Cloudera Runtime 7.2.14

Tuning Apache Kafka Performance

Date published: 2019-12-18
Date modified: 2022-02-24

CLOUDERA

Legal Notice

Contents

Handling large messages

Learn more about the options you have when handling large messages with Kafka.

Kafka can be tuned to handle large messages. This can be done by configuring broker and consumer properties relating to maximum message and file sizes. However, before doing so, Cloudera recommends that you try and reduce the size of messages first. Review the following options that can help you reduce message size.

- The Kafka producer can compress messages. For example, if the original message is a text-based format (such as XML), in most cases the compressed message will be sufficiently small.
- Use the compression.type producer configuration parameters to enable compression. gzip, lz4, Snappy, and Zstandard are supported.
- If shared storage (such as NAS, HDFS, or S3) is available, consider placing large files on the shared storage and using Kafka to send a message with the file location. In many cases, this can be much faster than using Kafka to send the large file itself.
- Split large messages into 1 KB segments with the producing client, using partition keys to ensure that all segments are sent to the same Kafka partition in the correct order. The consuming client can then reconstruct the original large message.

If none the options presented are viable, and you need to send large messages with Kafka, you can do so by configuring the following parameters based on your requirements.

Table 1: Broker Configuration Properties

Property	Default Value	Description
message.max.bytes	1000000 (1 MB)	Maximum message size the broker accepts.
log.segment.bytes	1073741824 (1 GiB)	Size of a Kafka data file. Must be larger than any single message.
replica.fetch.max.bytes	1048576 (1 MiB)	Maximum message size a broker can replicate. Must be larger than messa accept messages it cannot replicate, potentially resulting in data loss.

Table 2: Consumer Configuration Properties

Property	Default Value	Description
max.partition.fetch.bytes	1048576 (10 MiB)	The maximum amount of data per-partition the server will return.
fetch.max.bytes	52428800 (50 MiB)	The maximum amount of data the server should return for a fetch request

Note: The consumer is able to consume a message batch that is larger than the default value of the max. partition.fetch.bytes or fetch.max.bytes property. However, the batch will be sent alone, which can cause performance degradation.

Cluster sizing

Learn how to size your Kafka cluster.

Sizing estimation based on network and disk message throughput

Learn how to estimate the size of your cluster based on network and disk throughput requirements.

There are many variables that go into determining the correct hardware footprint for a Kafka cluster. The most accurate way to model your use case is to simulate the load you expect on your own hardware. You can do this using the load generation tools that ship with Kafka, kafka-producer-perf-test and kafka-consumer-perf-test.

However, if you want to size a cluster without simulation, a very simple rule could be to size the cluster based on the amount of disk-space required (which can be computed from the estimated rate at which you get data times the required data retention period).

A slightly more sophisticated estimation can be done based on network and disk throughput requirements. To make this estimation, let's plan for a use case with the following characteristics:

- W - MB/sec of data that will be written
- R - Replication factor
- C - Number of consumer groups, that is the number of readers for each write

Kafka is mostly limited by the disk and network throughput.

The volume of writing expected is W * R (that is, each replica writes each message). Data is read by replicas as part of the internal cluster replication and also by consumers. Because every replicas but the master read each write, the read volume of replication is (R-1) * W. In addition each of the C consumers reads each write, so there will be a read volume of C * W. This gives the following:

- Writes: W * R
- Reads: (R+C- 1) * W

However, note that reads may actually be cached, in which case no actual disk I/O happens. We can model the effect of caching fairly easily. If the cluster has M MB of memory, then a write rate of W MB/second allows M/(W * R) seconds of writes to be cached. So a server with 32 GB of memory taking writes at 50 MB/second serves roughly the last 10 minutes of data from cache. Readers may fall out of cache for a variety of reasons—a slow consumer or a failed server that recovers and needs to catch up. An easy way to model this is to assume a number of lagging readers you to budget for. To model this, let's call the number of lagging readers L. A very pessimistic assumption would be that L = R + C -1, that is that all consumers are lagging all the time. A more realistic assumption might be to assume no more than two consumers are lagging at any given time.

Based on this, we can calculate our cluster-wide I/O requirements:

- Disk Throughput (Read + Write): W * R + L * W
- Network Read Throughput: (R + C -1) * W
- Network Write Throughput: W * R

A single server provides a given disk throughput as well as network throughput. For example, if you have a 1 Gigabit Ethernet card with full duplex, then that would give 125 MB/sec read and 125 MB/sec write; likewise 6 7200 SATA drives might give roughly 300 MB/sec read + write throughput. Once we know the total requirements, as well as what is provided by one machine, you can divide to get the total number of machines needed. This gives a machine count running at maximum capacity, assuming no overhead for network protocols, as well as perfect balance of data and load. Since there is protocol overhead as well as imbalance, you want to have at least 2x this ideal capacity to ensure sufficient capacity.

Related Information

kafka-*-perf-test

Choosing the number of partitions for a topic

Learn how to determine the number of partitions each of your Kafka topics requires.

Choosing the proper number of partitions for a topic is the key to achieving a high degree of parallelism with respect to writes to 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.
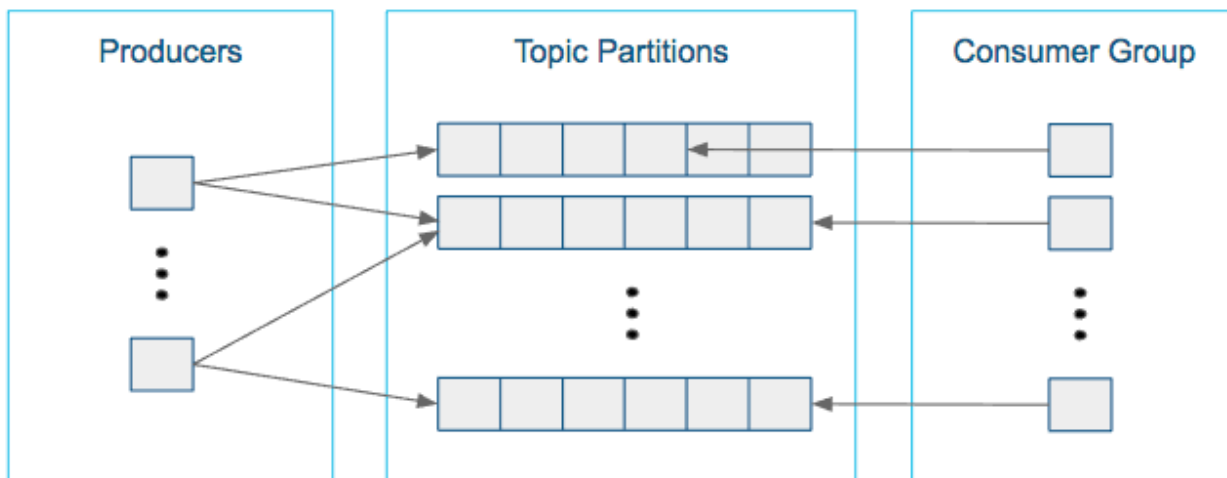
For example, if you want to be able to read 1 GB/sec, but your consumer is only able process 50 MB/sec, then you need at least 20 partitions and 20 consumers in the consumer group. Similarly, if you want to achieve the same for producers, and 1 producer can only write at 100 MB/sec, you need 10 partitions. 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
```

Related Information
Managing Apache kafka

Broker Tuning

JVM and garbage collection

Learn more about garbage collection for Kafka.

Garbage collection has a huge impact on performance of JVM based applications. It is recommended to use the Garbage-First (G1) garbage collector for Kafka broker. In Cloudera Manager specify the following under Additional Broker Java Options in the Kafka service configuration:

```
-server -XX:+UseG1GC -XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35 -XX:+DisableExplicitGC
-Djava.awt.headless=true -Djava.net.preferIPv4Stack=true
```

Cloudera recommends to set 4-8 GB of JVM heap size memory for the brokers depending on your use case. As Kafka's performance depends heavily on the operating systems page cache, it is not recommended to collocate with other memory-hungry applications.

- Large messages can cause longer garbage collection (GC) pauses as brokers allocate large chunks. Monitor the GC log and the server log.

  Add this to Broker Java Options:

  ```
   -XX:+PrintGC -XX:+PrintGCDetails
   -XX:+PrintGCTimeStamps
  -Xloggc:</path/to/file.txt>
  ```

- If long GC pauses cause Kafka to abandon the ZooKeeper session, you may need to configure longer timeout values, see Kafka-ZooKeeper performance tuning for details.

Related Information
Kafka-ZooKeeper performance tuning

Network and I/O threads

Learn more about the configuration properties related to network and I/O threads.

Kafka brokers use network threads to handle client requests. Incoming requests (such as produce and fetch requests) are placed into a requests queue from where I/O threads are taking them up and process them. After a request is

processed, the response is placed into an internal response queue from where a network thread picks it up and sends response back to the client.

- num.network.threads is an important cluster-wide setting that determines the number of threads used for handling network requests (that is, receiving requests and sending responses). Set this value mainly based on number of producers, consumers and replica fetchers.
- queued.max.requests controls how many requests are allowed in the request queue before blocking network threads.
- num.io.threads specifies the number of threads that a broker uses for processing requests from the request queue (might include disk I/O).

ISR management

Learn more about in-sync replica management.

An in-sync replica (ISR) set for a topic partition contains all follower replicas that are caught-up with the leader partition, and are situated on a broker that is alive.

- If a replica lags "too far" behind from the partition leader, it is removed from the ISR set. The definition of what is too far is controlled by the configuration setting replica.lag.time.max.ms. If a follower hasn't sent any fetch requests or hasn't consumed up to the leaders log end offset for at least this time, the leader removes the follower from the ISR set.
- num.replica.fetchers is a cluster-wide configuration setting that controls how many fetcher threads are in a broker. These threads are responsible for replicating messages from a source broker (that is, where partition leader resides). Increasing this value results in higher I/O parallelism and fetcher throughput. Of course, there is a trade-off: brokers use more CPU and network.
- replica.fetch.min.bytes controls the minimum number of bytes to fetch from a follower replica. If there is not enough bytes, wait up to replica.fetch.wait.max.ms.
- replica.fetch.wait.max.ms controls how long to sleep before checking for new messages from a fetcher replica. This value should be less than replica.lag.time.max.ms, otherwise the replica is kicked out of the ISR set.
- To check the ISR set for topic partitions, run the following command:

```
kafka-topics --zookeeper ${ZOOKEEPER_HOSTNAME}:2181/kafka --describe --t
opic ${TOPIC}
```

- If a partition leader dies, a new leader is selected from the ISR set. There will be no data loss. If there is no ISR, unclean leader election can be used with the risk of data-loss.
- Unclean leader election occurs if unclean.leader.election.enable is set to true. By default, this is set to false.

Log cleaner

An overview on configuration properties related to log compaction.

The following cluster-wide configuration settings can be used to fine tune log compaction:

- log.cleaner.threads controls how many background threads are responsible for log compaction. Increasing this value improves performance of log compaction at the cost of increased I/O activity.
- log.cleaner.io.max.bytes.per.second throttles log cleaner's I/O activity so that the sum of its read and write is less than this value on average.
- log.cleaner.dedupe.buffer.size specifies memory used for log compaction across all cleaner threads.
- log.cleaner.io.buffer.size controls total memory used for log cleaner I/O buffers across all cleaner threads.
- log.cleaner.min.compaction.lag.ms controls how long messages are left uncompacted.
- log.cleaner.io.buffer.load.factor controls log cleaner's load factor for the dedupe buffer. Increasing this value allows the system to clean more logs at once but increases hash collisions.
- log.cleaner.backoff.ms controls how long to wait until the next check if there is no log to compact.

System Level Broker Tuning

Operating system related kernel parameters affect overall performance of Kafka. These parameters can be configured via sysctl at runtime. To make kernel configuration changes persistent (that is, use adjusted parameters after a reboot), edit /etc/sysctl.conf. The following sections describe some important kernel settings.

File descriptor limits

Learn more about file descriptor requirements.

As Kafka works with many log segment files and network connections, the Maximum   Process File Descriptors setting may need to be increased in some cases in production deployments, if a broker hosts many partitions. For example, a Kafka broker needs at least the following number of file descriptors to just track log segment files:

```
(number of partitions)*(partition size / segment size)
```

The broker needs additional file descriptors to communicate via network sockets with external parties (such as clients, other brokers, Zookeeper, and Kerberos).

The Maximum Process File Descriptors setting can be monitored in Cloudera Manager and increased if usage requires a larger value than the default ulimit (often 64K). It should be reviewed for use case suitability.

- To review FD limit currently set for a running Kafka broker, run cat   /proc/$KAFKA\_BROKER\_PID$/limits, and look for Max open   files.
- To see open file descriptors, run:

```
lsof -p KAFKA_BROKER_PID
```

Filesystems

Learn how mount options such as noatime and relatime affect Kafka performance.

Linux records when a file was created (ctime), modified (mtime) and accessed (atime). The value noatime is a special mount option for filesystems (such as EXT4) in Linux that tells the kernel not to update inode information every time a file is accessed (that is, when it was last read). Using this option can result in write performance gain. Kafka is not relying on atime. The value relatime is another mounting option that optimizes how atime is persisted. Access time is only updated if the previous atime was earlier than the current modified time.

To view mounting options, run mount -l or cat /etc/fstab command.

Virtual memory handling

Learn about how virtual memory handling affects Kafka performance.

Kafka uses system page cache extensively for producing and consuming the messages. The Linux kernel parameter, vm.swappiness, is a value from 0-100 that controls the swapping of application data (as anonymous pages) from physical memory to virtual memory on disk. The higher the value, the more aggressively inactive processes are swapped out from physical memory. The lower the value, the less they are swapped, forcing filesystem buffers to be emptied. It is an important kernel parameter for Kafka because the more memory allocated to the swap space, the less memory can be allocated to the page cache. Cloudera recommends to set vm.swappiness value to 1.

- To check memory swapped to disk, run vmstat and look for the swap columns.

Kafka heavily relies on disk I/O performance. vm.dirty_ratio and vm.dirty_background_ratio are kernel parameters that control how often dirty pages are flushed to disk. Higher vm.dirty_ratio results in less frequent flushes to disk.

- To display the actual number of dirty pages in the system, run egrep   "dirty|writeback" /proc/vmstat

Networking parameters

An overview on Linux kernel parameters related to networking to fine tune for better Kafka performance.

Kafka is designed to handle a huge amount of network traffic. By default, the Linux kernel is not tuned for this scenario. The following kernel settings may need to be tuned based on use case or specific Kafka workload:

- net.core.wmem_default: Default send socket buffer size.
- net.core.rmem_default: Default receive socket buffer size.
- net.core.wmem_max: Maximum send socket buffer size.
- net.core.rmem_max: Maximum receive socket buffer size.
- net.ipv4.tcp_wmem: Memory reserved for TCP send buffers.
- net.ipv4.tcp_rmem: Memory reserved for TCP receive buffers.
- net.ipv4.tcp_window_scaling: TCP Window Scaling option.
- net.ipv4.tcp_max_syn_backlog: Maximum number of outstanding TCP SYN requests (connection requests).
- net.core.netdev_max_backlog: Maximum number of queued packets on the kernel input side (useful to deal with spike of network requests).

Use the Cloudera Enterprise Reference Architecture for Bare Metal Deploymentsas a guideline when specifying values for the parameters.

Configure JMX ephemeral ports

Learn more aboutKafka JMX ephermal ports and how to configure them.

Kafka uses two high-numbered ephemeral ports for JMX. These ports are listed when you view netstat -anp information for the Kafka broker process.

- You can change the number for the first port by adding a command similar to the following to the field Additional Broker Java Options (broker_java_opts) in Cloudera Manager.

```
-Dcom.sun.management.jmxremote.rmi.port=port
```

- The JMX_PORT configuration maps to com.sun.management.jmxremote.port by default.

  To access JMX via JConsole, run jconsole ${BROKER_HOST}:9393
- The second ephemeral port used for JMX communication is implemented for the JRMP protocol and cannot be changed.

Kafka-ZooKeeper performance tuning

Recommendations and parameters in connection with Kafka-Zookeeper performance tuning.

Kafka uses Zookeeper to store metadata information about topics, partitions, brokers and system coordination (such as membership statuses). Unavailability or slowness of Zookeeper makes the Kafka cluster unstable, and Kafka brokers do not automatically recover from it. Cloudera recommends to use a 3-5 machines Zookeeper ensemble solely dedicated to Kafka (co-location of applications can cause unwanted service disturbances).

- zookeeper.session.timeout.ms is a setting for Kafka that specifies how long Zookeeper shall wait for heartbeat messages before it considers the client (the Kafka broker) unavailable. If this happens, metadata information about

partition leadership owned by the broker will be reassigned. If this setting is too high, then it might take a long time for the system to detect a broker failure. On the other hand, if it is set to too small, it might result in frequent leadership reassignments.

- jute.maxbuffer is a crucial Java system property for both Kafka and Zookeeper. It controls the maximum size of the data a znode can contain. The default value, one megabyte, might be increased for certain production use cases.
- There are cases where Zookeeper can require more connections. In those cases, it is recommended to increase the maxClientCnxns parameter in Zookeeper.
- Note that old Kafka consumers store consumer offset commits in Zookeeper (deprecated). It is recommended to use new consumers that store offsets in internal Kafka topics as this reduces the load on Zookeeper.