How to Choose the Number of Topics/Partitions in a Kafka Cluster?

READ TIME: 7 MIN

Note

For the latest, check out the blog posts Apache Kafka® Made Simple: A First Glimpse of a Kafka Without ZooKeeper and Apache Kafka Supports 200K Partitions Per Cluster.

This is a common question asked by many Kafka users. The goal of this post is to explain a few important determining factors and provide a few simple formulas for when you are self-managing your Kafka clusters. If you are using Confluent Cloud, most of these operational concerns are taken care of by us here at Confluent.

More partitions lead to higher throughput

The first thing to understand is that a topic partition is the unit of parallelism in Kafka. On both the producer and the broker side, writes to different partitions can be done fully in parallel. So expensive operations such as compression can utilize more hardware resources. On the consumer side, Kafka always gives a single partition's data to one consumer thread. Thus, the degree of parallelism in the consumer (within a consumer group) is bounded by the number of partitions being consumed. Therefore, in general, the more partitions there are in a Kafka cluster, the higher the throughput one can achieve.

A rough formula for picking the number of partitions is based on throughput. You measure the throughout that you can achieve on a single partition for production (call it p) and consumption (call it c). Let's say your target throughput is t. Then you need to have at least max(t/p, t/c) partitions. The per-partition throughput that one can achieve on the producer depends on configurations such as the batching size, compression codec, type of acknowledgement, replication factor, etc. However, in general, one can produce at 10s of MB/sec on just a single partition as shown in this benchmark. The consumer throughput is often application dependent since it corresponds to how fast the consumer logic can process each message. So, you really need to measure it.

Although it's possible to increase the number of partitions over time, one has to be careful if messages are produced with keys. When publishing a keyed message, Kafka deterministically maps the message to a partition based on the hash of the key. This provides a guarantee that messages with the same key are always routed to the same partition. This guarantee can be important for certain applications since messages within a partition are always delivered in order to the consumer. If the number of partitions changes, such a guarantee may no longer hold. To avoid this situation, a common practice is to over-partition a bit. Basically, you determine the number of partitions based on a future target throughput, say for one or two years later. Initially, you can just have a small Kafka cluster based on your current throughput. Over time, you can add more brokers to the cluster and proportionally move a subset of the existing partitions to the new brokers (which can be done online). This way, you can keep up with the throughput growth without breaking the semantics in the application when keys are used.

In addition to throughput, there are a few other factors that are worth considering when choosing the number of partitions. As you will see, in some cases, having too many partitions may also have negative impact.

More partitions requires more open file handles

Each partition maps to a directory in the file system in the broker. Within that log directory, there will be two files (one for the index and another for the actual data) per log segment. Currently, in Kafka, each broker opens a file handle of both the index and the data file of every log segment. So, the more partitions, the higher that one needs to configure the open file handle limit in the underlying operating system. This is mostly just a configuration issue. We have seen production Kafka clusters running with more than 30 thousand open file handles per broker.

More partitions may increase unavailability

Kafka supports intra-cluster replication, which provides higher availability and durability. A partition can have multiple replicas, each stored on a different broker. One of the replicas is designated as the leader and the rest of the replicas are followers. Internally, Kafka manages all those replicas automatically and makes sure that they are kept in sync. Both the producer and the consumer requests to a partition are served on the leader replica. When a broker fails, partitions with a leader on that broker become temporarily unavailable. Kafka will automatically move the leader of those unavailable partitions to some other replicas to continue serving the client requests. This process is done by one of the Kafka brokers designated as the controller. It involves reading and writing some metadata for each affected partition in ZooKeeper. Currently, operations to ZooKeeper are done serially in the controller.

In the common case when a broker is shut down cleanly, the controller will proactively move the leaders off the shutting down broker one at a time. The moving of a single leader takes only a few milliseconds. So, from the clients perspective, there is only a small window of unavailability during a clean broker shutdown.

However, when a broker is shut down uncleanly (e.g., kill -9), the observed unavailability could be proportional to the number of partitions. Suppose that a broker has a total of 2000 partitions, each with 2 replicas. Roughly, this broker will be the leader for about 1000 partitions. When this broker fails uncleanly, all those 1000 partitions become unavailable at exactly the same time. Suppose that it takes 5 ms to elect a new leader for a single partition. It will take up to 5 seconds to elect the new leader for all 1000 partitions. So, for some partitions, their observed unavailability can be 5 seconds plus the time taken to detect the failure.

If one is unlucky, the failed broker may be the controller. In this case, the process of electing the new leaders won’t start until the controller fails over to a new broker. The controller failover happens automatically but requires the new controller to read some metadata for every partition from ZooKeeper during initialization. For example, if there are 10,000 partitions in the Kafka cluster and initializing the metadata from ZooKeeper takes 2 ms per partition, this can add 20 more seconds to the unavailability window.

In general, unclean failures are rare. However, if one cares about availability in those rare cases, it’s probably better to limit the number of partitions per broker to two to four thousand and the total number of partitions in the cluster to low tens of thousand.

More partitions may increase end-to-end latency

The end-to-end latency in Kafka is defined by the time from when a message is published by the producer to when the message is read by the consumer. Kafka only exposes a message to a consumer after it has been committed, i.e., when the message is replicated to all the in-sync

replicas. So, the time to commit a message can be a significant portion of the end-to-end latency. By default, a Kafka broker only uses a single thread to replicate data from another broker, for all partitions that share replicas between the two brokers. Our experiments show that replicating 1000 partitions from one broker to another can add about 20 ms latency, which implies that the end-to-end latency is at least 20 ms. This can be too high for some real-time applications.

Assuming a replication factor of 2, note that this issue is alleviated on a larger cluster. For example, suppose that there are 1000 partition leaders on a broker and there are 10 other brokers in the same Kafka cluster. Each of the remaining 10 brokers only needs to fetch 100 partitions from the first broker on average. Therefore, the added latency due to committing a message will be just a few ms, instead of tens of ms.

As a rule of thumb, if you care about latency, it's probably a good idea to limit the number of partitions per broker to 100 x b x r, where b is the number of brokers in a Kafka cluster and r is the replication factor.

More partitions may require more memory in the client

In the most recent 0.8.2 release which we ship with the Confluent Platform 1.0, we have developed a more efficient Java producer. One of the nice features of the new producer is that it allows users to set an upper bound on the amount of memory used for buffering incoming messages. Internally, the producer buffers messages per partition. After enough data has been accumulated or enough time has passed, the accumulated messages are removed from the buffer and sent to the broker.

If one increases the number of partitions, message will be accumulated in more partitions in the producer. The aggregate amount of memory used may now exceed the configured memory limit. When this happens, the producer has to either block or drop any new message, neither of which is ideal. To prevent this from happening, one will need to reconfigure the producer with a larger memory size.

As a rule of thumb, to achieve good throughput, one should allocate at least a few tens of KB per partition being produced in the producer and adjust the total amount of memory if the number of partitions increases significantly.

A similar issue exists in the consumer as well. The consumer fetches a batch of messages per partition. The more partitions that a consumer consumes, the more memory it needs. However, this is typically only an issue for consumers that are not real time.

Summary

In general, more partitions in a Kafka cluster leads to higher throughput. However, one does have to be aware of the potential impact of having too many partitions in total or per broker on things like availability and latency. In the future, we do plan to improve some of those limitations to make Kafka more scalable in terms of the number of partitions.