Kafka Best Practices

Zookeeper

- Do not co-locate zookeeper on the same boxes as Kafka
- We recommend zookeeper to isolate and only use for Kafka not any other systems should be depend on this zookeeper cluster
- Make sure you allocate sufficient JVM , good starting point is 4Gb
- Monitor: Use JMX metrics to monitor the zookeeper instance

Apache Kafka is a widely popular distributed streaming platform that thousands of companies like New Relic, Uber, and Square use to build scalable, high-throughput, and reliable real-time streaming systems. For example, the production Kafka cluster at New Relic processes more than 15 million messages per second for an aggregate data rate approaching 1 Tbps.

Kafka has gained popularity with application developers and data management experts because it greatly simplifies working with data streams. But Kafka can get complex at scale. A high-throughput publish-subscribe (pub/sub) pattern with automated data retention limits doesn't do you much good if your consumers are unable to keep up with your data stream and messages disappear before they're ever seen. Likewise, you won't get much sleep if the systems hosting the data stream can't scale to meet demand or are otherwise unreliable.

In hopes of reducing that complexity, I'd like to share 20 of New Relic's best practices for operating scalable, high-throughput Kafka clusters. We've divided these tips into four categories for working with:

1. Partitions

2. Consumers

3. Producers

4. Brokers

But First, a Quick Rundown of Kafka and Its Architecture

Kafka is an efficient distributed messaging system providing built-in data redundancy and resiliency while retaining both high-throughput and scalability. It includes automatic data retention limits, making it well suited for applications that treat data as a stream, and it also supports "compacted" streams that model a map of key-value pairs.

To understand these best practices, you'll need to be familiar with some key terms:

- **Message:** A record or unit of data within Kafka. Each message has a key and a value, and optionally headers.
- **Producer:** Producers publish messages to Kafka *topics*. Producers decide which topic partition to publish to, either randomly (round-robin) or using a partitioning algorithm based on a message's key.
- **Broker:** Kafka runs in a distributed system or *cluster*. Each node in the cluster is called a broker.
- **Topic:** A topic is a category to which data records — or messages — are published. Consumers subscribe to topics in order to read the data written to them.
- **Topic partition:** Topics are divided into partitions, and each message is given an *offset*. Each partition is typically replicated at least once or twice. Each partition has a leader and one or more replicas (copies of the data) that exist on followers, providing protection against a broker failure. All brokers in the cluster are both leaders and *followers*, but a broker has at most one replica of a topic partition. The leader is used for all reads and writes.
- **Offset:** Each message within a partition is assigned an offset, a monotonically increasing integer that serves as a unique identifier for the message within the partition.
- **Consumer:** Consumers read messages from Kafka topics by subscribing to topic partitions. The consuming application then processes the message to accomplish whatever work is desired.
- **Consumer group:** Consumers can be organized into logic consumer groups. Topic partitions are assigned to balance the assignments among all

consumers in the group. Within a consumer group, all consumers work in a load-balanced mode; in other words, each message will be seen by one consumer in the group. If a consumer goes away, the partition is assigned to another consumer in the group. This is referred to as a *rebalance*. If there are more consumers in a group than partitions, some consumers will be idle. If there are fewer consumers in a group than partitions, some consumers will consume messages from more than one partition.

- **Lag:** A consumer is lagging when it's unable to read from a partition as fast as messages are produced to it. Lag is expressed as the number of offsets that are behind the head of the partition. The time required to recover from lag (to "catch up") depends on how quickly the consumer is able to consume messages per second:

```
time = messages / (consume rate per second - produce rate per second)
```

Best Practices for Working With Partitions

- **Understand the data rate of your partitions to ensure you have the correct retention space.** The data rate of a partition is the rate at which data is produced to it; in other words, it's the average message size times the number of messages per second. The data rate dictates how much retention space, in bytes, is needed to guarantee retention for a given amount of time. If you don't know the data rate, you can't correctly calculate the retention space needed to meet a time-based retention goal. The data rate also specifies the minimum performance a single consumer needs to support without lagging.
- **Unless you have architectural needs that require you to do otherwise, use random partitioning when writing to topics.** When you're operating at scale, uneven data rates among partitions can be difficult to manage. There are three main reasons for this:
    - First, consumers of the "hot" (higher throughput) partitions will have to process more messages than other consumers in the consumer group, potentially leading to processing and networking bottlenecks.
    - Second, topic retention must be sized for the partition with the highest data rate, which can result in increased disk usage across other partitions in the topic.

  - Third, attaining an optimum balance in terms of partition leadership is more complex than simply spreading the leadership across all brokers. A "hot" partition might carry 10 times the weight of another partition in the same topic.

For a closer look at working with topic partitions, see Effective Strategies for Kafka Topic Partitioning.

Best Practices for Working With Consumers

- **If your consumers are running versions of Kafka older than 0.10, upgrade them.** In version 0.8.x, consumers use Apache ZooKeeper for consumer group coordination, and a number of known bugs can result in long-running rebalances or even failures of the rebalance algorithm (we refer to these as "rebalance storms"). During a rebalance, one or more partitions are assigned to each consumer in the consumer group. In a rebalance storm, partition ownership is continually shuffled among the consumers, preventing any consumer from making real progress on consumption.
- **Tune your consumer socket buffers for high-speed ingest.** In Kafka 0.10.x, the parameter is receive.buffer.bytes, which defaults to 64kB. In Kafka 0.8.x, the parameter is socket.receive.buffer.bytes, which defaults to 100kB. Both of these default values are too small for high-throughput environments, particularly if the network's bandwidth-delay product between the broker and the consumer is larger than a local area network (LAN). For high-bandwidth networks (10 Gbps or higher) with latencies of 1 millisecond or more, consider setting the socket buffers to 8 or 16 MB. If memory is scarce, consider 1 MB. You can also use a value of -1, which lets the underlying operating system tune the buffer size based on network conditions. However, the automatic tuning might not occur fast enough for consumers that need to start "hot."
- **Design high-throughput consumers to implement back-pressure when warranted**. It is better to consume only what you can process efficiently than it is to consume so much that your process grinds to a halt and then drops out of the consumer group. Consumers should consume into fixed-sized buffers (see the Disruptor pattern), preferably off-heap if running in a Java virtual machine (JVM). A fixed-size buffer will prevent a consumer from pulling so much data onto the heap that the JVM spends all of its time

performing garbage collection instead of the work you want to achieve — which is processing messages.

- **When running consumers on a JVM, be wary of the impact that garbage collection can have on your consumers.** For example, long garbage collection pauses can result in dropped ZooKeeper sessions or consumer-group rebalances. The same is true for brokers, which risk dropping out of the cluster if garbage collection pauses are too long.

Best Practices for Working With Producers

- **Configure your producer to wait for acknowledgments.** This is how the producer knows that the message has actually made it to the partition on the broker. In Kafka 0.10.x, the settings is acks; in 0.8.x, it's request.required.acks. Kafka provides fault-tolerance via replication so the failure of a single node or a change in partition leadership does not affect availability. If you configure your producers without acks (otherwise known as "fire and forget"), messages can be silently lost.
- **Configure retries on your producers.** The default value is 3, which is often too low. The right value will depend on your application; for applications where data-loss cannot be tolerated, consider Integer.MAX_VALUE (effectively, infinity). This guards against situations where the broker leading the partition isn't able to respond to a produce request right away.
- **For high-throughput producers, tune buffer sizes**, particularly buffer.memory and batch.size(which is counted in bytes). Because batch.size is a per-partition setting, producer performance and memory usage can be correlated with the number of partitions in the topic. The values here depend on several factors: producer data rate (both the size and number of messages), the number of partitions you are producing to, and the amount of memory you have available. Keep in mind that larger buffers are not always better because if the producer stalls for some reason (say, one leader is slower to respond with acknowledgments), having more data buffered on-heap could result in more garbage collection.
- **Instrument your application to track metrics** such as number of produced messages, average produced message size, and number of consumed messages.

Best Practices for Working With Brokers

- **Compacted topics require memory and CPU resources on your brokers.** Log compactionneeds both heap (memory) and CPU cycles on the brokers to complete successfully, and failed log compaction puts brokers at risk from a partition that grows unbounded. You can tune log.cleaner.dedupe.buffer.size and log.cleaner.threads on your brokers, but keep in mind that these values affect heap usage on the brokers. If a broker throws an OutOfMemoryError exception, it will shut down and potentially lose data. The buffer size and thread count will depend on both the number of topic partitions to be cleaned and the data rate and key size of the messages in those partitions. As of Kafka version 0.10.2.1, monitoring the log-cleaner log file for ERROR entries is the surest way to detect issues with log cleaner threads.
- **Monitor your brokers for network throughput.** Make sure to do this with both transmit (TX) and receive (RX), as well as disk I/O, disk space, and CPU usage. Capacity planning is a key part of maintaining cluster performance.
- **Distribute partition leadership among brokers in the cluster.** Leadership requires a lot of network I/O resources. For example, when running with replication factor 3, a leader must receive the partition data, transmit two copies to replicas, plus transmit to however many consumers want to consume that data. So, in this example, being a leader is at least four times as expensive as being a follower in terms of network I/O used. Leaders may also have to read from disk; followers only write.
- **Don't neglect to monitor your brokers for in-sync replica (ISR) shrinks, under-replicated partitions, and unpreferred leaders.** These are signs of potential problems in your cluster. For example, frequent ISR shrinks for a single partition can indicate that the data rate for that partition exceeds the leader's ability to service the consumer and replica threads.
- **Modify the Apache Log4j properties as needed.** Kafka broker logging can use an excessive amount of disk space. However, don't forgo logging completely — broker logs can be the best, and sometimes only, way to reconstruct the sequence of events after an incident.
- **Either disable automatic topic creation or establish a clear policy regarding the cleanup of unused topics.** For example, if no messages are seen for x days, consider the topic defunct and remove it from the cluster. This will avoid the creation of additional metadata within the cluster that you'll have to manage.

- **For sustained, high-throughput brokers, provision sufficient memory to avoid reading from the disk subsystem.** Partition data should be served directly from the operating system's file system cache whenever possible. However, this means you'll have to ensure your consumers can keep up; a lagging consumer will force the broker to read from disk.
- **For a large cluster with high-throughput service level objectives (SLOs), consider isolating topics to a subset of brokers.** How you determine which topics to isolate will depend on the needs of your business. For example, if you have multiple online transaction processing (OLTP) systems using the same cluster, isolating the topics for each system to distinct subsets of brokers can help to limit the potential blast radius of an incident.
- **Using older clients with newer topic message formats, and vice versa, places extra load on the brokers** as they convert the formats on behalf of the client. Avoid this whenever possible.
- **Don't assume that testing a broker on a local desktop machine is representative of the performance you'll see in production.** Testing over a loopback interface to a partition using replication factor 1 is a very different topology from most production environments. The network latency is negligible via the loopback and the time required to receive leader acknowledgements can vary greatly when there is no replication involved.