Evaluation of Kafka

Let's recap how Kafka fulfills its promised functionalities.

We'll cover the following

Kafka promised to be efficient in collecting data from multiple producers in parallel, retaining data, and delivering it to multiple consumers simultaneously. Moreover, it promised to deliver loads of data in real time. Let's go through some pieces of evidence as to how Kafka provides these functionalities by comparing the performance of Kafka with Apache ActiveMQ (a popular open-source implementation of Java Message Service (JMS)) and RabbitMQ (a messaging system known for its performance).

All the computational results and time spent on them that is stated in the text below are done on two Linux machines, both of which have eight 2GHz cores, 16 GB of memory, and 6 disks with RAID 10. Both the Linux machines are connected through a 1 GB network link. One is deployed as a broker, and the other performs the function of both the producer and consumer interchangeably. Though such an experimental setup might seem minuscule, Kafka can extract amazing throughput from this setup. Since Kafka is horizontally scalable, it will not be a stretch to

extrapolate these numbers for a larger setup (for example, for back-of-the-envelope calculations).

Performance improvements

To check the improved performance of Kafka, we'll have to analyze the messages going from producer to brokers and from brokers to consumers.

Producer throughput

ActiveMQ and RabbitMQ don't have any simple way to send batched messages, so only 1 message is sent to the broker at any given time. However, if we use a single producer at each system to produce 10 million messages, each message being 200 bytes in size, and send these messages in batches of 1 to 50, Kafka can publish 50,000 to 400,000 messages per second, respectively. The results achieved by Kafka are orders of magnitude better than ActiveMQ's results and twice as better as RabbitMQ's results.

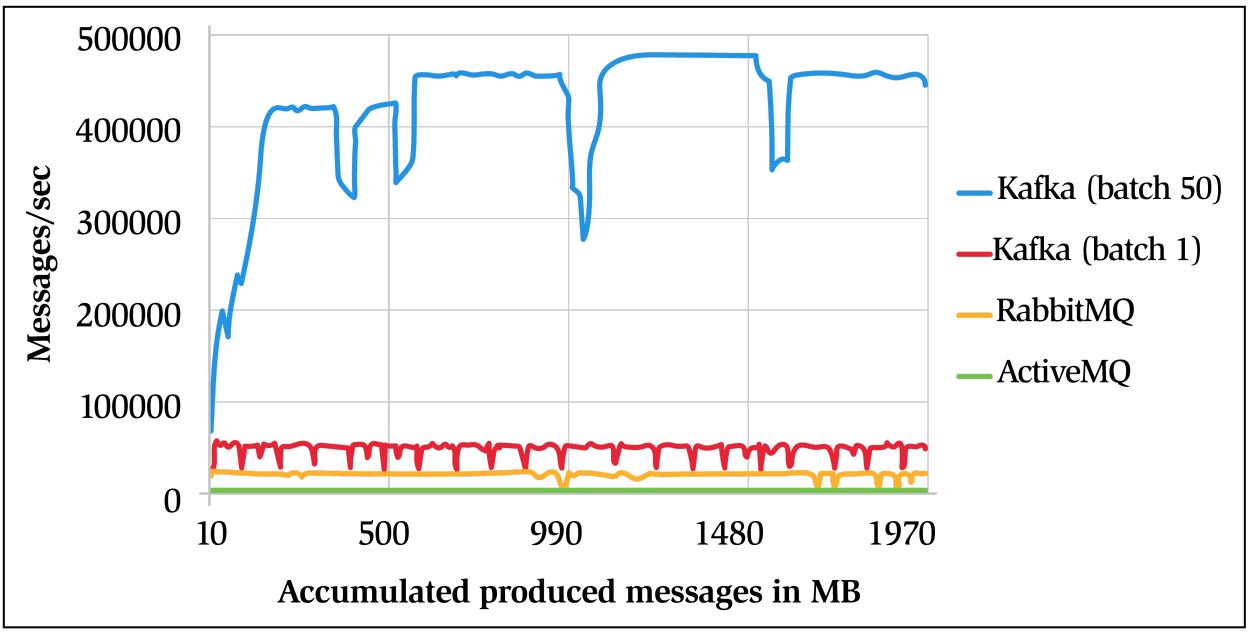
The reasons why Kafka's producer shows this improved performance are listed as follows:

- Kafka's producer sends as many messages to the broker as the broker can process without waiting for any kind of acknowledgment from it.
- Kafka possesses a simple and efficient storage system. On average, Kafka only had an overhead of 9 bytes per message as opposed to ActiveMQ's 144 bytes. ActiveMQ's overhead comes from two sources:
    - A large message header that JMS requires
    - Maintenance of indexing structures

Batch processing

Kafka's batching is the key to its achieved improvement in performance because sending a batch of messages also reduces the remote procedure call (RPC) overhead. Moreover, if the systems are far away from each other, batching will be able to make maximum use of the RTT. The improved throughput of Kafka's producer as compared to ActiveMQ and RabbitMQ and the magnitude of

improvement in batch processing adds to its performance. This can be seen in the following illustration.



Performance of the Kafka producer

If we observe the performance of a Kafka producer that is sending a batch of 50 messages, a fluctuation in its throughput can be clearly seen. There are two parameters in Kafka's producer, configurable by the users, which can cause this variance in throughput, i.e., batch size, which in this case is 50 and linger time. Kafka's producer has a buffer that collects messages until it has accumulated 50 messages to send them to the broker, which can affect the throughput if it is waiting long to collect these messages. Moreover, linger time is another parameter that decides the time for which the producer can wait before sending any messages it has collected to the broker. This variance can be minimized if these parameters are tuned carefully.

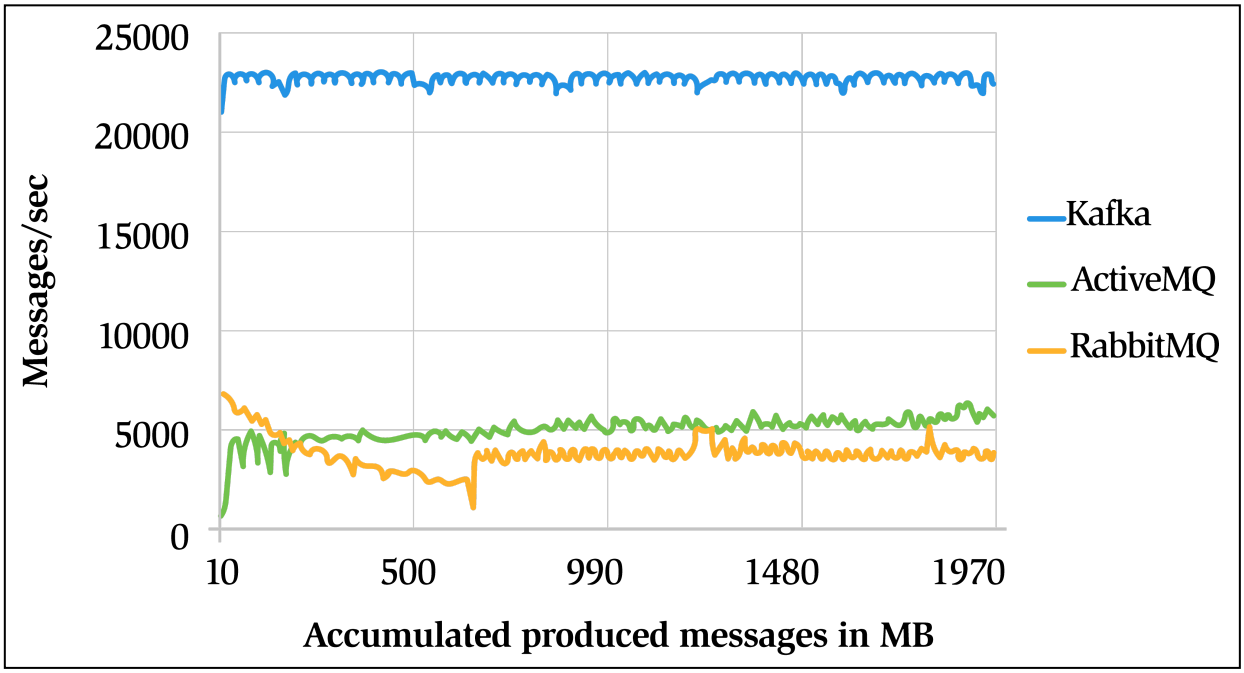
Consumer throughput

If we use a single consumer at each machine to retrieve a total of 10 million messages and configure each system to pull around 1000 messages or 200KB, Kafka outperforms ActiveMQ and RabbitMQ by 4 times by consuming 20,000 messages per second.

The reasons why Kafka's consumption is much better are listed as follows:

- Kafka brokers have very efficient storage. They send fewer bytes to the consumers.
- ActiveMQ and RabbitMQ brokers maintain the delivery state of each message.
- ActiveMQ's threads get busy writing KahaDB pages to the disk.
- Kafka brokers do not indulge in any disk writing activities because of the file systems page cache. (Messages will be delivered from the RAM, and as we read forward, the OS will prefetch more data ahead of time to keep the cache populated with data that will be asked next.)
- Moreover, Kafka reduces transmission overhead by the use of the sendfile API.

The comparison of the performance of Kafka, ActiveMQ, and RabbitMQ's consumers is shown in the illustration below.



Performance of the Kafka consumer

This was a basic implementation of Kafka with a small infrastructure, but we can always scale Kafka as per our needs. It was reported in August 2020 that Tencent runs the world's largest Kafka installation, which is capable of processing trillions of messages each day.

Point to ponder

Question

Why is consumption slower than production in the graphs shown above?

Hide Answer

Kafka consumers often lag the performance of producers because there are more producers producing messages than consumers consuming messages. Moreover, there can also be more accesses made on the ZooKeeper by the consumer due to consumer failures and rebalancing processes. However, we have only one producer and consumer here, and the consumer's performance is still lagging behind the producer's performance. This could be because of the consumer requesting a lesser number of bytes in each pull request from the brokers, or it might be that the delay in the requests is longer than needed. Both parameters can be configured by the user, and they should be configured to fully utilize the capacity of the consumers.

> Note: Above are some of the possible reasons. We can't say for sure because the study that reported these numbers didn't mention the exact reason.

Scalability and distribution support

Kafka provides scalability to the users. A user can always start with a single broker and can expand it to a cluster of more than one broker or into a production-scale cluster of several brokers if there is an increase in the data that is being produced. Moreover, if production is faster than consumption, consumers can also be increased. The broker cluster can be expanded even when the system is online without affecting the availability of Kafka as a whole. It also provides distribution

support to the users since they can define several partitions in a topic. Therefore, it distributes data among the brokers to consume in parallel without decreasing the performance. This provides an opportunity to replicate data and make it available even in case of failures in machines.

Data retention

Kafka can retain data in the brokers for consumers that are not always available for consumption without any danger of losing it, which makes it durable. However, the data is only retained with some configurable limits that are based on time and quantity of data.

Conclusion

In this chapter, we learned about a framework that can store and transfer huge volumes of data streams from producers to consumers. Kafka provides efficient storage and propagation of data from producers to consumers. These features help Kafka achieve high throughput and outperform frameworks like RabbitMQ and ActiveMQ. Unlike some existing messaging systems, Kafka provides distribution support through which it can handle multiple producers and consumers simultaneously.

System design wisdom in Kafka

- Kafka uses some unconventional design decisions that enable it to achieve much better throughput as compared to its competitors. Not keeping consumer-related state in brokers (that will change per message fetch) and assigning only one consumer per partition are some of those unconventional design decisions. It might seem to limit from the perspective of the end consumers. However, in reality, it pushes the users to think for parallelism (by using more consumers and partitions) and using external sources if there is a need to keep state (like Kafka used ZooKeeper). This is an example where a designer learns that we should not blindly follow conventional design wisdom —at times, going different ways opens up new opportunities.

- You might have been wondering why we kept Kafka in the “Bigdata Processing” chapter of the course while we haven’t discussed any new way of processing. In the real world, data (on which we need to run processing) comes from all over the place, and a major source of high latency is the slow collection and dissemination of data. Kafka fills that need and can provide data to processing engines in real time. This example teaches us that when we are solving some problem (for example, making processing fast), we should look for end-to-end opportunities to solve the problem (for example, trying to make the processing engine any faster might not help if most of the latency is coming due to slow data dissemination).

In conclusion, Kafka is a major step forward in the field of data streaming in real time.

← Back

Delivery Guarantees o...

Next →

Quiz on Kafka

Mark as Completed