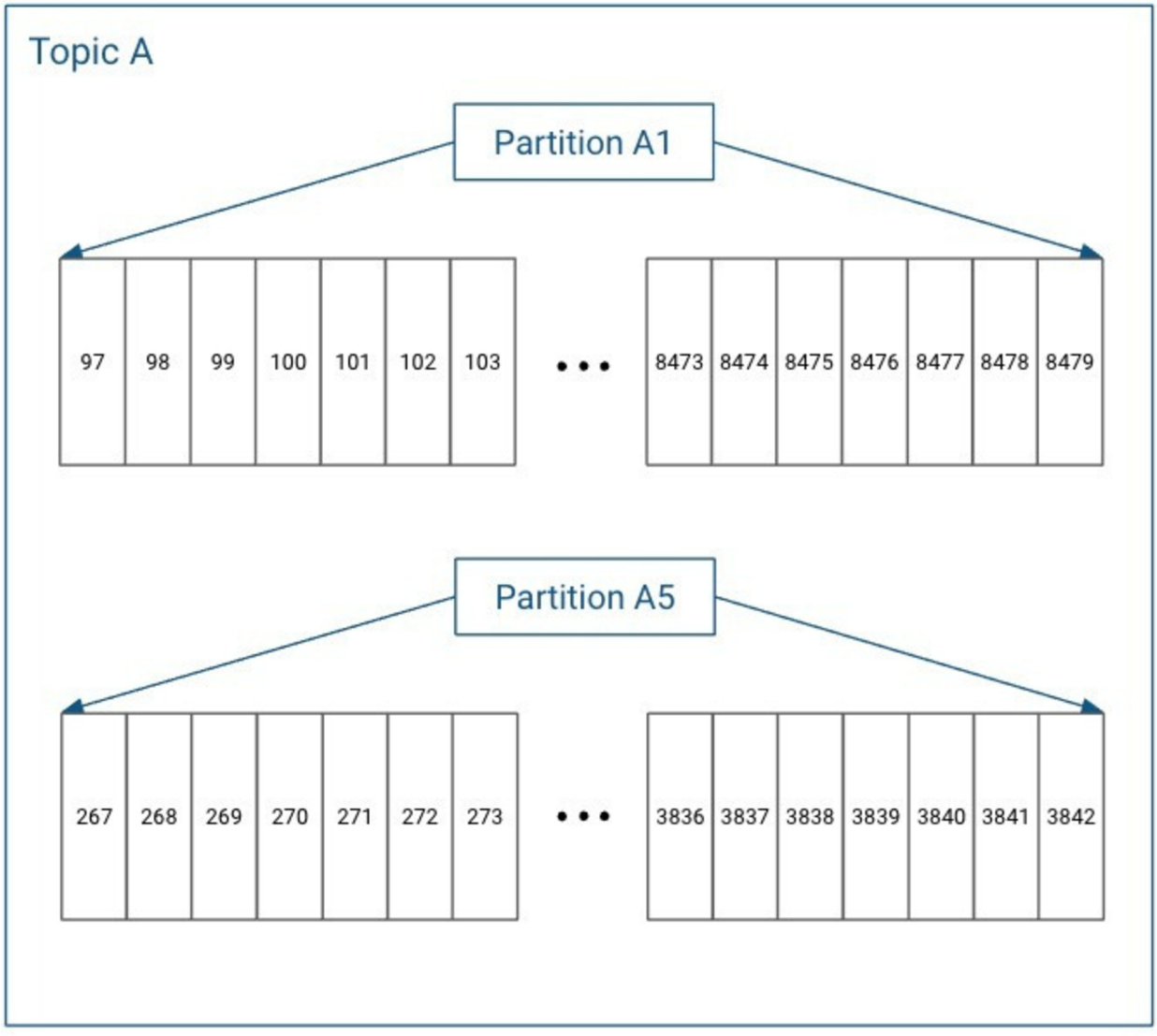
- The record can indicate a specific partition.
- The record can includes an assignment key.
  The hash of the key and the number of partitions in the topic determines which partition the record is assigned to. Including the same key in multiple records ensures all the records are appended to the same 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

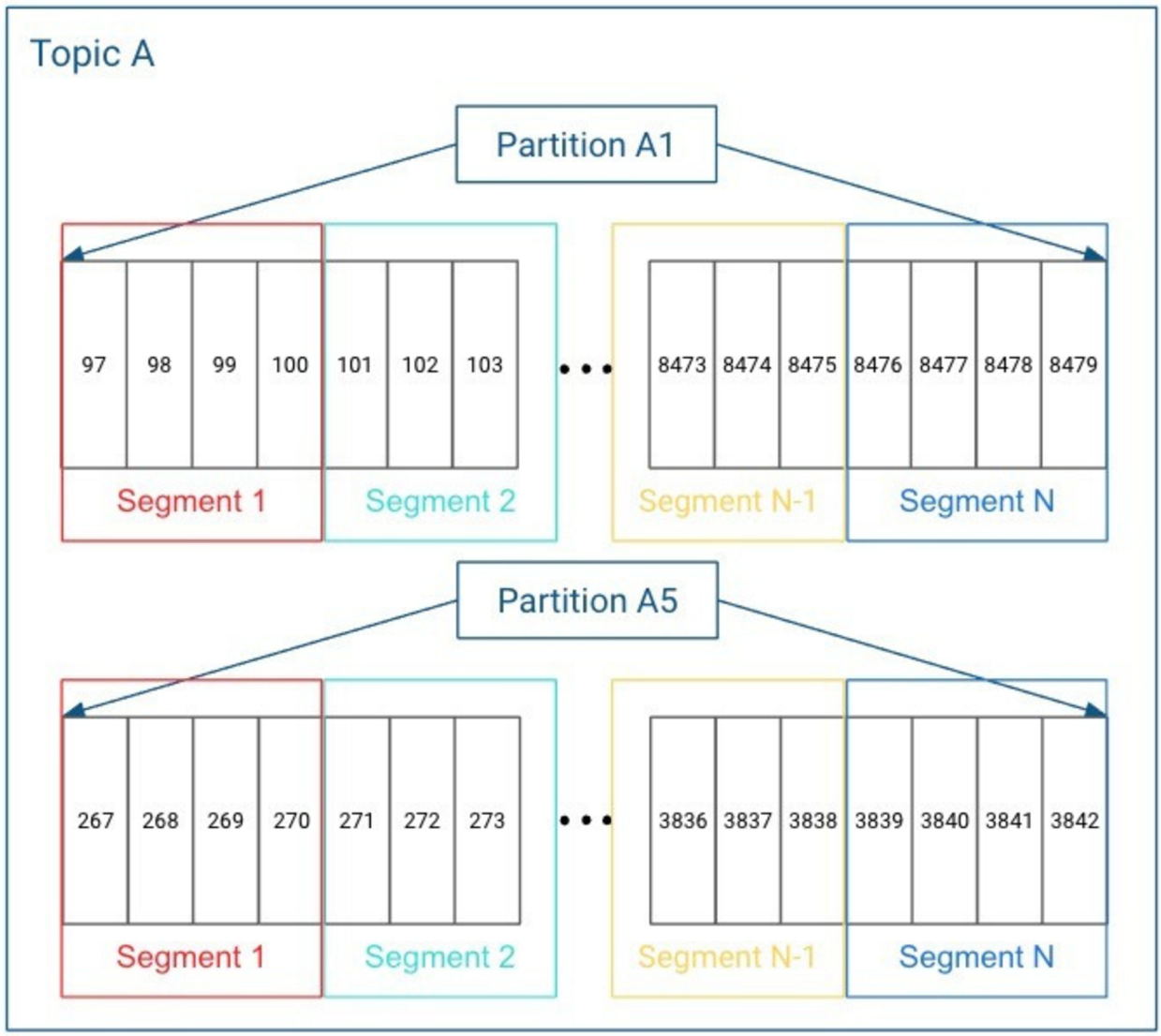


Note: These references to “log” should not be confused with where the Kafka broker stores their operational logs.

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

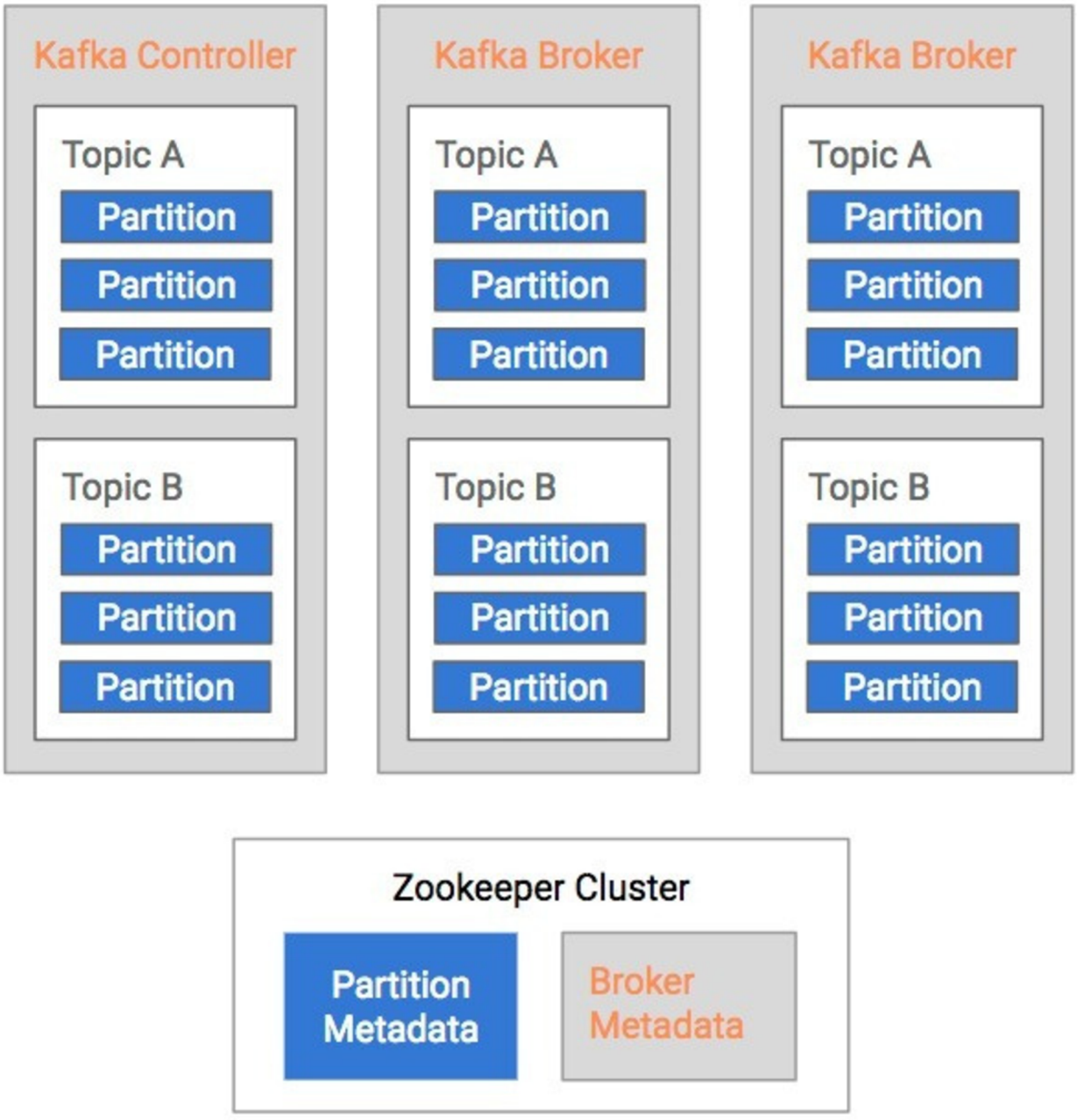
Learn about the types of data maintained in Zookeeper by the brokers.

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

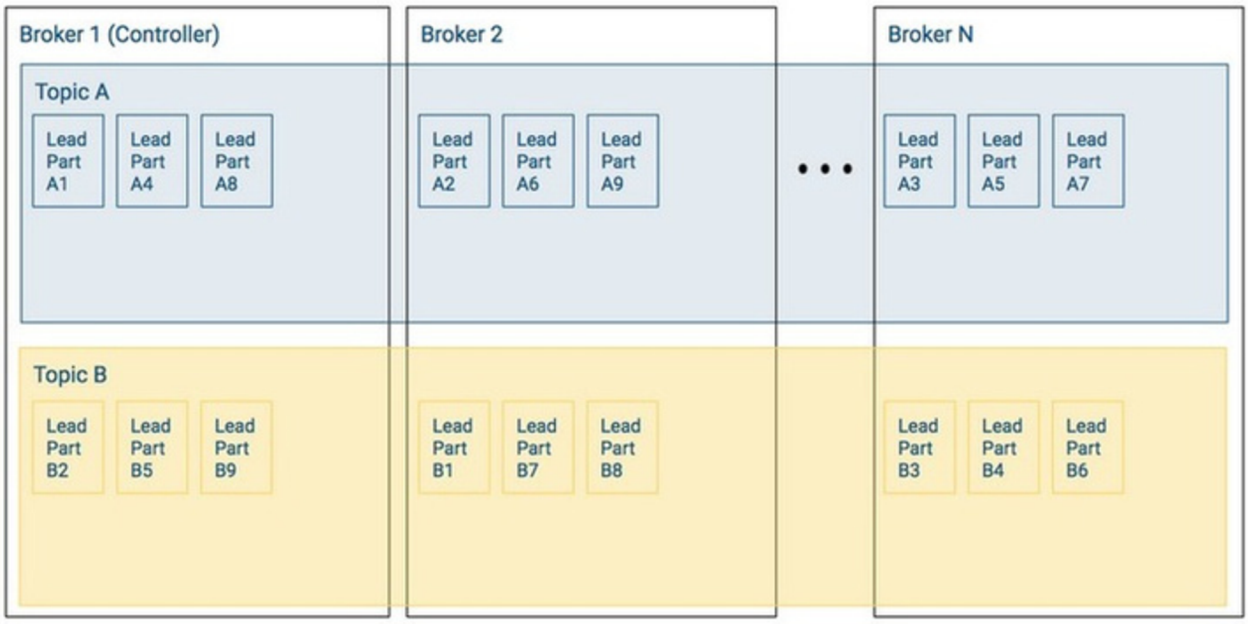


Leader positions and in-sync replicas

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:

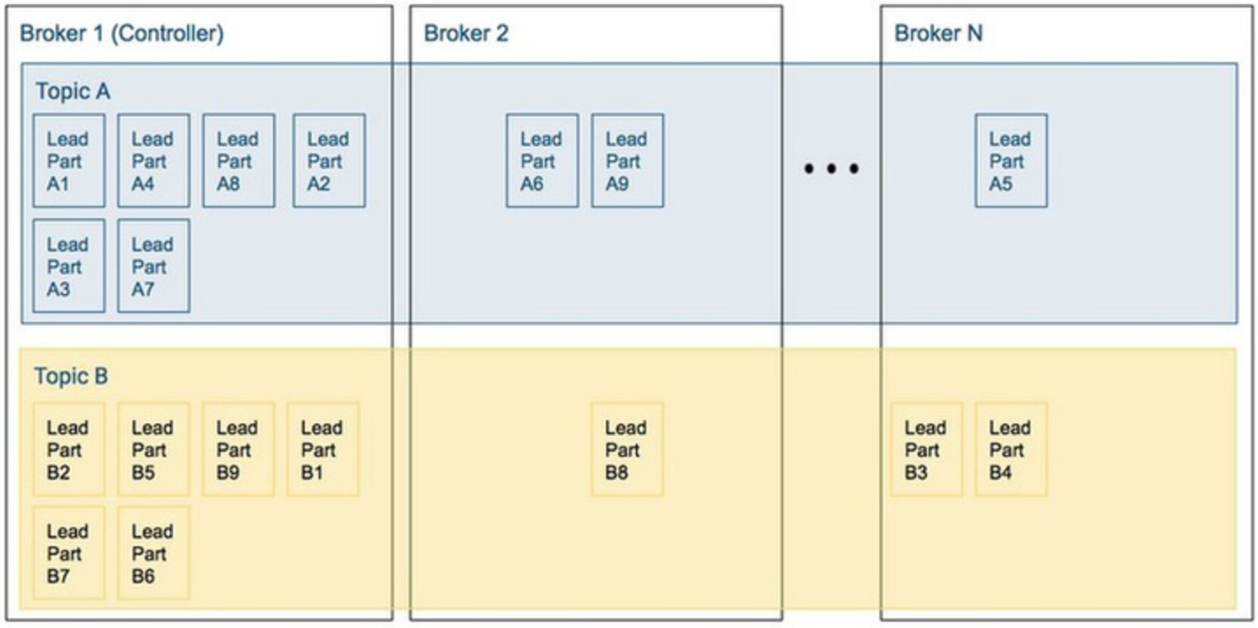
- Producer writes and consumer reads occur at the partition level.
- Leader partitions are responsible for ensuring that the follower partitions keep their records in sync.

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



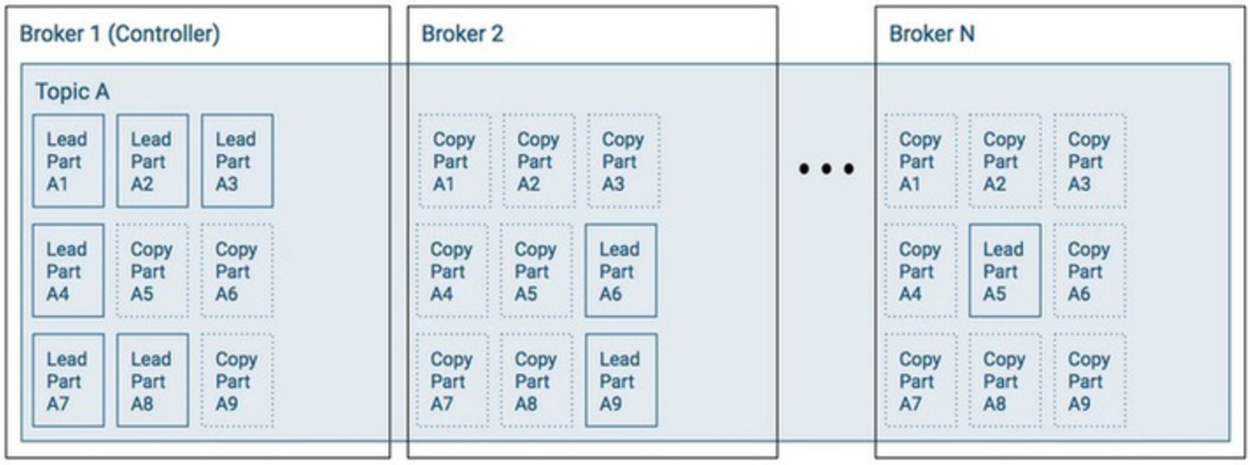
In a scenario like this a lot more of the overall Kafka workload occurs on Broker 1. Consequently this also causes a backlog of work, which slows down the cluster throughput, which will worsen the backlog. Even if a cluster starts with perfectly balanced topics, failures of brokers can cause these imbalances: 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 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 (Uber and LinkedIn) that showcase how well Kafka can scale when everything is done right.

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 fro 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.

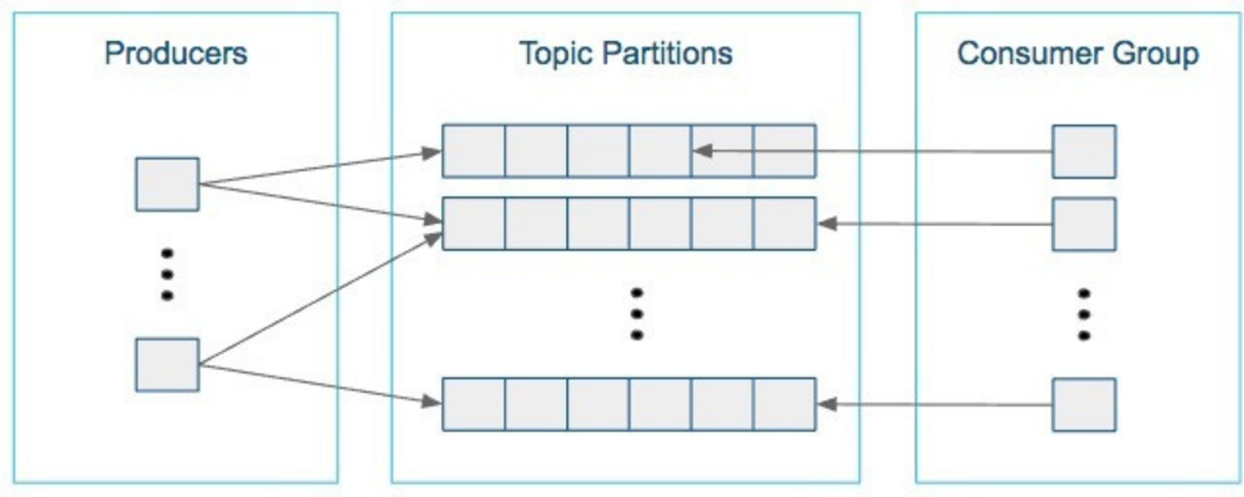
For global ordering, you have two options:

- 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 --gr
oup 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?

Cloudera Manager has monitoring for a Kafka cluster.

Currently, there are three GitHub projects as well that provide additional monitoring functionality:

- Doctor Kafka (Pinterest, Apache 2.0 License)
- Kafka Manager (Yahoo, Apache 2.0 License)
- Cruise Control (LinkedIn, BSD 2-clause License)

These projects are Apache-compatible licensed, but are not Open Source (no community, bug filing, or transparency).

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 (Flume, 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   1
 consumer-1
my-topic       1               2              3              consumer-1-69d6     /127.0.0.1     1
my-topic       2               2              3              consumer-2-9bb2 /127.0.
 consumer-1
my-topic
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 bi-directional 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 tend to be quite large (10s to 100s of GB), so reserving a significant amount of space could save you some headaches. For Kafka data, you need to perform estimates on message size, number of topics, and redundancy. Also 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 topics, leader partitions will be a tiny fraction of what a broker can handle (limited by software and hardware). To

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

We shuacvhe atws o2 bhlioggh pboasntsd wonid uths indga tKaa pfkaart witiiothn sF,l u4m mee:dium bandwidth data

- This updated version for CDH 5.8/Apache Kafka 0.9/Apache Flume 1.7: New in Cloudera Enterprise 5.8: Flafka Improvements for Real-Time Data Ingest

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.