Optimizing Space Amplification in RocksDB

Siying Dong[1], Mark Callaghan[1], Leonidas Galanis[1],
Dhruba Borthakur[1], Tony Savor[1] and Michael Stumm[2]

[1]Facebook, 1 Hacker Way, Menlo Park, CA USA 94025
{siying.d, mcallaghan, lgalanis, dhruba, tsavor}@fb.com

[2]Dept. Electrical and Computer Engineering, University of Toronto, Canada M8X 2A6
stumm@eecg.toronto.edu

ABSTRACT

RocksDB is an embedded, high-performance, persistent key-value storage engine developed at Facebook. Much of our current focus in developing and configuring RocksDB is to give priority to resource efficiency instead of giving priority to the more standard performance metrics, such as response time latency and throughput, as long as the latter remain acceptable. In particular, we optimize space efficiency while ensuring read and write latencies meet service-level requirements for the intended workloads. This choice is motivated by the fact that storage space is most often the primary bottleneck when using Flash SSDs under typical production workloads at Facebook. RocksDB uses *log-structured merge trees* to obtain significant space efficiency and better write throughput while achieving acceptable read performance.

This paper describes methods we used to reduce storage usage in RocksDB. We discuss how we are able to trade off storage efficiency and CPU overhead, as well as read and write amplification. Based on experimental evaluations of MySQL with RocksDB as the embedded storage engine (using TPC-C and LinkBench benchmarks) and based on measurements taken from production databases, we show that RocksDB uses less than half the storage that InnoDB uses, yet performs well and in many cases even better than the B-tree-based InnoDB storage engine. To the best of our knowledge, this is the first time a Log-structured merge tree-based storage engine has shown competitive performance when running OLTP workloads at large scale.

1. INTRODUCTION

Resource efficiency is the primary objective in our storage systems strategy at Facebook. Performance must be sufficient to meet the needs of Facebook's services, but efficiency should be as good as possible to allow for scale.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

Facebook has one of the largest MySQL installations in the world, storing many 10s of petabytes of online data. The underlying storage engine for Facebook's MySQL instances is increasingly being switched over from InnoDB to MyRocks, which in turn is based on Facebook's RocksDB. The switchover is primarily motivated by the fact that MyRocks uses half the storage InnoDB needs, and has higher average transaction throughput, yet has only marginally worse read latencies.

RocksDB is an embedded, high-performance, persistent key-value storage system [1] that was developed by Facebook after forking the code from Google's LevelDB [2, 3].[1] RocksDB was open-sourced in 2013 [5]. *MyRocks* is RocksDB integrated as a MySQL storage engine. With MyRocks, we can use RocksDB as backend storage and still benefit from all the features of MySQL.

RocksDB is used in many applications beyond just MySQL, both within and outside of Facebook. Within Facebook, RocksDB is used as a storage engine for Laser, a high query throughput, low latency key-value storage service [6], ZippyDB, a distributed key-value store with Paxos-style replication [6], Dragon, a system to store indices of the Social Graph [7], and Stylus, a stream processing engine [6], to name a few. Outside of Facebook, both MongoDB [8] and Sherpa, Yahoo's largest distributed data store [9], use RocksDB as one of their storage engines. Further, RocksDB is used by LinkedIn for storing user activity [10] and by Netflix to cache application data [11], to list a few examples.

Our primary goal with RocksDB at Facebook is to make the most efficient use of hardware resources while ensuring all important service level requirements can be met, including target transaction latencies. Our focus on efficiency instead of performance is arguably unique in the database community in that database systems are typically compared using performance metrics such as transactions per minute (e.g., tpmC) or response-time latencies. Our focus on efficiency does not imply that we treat performance as unimportant, but rather that once our performance objectives are achieved, we optimize for efficiency. Our approach is driven in part by the data storage needs at Facebook (that may well differ from that of other organizations):

1. SSDs are increasingly being used to store persistent data and are the primary target for RocksDB;
2. Facebook relies primarily on shared nothing configura-

[1]A Facebook blog post lists many of the key differences between RocksDB and LevelDB [4].

tions of commodity hardware in their data centers [12], where data is distributed across a large number of simple nodes, each with 1-2 SSDs;

3. the amount of data that needs to be stored is huge;

4. the read-write ratio is relatively low at roughly 2:1 in many (but not all) cases, given the fact that large memory-based caches are used extensively.

Minimizing space amplification is important to efficient hardware use because storage space is the bottleneck in environments like the one described above. In a typical production MySQL environment at Facebook, SSDs process far fewer reads/s and writes/s during peak times under InnoDB than what the hardware is capable of. The throughput level under InnoDB is low, not because of any bottleneck on the SSD or the processing node — e.g., CPU utilization remains below 40% — but because the query rate per node is low. The per node query rate is low, because the amount of data that has to be stored (and be accessible) is so large, it has to be sharded across many nodes to fit, and the more nodes, the fewer queries per node.

If the SSD could store twice as much data, then we would expect storage node efficiency to double, since the SSDs could easily handle the expected doubling of IOPS, and we would need far fewer nodes for the workload. This issue drives our focus on space amplification. Moreover, minimizing space amplification makes SSDs an increasingly attractive alternative compared to spinning disks for colder data storage, as SSD prices continue to decline. In our pursuit to minimize space amplification, we are willing to trade off some extra read or write amplification. Such a tradeoff is necessary because it is not possible to simultaneously reduce space, read, and write amplification [13].

RocksDB is based on Log-Structured Merge-Trees (LSM-trees). The LSM-tree was originally designed to minimize random writes to storage as it never modifies data in place, but only appends data to files located in stable storage to exploit the high sequential write speeds of hard drives [14]. As technology changed, LSM-trees became attractive because of their low write amplification and low space amplification characteristics.

In this paper, we describe our techniques for reducing space amplification within RocksDB. We believe some of the techniques are being described for the first time, including dynamic LSM-tree level size adjustment, tiered compression, shared compression dictionary, prefix bloom filters, and different size multipliers at different LSM-tree levels. We discuss how the space amplification techniques affect read and write amplification, and we describe some of the tradeoffs involved. We show by way of empirical measurements that RocksDB requires roughly 50% less storage space than InnoDB, on average; it also has a higher transaction throughput, and yet it increases read latencies only marginally, remaining well within the margins of acceptability. We also discuss tradeoffs between space amplification and CPU overhead, since the CPU may become a bottleneck once space amplification is significantly reduced.

We demonstrate, based on experimental data, that a storage engine based on an LSM-tree can be performance competitive when used on OLTP workloads. We believe this is the first time this is shown. With MyRocks, each MySQL table row is stored as a RocksDB key-value pair: the primary keys are encoded in the RocksDB key and all other row data is encoded in the value. Secondary keys, which are not necessarily unique, are stored as separate key-value pairs, where the RocksDB key encodes the secondary key appended with the corresponding target primary key, and the value is left empty; thus secondary index lookups are translated into RocksDB range queries. All RocksDB keys are prefixed by a 4-byte table-ID or index-ID so that multiple tables or indexes can co-exist in one RocksDB key space. Finally, a global sequence ID, incremented with each write operation, is stored with each key-value pair to support snapshots. Snapshots are used to implement multiversion concurrency control, which in turn enables us to implement ACID transactions within RocksDB.

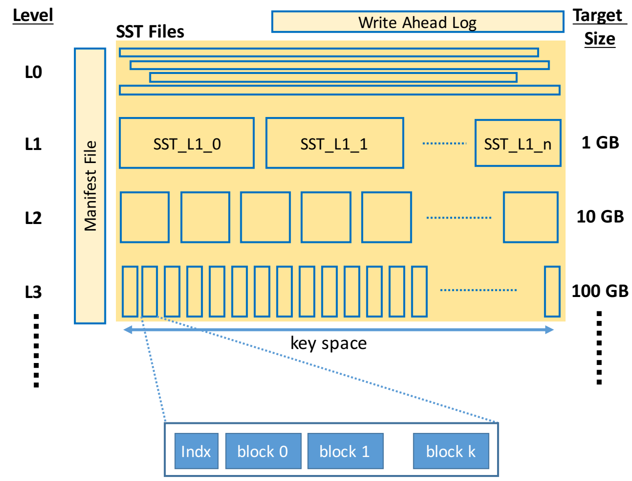

Figure 1: SST file organization. Each level maintains a set of SST files. Each SST file consists of unaligned 16KB blocks with an index block identifying the other blocks within the SST. Level 0 is treated differently in that its SST files have overlapping key ranges, while the SST files at the other level have non-overlapping key ranges. A *manifest file* maintains a list of all SST files and their key ranges to assist lookups.

In the next section, we provide a brief background on LSM-trees. In Section 3 we describe our techniques to reduce space amplification. In Section 4 we show how we balance space amplification with read amplification and CPU overhead. Finally, in Section 5 we present the results of experimental evaluations using realistic production workloads (TPC-C and LinkBench) and measurements taken from production instances of the database. We close with concluding remarks.

2. LSM-TREE BACKGROUND

Log Structured Merge Trees (LSM-trees) are used in many popular systems today, including BigTable [15], LevelDB, Apache Cassandra [16], and HBase [17]. Significant recent research has focused on LSM-trees; e.g., [18, 19, 20, 21, 22]. Here we briefly describe the LSM-tree as implemented and configured in MyRocks at Facebook by default.

Whenever data is written to the LSM-tree, it is added to an in-memory write buffer called *mem-table*, implemented as a skiplist having O(log n) inserts and searches. At the same time, the data is appended to a Write Ahead Log (WAL) for recovery purposes. After a write, if the size of the mem-table reaches a predetermined size, then (i) the current WAL and mem-table become immutable, and a new WAL and mem-

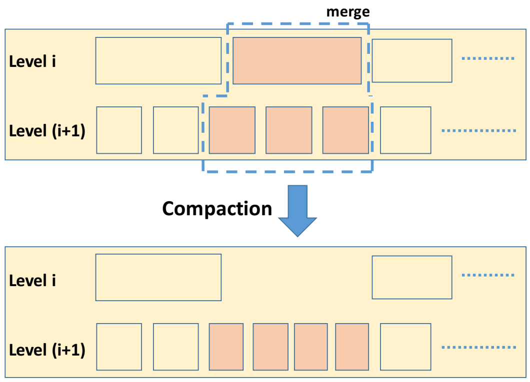

Figure 2: Compaction. The contents of a selected level-i SST file is merged with those SST files at level i+1 that have key ranges overlapping with the key range of the level-i SST. The shaded SST files in the top part of the figure are deleted after the merge process. The shaded SST files in the bottom part of the figure are new files created from the compaction process. The compaction process removes data that has become obsolete; i.e., data that has been marked as deleted and data that has been overwritten (if they are no longer needed for snapshots).

table are allocated for capturing subsequent writes, (*ii*) the contents of the mem-table are flushed out to a "*Sorted Sequence Table*" (SST) data file, and upon completion, (*iii*) the WAL and mem-table containing the data just flushed are discarded. This general approach has a number of favorable consequences: new writes can be processed concurrently to the flushing of an older mem-table; all I/O is sequential,[2] and, except for the WAL, only entire files are ever written.

Each of the SSTs stores data in sorted order, divided into unaligned 16KB blocks (when uncompressed). Each SST also has an index block for binary search with one key per SST block. SSTs are organized into a sequence of levels of increasing size, Level-0 – Level-N, where each level will have multiple SSTs, as depicted in Figure 1. Level-0 is treated specially in that its SSTs may have overlapping key ranges, while the SSTs of higher levels have distinct non-overlapping key ranges. When the number of files in Level-0 exceeds a threshold (e.g., 4), then the Level-0 SSTs are merged with the Level-1 SSTs that have overlapping key ranges; when completed, all of the merge sort input (L0 and L1) files are deleted and replaced by new (merged) L1 files. For L>0, when the combined size of all SSTs in level-L exceeds a threshold (e.g., $10^{(L-1)}$GB) then one or more level-L SSTs are selected and merged with the overlapping SSTs in level-(L+1), after which the merged level-L and level-(L+1) SSTs are removed. This is shown in Figure 2.

The merging process is called *compaction* because it removes data marked as deleted and data that has been overwritten (if it is no longer needed). This is implemented using multiple threads. Compaction also has the effect of gradually migrating new updates from Level-0 to the last level, which is why this particular approach is referred to as "leveled" compaction.[3] The process ensures that at any given time, each SST will contain at most one entry for any given key and snapshot. The I/O that occurs during compaction is efficient as it only involves bulk reads and writes of entire files: if a level-L file being compacted overlaps with only part of a level-(L+1) file, then nevertheless the entire level-(L+1) file is used as an input to the compaction and ultimately removed. A compaction may trigger a set of cascading compactions.

A single *Manifest File* maintains a list of SSTs at each level, their corresponding key ranges, and some other meta data. It is maintained as a log to which changes to the SST information are appended. The information in the manifest file is cached in an efficient format in memory to enable quick identification of SSTs that may contain a target key.

The search for a key occurs at each successive level until the key is found or it is determined that the key is not present in the last level. It begins by searching all memtables, followed by all Level-0 SSTs and then the SST's at the next following levels. At each of these successive levels, three binary searches are necessary. The first search locates the target SST by using the data in the Manifest File. The second search locates the target data block within the SST file by using the SST's index block. The final search looks for the key within the data block. Bloom filters (kept in files but cached in memory) are used to eliminate unnecessary SST searches, so that in the common case only 1 data block needs to be read from disk. Moreover, recently read SST blocks are cached in a block cache maintained by RocksDB and the operating system's page cache, so access to recently fetched data need not result in I/O operations. The MyRocks block cache is typically configured to be 12GB large.

Range queries are more involved and always require a search through all levels since all keys that fall within the range must be located. First the mem-table is searched for keys within the range, then all Level-0 SSTs, followed by all subsequent levels, while disregarding duplicate keys within the range from lower levels. Prefix Bloom filters (§4) can reduce the number of SSTs that need to be searched.

To get a better feel for the systems characteristics of LSM-trees, we present various statistics gathered from three production servers in the Appendix.

3. SPACE AMPLIFICATION

An LSM-tree is typically far more space efficient than a B-tree. Under read/write workloads similar to those at Facebook, B-tree space utilization will be poor [24] with its pages only 1/2 to 2/3 full (as measured in Facebook production databases). This fragmentation causes space amplification to be worse than 1.5 in B-tree-based storage engines. Compressed InnoDB has fixed page sizes on disk which further wastes space.

In contrast, LSM-trees do not suffer from fragmentation because it does not require data to be written to SSD page-aligned. LSM-tree space amplification is mostly determined by how much stale data is yet to be garbage-collected. If we assume that the last level is filled to its target size with data and that each level is 10X larger than the previous level, then in the worst case, LSM-tree space amplification will be 1.111..., considering that all of the levels up to the last level combined are only 11.111...% the size of the last level.

[2] There are usually concurrent streams of sequential IO that will cause seeks. However, the seeks will be amortized over LSM-tree's very large writes (many MB rather than KB).

[3] Leveled compaction is different that the compaction methods used, say, by HBase and Cassandra [23]. In this paper, all uses of the term *compaction* refer to leveled compaction.

RocksDB uses two strategies to reduce space amplification: (i) adapting the level sizes to the size of the data, and (ii) applying a number of compression strategies.

3.1 Dynamic level size adaptation

If a fixed size is specified for each level, then in practice it is unlikely that the size of the data stored at the last level will be 10X the target size of the previous level. In a worse case, the size of the data stored at the last level will only be slightly larger than the target size of the previous level, in which case space amplification would be larger than 2.

However, if we dynamically adjust the size of each level to be 1/10-th the size of the data on the next level, then space amplification will be reduced to less than 1.111....

The level size multiplier is a tunable parameter within an LSM-tree. Above, we assumed it is 10. The larger the size multiplier is, the lower the space amplification and the read amplification, but the higher the write amplification. Hence, the choice represents a tradeoff. For most of the Facebook production RocksDB installations, a size multiplier of 10 is used, although there are a few instances that use 8.

An interesting question is whether the size multiplier at each level should be the same. The original paper on LSM-trees proved that it is optimal to have the same multiplier at each level when optimizing for write amplification [14].[4] It is an open question of whether this also holds true when optimizing for space amplification, especially when considering that different levels may use different compression algorithms resulting in different compression ratios at each level (as described in the next section). We intend to analyze this question in future work.[5]

3.2 Compression

Space amplification can be further reduced by compressing the SST files. We apply a number of strategies simultaneously. LSM-trees provide a number of properties that make compression strategies more efficacious. In particular, SSTs and their data blocks in LSM-trees are immutable.

Key prefix encoding. Prefix encoding is applied on keys by not writing repeated prefixes of previous keys. We have found this reduces space requirements by 3% – 17% in practice, depending on the data workload.

Sequence ID garbage collection. The sequence ID of a key is removed if it is older than the oldest snapshot needed for multiversion concurrency control. Users can arbitrarily create snapshots to refer to the current database state at a later point in time. Removing snapshot IDs tends to be effective because the 7 byte large sequence ID does not compress well, and because most of the sequence IDs would no longer be needed after the corresponding snapshots that refer to them have been deleted. In practice, this optimization reduces space requirements from between 0.03% (e.g., for a database storing social graph vertexes that will have large values) and 23% (e.g., for a database storing social graph edges that will have empty values) .

Data compression. RocksDB currently supports several compression algorithms, including LZ, Snappy, zlib, and Zstandard. Each level can be configured to use any or none of these compression algorithms. Compression is applied on a per-block basis. Depending on the composition of the data, weaker compression algorithms can reduce space requirements down to as low as 40%, and stronger algorithms down to as low as 25%, of their original sizes on production Facebook data.

To reduce the frequency of having to uncompress data blocks, the RocksDB block cache stores blocks in uncompressed form. (Note that recently accessed compressed file blocks will be cached by the operating system page cache in compressed form, so compressed SSTs will use less storage space and less cache space, which in turn allows the file system cache to cache more data.)

Dictionary-Based Compression. A data dictionary can be used to further improve compression. Data dictionaries can be particularly important when small data blocks are used, as smaller blocks typically yield lower compression ratios. The dictionary makes it possible for smaller blocks to benefit from more context. Experimentally, we have found that a data dictionary can reduce space requirements by an additional 3%.

LSM-trees make it easier to build and maintain dictionaries. They tend to generate large immutable SST files that can be hundreds of megabytes large. A dictionary that is applied to all data blocks can be stored within the file so when the file is deleted, the dictionary is dropped automatically.

4. TRADEOFFS

LSM-trees have many configuration parameters and options, enabling a number of tradeoffs for each installation given the particulars of a target workload. Prior work by Athanassoulis et al. established that one can optimize for any two of space, read, and write amplification, but at the cost of the third [13]. For example, decreasing the number of levels (say by increasing the level multiplier) decreases space and read amplification, but increases write amplification.

As another example, in LSM-trees, a larger block size leads to improved compression without degrading write amplification, but negatively affects read amplification (since more data must be read per query). This observation allows us to use a larger block size for better compression ratios when dealing with write heavy applications. (In B-Trees, larger blocks degrade both write and read amplification.)

Tradeoffs in many cases involve judgement calls and depend on the expected workload and perceived minimal acceptable quality of service levels. When focusing on efficiency (as we do), it is exceedingly difficult to configure the system to properly balance CPU, disk I/O, and memory utilization, especially because it is strongly dependent on a highly varying workload.

As we show in the next section, our techniques reduce storage space requirements by 50% over InnoDB. This allows us to store twice as much data on each node, which in turn enables significant consolidation of existing hardware. At the same time, however, this also means that we double the workload (QPS) per server, which could cause the system to reach the limits of available CPU, random I/O, and RAM capacity.

[4]The original LSM-tree paper uses a fixed number of levels and varies the multiplier as the database gets larger, but keeping the multiplier the same at each level. LevelDB/RocksDB use a fixed multiplier but varies the number of levels as the database gets larger.

[5]Initial results indicate that adapting the size targets at each level to take into account the compression ratios achieved at each level lead to better results.

Tiered compression. Compression generally decreases the amount of storage space required, but increases CPU overheads, since data has to be compressed and decompressed. The stronger the compression, the higher the CPU overhead. In our installations, a strong compression algorithm (like zlib or Zstandard) is typically used at the last level even though it incurs higher CPU overhead, because most (close to 90%) of the data is located at that level, yet only a small fraction of reads and writes go to it. In various use cases, applying strong compression to the last level saves an additional 15%–30% in storage space over using lightweight compression only.

Conversely, we do not use any compression at levels 0–2 to allow for lower read latencies at the cost of higher space- and write-amplification, because they use up only a small proportion of the total storage space. Level-3 up to the last level use lightweight compression (like LZ4 or Snappy) because its CPU overhead is acceptable, yet it reduces space and write amplification. Reads to data located in the first three levels will more likely be located in (uncompressed) file blocks cached by the operating system because these blocks are frequently accessed. However, reads to data located in levels higher than 2 will have to be uncompressed, whether they are located in the operating system file cache or not (unless they are also located in the RocksDB block cache).

Bloom filters. Bloom filters are effective in reducing I/O operations and attendant CPU overheads, but at the cost of somewhat increased memory usage since the filter (typically) requires 10 bits per key. As an illustration that some tradeoffs are subtle, we do not use a Bloom filter at the last level. While this will result in more frequent accesses to last-level files, the probability of a read query reaching the last level is relatively small. More importantly, the last-level bloom filter is large (~9X as large as all lower-level Bloom filters combined) and the space it would consume in the memory-based caches would prevent the caching of other data that would be being accessed. We determined empirically that not having a Bloom filter for the last level improved read amplification overall, given our workloads.

Prefix Bloom filters. Bloom filters do not help with range queries. We have developed a prefix Bloom filter that helps with range queries, based on the observation that many range queries are often over a prefix; e.g., the userid part of a (userid,timestamp) key or postid of a (postid,likerid) key. We allow users to define prefix extractors to deterministically extract a prefix part of the key from which we construct a Bloom filter. When querying a range, the user can specify that the query is on a defined prefix. We have found this optimization reduces read amplification (and attendant CPU overheads) on range queries by up to 64% on our systems.

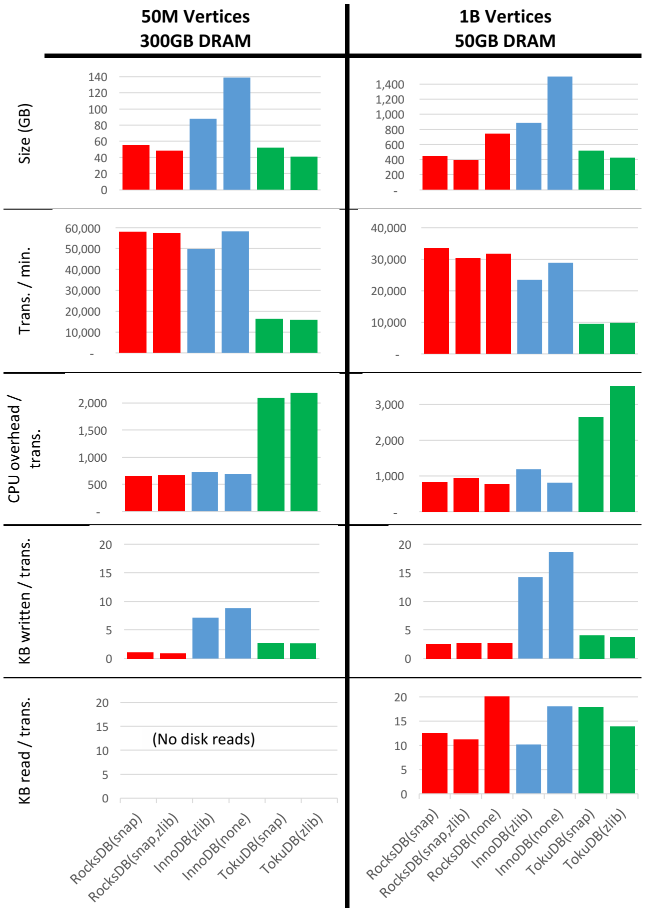

Figure 3: LinkBench benchmark. Statistics gathered from the 24th hour of 24 hour runs with 16 concurrent clients for 3 different storage engines: RocksDB (from Facebook MySQL 5.6) shown in red, InnoDB (from MySQL 5.7.10) shown in blue, and TokuDB (Percona Server 5.6.26-74.0) shown in green, configured to use the compression scheme(s) listed in brackets. (Sync-on-commit was disabled, binlog/oplog and redo logs were enabled.)

System setup: The hardware consisted of an Intel Xeon E5-2678v3 CPU with 24-cores/48-HW-threads running at 2.50GHz, 256GB of RAM, and roughly 5T of fast NVMe SSD provided via 3 devices configured as SW RAID 0. The operating system was Linux 4.0.9-30

Left hand side graphs: Statistics from LinkBench configured to store 50M vertices, which fits entirely in DRAM.

Right hand side graphs: Statistics from LinkBench configured to store 1B vertices, which does not fit in memory after constraining DRAM memory to 50GB: all but 50GB of RAM was mlocked by a background process so the database software, OS page cache and other monitoring processes had to share the 50GB. The MyRocks block cache was set to 10GB.

5. EVALUATION

A review of numerous MySQL installations at Facebook generally reveal that

1. the storage space used by RocksDB is about 50% lower than the space used by InnoDB with compression,

2. the amount of data written to storage by RocksDB is between 10% and 15% of what InnoDB writes out, and

3. the number and volume of reads is 10% – 20% higher in RocksDB that for InnoDB (yet well within the margin of acceptability).

For more meaningful metrics from controlled environments, we present the results of extensive experiments using two benchmarks with MySQL. The first benchmark, LinkBench, is based on traces from production databases that store "social graph" data at Facebook; it issues a considerable number of range queries [25]. We ran 24 1 hour intervals of

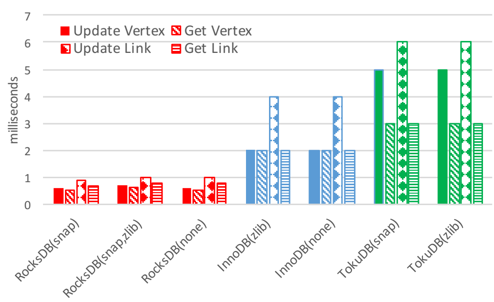

Figure 4: LinkBench Quality of Service: 99th percentile latencies for: Update Vertex, Get Vertex, Update Link, Get Link. The setup of the hardware and storage engines are as described in Figure 3. The database with 1B vertices was used with available DRAM constrained to 50GB.

LinkBench and measured statistics for the 24th interval to obtain numbers from a steady-state system.[6] The second benchmark is the standard TPC-C benchmark.

For both benchmarks, we experimented with two variants: one where the database fit in DRAM so that disk activity was needed only for writes to achieve durability, and one where the database did not fit in memory. We compare the behavior of RocksDB, InnoDB, and TokuDB, configured to use a variety of compression strategies. (TokuDB is another open source, high-performance storage engine for MySQL which at its core uses a fractal tree index tree data structure to reduce space and write amplification [26].)

Figure 3 shows the results from our LinkBench experiments. The hardware and software setup used is described in the figure caption. Some observations for the LinkBench benchmark with a database that does not fit in memory:

- **Space usage:** RocksDB with compression uses less storage space than any of the alternatives considered; without compression, it uses less than half as much storage space as InnoDB without compression.
- **Transaction throughput:** RocksDB exhibits higher throughput than all the alternatives considered: 3%-16% better than InnoDB, and far better than TokuDB. What is not visible in the graph is that in all cases, CPU is the bottleneck preventing throughput to further increase.
- **CPU overhead:** When stronger compression is used, RocksDB exhibits less than 20% higher CPU overhead per transaction compared to InnoDB with no compression, but less than 30% as much CPU overhead as TokuDB. RocksDB with strong compression incurs only 80% as much CPU overhead as InnoDB with compression.
- **Write Volume:** The volume of data written per transaction in RocksDB is less than 20% of the volume of data written by InnoDB.[7] RocksDB write volume is significantly lower than TokuDB write volume.

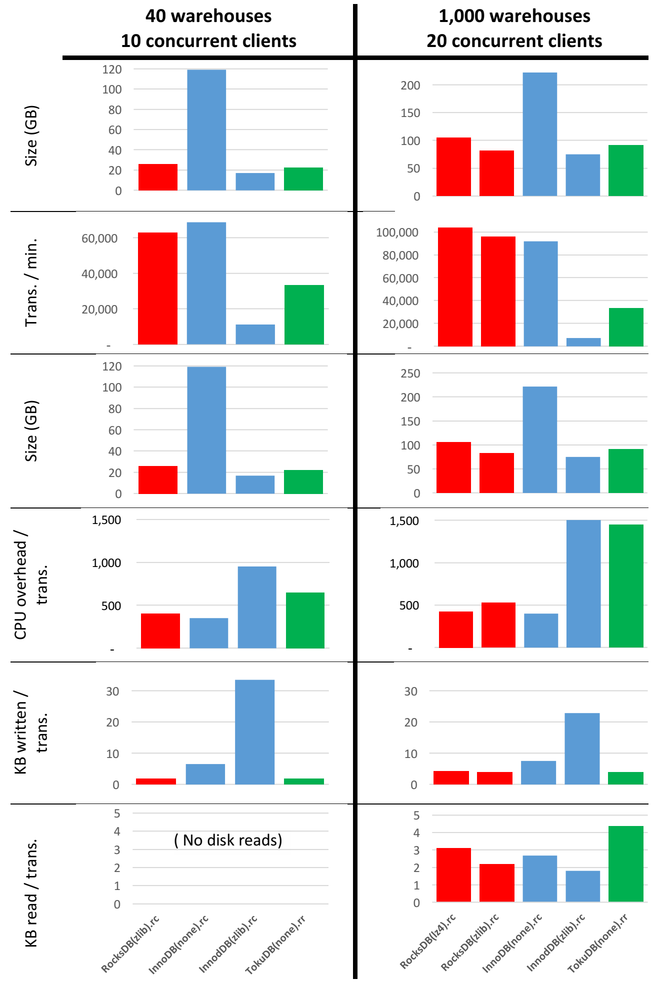

Figure 5: TPC-C Benchmarks. Metrics obtained using the same setup as in Figure 3.
Left hand side: configuration of 40 warehouses and 10 concurrent clients. The database fits entirely in memory. The statistics were gathered over the entire 15th hour after 14 hours of operation.
Right hand side: configuration of 1,000 warehouses and 20 concurrent clients. The statistics were gathered over the entire 12th hour after 11 hours of operation. The transaction isolation levels used are marked as "rc" for READ COMMITTED or "rr" for REPEATABLE READ.

- **Read Volume:** The volume of data read per read transaction in RocksDB is 20% higher than InnoDB when no compression is used, and between 10% and 22% higher when compression is used. RocksDB read volume is significantly less than TokuDB read volume.

Figure 4 depicts the quality of service achieved by the different storage engines. Specifically, it shows the 99-th percentile latencies for read and write requests on both vertices and edges in the LinkBench database. The behavior of RocksDB is an order of magnitude better than the behavior of all the other alternatives considered.

[6] We also gathered statistics when loading the full LinkBench database; the results (not shown) are in line with the steady-state numbers.

[7] The I/O volume numbers were obtained from *iostat*. The write volume numbers had to be adjusted because *iostat* counts TRIM as bytes written when in fact none are. RocksDB frequently deletes entire files (in contrast to InnoDB) and uses TRIM for that, which iostat reports as if the entire file had been written.

The results of the TPC-C benchmark are shown in Figure 5. The database size statistics are more difficult to interpret here because the TPC-C database grows with the number of transactions. For example, InnoDB with compression is shown to require a small storage footprint, but this is only the case because this InnoDB variant was able to process far fewer transactions up to the point the measurements were taken; in fact, InnoDB database size grows much faster in transaction time than RocksDB.

The figure clearly shows that RocksDB is not only competitive on OLTP workloads, but generally has higher transaction throughput while requiring significantly less storage space than the alternatives. RocksDB writes out less data per transaction than all the other configurations tested, yet reads only marginally more and requires only marginally more CPU overhead per transaction.

6. CONCLUDING REMARKS

We described how RocksDB was able to reduce storage space requirements by 50% over what InnoDB would need, while at the same time increasing transaction throughput and significantly decreasing write amplification, yet increasing average read latencies by a marginal amount. It did so by leveraging LSM-trees and applying a variety of techniques to conserve space.

A number of these techniques were, as far as we are aware, described for the first time, including: (*i*) dynamic LSM-tree level size adjustment based on current DB size; (*ii*) tiered compression where different levels of compression are used at different LSM-tree levels; (*iii*) use of a shared compression dictionary; (*iv*) application of Bloom filters to key prefixes; and (*v*) use of different size multipliers at different LSM-tree levels. Moreover, we believe this is the first time a storage engine based on an LSM-tree has been shown to have competitive performance on traditional OLTP workloads.

7. REFERENCES

[1] http://rocksdb.org.

[2] http://leveldb.org.

[3] https://github.com/google/leveldb.

[4] https://github.com/facebook/rocksdb/wiki/Features-Not-in-LevelDB.

[5] https://github.com/facebook/rocksdb.

[6] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at Facebook," in *Proc. Intl. Conf. on Management of Data*, 2016, pp. 1087–1098.

[7] A. Sharma, "Blog post: Dragon: a distributed graph query engine," https://code.facebook.com/posts/1737605303120405/dragon-a-distributed-graph-query-engine/, 2016.

[8] https://www.mongodb.com.

[9] https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights.

[10] https://www.youtube.com/watch?v=plqVp_OnSzg.

[11] http://techblog.netflix.com/2016/05/application-data-caching-using-ssds.html.

[12] http://opencompute.org.

[13] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan, "Designing access methods: the RUM conjecture," in *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, 2016.

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "BigTable: A distributed storage system for structured data," *ACM Trans. on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[16] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[17] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya, "Storage infrastructure behind Facebook messages: Using HBase at scale." *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 4–13, 2012.

[18] P. A. Bernstein, S. Das, B. Ding, and M. Pilman, "Optimizing optimistic concurrency control for tree-structured, log-structured databases," in *Proc. 2015 ACM SIGMOD Intl. Conf. on Management of Data*, 2015, pp. 1295–1309.

[19] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *14th USENIX Conf. on File and Storage Technologies (FAST 16)*, Feb. 2016, pp. 149–166.

[20] R. Sears and R. Ramakrishnan, "bLSM: a general purpose log structured merge tree," in *Proc. 2012 ACM SIGMOD Intl. Conf. on Management of Data*, 2012, pp. 217–228.

[21] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: a scalable log-structured database system in the cloud," *Proc. of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, 2012.

[22] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th European Conf. on Computer Systems*. ACM, 2014, p. 16.

[23] T. Hobbs, "Blog post: When to use leveled compaction," http://www.datastax.com/dev/blog/when-to-use-leveled-compaction, june 2012.

[24] A. C.-C. Yao, "On random 2—3 trees," *Acta Inf.*, vol. 9, no. 2, pp. 159–170, Jun. 1978.

[25] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the Facebook Social Graph," in *Proc. 2013 ACM SIGMOD Intl. Conf. on Management of Data*, 2013, pp. 1185–1196.

[26] I. Tokutek, "TokuDB: MySQL performance, MariaDB performance," http://www.tokutek.com/products/tokudb-for-mysql/, 2013.

APPENDIX

In this appendix, we present various statistics from three LSM tree-based systems currently in production. The intent is to provide the reader with a better feel for how LSM tree-based systems behave in practice.

The statistics we present were gathered from representative servers in production running MySQL/MyRocks servers in production serving Social Network queries. The workload is update-heavy. The data presented here was collected over an observation period of over one month.

Figure 6 depicts the number of files at each level at the end of the observation period. Figure 7 depicts the aggregate size of the files at each level at the end of the observation period. In both figures, the relative differences between the numbers of different levels provide more insight than the absolute numbers. The figures shows that the number of files and the aggregate size of the files grow by roughly a factor of 10 at each level, which is what one would expect given the description in Section 2.

Figure 8 shows the number of compactions that occurred at each level during the observation period. Again, the absolute numbers are not very meaningful, especially since they are highly affected by configuration parameters. The first set of bars on the left represent instances of copying a mem-table to an L0 file. The second set of bars represent the merging of *all* L0 files into L1; hence each such compaction involves far more data than, say, a compaction merging an L1 file into L2.

Figure 9 depicts the amount of data written to disk (in GB) for each level during the observation period, broken down into writes for new data, writes for data being updated, and writes for data being copied from an existing SST at the same level. Writes to Level 0 are almost entirely new writes, represented by the red bars. The data needs to be interpreted with care: a new data write at level Li only implies that the key of the kv-pair was not present at that level when the data was written but the key may well be present at level Lj with $j > i$.

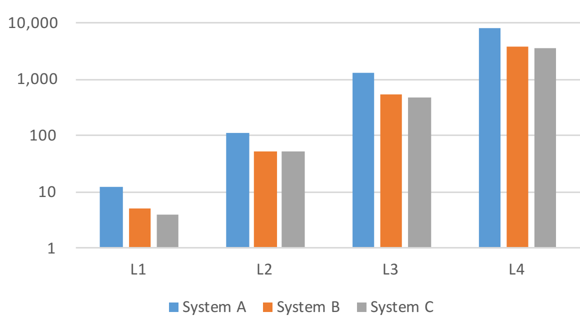

Figure 6: Number of files at each level. Note the logarithmic scale of the y-axis. Level 0 is not included as the number of files for that level oscillates between 0 and 4 and the reported value largely depends on when the snapshot is taken.

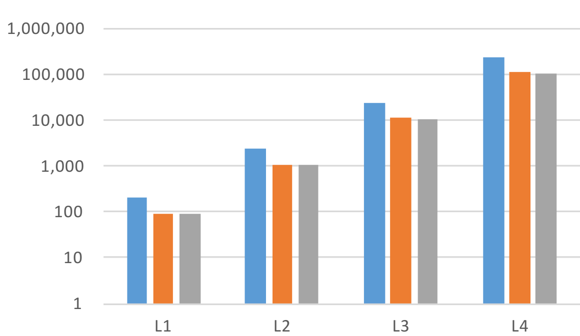

Figure 7: Aggregate size of all files for each level in Megabytes. Note the logarithmic scale of the y-axis. Level 0 is not included for the same reason described as in Fig. 6.

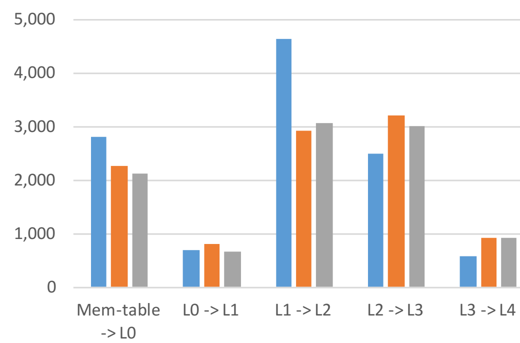

Figure 8: Number of compactions at each level for the three systems during the observation period. Note that the y-axis is *not* in logarithmic scale.

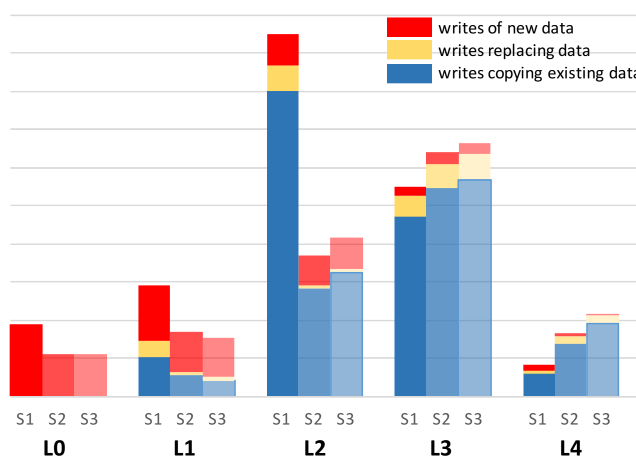

Figure 9: Disk writes at each level for the three systems. The height of each bar represent the total number of bytes written to disk at each level during the observation period. This is further broken down into (i) in red: writes of new data, i.e., with a key that does not currently exist at that level, (ii) in yellow: writes of data being updated. i.e., with a key that already exists at that level, and (iii) in blue: writes for data being copied from an existing SST at that same level. The y-axis has been hidden as the absolute numbers are not informative. The y-axis is *not* scaled logarithmically.

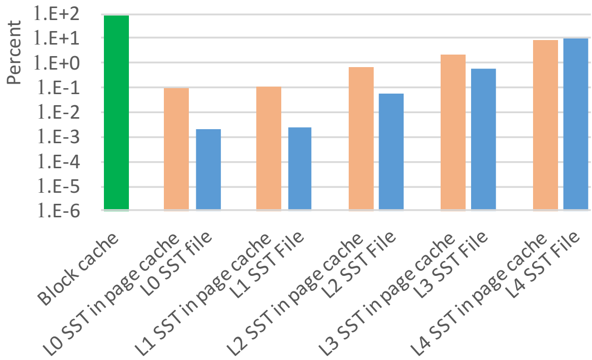

Figure 10: Location of target read query data. Each bar shows the percentage of read queries served from each possible source. Note that the graph has a logarithmic y-axis. The read queries represented in this graph are those aggregated over the three systems being monitored. 79.3% of all read queries are served from the RocksDB block cache. Read queries not served from the block cache are served from L0, L1, L2, L3, or L4 SST files. Some of those are served from the operating system page cache, which therefore do not require any disk accesses. Less than 10% of read queries result in a disk access.

Figures 10 and 11 depict the location from where read query data is served. For this figure, we aggregated the number of reads from all three systems. A majority of read requests are successfully served by the RocksDB block cache: for data blocks, the hit rate is 79.3%, and for meta data blocks containing indices and Bloom filters, the hit rate is 99.97%. Misses in the RocksDB cache result in file system reads. The figure shows that the majority (52%) of file system reads are serviced from the operating system page cache, with the page cache hit rate being 98%, 98%, 93%, 77%, and 46% for levels L0-L4, respectively. However, given the fact that almost 90% of the disk storage space is used to hold data from the last level, L4, and given that L4 data has a poorer RocksDB block cache hit rate, 92% of all read queries that miss in the block cache are served by L4. Since most file accesses are to L4 SSTs, it is clear that Bloom filters are helpful in reducing the number of file accesses at levels L0–L3 for these read queries.

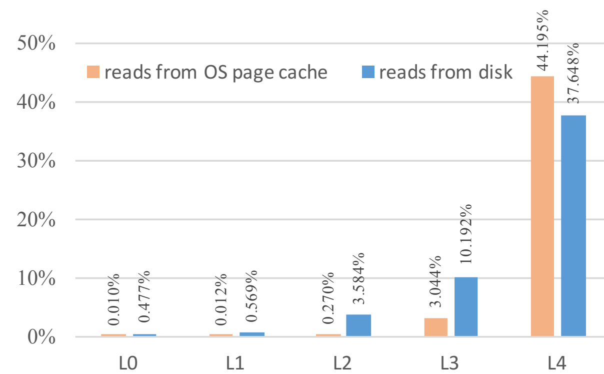

Figure 11: Location of target read query data when not in the RocksDB block cache. The graph depicts the same data as in Figure 10 but for read accesses that miss in the RocksDB block cache. Note that the graph does *not* have a logarithmic y-axis. The graph shows that the overwhelming majority (82%) of read queries that miss in the RocksDB block cache are served from L4, the last level. For those read queries, the data is located in the OS file page cache in 46% of the cases.