Chapter 12: Batching and Compression

Chapter 10: Client Configuration mostly attended to the aspects of client configuration related to functionality and safety, deliberately sidestepping any serious discussions on performance. The intent of this chapter is to focus on one specific area of performance optimisation — batching and compression. The two are related, collectively bearing a significant impact on the performance of an event streaming system.

Comparing disk and network I/O

Kafka utilises a segmented, *append-only log*, largely limiting itself to *sequential I/O* for both reads and writes, which is fast across a wide variety of storage media. There is a wide misconception that disks are slow; however, the performance of storage media (particularly rotating media) is greatly dependent on access patterns. The performance of random I/O on a typical 7,200 RPM SATA disk is between three and four orders of magnitude slower than sequential I/O. Furthermore, a modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes. Because of this, the difference between sequential I/O and random I/O is still evident in flash and other forms of solid-state non-volatile media, although the effects are less dramatic compared to rotating media.

Sequential I/O is comparable to the peak performance of network I/O. Furthermore, disk I/O is local to the host, whereas network I/O is shared. In practice, this means that a well-designed log-structured persistence layer will keep up with the network traffic. In fact, often the bottleneck with Kafka's performance isn't the disk, but the network. Stated bluntly, the network fabric will run out of steam before the broker.

Producer record batching

To counteract the limitations of the network, Kafka clients will batch multiple records together before sending them over the network. This is independent of, and in addition to, the low-level batching provided by the OS at the TCP socket layer. Batching of records amortises the overhead of the network round-trip, using larger packets and improving bandwidth efficiency.

Batching in Kafka can act end-to-end. The producer client uses an accumulator buffer for staging records prior to forwarding them to the leader broker. Once the records are batched, the broker will (in most cases) persist them as-is, without unpacking the batch or performing any intermediate

manipulations of the stored records. This is carried through to the consumer. When polling for records, the consumer is served batches of records by the broker — the same batches that were originally published. There are cases where a batch is not end-to-end; for example, when a broker is instructed to apply a compression scheme that differs from the producer. More on that later.

Being a largely end-to-end concern, batching is controlled by the producer. The `batch.size` and `linger.ms` properties collectively limit the extent to which the producer will attempt to batch queued records in order to maximise the outgoing transmission efficiency. The default values of `batch.size` and `linger.ms` are `16384` (number of bytes) and `0` (milliseconds), respectively.

The producer combines any records that arrive between request transmissions into a single batched request. Normally this only occurs under load, when records arrive faster than they can be transmitted. In some circumstances, the client may want to reduce the number of requests even under moderate load. The `linger.ms` setting accomplishes this by adding a small amount of artificial delay — rather than immediately sending a record the moment it is enqueued, the producer will wait for up to a set delay to allow other records to accumulate in a batch, maximising the amount of data that can be transmitted in one go. Although records may be allowed to linger for up to the duration specified in `linger.ms`, the `batch.size` property will have an overriding effect, dispatching the batch once it reaches the set maximum size.

While the `linger.ms` property only comes into the picture when the producer is lightly loaded, the `batch.size` property is continually in effect, acting as the overarching and unremitting limiter of the staged batch.

> The `linger.ms` setting is often likened to the *Nagle's Algorithm* in TCP, as they both aim to improve the efficiency of network transmission by combining small and intermittent outgoing messages and sending them at once. The comparison was originally made by the Apache Kafka design team. It has since found its way into the official documentation and appears to reverberate strongly within the user community.
>
> While the similarities are superficial, there are two crucial differences. Firstly, Nagle's Algorithm does not indiscriminately buffer data on the basis of its apparent intermittence. It only takes effect when there is *at least one outstanding packet* that is yet to be acknowledged by the receiver. Secondly, Nagle's Algorithm does not impose an artificial time limit on the delay. Data will be buffered until a full packet is formed, or the `ACK` for the previous packet is received, whichever occurs first.
>
> Combined, the two instruments make the algorithm self-regulating; rather than relying on arbitrary, user-defined linger times, the algorithm buffers data based on the network's observed performance. Free-flowing networks with frequent `ACK`s result in lighter buffering and reduced transmission latency. As congestion increases, the buffering becomes more pronounced, improving transmission efficiency at the expense of latency, and helps avoid a congestive collapse.

Congestive collapse is a phenomenon that occurs when a network is overwhelmed by a high packet rate, usually at known choke points, leading to increased failures (packet loss and timeouts) and a corresponding escalation of retries. This creates a perpetuating feedback loop that degenerates to a parasitic stable state where the traffic demand is high, but there is little useful throughput available.

The batching algorithm used by Kafka is crude by comparison. It requires careful tuning and imposes a fixed penalty on intermittent publishing patterns regardless of the network's performance or the observed latency. This does not necessarily render it ineffective. On the contrary, the batching algorithm can be very effective over high-latency networks. Its main issue is the lack of adaptability, making it suboptimal in dynamic network climates or in the face of varying cluster performance — both factors affecting publishing latency. Kafka places the onus of tuning the algorithm parameters on the user, requiring careful and extensive experimentation to empirically arrive at the optimal set of parameters for a given network and cluster profile. These parameters must also be periodically revised to ensure their continued viability.

Even with the default `linger.ms` value of `0`, the producer will still allow for some buffering due to the asynchronous nature of the transmission: calling `send()` serializes the record, assigns a partition number, and places the serialized contents into the accumulator buffer, but does not transmit it. The actual communications are handled by a background I/O thread. So if `send()` is called multiple times in rapid succession, at least some of these records will likely be batched.

Due to the diminishing returns exhibited with larger batch sizes and the lack of self-regulation in Kafka's batching algorithm, it may be prudent to err on the side of smaller values of `linger.ms` initially — from zero to several milliseconds. The non-functional requirements of the overall system should also specify the tolerance for latency, which ought to be honoured over any gains in network efficiency or storage savings on the brokers. The extent of batching may be increased further if the network is becoming a genuine bottleneck; however, such changes should be temporary — the focus should be on addressing the root cause.

Compression

The effectiveness of batching increases substantially when complemented by record compression. As compression operates on an entire batch, the resulting compression ratios increase with the batch size. The effects of compression are particularly pronounced when using text-based encodings such as JSON, where the records exhibit low information entropy. For JSON, compression ratios ranging from 5x to 7x are not unusual, which makes enabling compression a no-brainer. Furthermore, record batching and compression are largely done as a client-side operation, which transfers the load onto the client and has a positive effect not only on the network bandwidth, but also on the brokers' disk I/O and storage utilisation.

The biggest gains will be felt when starting from small batches. As the batch size increases, the laws of diminishing returns take effect — an increase in the batch size will yield a proportionally smaller gain in compression ratio. The incremental reduction in the number of packets used, and hence the transmission efficiency, will also be less noticeable with higher batch sizes.

The `compression.type` client configuration property controls the algorithm that the producer will use to compress record batches before forwarding them on to the partition leaders. The valid values are:

- `none`: Compression is disabled. This is the default setting.
- `gzip`: Use the *GNU Gzip* algorithm — released in 1992 as a free substitute for the proprietary `compress` program used by early UNIX systems.
- `snappy`: Use Google's *Snappy* compression format — optimised for throughput at the expense of compression ratios.
- `lz4`: Use the *LZ4* algorithm — also optimised for throughput, most notably for the speed of decompression.
- `zstd`: Use Facebook's *ZStandard* — a newer algorithm introduced in Kafka 2.1.0, intended to achieve an effective balance between throughput and compression ratios.

Kafka compression applies to an entire record batch, which as we know *can be* end-to-end — depending on the settings on the broker. Specifically, when the broker is configured to accept the producer's preferred compression scheme, it will not interfere with the contents of the batch. The batch flows from the producer to the broker, is persisted across multiple replicas, and is eventually served to one or more consumers — all as a single, indivisible chunk. The broker simply acts as a relaying party — it does not decompress and re-compress the record as part of its role. Each chunk has a header that notes the algorithm that was used during compression, allowing the consumers to apply the same when unpacking the chunk.

The `compression.type` broker property applies to all topics by default, overriding the producer property. In addition, it is possible to configure individual topics by using the `kafka-configs.sh` CLI to specify a dynamic configuration for the `topics` entity type.

On the flip side, end-to-end compression has a subtle drawback, which can catch out unsuspecting users. Because the broker is unable to mediate the interchange format, the result is a latent coupling between the producer's capabilities and that of the consumer clients. Although Kafka strives to maintain binary protocol compatibility between minor releases, this promise does not cover end-to-end contracts, such as compression and checksumming. As such, the producer application must use a compression format this is compatible with the oldest consumer version.

The introduction of ZStandard is a good example of a breaking change, and may be considered as a 'gotcha' depending on your client ecosystem. When operating a mixture 2.1.0+ and pre-2.1.0 consumers, the use of `compression.type=zstd` on a producer will render the records unreadable for the older clients, resulting in an `UNSUPPORTED_COMPRESSION_TYPE` error. The correct way to enable ZStandard is to upgrade all consumers first, and only when the last pre-2.1.0 consumer has been retired, allow the use of `compression.type=zstd`. Alternatively, one can enable re-compression on the broker to maintain compatibility with older clients; however, this leads to increased resource utilisation on the broker.

End-to-end compression has a profoundly positive impact on performance. Compression is a processor-intensive operation and consumes additional memory, particularly during the encoding phase. (Comparatively, decoding a compressed stream is cheaper; the difference may be an order of magnitude in extreme cases.) By eliminating the broker from the equation, the cost of compression is absorbed by the clients. The distribution of load to the periphery dovetails into Kafka's broader scaling strategy — it is typically much easier to scale the clients than the broker, at least for well-architected, stateless applications.

Kafka additionally offers compression at the broker level, for those scenarios where it is necessary to change the compression scheme. This is controlled by the `compression.type` broker property. Its default value is `producer`, meaning that the broker will revert to the producer-assigned compression scheme; in other words, the broker will not meddle the batch. Alternatively, the value of `compression.type` may be set to one of the supported compression schemes (as per the producer-side property). The only difference is when disabling compression: the producer property accepts `none`, whereas the broker property accepts `uncompressed`.

While broker-level configuration can provide for more fine-grained control of the compression scheme, its main drawback is the increased CPU utilisation and the forfeiting of the *zero-copy* optimisation, as the batches are no longer end-to-end.

Zero-copy describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. In a typical I/O scenario, the transfer of data from a network socket to a storage device occurs without the involvement of the CPU and with a reduced number of context switches between the kernel and user mode.

The use of compression has no functional effect on the system. Its effect on bandwidth utilisation, disk space utilisation, and broker I/O are, in some cases, astounding. As a performance optimisation, and depending on the type of data transmitted, the effect of compression can be so profound that it thwarts any potential philosophical debates over the viability of 'premature' optimisation.

Compression algorithms achieve a reduction in the encoded size relative to the uncompressed original by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. For compression to be effective, the data must contain a large amount of repetition and be of sufficient size so as to warrant any structural

overheads, such as the introduction of a dictionary. Text formats such as JSON tend to be highly compressible as they are verbose by nature and contain repeated character sequences that carry no informational content. They also fail to take advantage of the full eight bits that each byte can theoretically accommodate, instead representing characters with seven of the lower order bits. Binary encodings tend to exhibit more informational density, but may still be highly compressible, depending on their internal structure. Binary streams containing digital media — for example, images, audio or video — are high-entropy sources and are virtually incompressible.

Information entropy is the measure of the informational content conveyed by an element in a stream. It is determined by the likelihood of predicting the value of an element in a stream based on the observations of prior elements, with the resulting score varying between zero (no entropy, perfectly predictable) and one (highest entropy, completely unpredictable). In the context of a data record, an element might be an individual bit or a byte in the record. The easier it is to predict the value of the next byte, the less new information it carries. As an example of a low entropy source, consider a formatted JSON document. Having observed a newline character, it is extremely likely that the next element is a whitespace character, with the next likely candidate being a double quote, followed in relative likelihood by a closing brace. Compare this to a truly randomly sequence of bytes. The likelihood of predicting the next byte is equivalent to chance; this is an example of maximum entropy. Compressing this type of data will only lead to an increase in the output size as the overheads of the compression algorithm will be added into the output, not offset by any gains in entropy.

Without delving into a harrowing analysis of the different compression schemes, the recommendation is to always enable compression for text encodings as well as binary data (unless the latter is known to contain a high information entropy payload). In some cases, the baseline impact of enabling compression may be so substantial that the choice of the compression algorithm hardly matters. In other cases, the choice of the algorithm may materially impact the overall performance, and a more careful selection is warranted.

As a rule of thumb, use LZ4 when dealing with legacy consumers, transmitting over a network that offers capacity in excess of your peak uncompressed data needs. In other words, the network is able to sustain your traffic flow even without compression. If the network has been identified as the bottleneck, consider switching to Gzip and also increasing `batch.size` and `linger.ms` to increase the size of transmitted batches to maximise the effectiveness of compression at the expense of latency.

If all consumers are at a version equal to or greater than 2.1.0, your choices are basically LZ4 and ZStandard. Use LZ4 for low-overhead compression. When the network becomes the bottleneck, consider ZStandard, as it is able to achieve similar compression ratios to Gzip at a fraction of the compression and decompression time. You may also need to increase `batch.size` and `linger.ms` to maximise compression effectiveness.

The guidelines above should not be taken to mean that LZ4 always outperforms Snappy or that

ZStandard is should always be preferred over Gzip. When undertaking serious performance tuning, you should carefully consider the shape of your data and conduct studies using synthetic records that are representative of the real thing, or better still, using historical data if this is an option. When benchmarking the different compression schemes, you should also measure the CPU and memory utilisation of the producer and consumer clients, comparing these to the baseline case (when compression is disabled). In some cases you may find that Snappy or Gzip indeed offer a better compromise. The guidelines presented here should be used as the starting point, particularly when one's copious free time is prioritised towards dealing with the matters of building software, over conducting large-scale performance trials.

This chapter has given the reader an insight into Kafka's performance 'secret sauce' — namely, the use of log-structured persistence to limit access patterns to sequential reads and writes. The implication: a blazingly fast disk I/O subsystem that can outperform the network fabric, requiring further client-side optimisations to bring the two into parity.

We looked at two controls available on the producer client — batching and compression. The potential performance impacts of these controls are significant, particularly in the areas of throughput and latency. Getting them right could entail significant gains with relatively little effort.