COMPRESSION IN CASSANDRA

Operators can set up compression on a per-table basis with Cassandra. By compressing the SSTable in user-configurable compression chunk\_length\_in\_kb, compression minimizes the amount of data on disk. The CPU cost of compressing data is only required when the SSTable is written since Cassandra SSTables are immutable. Since updates to the data will appear in separate SSTables, Cassandra won't need to decompress, overwrite, and recompress data when UPDATE instructions are sent. When a read occurs, Cassandra finds the pertinent compressed chunks on disk, decompresses the entire chunk, and then continues with the rest of the read path (read repair, merging data from disks and memtables, etc.).

Usually, compression methods compromise in one of the following three areas:

Compression speed: The rate at which data is compressed using the compression algorithm. The fact that data must be compressed before being written to disk makes this crucial for the flush and compaction pathways.

Decompression speed: The rate at which data is decompressed using the compression process. This is crucial for the read and compaction paths since, in order to be returned, data must first be fully read off disk and decompressed.

Ratio: The ratio that reduces the uncompressed data. This is commonly measured by Cassandra as the ratio of the disk's data size to its uncompressed size. A ratio of 0.5, for instance, indicates that 50% of the uncompressed data is stored on disk. This ratio is disclosed by Cassandra for each table as the SSTable Compression Ratio.

By default, Cassandra provides five compression algorithms that vary in these aspects. The following table, which provides an extremely rough grading of the various options by their performance in these areas (A is relatively good, F is relatively bad), should help you choose a starting point based on the requirements of your application, even though benchmarking compression algorithms depends on many factors (algorithm parameters such as compression level, the compressibility of the input data, underlying processor class, etc.):

| Compression Algorithm | Cassandra Class | Compression | Decompression | Ratio | C\* Version |
| --- | --- | --- | --- | --- | --- |
| [LZ4](https://lz4.github.io/lz4/) | LZ4Compressor | A+ | A+ | C+ | >=1.2.2 |
| [LZ4HC](https://lz4.github.io/lz4/) | LZ4Compressor | C+ | A+ | B+ | >= 3.6 |
| [Zstd](https://facebook.github.io/zstd/) | ZstdCompressor | A- | A- | A+ | >= 4.0 |
| [Snappy](http://google.github.io/snappy/) | SnappyCompressor | A- | A | C | >= 1.0 |
| [Deflate (zlib)](https://zlib.net/) | DeflateCompressor | C | C | A | >= 1.0 |

In general, LZ4 is the best option for a performance-critical (latency or throughput) application since it achieves a great ratio per CPU cycle used. For this reason, Cassandra sets it as the default option.

However, because Zstd may achieve a sizable extra ratio over LZ4, it might be a superior option for storage-critical applications (disk footprint).

For backwards compatibility, Snappy is retained, however LZ4 is usually better.

For backwards compatibility, deflate is retained, however Zstd is usually preferred.

COMPACTION CONFIGARATION :

Compression is set up as an optional input to CREATE TABLE or ALTER TABLE, per-table. With every compressor, there are three alternatives available:

class: designates the compression class to be used (default: LZ4Compressor). ZstdCompressor and DeflateCompressor are the two "good" ratio compressors, whereas LZ4Compressor and SnappyCompressor are the two "fast" compressors.

chunk\_length\_in\_kb: indicates how many kilobytes of data are contained in each compression chunk (default: 16KiB). Bigger chunk sizes enhance the ratio and provide compression algorithms with more context, but they also necessitate more reads to deserialize and read data off disk. This is the main tradeoff in this situation.

The following extra options are supported by the LZ4Compressor:

lz4\_compressor\_type (default fast): indicates whether to use the fast (also known as LZ4) or high (also known as LZ4HC) ratio version of LZ4. With the help of the lz4\_high\_compressor\_level option, operators can adjust the performance <→ ratio tradeoff in the high mode thanks to its variable level. Keep in mind that using the Zstd compressor might be better in versions 4.0 and higher.

lz4\_high\_level\_of\_compressor (default 9): a figure that ranges from 1 to 17, inclusive, that indicates how much CPU time should be used to increase the compression ratio. Higher levels receive greater compression ratio while being slower overall, whereas lower levels are typically "faster" but receive less ratio.

In addition, the ZstdCompressor supports the following options:

compression\_level (three by default): An integer representing the amount of CPU time to be spent attempting to increase the compression ratio, ranging from -131072 to 22 inclusive. The speed increases with decreasing level (at proportionate cost).

However, operators should be mindful that compression changes take time to take effect. Since SSTables are immutable, the compression of the data is applied during the writing process and remains unaltered until the table is compacted. If an operator needs compression changes to take effect immediately, they can use nodetool scrub or nodetool upgradesstables -a to trigger an SSTable rewrite. These commands will rebuild the SSTables on disk and re-compress the data. If an operator issues an ALTER TABLE change to the compression options, the existing SSTables will not be modified until they are compacted.