Compression: CPU vs. Memory Tradeoff

Karthik Rao, Raahul Acharya, Teddy Liu, Tomislav Zabcic-Matic
Harvard College

ABSTRACT

We present a test data system, compression architecture, and evaluation of compression schemes used in modern-day LSM trees. Our aim is to quantify and derive the tradeoffs of using compression schemes to store on-disk data in the levels of an LSM tree, a prominent noSQL data structure. From this paper, we derive conclusions on when compression should be used and if so, what compression schemes work best theoretically and experimentally. We also use our evaluation to draw information on what workload types and data structure specifications are best suited for compression, and what benefits compression brings to handling these workloads in terms of memory minimization, computational efficiency, and CPU utilization. We found that while compression can reduce data size (in bytes) by up to 99%, this does not translate to faster query run-times due to extremely high overhead costs in decoding during reads and both encoding and decoding during writes.

1. INTRODUCTION

Over the last several years, Log-Structured Merge-based (LSM) Trees have emerged as the primary data structure in noSQL systems. Currently used for write-intensive workloads with not many storage constraints, LSM trees have been integrated by the likes of Facebook and Google to store a multi-terabyte inflow of daily data, much of which will not be read. Because LSM trees store all their non-buffer data on disk, accesses to data during reads can be very costly due to high I/O costs for disk to main memory data movement. As more and more data is stored in an LSM tree, on average more and more I/Os are needed to scan through the levels looking for a particular key or range of keys. For this reason, auxillary in-memory structures like bloom filters or fence pointers are used, but even these cannot fully minimize the number of calls to disk needed for a scan.

Compression schemes, applied to a wide range of data types and structures, are a potential solution to this problem. They can reduce file size immensely, reducing storage costs by up to 99%. However, they are very variable and do not always achieve strict and constant results on different data inputs. Normally, this would cause great concern in rigid data structures like relational databases with multiple indexes over the data. For best improvement in such SQL-based systems, a uniform spacing between data (equivalent number of data points per page) is key. LSM Trees, however, do not hold such a rigid structure, making them a potential application for powerful compression schemes. They are quite abstract and centered around the fundamental concept of storing data in files on disk. Once stored in-file, the data can be adjusted and compressed in whatever way one sees fit and adjustments can be made to the tree metadata as well. As long as these compressed files can be accessed and then decomposed into standard format before being read, and then compressed after being written to, an LSM tree can very efficiently accommodate compression. Therefore, this is potentially a very interesting problem to explore and one that has not been done yet. *Optimizing Space Amplification in RocksDB* and *Rose: Compressed Log Structured Replication* both mentioned the potential of using compression to improve LSM trees, but they 1) only focused on the overhead costs (storage and cpu) of compression, 2) only considered a minimal number of knobs and adjustments over compression schemes, and 3) did not concisely and effectively convey the potential tradeoffs. Therefore, we would like to:

- Expand the current semantics of LSM trees to include knobs for compression among different levels and tiers.
- Theoretically model the overhead costs, compression efficiency, and I/O improvements of certain compression schemes on LSM trees with parameterized workloads.
- Test different compression schemes on various data sizes and distributions, analyzing where optimizations are observed and why.
- Provide a comprehensive conclusion on where compression schemes are useful, what

compression schemes are useful, and on which workloads/tree structures compression schemes are useful.

2. LSM MODEL

We architected and implemented an tiered LSM tree tailored to ingesting both uncompressed and compressed data and loading, reading from, and writing to this data. Our main goal was to be able to test the tradeoffs of storing compressed data versus uncompressed data on various facets of an LSM tree, and our structure allows us to efficiently do so.

During instantiation, we predefine a compression scheme (or none) for all levels that will guide how the tree flushes data to disk and decompresses it for scans. We also pre-set a tunable level ratio and buffer size. The level ratio represents the number of components (contiguous runs of sorted key,values) per level, and the scaling factor by which a component is increased in size in sequential levels. The buffer size represents the number of key-value pairs that are written to main memory before being flushed to disk. Once the buffer is filled and flushed to disk, it forms a new component in Level 1 of the tree and pushes all other components back, merging and cascading components where necessary.

However, subcomponents, not components, are the most granular structure in our tree. Each subcomponent holds reference to a file that contains one buffer-size worth of sorted key-value data and metadata on the max and min keys within this file. Because these data sizes are fixed, each component is made up of a variable number of subcomponents depending on the level it is in. Also, when performing reads, data is pulled to disk one subcomponent, not component, at a time.

Given this structure, we mathematically modeled the parameters and dependencies on which read and write run time relies. These results guided our experimentation and provided direction for further work.

Firstly, we will define in the following table the high level parameters used throughout these calculations. Note, that all sizes are measured in bytes for consistency.

Table 1: Model Parameters

Parameter	Definition
B	*buffer size*
p	*page size*
N	*full workload size*
r	*compression ratio (1 when no compression)*
k	*key*
L	*number of levels in tree*
n	*number of key-value pairs*
X(n)	*uncompressed size of n key-value pairs*
s	*selectivity of workload (0-100)*

Next, we will define run-time functions and attempt to approximate their values based on computational and I/O costs.

Table 2: High-Level Functions

Parameter	Definition
C(X(n),r)	*time it takes to compress n key-values with ratio r and store*
D(X(n),r)	*time it takes to pull and decode n key-values compressed with ratio r*
Φ(k)	*Number of subcomponents scanned before key k is found*

To approximate these three functions, we note that cost comes from 1) data movement and 2) computation. Data movement depends solely on the size of the data being moved to and/or from memory and computation depends on the search/sort algorithm being used. *Φ(k)* is unique in that it depends both on key distribution and selectivity. We will assume for now the keys follow a uniform distribution.

The cost of encoding X(n) is a linear computational cost of iterating through n key-values and performing a compression after observing a group of elements (the schemes we work with are primarily element-wise). The

data movement cost comes from writing the compressed data to disk, which is now of size rX. Data is written in pages and therefore the number of pages written times some constant representing movement cost per page read/written defines the total time writing to disk. Decoding works the same way but now the data movement cost comes from moving rX bytes *from* disk *to* memory. Because computational decoding is also a linear cost (for similar schemes), the on-memory computational cost remains the same as well.

$\phi(k)$ was trickier but for simplicity's sake we defined it to be agnostic to key and also to fall somewhere between 1 page search and the total number of pages in the tree. When assuming a uniform key-distribution, we can deem fence pointers to be trivial and therefore assume the number of subcomponents we search to fall on a uniform distribution (from 1 to the total number of subcomponents in the tree) as well. Next, proxy defining per-component selectivity to be the probability that a key will be found in a qualifying component, we can scale the number of pages up or down based on s.

Table 3: Function Approximations

Parameter
$C(X(n),r) = O(n) + c\lfloor rX/p \rfloor + 1$
$D(X(x),r) = O(n) + c\lfloor rX/p \rfloor + 1$
$\phi(k) = (1-s)(\lfloor LN/B \rfloor) + 1$

We can now define the cost of reads and writes. We define point read run-time as the cost of decoding (and simultaneously scanning) one subcomponent times the expected number of components that should be scanned before a key is found. Similarly, we envisioned range queries from key1 to key 2 as a set of point queries for all the keys in this range. Therefore, the runtime of a range query is additive over the runtimes for all the point-queries involved. However, this must be capped at the number of pages in the tree, so a min() function is called on it.

Writes are mathematically impossible to model because they fall on such a variable runtime scale. A write can be anything from an append on an in-memory array to a full LSM tree cascade and merge algorithm which involves L external sorts and essentially compresses, decompresses, and sorts the whole tree in the process. It can also be anything in between. Therefore, we choose to only model a worst-case and best-case run time for writes, and then assert that a write runtime will fall anywhere in between but not outside of this range.

Table 4: Read-Write Runtimes

Parameter
Point Query $PQ(k) = D(B)\phi(k)$
Range Query $RQ(k_1,k_2) = min((k_2 - k_1)D(B)\phi(k),\lfloor LN/B \rfloor+1)$
Write $[O(1),O(NLogN) + O(BlogB) + C(N) + D(N)]$

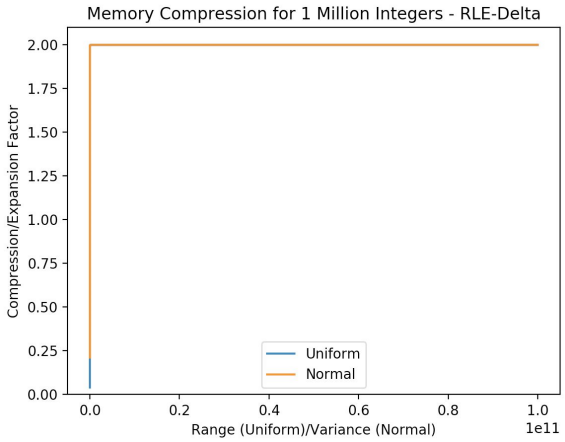
Given these models, we can posit hypotheses on what alterations in workload or improvements in tree structure will reduce query costs. See Section 5 for further steps on this exploration.

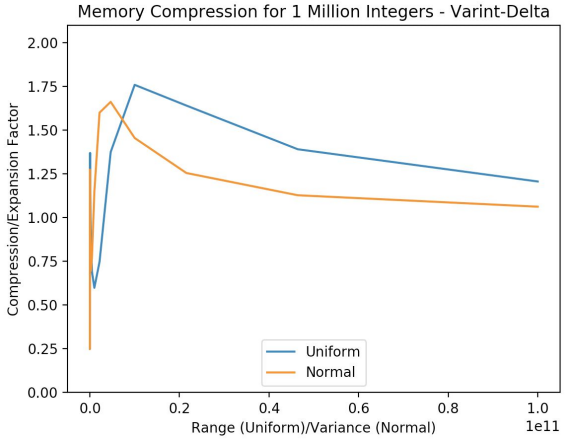
3. COMPRESSION SCHEMES

We focused on two compression schemes, one involving run-length encoding (RLE) and the other involving variably-sized integers (Varint Encoding). Both schemes used delta encoding as a baseline for reducing the apparent mean and variance of the underlying data. Subsequently, either RLE or Varint Encoding was applied to the delta-encoded data.

Run-Length Encoding represents a sequence of integers as a list of pairs, where the first element is an integer that represents a real data element, and the second element is an integer representing the number of occurrences in a row of the first element in the pair. This form of encoding performs exceedingly well when the underlying data contains long runs of elements that are the same, so that there are not many necessary additional metadata points that cause memory expansion. If there are too many short runs in the RLE, the encoding will actually possibly result in memory expansion rather than compression, with a worse-case expansion of 2x the original data size. Thus, RLE-delta encoding is best suited to datasets which are roughly sequential, or have many consecutive increments of the same size. Particularly in the case of sequential keys, such as in an auto-incrementing key generated for each new value, this method results in a fixed 16 bytes of data needed in order to represent the underlying data.

Varint Encoding takes advantage of the fact that many integers do not require the full four bytes available to standard C integers in order to properly represent them, particularly in the case of the integers produced as a result of delta encoding. This causes Varint-Delta to have far better average case performance than RLE-Delta performance. However, in the best case, Varint-Delta is limited to just above a ¼ compression factor, whereas the ratio of compressed data to raw data is infinitesimal in the case of RLE-Delta. Due to the metadata required to denote a run of integers of a specific size (1, 2, or 4 bytes), Varint-Delta may result in memory expansion. However, it may be possible to limit this effect, and future work would focus on more sophisticated manners of reducing the possibility of memory expansion in Varint-Delta.





The above two figures show the memory compression or expansion rates for RLE-Delta and Varint-Delta Encoding for a set of 1,000,000 integers (as a ratio of the size of the compressed data divided by the size of the raw data). Blue lines represent the case in which the raw data is distributed uniformly at random, and orange lines represent normally-distributed raw data. Due to the nature of LSM trees, all data is sorted before compression. The x-axis represents the range of integers in the uniform case, and the variance in the normal case. Results are only presented for 1 Million keys as other numbers of keys have the same outcomes for identical ratios of range/variance vs the number of keys.

4. EVALUATION

We implemented a prototype LSM tree in C++ using ~1000 lines of code. We also built dual-layered compression schemes for testing by combining fundamental encoding methods in RLE (run length encoding), varInt, and delta encoding. Using these building blocks, we constructed an RLE-over-Delta and an RLE-over-varInt compression scheme that essentially applied RLE to files after they had first been compressed by delta and varInt encoding, respectively. Double compression obviously increased the overhead encoding and decoding costs, but provided much more powerful disk storage reductions.

We stored decompressed component data on disk as binary files where integers were contiguously stored (with no symbol separating them) as key,value,key,value pairs. Given this architecture choice and the fact that keys could range from 0 to 2^{32} and values from -2^3 to 2^{32}, we realized that, regardless the key distribution or value distribution in a given workload, it was highly unlikely that we would come across continuous runs of key,value pairs that were largely repetitive or sequentially increasing/decreasing, making RLE and Delta encoding quite useless by themselves. In fact, we found that when applied over files, RLE in fact increased file size by around 45%. However, RLE applied over Delta encoded data provided extremely large file size reductions for certain workloads, particularly where keys and values were distributed sequentially and differences between keys and values were often the same.

For our compression efficiency tests, we recorded compression ratios for files with various numbers of key,value pairs stored in them- ranging their size from 10 key,value pairs to 1,000,000. We found that compression ratios for our RLE-over-Delta encoding schemes (defined as compressed file size [in bytes] over original file size) decreased from over 100% to 1% at the 1,000,000 entry file. This was unique to the sequentially distributed key,value workload, but similar reductions were noticed across other workloads as well, making RLE-over-Delta a very powerful compression scheme for data in LSM trees. We predict that this compression ratio reflects potentially

further reductions given larger file sizes. 1,000,000 key-value pairs is not extremely large by any means for a single file, and compression efficiency was very high still. We believe this trend will continue for even larger files and compression ratio will further decline for such workloads.

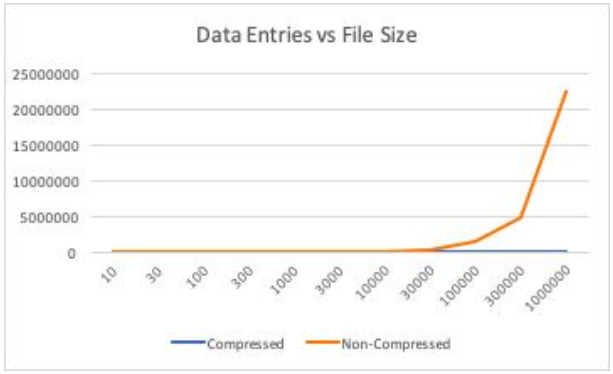


Figure 1: Data Entries (number of key,value pairs) per file plotted against their respective file sizes in bytes for compressed files versus uncompressed files.

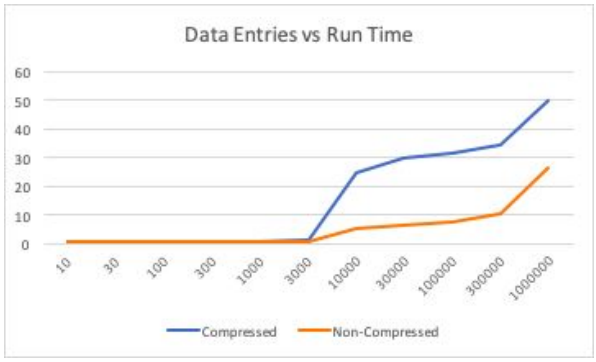


Figure 2: Data entries in-file plotted against how long running 100,000 queries (50% reads) took (in seconds)

For our computational speedup and CPU utilization tests, we used a Linux Virtual Box in Ubuntu to run tests on a constant environment. We found that CPU utilization for all data sizes, buffer sizes, and workload configurations ranged around 99%, making it a given constant for our tests. For our most revealing run-time tests, we ran 100,000 queries with equal amounts of reads and writes on various data sizes. Interestingly, we found that compression algorithms do not, in fact, speed up querying on workloads with such read/write distributions. Run-times for compressed and uncompressed files are around the same for data sizes up to 3000 key,value pairs, but then after that compressed file run-time shoots up. At around 300,000 key,value pairs, the run-time differences between compressed and uncompressed values seem to level out at around 25.0 seconds, which is quite inexplicable, but possibly due to an offsetting effect between overhead compression costs and reduced I/Os for compressed files.

These results, however, could have stemmed largely from our LSM tree structural choices. One of our tunable knobs was the buffer size, which represents both the number of integers written on main-memory before a flush to disk and the number of data entries stored in a sub-component. For a given point read, each subcomponent whose potential key range (min-max) includes the key in question is pulled to disk and decompressed before it is scanned (until the key is found). The I/O benefits from compression theoretically come from the reductions in I/Os needed to pull a whole subcomponent to main-memory. Therefore, when the buffer size is chosen such that both the compressed file size and decompressed file size fit within one CPU page (4096 bytes) or generally the same number of CPU pages, the same number of I/Os is needed to read a subcomponent in uncompressed and compressed format. This completely eradicates the need for compression in the first place, and only adds the overhead cost of encoding and decoding to the overall runtime.

Therefore, we decided to test what an optimal buffer size would be given our RLE-Delta encoding scheme. On a data set of 10,000,000 key,value pairs, we ran 1 load query (loading the whole dataset onto disk), and 100,000 read queries on 8 different buffer sizes and tested runtimes for load and reads. Note, however, that in our graphical representations these buffer sizes correspond to the number of key,value pairs represented per subcomponent- to get the total number of uncompressed bytes supported in our buffer, multiply this number times 8 (4 bytes per integer and 2 integers per key,value pair). Interestingly, we found that for both reads and loads, there was an optimal buffer size, and it was the same at 2048 key,value pairs (16,384 bytes).

This optimization was quite interesting but we hypothesize that it depends largely on compression ratio. For these datasets/data sizes, we were seeing compression ratios of around 25%, meaning that subcomponent sizes, when compressed, would be ¼ the number of bytes. Given this knowledge, a buffer/subcomponent of 16,384 bytes would be compressed to around 4096 bytes, which is equivalent to 1 page size. That provides much light on this convex optimality, because now we know that running reads and loads is fastest at 2,048 key-value pairs because it creates compressed subcomponents that can be fully pulled into

main-memory with one I/O. Buffer sizes smaller than this would lead to files that would not fully fill a page pull and buffer sizes larger than this would create files that require multiple page pulls (the last of which will inevitably be not fully filled).

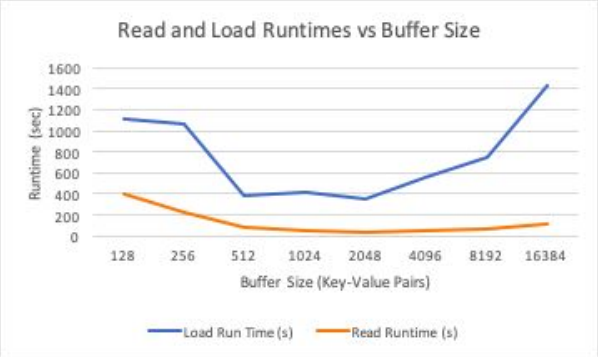


Figure 3: Run-times for loading 10,000,000 key-values onto disk and executing 100,000 read queries at numerous buffer sizes

5. FURTHER STEPS

The data we found pointed to the fact that compression was highly efficient on specific workloads, but file encoding and decoding overhead costs during reads and writes were larger than the reduced I/O costs provided by compression. This is unsurprising, as compressing a file on disk requires scanning through the whole file. This in itself requires a large number of I/Os, and once files get large enough this is very costly. Moreover, the way our testing LSM tree was built, a full file compression was required every time the in-memory buffer was filled. Further, if a full-level merge was called upon due to cascading effects, this could become extremely costly as a whole level's worth of files would have to be decompressed, sorted, and then compressed once more. We could theoretically make our LSM tree one infinitely sized on-disk level requiring no cascade merges, but this would make reads much longer as the data structure would essentially become an array with fence pointers on every tier. Therefore, we would like to take further steps into fleshing out what LSM tree specifications work best with compression schemes by altering our leveling ratio, building and testing on leveled trees as opposed to tiered ones, and looking for an optimal buffer size (if that exists). Considering merge/write optimization, we would like to look into compression schemes that are sortable in compressed format, because these can be merged without a prior decompression and then recompression, addressing the bottleneck mentioned above.

Another step that we plan to explore uses our run-time modeling from Section 2 to find workload-specific compression schemes. We derived from that section, given a predefined set of constants, a theoretical run time for point and range reads and point writes for compressed and uncompressed data. Depending on our query workload, we know that compressed data becomes faster to work with when reads or writes (or both) run faster on it as opposed to on uncompressed data. Two of our primary parameters defining read and write times were r and $\Phi(k)$ which represent compression ratio and average # number of subcomponents read before success. Both of these terms are specific to the data being read, and so we were not able to define them earlier. However, we aim to use data from various workloads to derive an empirical model that can provide approximations for these functions given an inputted workload. This will require extensive workload distribution models and techniques that can map data to numerical distributions and queries to selectivity levels.

Once this is achieved, we can take a given workload w, define a theoretical compression ratio r that a given compression scheme s will achieve with w, approximate the average number of pages that will have to be pulled to memory per read using this compression scheme, and then use this to model the runtime under compression versus not-under compression. If reads or writes are faster using scheme s, then the LSM tree will potentially compress the incoming data. This holds immense potential, in that it can answer questions such as whether to compress an LSM tree or not, but also can be expanded to answer questions like should different compression schemes be used for different levels of a tree due to changing workloads.

6. CONCLUSION

In this paper, we presented the results of using compression schemes on LSM trees on storage costs, query run time, and CPU utilization. We empirically and mathematically modeled the compression ratios certain schemes achieved on specific workloads, and then used these to model the theoretical read and write run-times on an LSM tree. We found that while encoding can achieve compression ratios of up to 1%, this did not necessarily improve run-time for reads and certainly not for writes. However, we do admit that there is much further to take this subject, and pointed out a few directions in the previous section.