KIP-780: Support fine-grained compression options

Status

Current state: Under Discussion

Discussion thread: here

JIRA: KAFKA-13361

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

As a following work of KIP-390: Support Compression Level, this proposal suggests adding support for per-codec configuration options to Producer, Broker, and Topic configurations, which enable a fine-tuning of compression behavior.

There are many kinds of compression codecs with intrinsic mechanisms; generally, they provide various configuration options to cope with diverse kinds of input data by tuning the codec's compression/decompression behavior. However, In the case of Apache Kafka, it has been allowed to compress the data with the default settings only, and only the compression level made configurable in KIP-390 (not shipped yet as of 3.0.0.).

Making finely-tune the compression behavior is so important with the recent trend. The bigger and more data feeds into the event-driven architecture, the more multi-region Kafka architecture gains popularity, and Apache Kafka is working as a single source of the truth, not a mere messaging system. The more data is compressed, the more easily transferred within the computer network and stored in a disk with less footprint, with a longer retention period. For example, the user can take a speed-first approach compressing messages to the regional Kafka cluster and compressing the data extremely in the aggregation cluster to keep them over a long period.

This feature opens all possibility of the cases described above.

Public Interfaces

It adds the following options into the Producer, Broker, and Topic configurations:

- `compression.gzip.buffer`: the buffer size that feeds raw input into the Deflator or is fed by the uncompressed output from the Deflator. (available: [512,), default: 8192(=8kb).)
- `compression.snappy.block`: the block size that snappy uses. (available: [1024,), default: 32768(=32kb).)
- `compression.lz4.block`: the block size that lz4 uses. (available: [4, 7], (means 64kb, 256kb, 1mb, 4mb respectively), default: 4.)
- `compression.zstd.window`: enables long mode; the log of the window size that zstd uses to memorize the compressing data. (available: 0 or [10, 22], default: 0 (disables long mode.))

All of the above are different but somewhat in common from the point of compression process in that it impacts the memorize size during the process.

Proposed Changes

Producer

The user can apply a detailed compression configuration into the producer; Since the default values are set to the currently used values, it will make no difference if the user doesn't make an explicit change.

Broker

Like the present, the broker will use the newly introduced compression settings to recompress the delivered record batch. The one thing to note is, we only recompress the records only when the given record batch is compressed with the different codec from the topic's compression settings - it does not care about the detailed configurations. For example, if the given record batch is compressed with gzip, level 1, and the topic's compression config is gzip with level 9, it does not recompress the batch since the codec is the same.

Whether to recompress or not is up to the codec difference only, like the present. The reason for this behavior will be discussed in the Rejected Alternatives section below.

Consumer

If the user uses gzip, the buffer size of `BufferedInputStream`, which reads the decompressed data from `GZIPInputStream`, is impacted by `` `compression.gzip.buffer` ``. In other cases, there are no explicit changes.

However, since the characteristics of the compressed data would be changed, the consumer may experience some performance impact per the data given. I will discuss this aspect in detail in the Benchmark section.

Benchmarks

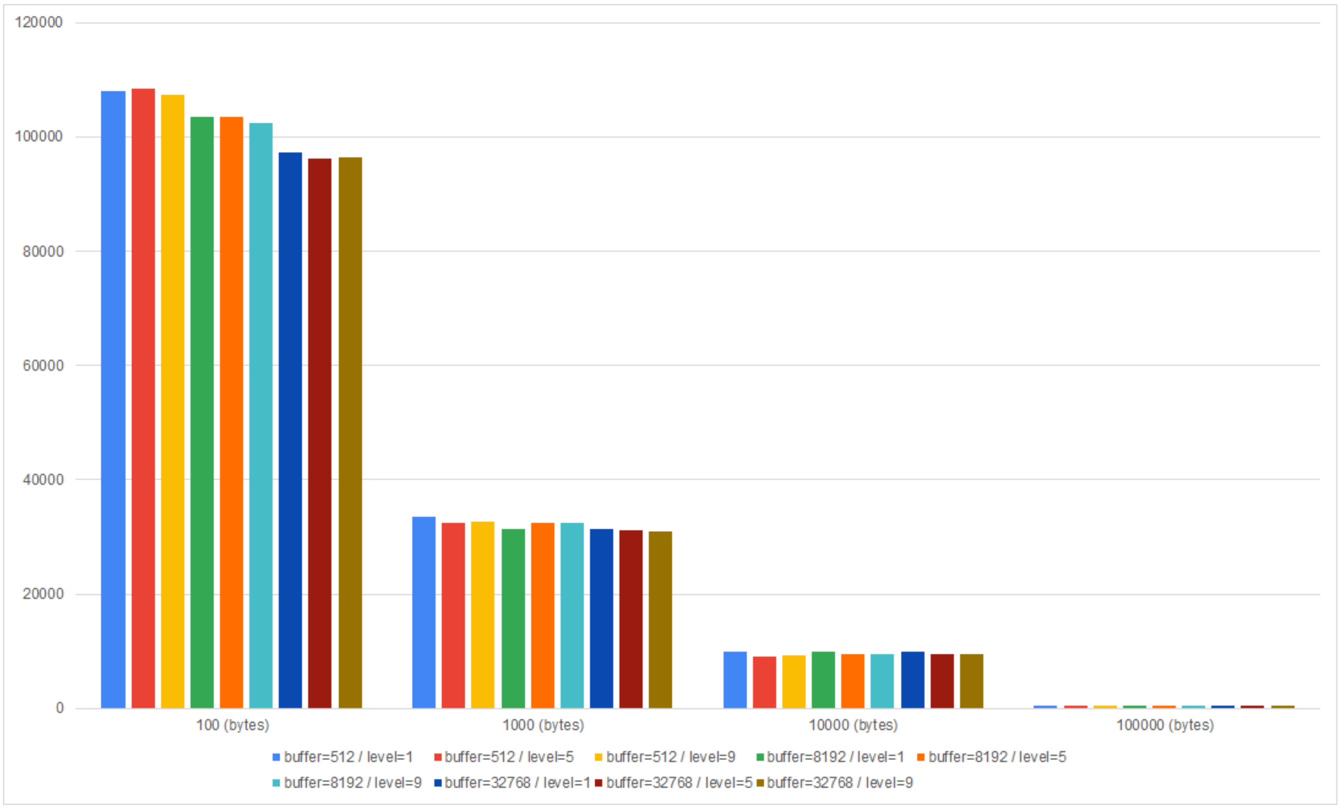
This section discusses the performance aspect impacted by the newly introduced configurations and the compression level proposed in KIP-390.

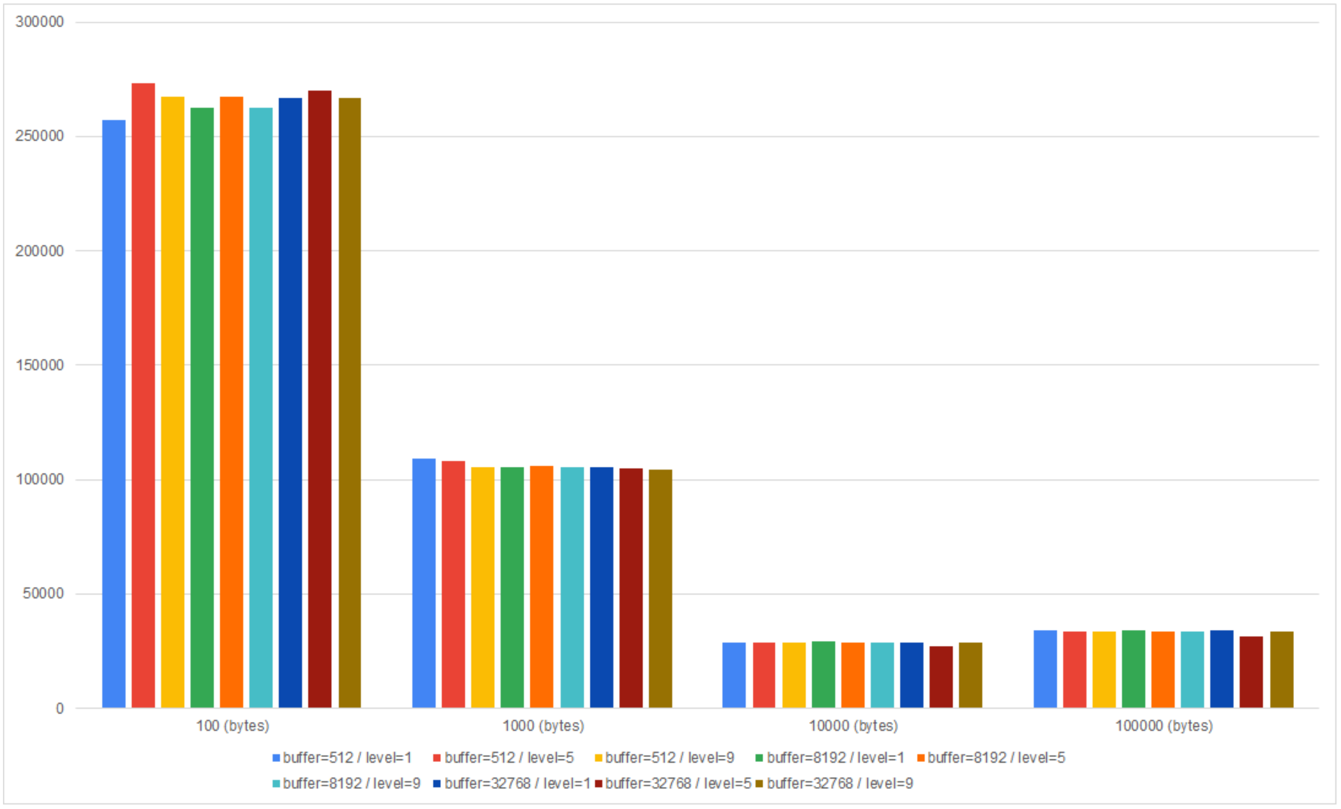
All tests were conducted on Intel i7-8565U CPU, 32GB RAM, Ubuntu 20.04, with Azul Zulu JDK 1.8.0_302. You can also see the raw data sheet here.

With Randomly Generated Dataset

Two microbenchmarks were run to see how each configuration option interacts with the other one; each benchmark measured the read or write ops/sec for various compression options.

Gzip

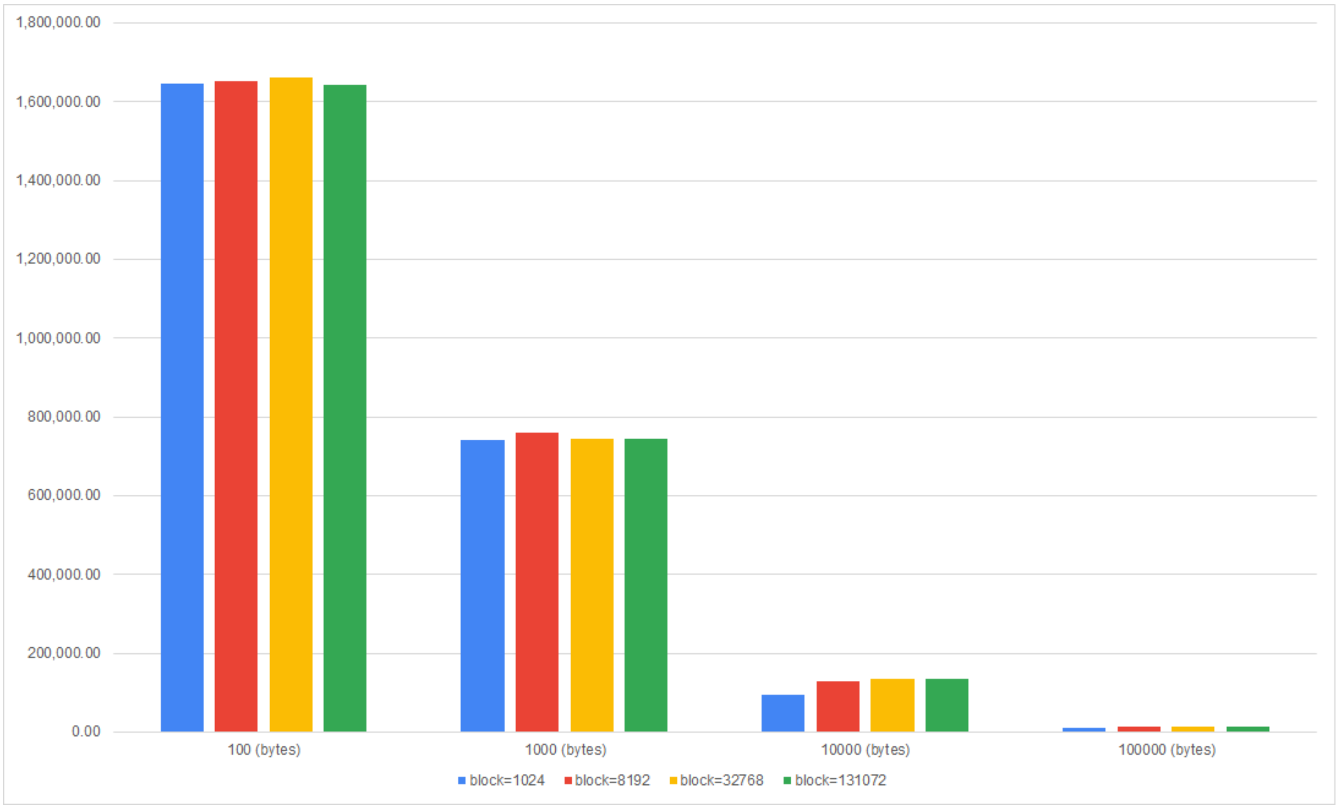


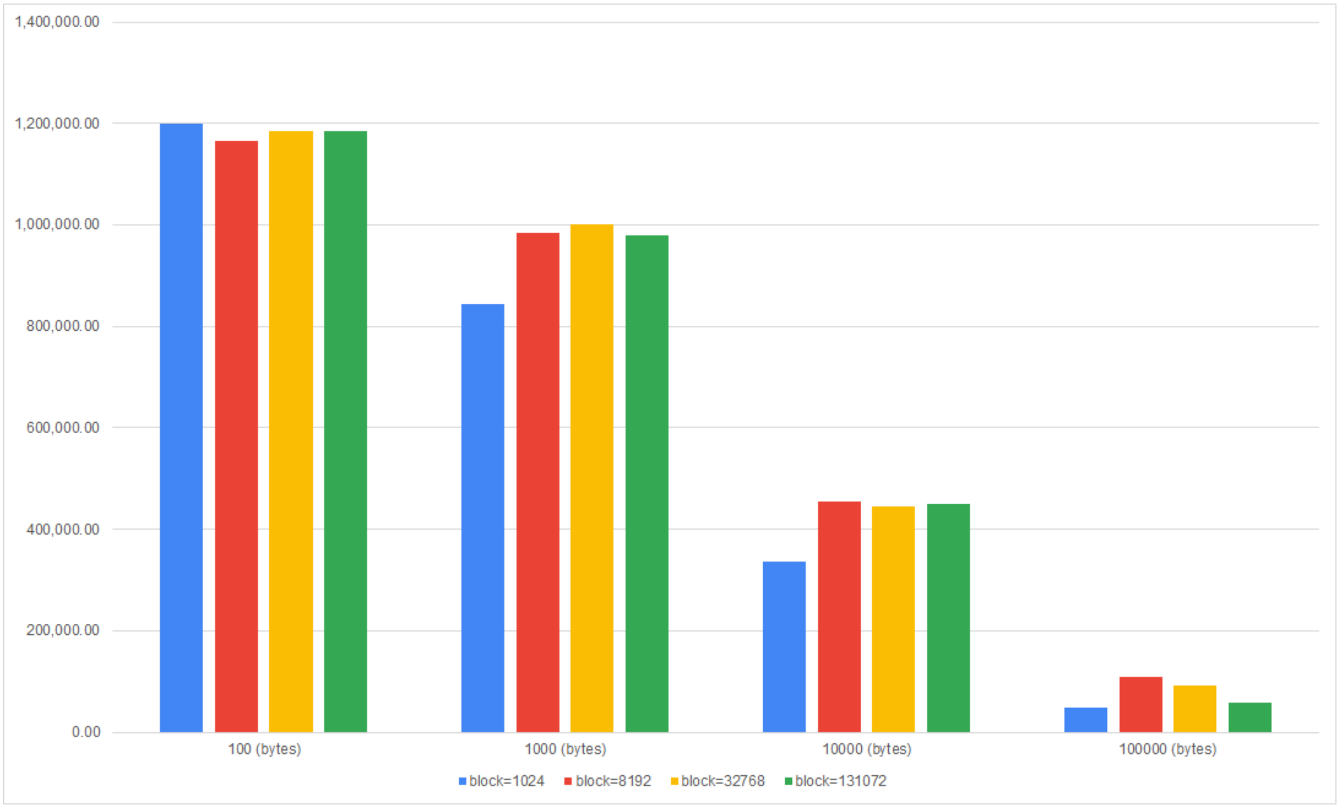


(Left: Gzip (Write ops/sec), Right: Gzip (Read ops/sec))

Compression level and buffer size did not significantly impact both writing and reading speed; The most significant factor was data size. The impact of compression parameters was restricted only when the data size was very small, and made only ~11% of differences.

Snappy





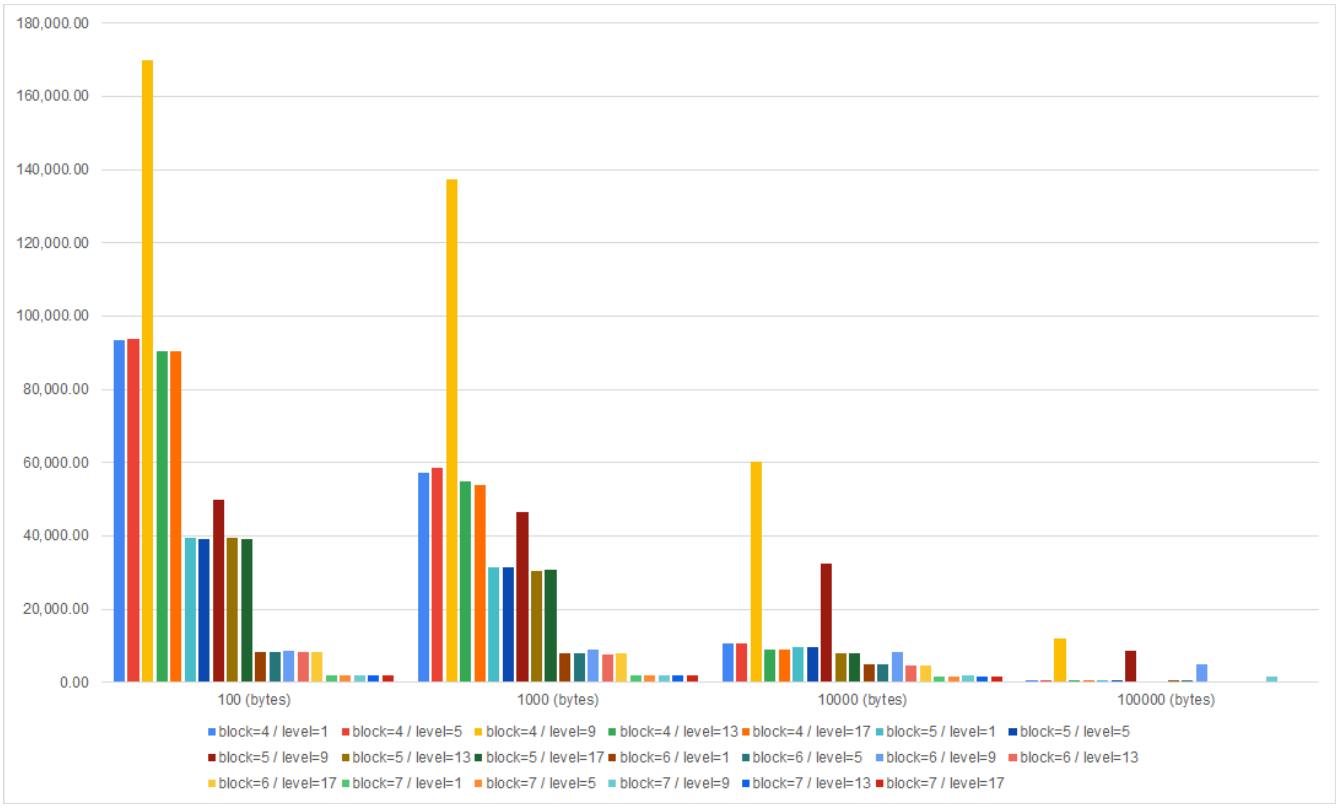
(Left: Snappy (Write ops/sec), Right: Snappy (Read ops/sec))

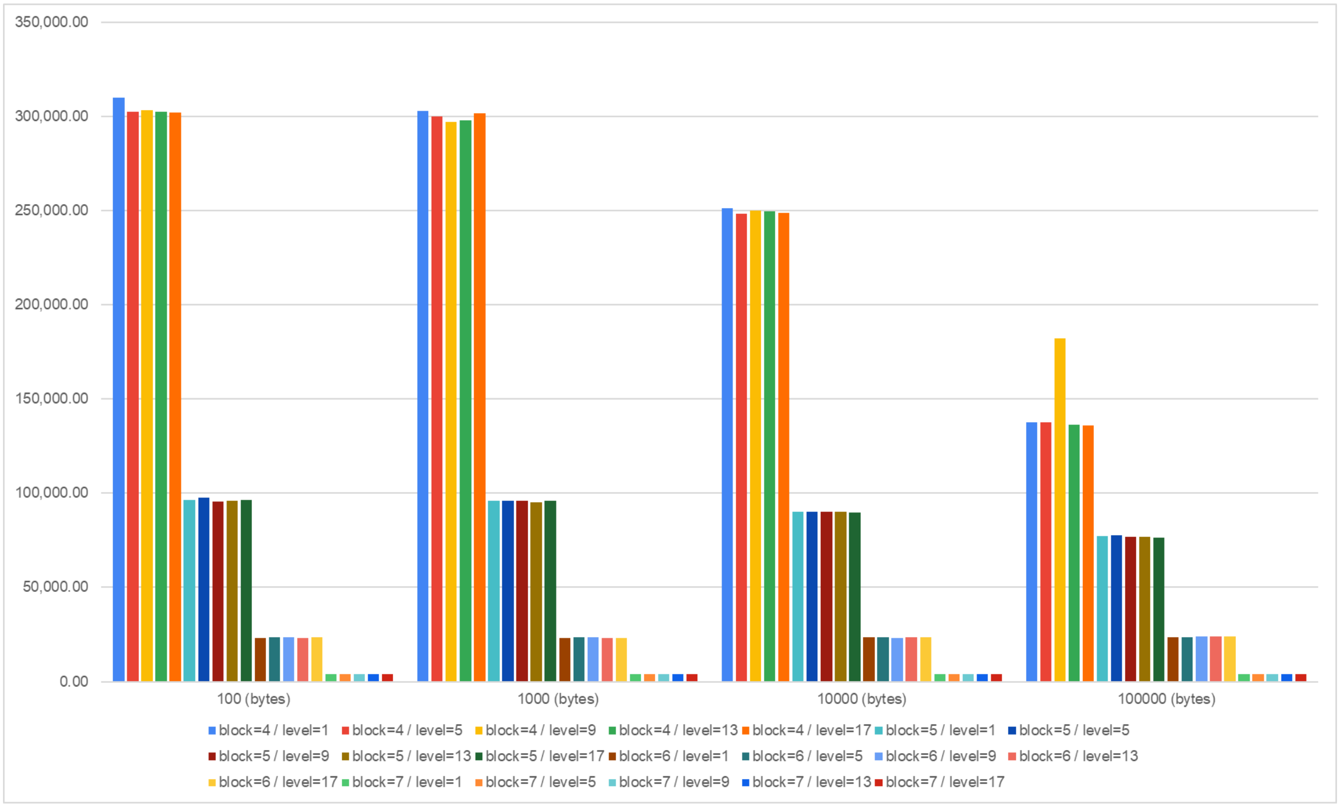
Changing the block size did not drastically affect the writing speed - however, if you stick to the too-small block size for large records, it will degrade the writing speed. For example, using a block size of 1024 with 100 bytes-sized data degrades the compression speed 0.01% only. In contrast, if the data size is 10000 or 100000 bytes, the gap in writing speed is 30% or 38%, respectively.

If the record size is so small (like less than 1kb), reducing the block size to smaller than 32kb (default) would be reasonable for both of compression speed and memory footprint. However, if the record size is not so small yet, reducing the block size is not recommended.

Block size also had some impact on reading speed; As the size of data increases, small block size degraded the read performance.

LZ4



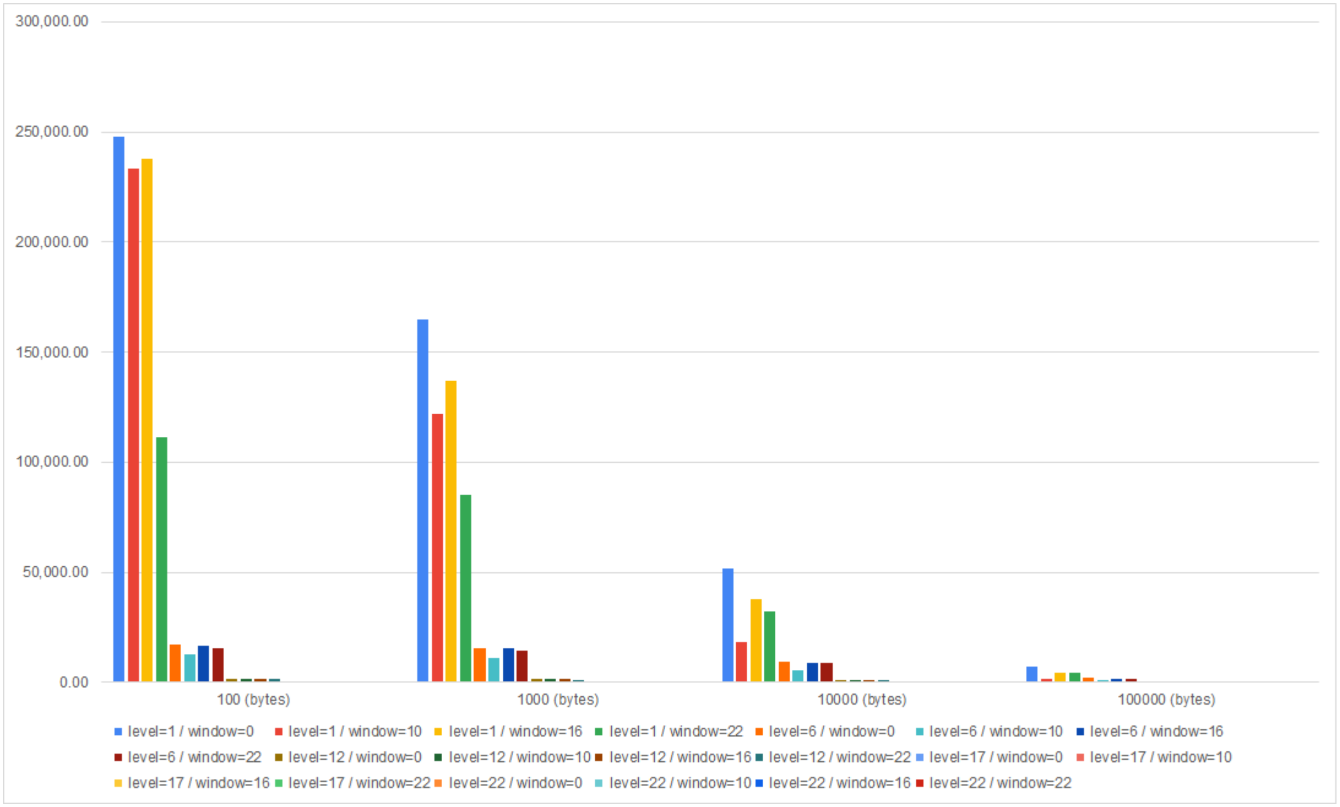


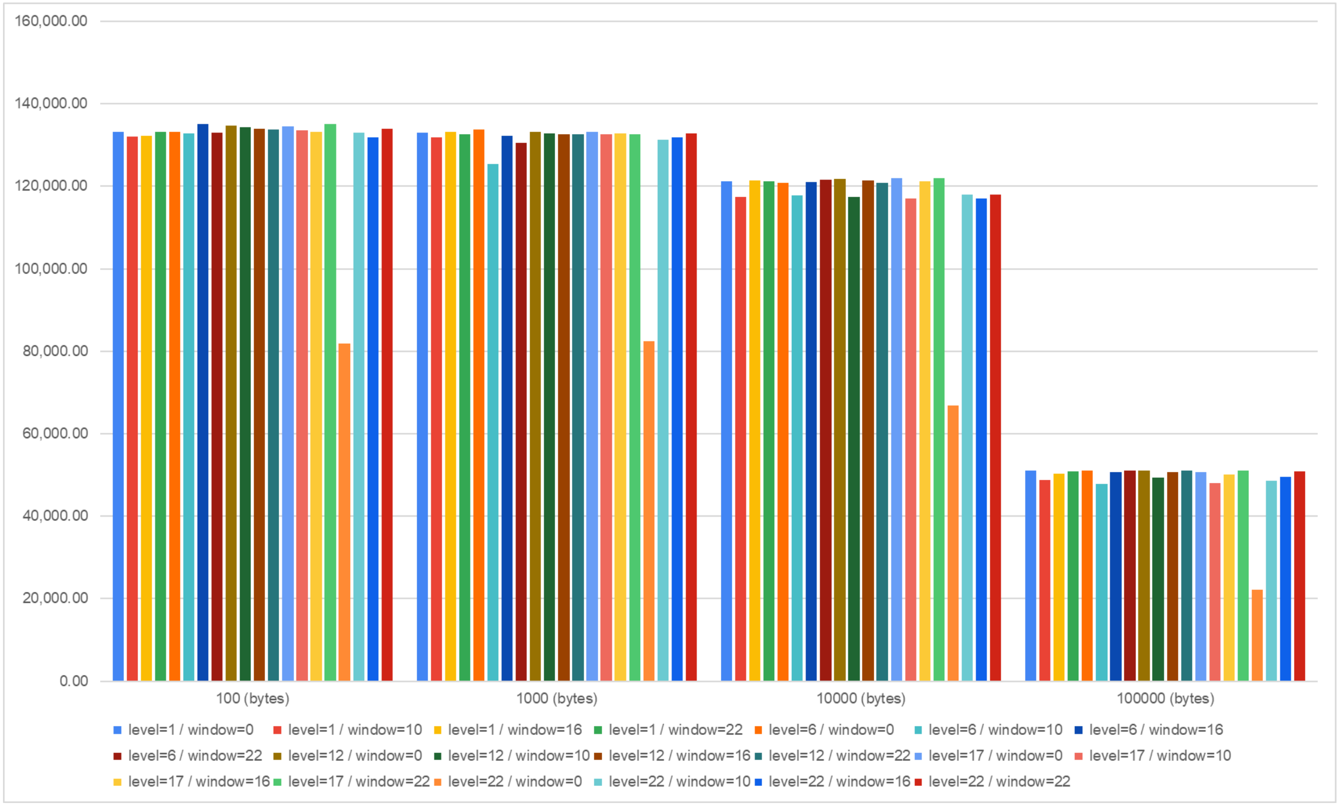
(Left: LZ4 (Write ops/sec), Right: LZ4 (Read ops/sec))

Level 9 (default) always did the best; The differences between the compression level were neglectable, and as the block size increased, and in these cases, level 9 also showed similar speed with the other levels.

Overall, as the block size increased, the writing speed decreased for all data sizes. In contrast, the reading speed was almost the same at <= 1.000 bytes. If the block size is too big, the ops/sec for reading operation also degrades.

Zstd





(Left: Zstd (Write ops/sec), Right: Zstd (Read ops/sec))

For the writing speed, as the data size increases, the ops/sec differences between the levels disappear. Disabling the long mode was better than enabling, but if the long mode is turned on, the windowLog of 16 resulted in the best.

The reading speed was almost the same at < 100,000 bytes. Interestingly, level 6 with non-long mode showed the worst case with all data sizes.

Conclusion

The Randomly-generated dataset test shows the following:

1. Focus the writing speed and the compressed size; the compression options in general relatively more minor impact than the reading speed.
2. Before deciding which codec you will use, check the data size first; for all kinds of codecs, allocating enough buffer or block was the most critical factor for both Write/Read ops/sec.
3. With Zstd, turning on the long mode with a smaller window size is not recommended.

Real World Dataset

A benchmark with a real-world dataset was run to see the actual compression ratio and speed.

Since the traditional `TestLinearWriteSpeed` tool measures the disk-writing speed of 'compressed' data only, not the compressed size or compression speed, I implemented another tool named `TestCompression`. It measures:

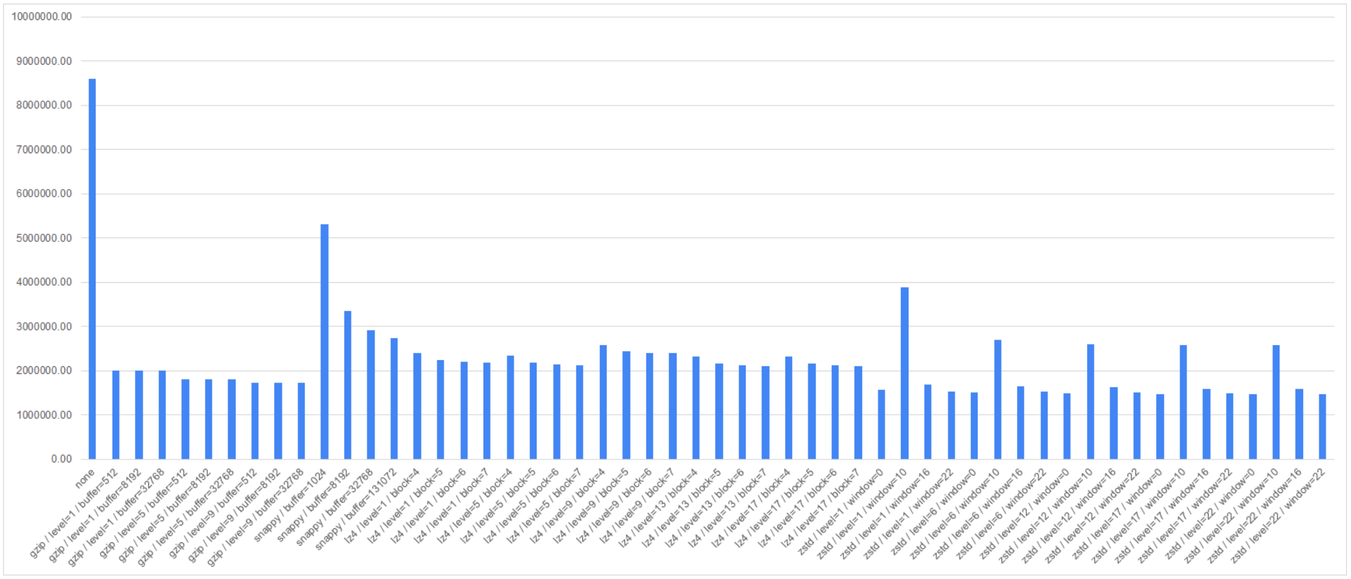
1. The size of the compressed data from given files or randomly generated binary payload.
2. The average elapsed time to convert the input into the compressed MemoryRecords instance.

This tool also aims to provide a reasonable expectation on the compressed size and speed for the users, who want to tune their compression settings. You can run this tool like the following:

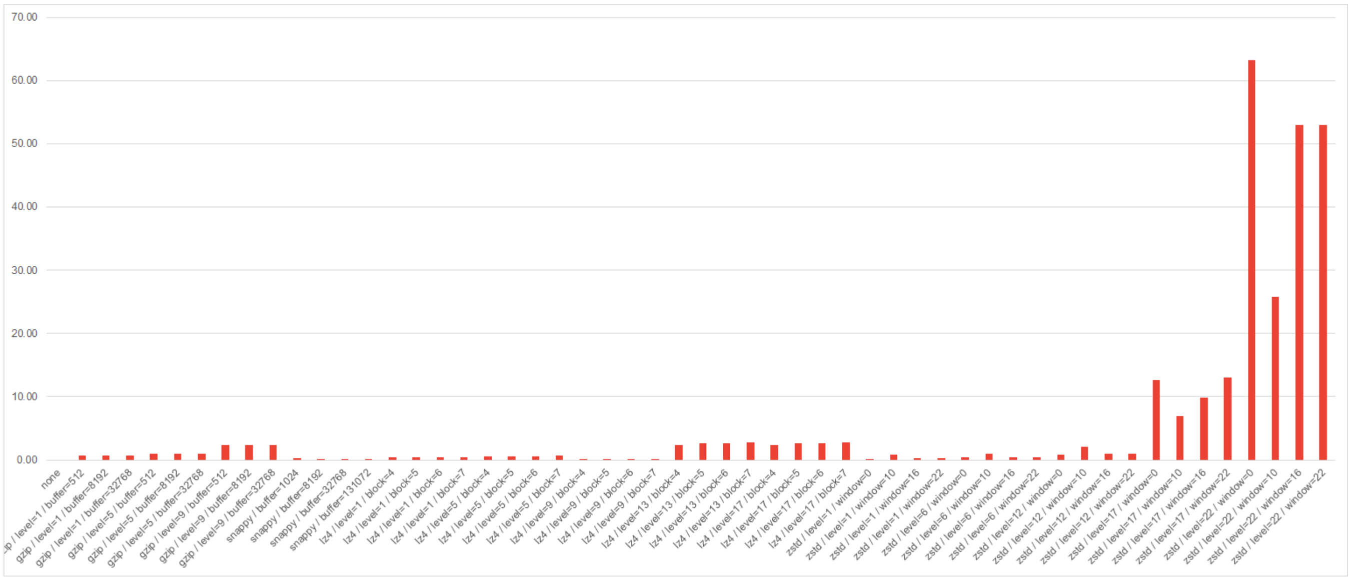
```
./gradlew clean jar testJar

# 100 batches, 1000 records, with zstd / level = 1 / window = 0
KAFKA_HEAP_OPTS="-Xms12G -Xmx12G" INCLUDE_TEST_JARS=true bin/kafka-run-class.sh kafka.TestCompression --dir ~
/dataset --batch-size 10 --batch-count 100 --compression zstd --property "compression.zstd.window=0" --property
"compression.zstd.level=1"
```

The dataset used consists of json data files collected from IoT sensors and measured the average compressed size / elapsed time of 100 batches, each consisting of 10 json files from the dataset.



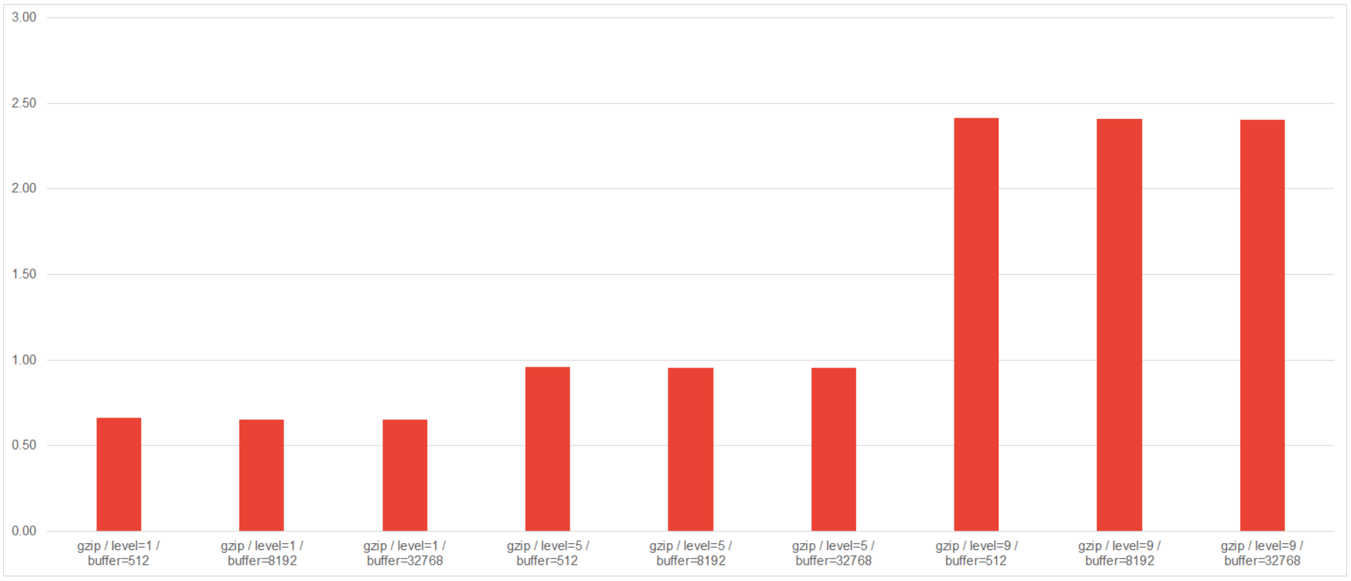
(Compressed Size in bytes, overall)



(Elapsed in sec, overall)

Since the variance in elapsed time is so significant, I will show the detailed graph per codec for better understanding in the following sections.

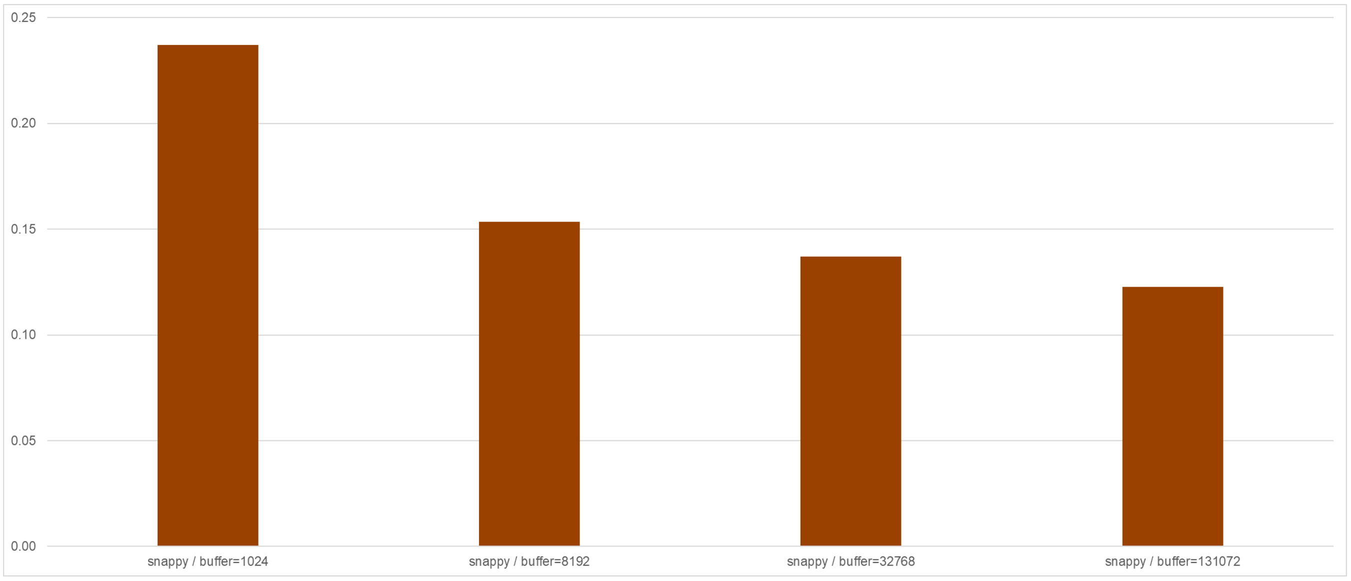
Gzip



(Gzip Elapsed time, detail)

Gzip took second place in terms of the compression ratio. However, although the variance in the compressed size is less than 15%, as we took the higher compression level, the elapsed time also increases, regardless of the buffer size. So, setting the level to 1 and assigning enough buffer to the consumer would be a preferable strategy in terms of producing and consuming both.

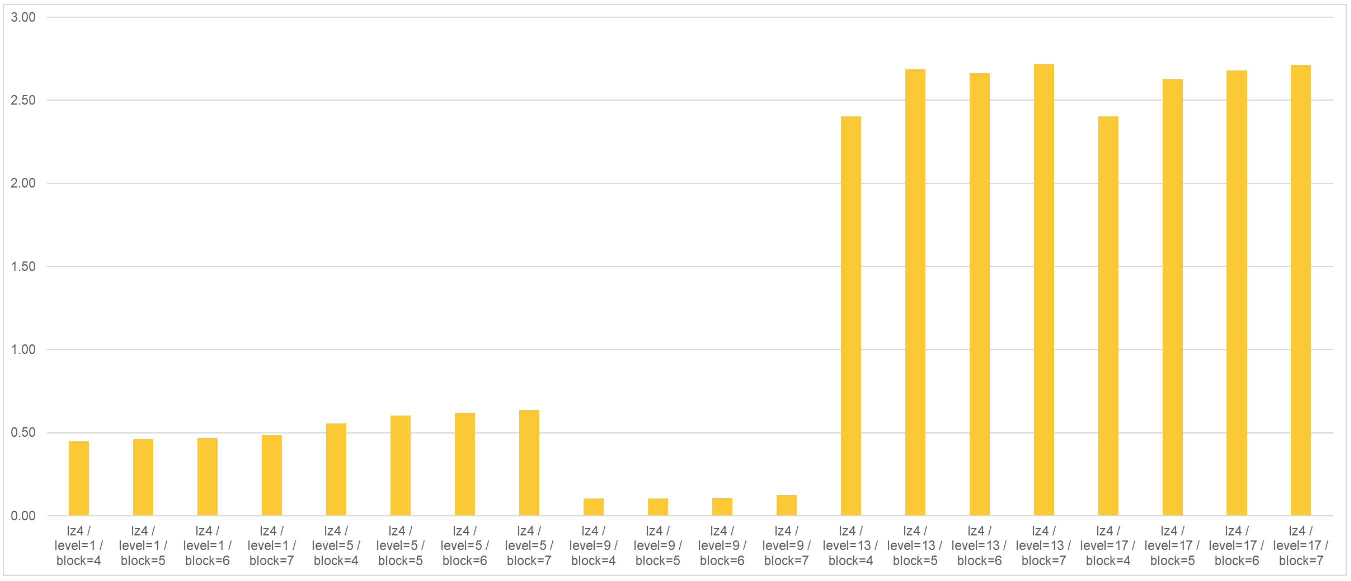
Snappy



(Snappy Elapsed time, detail)

Snappy took the bottom in terms of the compression ratio. The bigger block size snappy uses, both of the compression ratio and speed increased. So, finding the optimal block size seems to be the essential thing in tuning snappy.

LZ4



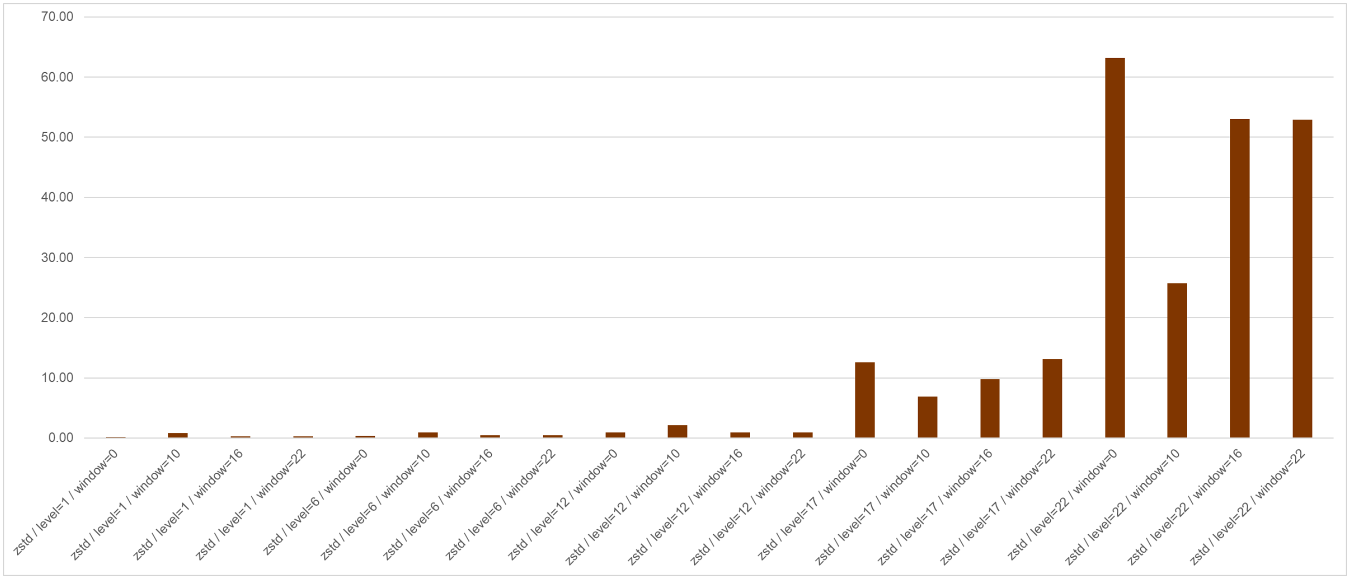
(LZ4 Elapsed time, detail)

The third place in compression ratio. The default configuration (level = 9, block = 4) outperformed all the other configurations in speed and size both.

Against all exceptions, the higher compression level did not reduce the compressed size - instead, the bigger block size did it.

In general, when the user has to use LZ4, using the default configuration would be the most reasonable strategy. But, if the user needs the smaller compressed data size, enlarging the block size and lowering the level would be an option.

Zstd



(Zstd Elapsed time, detail)

Zstd outperformed all the other configurations, both in compression ratio and speed. Comparing gzip / level=9 / buffer=32768 and zstd / level=1 / window=0, the best configuration among all the other and the most speed-first strategy in zstd, zstd outplayed gzip with only 90% of the size and 7% of compression time.

Turning on the long mode with a small window size (=10) did not help in both size and speed; as the compression level increased, the compression ratio also improved but, they had a small gain of size compared with zstd / level=1 / window=0 (less than 10%) with being overwhelmingly slow. If the smaller compressed size is necessary at the cost of the speed with zstd / level = 1 fixed, enlarging the window size to 22 would be a good approach - it takes twice compressing time but is still much faster than the other configurations.

Conclusion

In general, lowering the level and making the buffer/block/window size enough to the given data size resulted in the most satisfactory result. This result agrees with the randomly generated dataset-based result and the producer/consumer benchmark conducted in KIP-390 - which recommended lower the compression level to boost the producing speed.

Compatibility, Deprecation, and Migration Plan

Since the default values of newly introduced options are all identical to the currently used values, there are no compatibility or migration issues.

Rejected Alternatives

Recompression is enabled even when the detailed codec configuration is different

To enable this feature, we have to store the detailed compression configuration into the record batch. (currently, only the codec is stored.) It requires the modification of the record batch's binary format.

1. Do we need to use our invaluable reserved bits for this feature? Is that so important?
2. If the compression codec adds a new feature (like zstd's long mode), we must also repeatedly update the format. Is that worth it?

For the reasons above, storing detailed configuration options into the record batch is not feasible; since the broker can't retrieve the detailed options from the given record batch, recompression should be done only when the specified codec is different from the record batch's one.

Final Notes

As I stated earlier, this KIP is the following work of KIP-390: Support Compression Level. If this KIP is passed, it would be good to merge this feature with KIP-390 and separate the original config name proposed by it, '`compression.level`', into '`compression.[gzip,lz4,zstd].level`'. Here is why.

When I was first working on KIP-390, I had no idea of the per-codec configuration approach proposed in this KIP. It is why I chose to use '`compression.level`' for all codec cases, also with consistency with '`compression.codec`'. However, with some real-world tests, I found that it has some shortcomings:

1. Hard to validate. The valid level range is different from codec to codec (even snappy doesn't support it), so the validation logic inevitably becomes complex and hard to update.

2. Hard to update. Every time we have to change the codec, we also need to update the '`compression.level`' accordingly. Leaving '`compression.level`' inconsistent to '`compression.codec`' was such a common mistake and made problems repeatedly.

However, If we adopt per-codec config scheme ('`compression.[gzip,lz4,zstd].{xxx}`'), there are many advantages like:

1. Consistency: we can restrict all the codec-specific configurations into '`compression.{codec}.{xxx}`' format. This scheme can cope with the unique features introduced by the codecs in the future easily. (like: zstd's long mode.)
2. Easy to validate. Every time what we only need to do is validate '`compression.{compression.type}.{xxx}`' only. It dramatically simplifies the validation logic.
3. Easy to update. Predefine the codec-specific configurations beforehand, and if changing the codec is needed, update the '`compression.codec`' only. It is not only handy but also very error-free.

As a note, the public preview (based on 3.0.0) which provides both functionality is already following this scheme.