MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph

Yoshinori Matsunobu
Facebook
yoshinori@fb.com

Siying Dong
Facebook
siying.d@fb.com

Herman Lee
Facebook
herman@fb.com

ABSTRACT

Facebook uses MySQL to manage tens of petabytes of data in its main database named the User Database (UDB). UDB serves social activities such as likes, comments, and shares. In the past, Facebook used InnoDB, a B+Tree based storage engine as the backend. The challenge was to find an index structure using less space and write amplification [1]. LSM-tree [2] has the potential to greatly improve these two bottlenecks. RocksDB, an LSM tree-based key/value store was already widely used in variety of applications but had a very low-level key-value interface. To overcome these limitations, MyRocks, a new MySQL storage engine, was built on top of RocksDB by adding relational capabilities. With MyRocks, using the RocksDB API, significant efficiency gains were achieved while still benefiting from all the MySQL features and tools. The transition was mostly transparent to client applications.

Facebook completed the UDB migration from InnoDB to MyRocks in 2017. Since then, ongoing improvements in production operations, and additional enhancements to MySQL, MyRocks, and RocksDB, provided even greater efficiency wins. MyRocks also reduced the instance size by 62.3% for UDB data sets and performed fewer I/O operations than InnoDB. Finally, MyRocks consumed less CPU time for serving the same production traffic workload. These gains enabled us to reduce the number of database servers in UDB to less than half, saving significant resources. In this paper, we describe our journey to build and run an OLTP LSM-tree SQL database at scale. We also discuss the features we implemented to keep pace with UDB workloads, what made migrations easier, and what operational and software development challenges we faced during the two years of running MyRocks in production.

Among the new features we introduced in RocksDB were transactional support, bulk loading, and prefix bloom filters, all are available for the benefit of all RocksDB users.

PVLDB Reference Format:
Yoshinori Matsunobu, Siying Dong, Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *PVLDB*, 13(12): 3217 - 3230, 2020.
DOI: https://doi.org/10.14778/3415478.3415546

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415546

1. INTRODUCTION

The Facebook UDB serves the most important social graph workloads [3]. The initial Facebook deployments used the InnoDB storage engine using MySQL as the backend. InnoDB was a robust, widely used database and it performed well. Meanwhile, hardware trends shifted from slow but affordable magnetic drives to fast but more expensive flash storage. Transitioning to flash storage in UDB shifted the bottleneck from Input/Output Operations Per Second (IOPS) to storage capacity. From a space perspective, InnoDB had three big challenges that were hard to overcome, index fragmentation, compression inefficiencies, and space overhead per row (13 bytes) for handling transactions. To further optimize space, as well as serving reads and writes with appropriate low latency, we believed an LSM-tree database optimized for flash storage was better in UDB. However, there were many different types of client applications accessing UDB. Rewriting client applications for a new database was going to take a long time, possibly multiple years, and we wanted to avoid that as well.

We decided to integrate RocksDB, a modern open source LSM-tree based key/value store library optimized for flash, into MySQL. As seen in Figure 1, by using the MySQL pluggable storage engine architecture, it was possible to replace the storage layer without changing the upper layers such as client protocols, SQL and Replication.

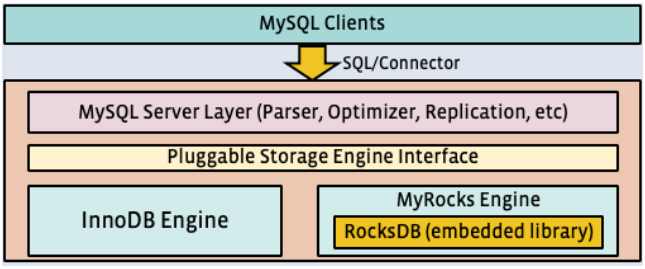

Figure 1: MySQL and MyRocks Storage Engine

We called this engine MyRocks. When we started the project, our goal was to reduce the number of UDB servers by 50%. That required the MyRocks space usage to be no more than 50% of the compressed InnoDB format, while maintaining comparable CPU and I/O utilization. We expected that achieving similar CPU utilization vs InnoDB was the hardest challenge, since flash I/O had sufficient read IOPS capacity and the LSM-tree database had less write amplification. Since InnoDB was a fast, reliable database with many features on which our Engineering team relied, there were many challenges ensuring there was no gap between InnoDB and MyRocks.

Among the significant challenges were: (1) Increased CPU, memory, and I/O pressure. MyRocks compresses the database size

by half which requires more CPU, memory, and I/O to handle the 2x number of instances on the host. (2) A larger gap between forward and backward range scans. The LSM-tree allows data blocks to be encoded in a more compacted form. As a result, forward scans are faster than backward scans. (3) Key comparisons. LSM-tree key comparisons are invoked more frequently than B-tree. (4) Query performance. MyRocks was slower than InnoDB in range query performance. (5) LSM-tree performance needs memory-based caching bloom filters for optimal performance. Caching bloom filters in memory is important to LSM-tree performance, but this consumes a non-trivial amount of DRAM and increases memory pressure. (6) Tombstone Management. With LSM-trees, deletes are processed by adding markers, which can sometimes cause performance problem with frequently updated/deleted rows. (7) Compactions, especially when triggered by burst writes, may cause stalls.

Section 3 provides the details for how those challenges were addressed. In short, the highlighted innovations implemented are the (1) prefix bloom filter so that range scans with equal predicates are faster (Section 3.2.2.1), the (2) mem comparable keys in MyRocks allowing more efficient character comparisons (Section 3.2.1.1), a (3) new tombstone/deletion type to more efficiently handle secondary index maintenance (Section 3.2.2.2), (4) bulk loading to skip compactions on data loading, (Section 3.2.3.4), (5) rate limited compaction file generations and deletions to prevent stalls (Section 3.2.3.2), and (6) hybrid compressions – using a faster compression algorithm for upper RocksDB levels, and a stronger algorithm for the bottommost level, so that MemTable flush and compaction can keep up with write ingestion rates with minimal space overhead (Section 3.3.4).

MyRocks also has native performance benefits over InnoDB such as not needing random reads for maintaining non-unique secondary indexes. More writes can be consolidated, with fewer total bytes written to flash. The read performance improvements and write performance benefits were evident when the UDB was migrated from InnoDB to MyRocks with no degradation of CPU utilization.

Comprehensive correctness, performance and reliability validations were needed prior to migration. We built two infrastructure services to help the migration. One was MyShadow, which captured production queries and replayed them to test instances. The other was a data correctness tool which compared full index data and query results between InnoDB and MyRocks instances. We ran these two tools to verify that MySQL instances running MyRocks did not return wrong results, did not return unexpected errors, did not regress CPU utilizations, and did not cause outstanding stalls. After completing the validations, the InnoDB to MyRocks migration itself was relatively easy. Since MySQL replication was independent of storage engine, adding MyRocks instances and removing InnoDB instances were simple. The bulk data loading feature in MyRocks greatly reduced data migration time as it could load indexes directly into the LSM-tree and bypass all MemTable writes and compactions.

The InnoDB to MyRocks UDB migrations were completed in August 2017. For the same data sets, MyRocks and modern LSM-tree structures and compression techniques reduced the instance size by 62.3% compared to compressed InnoDB. Lower secondary index maintenance overhead and overall read performance improvements resulted in slightly reduced CPU time. Bytes written to flush storage went down by 75%, which helped not to hit IOPS bottlenecks, and opened possibilities to use more affordable flash storage devices that had lower write cycles. These enabled us to reduce the number of database servers in UDB to less than half with MyRocks. Since 2017, regressions have been continuously tracked via MyShadow and data correctness. We improved compaction to guarantee the removal of stale data, meeting the increasing demands of data privacy.

This practice is valuable because: (1) Since SQL databases built on LSM-tree are gaining popularity, the practical techniques of tuning and improving LSM-tree are valuable. To the best of our knowledge, this is the first time these techniques have been implemented on a large-scale production system. (2) While some high-level B-tree vs LSM-tree comparisons are documented, our work exposed implementation challenges for LSM-tree to match B-tree performance, extra benefits from a LSM-tree, and optimizations that can narrow the gap. (3) Migrating data across different databases or storage engines is common. This paper shares the processes used to migrate the database to a different storage engine. The experience is more interesting because the storage engine moved to is relatively immature.

In this paper, we describe three contributions:

1. UDB overview, challenges with B-Tree indexes and why we thought LSM-tree database optimized for flash storage was suitable (Section 2).
2. How we optimized MyRocks for various read workloads and compactions (Section 3).
3. How we migrated to MyRocks in production (Section 4).

Then we show migration results in Section 5, followed by lessons learned in Section 6. Finally, we show related work in Section 7, and concluding remarks in Section 8.

2. BACKGROUND AND MOTIVATION

2.1 UDB Architecture

UDB is our massively sharded database service. We have customized MySQL with hundreds of features to operate the database for our needs. All customized extensions to MySQL are released as open source [4].

Facebook has many geographically distributed data centers across the world [5] and the UDB instances are running in some of them. Where other distributed database solutions place up to three copies in the same region and synchronously replicate among them, the Facebook ecosystem is so large that adopting this architecture for UDB is not practical as it would force us to maintain more than 10 database copies. We only maintain one database copy for each region. However, there are many applications which relied on short commit latency and did not function well with tens of millisecond for synchronous cross region transaction commits. These constraints led us to deploy a MySQL distributed systems architecture as shown in Figure 2.

We used traditional asynchronous MySQL replication for cross region MySQL replication. However, for in-region fault tolerance, we created a middleware called Binlog Server (Log Backup Unit) which can retrieve and serve the MySQL replication logs known as Binary Logs. Binlog Servers only retain a short period of recent transaction logs and do not maintain a full copy of the database. Each MySQL instance replicates its log to two Binlog Servers using MySQL Semi-Synchronous protocol. All three servers are spread across different failure domains within the region. This architecture

made it possible to achieve both short (in-region) commit latency and one database copy per region.

UDB is a persistent data store of our social graphs. On top of UDB, there is a huge cache tier called TAO [3]. TAO is a distributed write through cache handling social graphs and mapping them to individual rows in UDB. Aside from legacy applications, most read and write requests to UDB originate from TAO. In general, applications do not directly issue queries to UDB, but instead issue requests to TAO. TAO provides limited number of APIs to applications to handle social graphs. Limiting access methods to applications helped to prevent them from issuing bad queries to UDB and to stabilize workloads.

We use MySQL's Binary Logs not only for MySQL Replication, but also for notifying updates to external applications. We created a pub-sub service called Wormhole [6] for this. One of the use cases of Wormhole is invalidating the TAO cache in remote regions by reading the Binary Log of the region's MySQL instance.

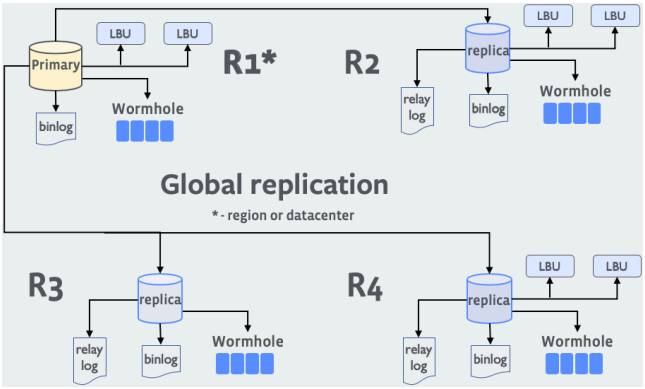

Figure 2: UDB Architecture

2.2 UDB Storage

UDB was one of the first database services built at Facebook. Both software and hardware trends have changed significantly since that time. Early versions of UDB ran on spinning hard drives that had a small amount of data because of low IOPS. Workload was carefully monitored to prevent the server from overwhelming the disk drives.

In 2010, we started adding solid state drives to our UDB servers to improve I/O throughput. The first iteration used a flash device as a cache for the HDD. While increasing server cost, Flashcache [7] improved IOPS capacity from hundreds per second to thousands per second, allowing us to support much more data on a single server. In 2013, we eliminated the HDD and switched to a pure flash storage. This setup was no longer bounded by read I/O throughput, but overall cost per GB was significantly higher than HDD or Flashcache. Reducing the space used by the database became a priority. The most straight-forward solution was to compress the data. MySQL's InnoDB storage engine supports compression and we enabled it in 2011. Space reduction was approximately 50%, which was still insufficient. In studying the storage engine, we found the B-Tree structure wasted space because of index fragmentations. Index fragmentation was a common issue for B-Tree database and approximately 25% to 30% of each InnoDB block space was wasted. We tried to mitigate the problem with B-tree defragmentation, but it was less effective on our workload than expected. In UDB, a continuous stream of mostly random writes would quickly fragment pages that were just defragmented. In order to keep the space usage down, we needed to defragment constantly and aggressively, which in turn, reduced server performance and wore out the flash faster. Flash durability was already becoming a concern since higher durability drives were more expensive.

Compression was also limited in InnoDB. Default InnoDB data block size was 16KB and table level compression required predefining the after-compressed block size (key_block_size), to one of 1KB, 2KB, 4KB or 8KB. This is to guarantee that pages can be individually updated, a basic requirement for B-tree. For example, if key_block_size was 8KB, then even if 16KB data was compressed to 5KB, actual space usage was still 8KB, so the storage savings was capped at 50%. Too small a block size results in high CPU overhead for increased page splits and compression attempts. We used 8KB for most tables, and 4KB for tables updated infrequently, so overall space saving impacts were limited.

Another issue we faced with InnoDB on flash storage was higher write amplification and occasional stalls caused by writes to flash. In InnoDB, a dirty page is eventually written back to a data file. Because TAO is responsible for most caching, to be efficient, MySQL runs on hardware where the working set is not cached in DRAM, so writing back dirty pages happens frequently. Even a single row modification in an InnoDB data block results in the entire 8KB page to be written. InnoDB also has a "double-write" feature to protect from torn page corruptions during unexpected shutdowns. These amplified writes significantly. In some cases, we hit issues where burst write rates triggered I/O stalls on flash.

Based on issues we faced in InnoDB in UDB, it was obvious we needed a better space optimized, lower write amplification database implementation. We found that LSM-tree database fitted well for those two bottlenecks. Another reason we got interested in LSM-tree was its friendliness to tiered storage, and new storage technologies created more tiering opportunities. Although we have not yet taken advantage of it, we anticipate benefits from it in the future.

Despite potential benefits, there were several challenges to adopting a LSM-tree database for UDB. First, there was not a production proven database that ran well on flash back in 2010. The majority of popular LSM databases ran on HDD and none had a proven case study for running on flash at scale.

Secondly, UDB still needed to serve a lot of read requests. While TAO has a high hit rate, read efficiency was still important because TAO often issued read requests from UDB for batch-style jobs that had low TAO cache hit rate. Also, TAO often went through "cold restart" to invalidate caches and refresh from UDB. Write requests also triggered read requests. All UDB tables had primary keys, so inserts needed to perform unique key constraint checks and updates/deletes needed to find previous rows. Delete or update via non-primary keys needed to read to find primary keys.

For these reasons, it was important to serve reads efficiently as well. A B-Tree database like InnoDB is well suited for both read and write workloads, while LSM-tree shifted more for write and space optimizations. So, it was questionable if LSM-tree databases could handle read workloads on flash.

2.3 Common UDB Tables

UDB has mainly two types of tables to store social data – one for objects and the other for associations of the objects [3]. Each object

and association have types (fbtype and assoc_type) defining its characteristics. The fbtype or assoc_type determines the physical table that stores the object or association. Common object table is called fbobj_info, which stores common objects, keyed by object type (fbtype) and identifiers (fbid). The object itself is stored in a "data" column in a serialized format, and its format is dependent on each fbtype. Association tables store associations of the objects. For example, the assoc_comments table stores associations of comments on Facebook activities (e.g. posts), keyed by pair of identifiers (id1 and id2) and type of the association (assoc_type). The association tables have secondary indexes called id1_type. Secondary index of the association table (id1_type index) was designed to optimize range scans. Getting list of ids (id2) associated to id (id1) is a common logic on Facebook, such as getting a list of people's identifiers who liked a certain post.

From schema point of view, object tables are accessed like a key value store than a relational model. On the other hand, association tables have meaningful schema such as pair of ids. We adopted an optimization called "covering index" [8] in id1_type secondary index, so that range scans can be completed without randomly reading from primary keys, by including all relevant columns in the index. Typical social graph updates modify both object tables and association tables for the same id1s in one database transaction, so having both tables inside one database instance makes sense to get benefits of ACID capabilities of the transactions.

2.4 RocksDB: Optimized for Flash

Utilizing Flash Storage is not unique to MySQL and other applications at Facebook already had years of experience. In order to address similar challenges faced by other applications, in 2012, a new key/value store library, RocksDB [9] was created for flash. By the time we started to look for an alternative storage engine for MySQL, RocksDB was already used in a list of services, including ZippyDB [10] Laser [11] and Dragon [12].

RocksDB is a key/value store library optimized for characteristics of flash-based SSDs. When choosing the main data structure of the engine, we studied several known data structures and chose LSM-tree for its good write amplification feature, with a good balance of read performance [1]. The implementation is based on LevelDB [13].

2.4.1 RocksDB Architecture

Whenever data is written to RocksDB, it is added to an in-memory write buffer called MemTable, as well as Write Ahead Log (WAL). Once the size of the MemTable reaches a predetermined size, the contents of the MemTable are flushed out to a "Sorted Strings Table" (SST) data file. Each of the SSTs stores data in sorted order, divided into blocks. Each SST also has an index block for binary search with one key per SST block. SSTs are organized into a sequence of sorted runs of exponentially increasing size, called level, where each level will have multiple SSTs, as depicted in Figure 3. In order to maintain the size for each level, some SSTs in level-L are selected and merged with the overlapping SSTs in level(L+1). The process is called compaction. We call the last level as Lmax.

In the read path, a key lookup 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, a whole binary search is used. Bloom filters are kept in each SST file and used to eliminate unnecessary search within an SST file.

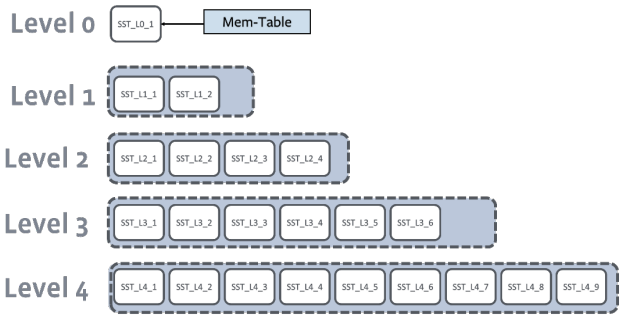

Figure 3: RocksDB Architecture

2.4.2 Why RocksDB?

As mentioned in Section 2.2, space utilization and write amplification are two bottlenecks of UDB. Write amplification is the initial optimization goal for RocksDB, so it is a perfect fit. LSM-tree is more effective because it avoids in-place updates to pages, which eventually caused page writes with small updates in UDB. Updates to a LSM-tree are batched and when they are written out, pages only contain updated entries, except for the last sorted run. When updates are finally applied to the last sorted run, lots of updates are already accumulated, so that a good percentage of page would be newly updated data.

Besides write amplification, we still need to address the other major bottleneck: space utilization. We noticed that, LSM-tree does significantly better than B-tree for this metric too. For InnoDB, space amplification mostly comes from fragmentation and less efficient compression. As mentioned in Section 2.2, InnoDB wasted 25-30% space in fragmentation. LSM-tree does not suffer from the problem and its equivalence is dead data not yet removed in the tree. LSM-tree's dead data is removed by compaction, and by tuning compaction, we are able to maintain the ratio of dead data to as low as 10% [14]. RocksDB also optimizes for space because it works well with compression. If 16KB data was compressed to 5KB, RocksDB uses just 5KB while InnoDB aligns to 8KB, so RocksDB is much more space efficient. Also, InnoDB has significant space overhead per row for handling transactions (6-byte transaction id and 7-byte roll pointer). RocksDB has 7-byte sequence number for each row, for snapshot reads. However, RocksDB converts sequence numbers to zero during compactions, if no other transaction is referencing them. Zero sequence number uses very little space after compression. In practice, most rows in Lmax have zero sequence numbers, so space saving is significant, especially if average row size is small.

Since RocksDB is a good fit to address the performance and efficiency challenges of UDB workloads, we decided to build MyRocks, a new MySQL storage engine on top of RocksDB. The engine implemented in MySQL 5.6 performs well compared to InnoDB in TPC-C benchmark results [14]. As Oracle releases newer versions of MySQL, we will continue to port MyRocks forward. The development work can be substantial because of new requirements for storage engines.

3. MYROCKS/ROCKSDB DEVELOPMENT

3.1 Design Goals

Re-architecting and migrating a large production database is a big engineering project. Before starting, we created several goals. While increasing efficiency was high priority, it was also important that many other factors, such as reliability, privacy, security, and simplicity, did not regress when transitioning to MyRocks.

3.1.1 Maintained Existing Behavior of Apps and Ops

Implementing a new database was only part of our project. Successfully migrating a continuously operating in UDB was also important. Hence, we made the ease of migration and operation a goal. The pluggable storage engine architecture in MySQL enabled that goal. Using the same client and SQL interface meant UDB client applications did not have to change, and many of our automation tools, such as instance monitoring, backups, and failover, continued to function with no usability loss.

3.1.2 Limited Initial Target Scope

We did not want to spend many years on a new database project. Spending five or more years to implement a great database, then spending additional multiple years to migrate, was not a reasonable direction for us. Our UDB databases kept growing, so we wanted to save space earlier rather than later.

We decided to limit the initial MyRocks product scope to UDB. Since UDB had specific table structures and query patterns, we believed it was feasible to make MyRocks beat our efficiency goals on UDB. On the other hand, fundamental designs such as on-disk index and row formats were discussed in the early stages. These were needed to support all workloads and were harder to change once implemented.

During MyRocks development, we continuously benchmarked against InnoDB based on UDB equivalent workloads. We used LinkBench [15], an open source benchmark tool that simulated UDB-like workloads. We also analyzed UDB production workloads. Based on this data, we drew up development tasks and prioritized accordingly. Once UDB on MyRocks reached production quality, we started supporting additional use cases.

3.1.3 Set Clear Performance and Efficiency Goals

As described previously, MyRocks was an efficiency driven project, so the focus was significant efficiency gains without sacrificing consistency. There were two goals compared to InnoDB in UDB. The first was a reducing database space by at least 50%, and the second, to do so without regressing CPU and I/O usage. Saving 50% disk space means that MySQL instance density is doubled per host, so a single database host needs to serve twice the amount of traffic. As more CPU and I/O pressure was expected, the specific goal was that they did not regress. Most UDB tables had secondary indexes, and LSM-tree database could manipulate secondary indexes more efficiently than InnoDB. We anticipated MyRocks could use less CPU and I/O for writes, while it was also expected to use more for reads.

Not all production workloads could migrate from InnoDB to MyRocks. We could not make a database that was better than InnoDB in all aspects. We picked LSM-tree over B-Tree to save space at the expense of read performance. For read intensive databases where all data resides in memory, MyRocks was hardly better than InnoDB and the space savings benefit was minimal. We made it clear we did not target such use cases (RUM Conjecture compromise [1]).

3.1.4 Design Choices

3.1.4.1 Contributions to RocksDB

We added features to RocksDB where possible. RocksDB is a widely used open source software and we thought it would benefit other RocksDB applications. MyRocks used the RocksDB APIs.

3.1.4.2 Clustered Index Format

UDB took advantage of the InnoDB clustered index structure. Primary key lookups could be completed by a single read since all columns are present. We adopted the same clustered index structure for MyRocks. Secondary key entries include primary key columns to reference corresponding primary key entries. There is no row ID.

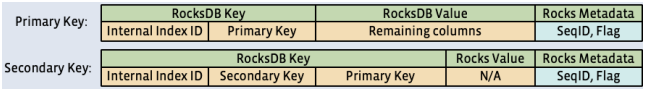

Figure 4: MyRocks Index Format

3.2 Performance Challenges

We made several read performance optimizations so that overall resource utilization was comparable to InnoDB. This section discusses these optimization improvements and features. Since RocksDB was a LSM-tree database, worse read performance as compared to InnoDB was expected. As we measured read performance gaps, we noted optimization opportunities that could fill the gaps. During early stage benchmarks, we also found that improving CPU efficiency was more important than I/O. Modern flash had sufficient read IOPS and since RocksDB wrote much less to flash, I/O was the lesser concern.

Another big challenge for LSM-tree databases was large number of tombstones can greatly slow down range scans. We implemented several features to mitigate the negative impact of delete markers.

3.2.1 CPU Reduction

3.2.1.1 Mem-comparable Keys

With LSM-tree, more key comparisons are made when executing queries as compared to InnoDB. Although RocksDB did several optimizations for it, the number is still significantly higher than InnoDB, especially in range queries. To look for the start key of a range, we only need one binary search in a B-tree, while we need to do one binary search for each sorted run in a LSM-tree and merge them using a heap. This can lead to several times more key comparisons. Similarly, simple key advancement does not require any key comparison in B-tree, while in LSM-tree, at least one key comparison is needed to adjust the heap, while often another one is needed to identify whether a record represents an older version. As a result, RocksDB is more sensitive to key comparison cost than InnoDB.

For example, most MySQL storage engines, including InnoDB, support case insensitive collations. This allows “ABC” to match “abc” on character comparisons, but it comes with a performance cost because each key comparison involves multiple steps, including key de-serialization. Even case sensitive collations data types may be required to go through some of these steps. In MyRocks, we always encode MySQL data to RocksDB keys in a

bytewise-comparable way which is much more efficient for comparisons.

3.2.1.2 Reverse Key Comparator

In RocksDB, iterating keys in forward order is much faster than reverse order. There are several reasons and most of them are fundamental to LSM-tree. Firstly, LSM-tree allows RocksDB to use key delta encoding inside each data block, but delta encoding is unfriendly to reverse iteration. Secondly, with LSM-tree, stale records may be present when we iterate through data. The records are stored in the tree in the reverse order of the key versions. This order guarantees fast forward iteration, but also slows reverse iteration because RocksDB needs to read one extra record for a key to find the latest version. Finally, the MemTable is implemented using a skip list with single direction pointers, so reverse iteration requires another binary search. As a result, ORDER BY query direction with reverse iteration is much slower than a forward one.

Fortunately, most UDB queries are uniform so that we can tune data placement based on common queries. As described in Section 2.3, we have two major data models in UDB – objects and associations. Associations are more expensive because they are fetched by range scans as opposed to objects which are fetched by point lookups. Range scans might span thousands of edges, and thus it was important to optimize them for social graph workloads.

TAO issues association range scans in descending order sorted by update time. To optimize descending scan performance, we implemented a reverse key comparator in RocksDB. It stores keys in inverse bytewise order. We adopted reverse key comparator for association secondary keys, so the descending scan by time internally executes a forward iteration scan, which was efficient. Our reverse key comparator improved descending scan throughput by approximately 15% in UDB.

3.2.1.3 Faster Approximate Size to Scan Calculation

As a MySQL storage engine, MyRocks needs to tell the MySQL optimizer the estimated cost to scan for each query plan candidate. For each query plan candidate, MySQL passes both minimum key and maximum key values to the storage engine, and MyRocks estimates the cost to scan the range and returns the cost to MySQL. RocksDB implements the functionality by finding the block location of both of minimum and maximum keys and calculating the size distance between the two blocks, including MemTables. This can cause nontrivial CPU overhead. We implemented two features to improve the performance of query cost calculation. One was completely skipping cost calculation when a hint to force a specific index was given. This is effective because the social graph queries are highly uniform so adding this hint in several SQL queries can reduce most of the approximate size overheads. We also improved the RocksDB algorithm to get the estimated size of scan ranges by estimating total size of full SST files within the range first and skipping the remaining of partial files as soon as RocksDB determines that they would not significantly change the result. RocksDB also tries to combine diving for minimum and maximum keys into one operation.

3.2.2 Latency Reduction/Range Query Performance

3.2.2.1 Prefix Bloom Filter

UDB had lots of range scans and it was more challenging in LSM-tree database. In B-tree, a range query starts from one leaf page, and a short range query only needs to read from one or two leaf pages. With LSM-tree, it has two parts – seek and next. The LSM-tree has much higher seek overhead. We usually need to read one data block from each sorted run, and it needs to be done even for sorted runs where there are no keys in the range. Reading more blocks means potentially more I/O and CPU for decompression.

To mitigate short range scan performance, we introduced the prefix bloom filter in RocksDB. Users specify the number of bytes as a "prefix", so that users can skip all sorted runs that do not contain any key starting with specific prefix.

Our association range scan was done by equal predicates from prefixes of indexed columns. Association range scan used the id1_type secondary key. The secondary key started with equal predicates (id1, assoc_type) and included several other columns, including timestamp. The latter columns were used to determine the sort order of the scan. For most TAO range scans, equal predicates were used for only the first prefix (id1, assoc_type) columns.

The prefix bloom filter is 20 bytes and is composed of the internal index id (4 bytes), the id1 (8 bytes), and the assoc_type (8 bytes). These two columns are always set in the WHERE clause with equal predicates for the id1_type secondary key. By supporting the prefix bloom filter, common index scans with equal predicates can use the bloom filter and improve read performance.

3.2.2.2 Reducing Tombstone on Deletes and Updates

Compaction was a major factor in RocksDB performance, impacting both read and write performance. One of the biggest pain points was how to handle deletions (called "tombstones") more efficiently. In the early stages, benchmarks like LinkBench were used to simulate UDB workloads. We observed an issue where the association range scan performance gradually degraded as the number of delete tombstones gradually increased in RocksDB data files. As mentioned in Section 2.3, the secondary index id1_type had many columns, including timestamp and version, and was a covering index [8]. This made range scans faster since random reads from primary keys were not needed for these columns. However, every update query modified the timestamp and version fields, resulting in constant updates to the secondary index.

Updating indexed columns in MyRocks means changing "keys" in RocksDB. Changing the RocksDB keys requires a Delete for the old key and a Put for the new key. The delete tombstone removes the matching Put during the MemTable Flush or Compactions, but the tombstone itself remains, since it is possible that a Put for the same key may exist in lower level SST files. The RocksDB compaction process drops tombstones during compaction once they reached Lmax. Updating MyRocks index key fields many times means generating huge number of tombstones. This makes range scans significantly expensive, since it needs to scan all tombstones. In UDB, changing primary key columns does not happen in usual workloads, so changing the primary keys incurred no performance implications. However, changing secondary keys does occur in steady state workloads. The association secondary keys were heavily used for range scans, such as returning a list of IDs who liked a specific photo.

To resolve tombstone inefficiencies, we introduced a new RocksDB Deletion type called SingleDelete. Compared to Delete, SingleDelete can immediately be dropped when removing a matched Put. The expected shorter lifespan of SingleDelete tombstones maintained range scan performance even with a steady

stream of secondary index changes. SingleDelete does not work when multiple Puts occur to the same key. For example, Put(key=1, value=1), Put(key=1, value=2) then SingleDelete(key=1) ends up as (key=1, value=1) still remaining, while Delete(key=1) hides both Puts. This is a data inconsistency scenario and SingleDelete should not be used in this case. The MyRocks secondary key prevents multiple Puts to the same key. If a secondary key does not change, MyRocks does not issue a RocksDB update call for the secondary key. If the secondary key changes, MyRocks issues SingleDelete(old_secondary_key) and Put(new_secondary_key). Multiple Puts to the same secondary_key without SingleDelete never occurs.

3.2.2.3 Triggering Compaction based on Tombstones

When deleting large numbers of rows, some SST files might become filled with tombstones and impact range scan performance. RocksDB extended compaction to track close-by tombstones. When a key range with a high density of tombstones is detected during flush or compaction, it immediately triggers another compaction. This helps reduce range scan performance skews caused by scanning an excessive number of tombstones. We call it the feature Deletion Triggered Compaction (DTC).

Figure 5 was a LinkBench results with and without DTC and SingleDelete. X-axis was time spent and Y-axis was query per second. With no optimization, QPS significantly degraded with time spent, and did not come back until most tombstone disappeared by compactions. SingleDelete showed similar behavior but QPS drop was lower and maximum throughput was higher. DTC made overall QPS drop much less significant. In production, providing stable performance was important so using both DTC and SingleDelete was more valuable to us.

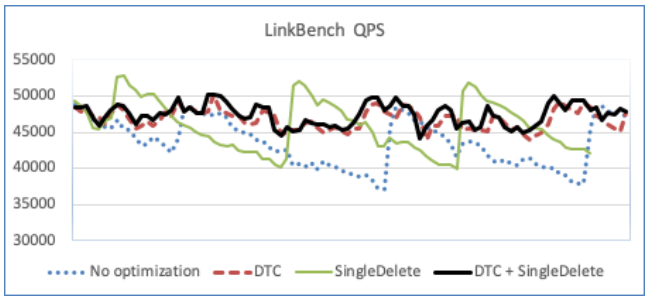

Figure 5: Linkbench with deletion optimizations

3.2.3 Space and Compaction Challenges

3.2.3.1 DRAM Usage Regression

The bloom filter is important for LSM-tree performance and needs to be cached in DRAM to be effective. This caused a significant DRAM usage regression compared to InnoDB. Our design used the typical bloom filter size of 10 bits per key. In order to reduce DRAM usage by bloom filters, we extended RocksDB to optionally allow it to skip creating the bloom filter in the last sorted run. By tuning the level size multiplier to 10 and the last run contain 90% of the data, the total bloom filter size was reduced by 90%, while the bloom filter is still effective. Skipping the bloom filter when using compression in the last level had the side effect that empty key lookups, such as unique key check by INSERTs, become more expensive. We chose memory efficiency over extra CPU time there.

3.2.3.2 SSD Slowness Because of Compaction

MyRocks relies on SSD's Trim command to reduce SSD's internal write amplification and improve performance [16]. However, we noticed that performance for some SSDs may temporarily drop after a spike of Trim commands. Compactions may create hundreds of megabytes to gigabytes of SST files. Deleting all of those files at once may cause a spike in trims resulting in potential performance issues or even stalls on flash storage. The solution was to add rate limiting to file deletion speeds to avoid such stalls.

Compaction I/O requests may also compete with user query I/O requests, causing query latency to increase. We added rate limits to compaction I/O requests to mitigate their effect on user query I/O.

3.2.3.3 Physically Removing Stale Data

In UDB, migration jobs are scheduled to remove unused data. Deletions have several types. Normal deletions, which execute the DELETE statement and use Delete/SingleDelete RocksDB APIs are typically related to user driven data deletion requests. Logical deletions, which execute UPDATE statements and use Put RocksDB APIs, are usually driven for space savings optimizations, such as overwriting unused data column to NULL but not deleting the record.

Our social graph workload typically allocates ever increasing fbid for object tables. The rate of modifications for an object usually decreases over time. If the object is deleted, no further modifications will be made to it. Since object tables use fbid as a primary key and since most insert/update/delete queries are for newly allocated fbid, there is a high chance that SST files containing both normal and logical deletions for old objects/fbids do not get picked up for compaction because all new changes come from newer fbids. These Delete or Put RocksDB operations for old fbids made their way into L1~L2, but compactions never continued to push them into Lmax. As a result, the row images containing the data remained in Lmax and they continued to take up space.

This was resolved by implementing Periodic Compaction in RocksDB. The feature checked the age of the data in the SST files, and if it was older than a settable threshold, it triggered compactions until it reached Lmax. This ensured that both Delete and Put eventually reached Lmax within reasonable time frames and that the space can be reclaimed.

3.2.3.4 Bulk Loading

One of the most common causes of stalls in LSM-tree databases like RocksDB was burst writes. Several types of data migration jobs, such as online schema changes and data loading, generated massive data writes. For example, the InnoDB to MyRocks migration needed to dump from InnoDB and load into MyRocks. MemTable flush and compaction are unable to keep up with heavy write ingestion rates. Writes, including those from user queries, would then be stalled until flush and compaction has made sufficient progress. Throttling was one way to mitigate, but it increased overall migration time.

To optimize burst writes and ensure they do not interfere with user queries, we implemented a bulk loading capability in RocksDB and MyRocks takes advantage of it. With bulk loading, MyRocks used RocksDB File Ingestion API to create SST files. The new SST files are directly ingested into Lmax, automatically and atomically updating the RocksDB Manifest to reflect the data files. Using Bulk Loading allows data migration jobs to bypass the MemTable and

compactions through each RocksDB sorted run level, thus eliminating any write stalls from those applications. Bulk loading requires that ingested key ranges never overlap with existing data. For us, massive data writes typically occurred when creating new tables, so bulk loading supports this scenario. Figure 6 was a benchmark to load our associations tables, which had both primary and secondary keys, into InnoDB and MyRocks, with and without bulk loading. X-axis was the number of tables to load in parallel, and Y-axis was rows inserted per second. With MyRocks bulk loading, insert throughput linearly scaled at least up to 20 tables and throughput was higher than InnoDB by 2.2 times (with one table, one concurrency) to 5.7 times (with 20 tables, 20 concurrency). Bulk loading eliminated stalls caused by these migration jobs.

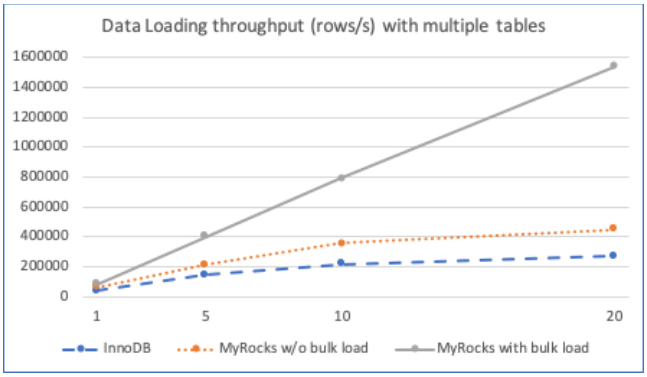

Figure 6: Bulk Loading throughput

3.3 Extra Benefits of Using MyRocks

While we had to overcome many challenges by using LSM-tree, in addition to space and write efficiency, the development effort also yielded the following benefits.

3.3.1 Online Backup and Restore Performance

We take logical backups from the database for disaster recovery and take binary log backups for point in time recovery. We implemented a read only snapshot to take consistent reads in MyRocks and let our logical backup tool use it. Compared to InnoDB, long running consistent reads were more efficient in MyRocks. InnoDB implemented UNDO logs [17] as a linked list and needed to keep all changes in the list after creating a transaction snapshot. It also needed to rewind the list to find the row based on the consistent snapshot. This caused significant slowdown if there were hot rows that changed a lot and a long running select needed to read the row after creating a snapshot. In MyRocks, a long running snapshot can maintain a reference to the specific version of the row needed.

Restore from logical backups is much faster than InnoDB since MyRocks can utilize bulk loading features (Figure 6).

We use physical backups for cloning a replica instance. Cloning an instance is done by creating a checkpoint in RocksDB and then sending all SST files and WAL files to a destination location. A RocksDB checkpoint creates hard links of the SST files. Since SST files are immutable, no modification to the original files can be made, allowing the checkpoint to point to a snapshot in time. Cloning to a host in a different region may take hours due to network transfer rates, especially if the source instance is large. A newly cloned instance will then replicate from the MySQL primary instance. This process to catch up the clone on transactions that took place after checkpoint creation may also take hours, depending on the rate of changes made on primary. To reduce replica synchronization time, we periodically re-create checkpoints during cloning, and continuously send newly hard linked SST files to the destination. Replication synchronization is only needed between the last checkpoint and the end of the copy, which could be limited regardless of database instance size.

3.3.2 Scales Better with Many Secondary Indexes

In MyRocks, manipulating secondary indexes can be done without random reads. Internally MyRocks issues RocksDB Put for inserting, SingleDelete and Put for updating, and SingleDelete for deleting secondary indexes, but they do not need to call Get/GetForUpdate. This was a significant advantage over InnoDB for heavily modified tables with multiple secondary indexes. Figure 6 shows that MyRocks had better insert throughput than InnoDB for a table that had one secondary index.

3.3.3 Replace and Insert Without Reads

MySQL has a REPLACE syntax, which blindly inserts/overwrites a new row. Internally REPLACE reads from a primary key to discover a matching row by the key. If the row exists, it is deleted and a new row overwrites it with the new value. Otherwise it behaves like INSERT. MyRocks optimized REPLACE to issue the RocksDB Put and skip unique key checks. The LSM-tree database made it possible to skip the random read and improved the write throughput.

MyRocks also has an option for INSERT statements to skip checking unique key constraints. If it is skipped, MyRocks does not need to perform a random read, while InnoDB still needs to issue the read. The drawback is these blind insertions may easily introduce data consistency bugs when interacting with other MySQL features like triggers and replication. We were not willing to take that risk in UDB so we took the safer direction to disable these optimizations. This performance feature is still available to users.

3.3.4 More Compression Opportunities

LSM-tree databases like RocksDB have multiple compression opportunities. We highlight one effective compression optimization, the “per level compression” algorithm.

RocksDB has multiple compression levels from L0 to Lmax. While RocksDB has 90% of the data in Lmax by default, most compactions typically happen at levels other than Lmax. This drove us to configure different compression algorithms between levels. Using a strong compression algorithm for Lmax and faster algorithm such as LZ4, or even no compression in non-Lmax levels, makes sense. RocksDB explicitly sets specific compression algorithms in Lmax. In UDB, we used Zstandard for Lmax, and LZ4 for other levels. Using faster compression algorithm for levels like L0 and L1 was helpful for MemTable flush and compactions to keep up with write ingestions. In UDB, approximately 80% of compaction bytes are completed in non-Lmax levels. By adopting LZ4 for non-Lmax levels, overall compaction time spent could drop to one third, compared to using Zstandard for all levels.

4. PRODUCTION MIGRATION

We started migrating from InnoDB to MyRocks in UDB after a year and a half of development and testing. Since new software will likely have bugs, we first added MyRocks as a “disabled replica”,

which meant executing replication traffic from the InnoDB primary instance, while not serving production read traffic. This enabled testing of MyRocks functionality without affecting production services. We could verify if MyRocks could serve writes without replication being stopped or inducing inconsistent data. We could validate data recovery correctness if the MyRocks instance crashed. Various automation tools, such as instance upgrade and database backup, can test interactions with the instance.

We created a disabled MyRocks replica instance by logically exporting InnoDB tables and then importing them into MyRocks using the bulk loading feature. Since InnoDB had clustered index structures, exported data was already sorted by primary key, which could be bulk loaded into MyRocks without extra sorting. For each replica set, we could migrate 200~300GB of InnoDB data per hour.

4.1 MyShadow – Shadow Query Testing

MyRocks needed to be tested before enabling it in a production environment. Of specific concern was how to verify that MyRocks could serve queries reliably, including verifying CPU and I/O utilization as compared to InnoDB, and checking for unexpected crashes or query plan regressions.

To test production queries, we created a shadow testing tool called MyShadow. At Facebook, we had a custom MySQL Audit Plugin that captured production queries and recorded them to our internal logging service. MyShadow read these captured queries from the logging service and replayed them to target MySQL instances. We used MyShadow first to capture read queries from the production InnoDB replica, then replayed the queries to a test MyRocks replica, allowing us to fix any query regressions before serving production read requests in MyRocks.

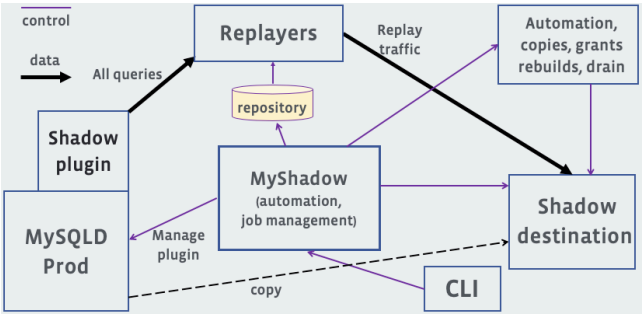

Figure 7: MyShadow Architecture

4.2 Data Correctness Checks

When starting to use a new database in production, it was a big challenge to validate the new database stored and returned correct data. We have InnoDB as the reference implementation to validate data correctness by comparing its data to MyRocks's. We created a data correctness tool to compare data returned from each storage engine. Our tool had three modes: Single, Pair and Select.

Single mode checked consistency between primary key and secondary keys of the same table in the instance by verifying if row counts and checksum of overlapping columns were identical. Single mode could find some internal MyRocks or RocksDB bugs that did not update either of the index correctly. For example, we could find a few RocksDB compaction bugs that ended up not deleting keys correctly, which showed up as index inconsistencies.

Pair mode ran full table scans to check row counts and checksum of the primary keys from two instances, based on a consistent snapshot at the same transaction state (Figure 8). Pair mode could compare InnoDB and MyRocks instances. It could find bugs that was not covered by Single mode, such as missing or extra rows in one of the instances.

Select mode was like pair mode, but instead of full scan statements, it ran select statements captured from MyShadow, and compared results between two instances. If select statement results were not consistent, it indicated inconsistencies so we could investigate further. Select mode could find inconsistencies reliably, but it also required filtering out false positives from nondeterministic queries (e.g. use of functions like NOW(), which returns the current time).

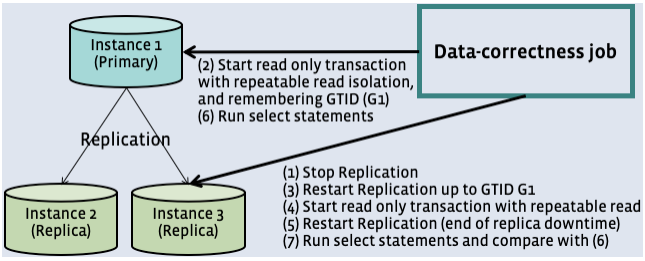

Figure 8: Data Correctness algorithm (Pair and Select mode)

4.3 Fixed Incompatible Queries

MyShadow and Data Correctness Tests revealed several issues that we could not find during unit tests or using common benchmarks like sysbench and LinkBench. Single mode Data Correctness found some RocksDB compaction bugs that did not handle Delete/SingleDelete correctly. Select mode Data Correctness found interesting bugs in the prefix bloom filter where some range scans with equal predicates returned fewer rows than expected. These were fixed prior to enabling MyRocks.

We will highlight one outstanding issue we found during MyShadow write traffic tests.

4.3.1 Gap Lock and Isolation Behavior Differences

Default transaction isolation level in MySQL is Repeatable Read, and is used in UDB. Repeatable Read implementation in InnoDB was unique compared to other database products. InnoDB's locking reads behaved as Read Committed, by returning the current snapshot. It was not strictly Repeatable Read. Instead, InnoDB locks ranges (Gap) on locking reads, and holds the locks even when the rows do not exist, to block other transactions from updating the same ranges. Statement Based Binary Logging also requires Gap Locks for correctness.

On the other hand, MyRocks adopted Snapshot Isolation model for Repeatable Read, which was the same as found in PostgreSQL. We considered the InnoDB style Gap Lock based implementation as well, but we concluded that Snapshot Isolation was simpler to implement. We can also switch our replicas to use Row Based Binary Logging and obviate the need for Gap Lock support.

This behavior difference caused issues when testing MyShadow write traffic. Snapshot Isolation based Repeatable Read returned errors if rows were conflicted, while InnoDB style locking reads did not conflict since it was essentially Read Committed. As a result, MySQL primary instances running MyRocks returned visibly higher number of errors than InnoDB primary instances

when using Repeatable Read isolation level. To reduce error rates, we investigated common conflicting queries, discussed the issue with application developers, and switched to Read Committed when we confirmed they were safe. We also added a logging feature to log queries using Gap Locks. Some apps explicitly depended on Gap Locks while others benefited from it by accident, and logging helped uncover these different cases.

4.4 Actual Migration

After passing MyShadow and Data Correctness testing, we started enabling MyRocks instances in production. We started with replica instances that served production read traffic and replicated from the primary instance.

Figure 9 shows migration steps we took in UDB. When we migrated to MyRocks in 2016-2017, we had six MySQL instances for each MySQL Replica Set (one primary and five replica instances replicating from the primary). We configured four InnoDB and two MyRocks instances in the same replica set, and the primary instance was fixed to InnoDB. MyRocks instances replicated from the InnoDB primary. MySQL separated the storage engines (InnoDB/MyRocks) from the replication streams (Binary Logs), so Binary Log formats were independent from storage engines. This architecture made the MyRocks deployment much less complex.

This configuration ran for a few months to validate that MyRocks could serve production read traffic reliably. We also continued intensive MyShadow write traffic tests to prepare for making MyRocks the primary.

Promoting MyRocks to primary was the culmination of years of effort since there was only one primary in each MySQL replica set. Despite all the planning and testing, there was still some trepidation. It required a leap of faith that we found all problems. When we enabled MyRocks as the primary, we monitored all applications closely for any unexpected behavior, but it all went smoothly. We continued to maintain a number of InnoDB replica instances in case we needed to revert MyRocks, but they were never needed. At this stage, we kept three MyRocks instances, including one primary, and three InnoDB replica instances for each replica set, and kept the topology up for another few months until we were confident that we could remove the InnoDB.

Having two or more instances with the same storage engine was essential during the transition phase. We had tens of thousands of replica sets, so losing some instances by hardware or software failures was normal. Had we lost all InnoDB or MyRocks instances in the same replica set, we would have had to perform a logical copy (exporting from InnoDB/MyRocks then importing into MyRocks/InnoDB), which was much more expensive than physical copy. Migrating from MyRocks to InnoDB was very painful because InnoDB did not have bulk loading capabilities. We kept three InnoDB instances for each replica set so that losing all instances at the same time was very unlikely.

We started enabling MyRocks replicas to serve production read traffic in mid 2016. We added MyRocks primary instances in early 2017. We gradually promoted more MyRocks instances to primaries. We removed almost all InnoDB instances by August 2017. Until migrations were complete, we kept two or more InnoDB and MyRocks instances in the same replica set, to avoid logical migrations when losing instances.

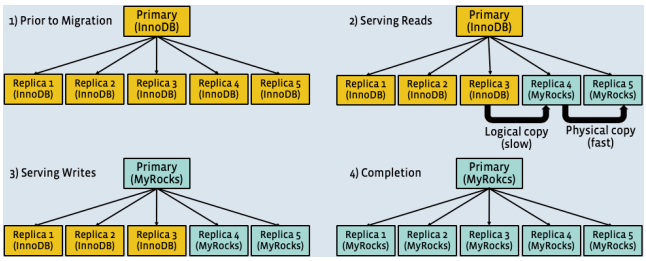

Figure 9: MyRocks instance migration steps.

5. RESULTS

These days we operate tens of thousands of MySQL replica sets in UDB, and the majority are only MyRocks. We keep a very small number of replica sets containing both InnoDB and MyRocks. These replica sets help us continuously benchmark against InnoDB based on our production workloads, as well as finding any unexpected bugs, such as query correctness or optimizer plan regressions.

5.1 MyRocks vs InnoDB Efficiency in UDB

Table 1 reflects a set of statistics from one of the UDB replica sets. It includes database instance size and the average CPU utilizations during peak time for two scenarios, with and without user read requests. Both scenarios include replication write requests. The data excludes DBA batch jobs such as backups and schema changes. We typically monitor space and CPU utilization for capacity planning. Common benchmarks like sysbench and TPC-C measure throughput rather than CPU utilization. In production database operations, we are working to save CPU time for given workloads. If a replica hits 100% CPU time, it will not be able to reliably serve more requests. So it is more important to track how much CPU time is spent on given workloads.

The MyRocks instance size was 37.7% compared to the InnoDB instance size with the same data sets. Our InnoDB instance was using compressed format. This shows that LSM-based RocksDB can be much more space efficient compared to B+Tree based compressed InnoDB. When we first deployed MyRocks in production in 2016, approximate space reduction was slightly above 50%. Since then, we changed the compression algorithm from Zlib to Zstandard, and support PeriodicCompaction to periodically reclaim stale data, realizing even greater space savings.

MyRocks was nearly 40% more CPU efficient than InnoDB when not serving read traffic and only serving write requests through replication. We have secondary indexes in most of the tables including objects and associations. When modifying rows MyRocks could maintain secondary indexes without random reads, which made a big difference compared to InnoDB.

CPU time spent serving both read and write traffic was slightly lower in MyRocks than in InnoDB. 1.65 means the MySQL server process (mysqld) used 1.65 CPU seconds for both user and system per second. Commodity servers have tens of CPU cores. We are targeting aggregated CPU utilizations by mysqld process under 40% of the total CPU cores during peak time. On typical hardware, we run many MyRocks instances per host. The MyRocks instances were more than 60% smaller compared to InnoDB, which means instance density became 2.5 times larger, and resulted in much higher aggregated CPU utilization. Note that these numbers were from replica instances. Primary instances used more CPU time for

sending binary logs to replicas, which were equally expensive between InnoDB and MyRocks. Some batch operations like logical backups also added CPU time although we did not usually run them during peak time. Right now our CPU utilization in UDB is still under target, but optimizing further is one of our long-term goals. Latency of read requests were comparable between InnoDB and MyRocks.

Overall, we could operate MyRocks with much smaller footprint and slightly lower CPU utilizations for our production UDB, compared to the compressed InnoDB we primarily used until 2017.

There were multiple factors where InnoDB could have been more efficient. For example, InnoDB supported only Zlib compression algorithm, while MyRocks had additional options like LZ4 and Zstandard, which were more CPU and/or space efficient compression algorithms. Also, our MySQL server binary was built with feedback-directed optimization (FDO), which optimized the binary based on MyRocks workloads. These reduced the CPU usage of MyRocks instances by approximately 7~10%. Due to maintenance considerations, we do not keep binaries optimized for InnoDB. By supporting Zstandard for InnoDB compression and building InnoDB optimized binary, CPU utilization of InnoDB instance will be comparable to MyRocks in our UDB workloads. Since we are already operating MyRocks everywhere in UDB, and due to density reasons, MyRocks is serving much more traffic than InnoDB per same space, we are focusing more optimizing CPU efficiencies in MyRocks.

Table 1: UDB statistics compared to InnoDB and MyRocks

Engine	Space	CPU seconds/s for writes	CPU seconds/s for reads + writes	Bytes written per second
InnoDB	2187.4GB	0.89	1.83	13.34MB
MyRocks	824.4GB	0.55	1.65	3.42MB

5.2 Migrated Facebook Messenger Backend from HBase to MySQL With MyRocks

HBase [18] was the choice for the Facebook Messenger backend database since its launch in 2010. At that time, HBase was chosen as the Facebook Messenger database because Messenger was write intensive and HBase was a good fit for it. HBase is based on a LSM algorithm, is strongly consistent, and runs well on HDD [19]. InnoDB was not chosen because it was not write and space optimized. Several years since then, flash storage has come to dominate database storage, and we have encountered issues where HBase could not use flash storage capacity because it exhausted CPU quickly, caused mainly by Garbage Collections.

The successful InnoDB to MyRocks migration in UDB suggested a direction to migrate from HBase to MySQL with the MyRocks engine, too. The migration was more complex than UDB since HBase and MySQL were very different database products. The migration was done in conjunction with refactoring the Facebook Messenger application. The data migration was done using MapReduce jobs to extract the data from HBase and loading it into MySQL. During the transition phase, consistency was checked by double writing and reading. The full details are discussed in [20].

UDB (InnoDB to MyRocks) and Facebook Messenger (HBase to MySQL/MyRocks) were very large OLTP database services at Facebook, and we were very excited that our research projects, which started in mid 2014, have come this far. We decided to move away from HBase to MySQL/MyRocks primarily since the latter ran much better on flash storage. MyRocks used less CPU time and stalled less often. People might be surprised that we migrated from NoSQL to SQL because of performance, since NoSQL was supposed to be faster. But in general, fundamentals like database architecture, data modeling, data access algorithm and ability to tune them easily mattered more than CPU time to parse SQL statements.

6. LESSONS LEARNED

The MySQL team at Facebook had Software Engineers who modified the MySQL server code base for Facebook workloads, and Production Engineers (PE) who made MySQL infrastructure automated and reliable day to day. PE were also heavily involved in MyRocks and RocksDB developments from very early stages. The PE spent significant efforts to stabilize InnoDB in UDB in the past, so they had better predications about what kinds of issues might happen during migrations. This helped prioritize and determine the features and directions for improving MyRocks and RocksDB. Ultimately, this reduced the time it took to successfully migrate the UDB backends. Driving migration projects without understanding the current production database workloads is harder to succeed.

When developing a new storage engine, it is very important to understand how core components worked, including flash storage and the Linux Kernel, from development to debugging production issues. It is important not only for database servers, but also for operational tools like backups. While it may be simpler to treat underlying components as black boxes, we believe this may lead to missed opportunities for improvement. Also, problems that occur between an application and the underlying component are harder to debug without first building expertise in these areas.

Outliers should not be ignored. Many of our production issues happened on only one or a few instances. Checking at the p90 or p99 data points would not have caught such issues. At Facebook, we have invested a lot in monitoring to quickly catch such outliers. Running correctness checks, even after production deployment, was important for finding bugs.

From operational standpoint, SQL compatibility was extremely helpful to migrate within reasonable time. Many of our important tools, such as MyShadow and data correctness, worked for both InnoDB and MyRocks thanks to SQL compatibility.

We realized that a LSM-tree database like RocksDB has many more adjustable parameters. It was much more workload dependent and harder to tune correctly. Applications generating many tombstones affected performance more severely than InnoDB and calibrating MyRocks properly was challenging. It is our long term goal to make MyRocks require less tuning to support a greater range of workload patterns. The LSM-tree database had features to speed up writes by sacrificing consistency, but we currently favor more conservative settings.

6.1 Memory Stalls and Efficiency

We have learned several lessons from memory allocation reliability and efficiency. RocksDB is more heavily dependent on memory allocator implementations than InnoDB. RocksDB allocates memory for each block and the actual block size differs from block

to block. We used jemalloc [21] and it was crucial for our workloads.

RocksDB used to rely on Linux's page cache for Buffered I/O with simply POSIX fadvise hints, while InnoDB supports Direct I/O (Linux O_DIRECT). We started using MyRocks with Buffered I/O in production, however we faced two challenges. One was transaction commit stalls triggered by the Linux page cache allocations, and the other was higher Linux kernel memory usage leading to swapping.

MyRocks transaction commit paths perform many memory operations (e.g. MemTable writes, binlog/wal writes), which caused Kernel Memory allocation stalls with the Linux Kernel 4.0. We upgraded Linux to 4.6, which solved most VM allocation issues and mitigated the problem.

6.1.1 Transitioning to Direct I/O in RocksDB

To achieve higher efficiency gains, we tried to use bigger flash storage capacity without increasing DRAM size, but we started seeing swaps triggered by memory pressure. We found that the Linux kernel allocated approximately 2~3GB of slab memory per 1TB of RocksDB SST files. As the storage size increased the overhead was not negligible, especially with a smaller DRAM configuration. Though our Linux Kernel team at Facebook implemented a new radix tree structure to reduce memory overhead to manage large data files, we decided to support Direct I/O in RocksDB and make it less dependent on a Linux kernel distribution [22]. Making MyRocks less dependent on specific operation system optimizations is important for the open source community. Most database users do not have a dedicated Linux kernel team, and some use proven stable kernel versions, which are relatively old. They could see better benchmark results even with older kernel. After using Direct I/O, our average slab size dropped by over 80%.

One notable challenge transitioning from buffered I/O to direct I/O in production was that we had to make sure we did not mix buffered and direct I/O to the same file. It was undefined behavior in Linux and caused significant performance slowdown. We adjusted our tools reading the files to match the access pattern. For example, our online MyRocks binary backup tool copied RocksDB SST files either using direct or buffered I/O, based on the instance's setting.

7. RELATED WORK

LSM-tree based databases have existed for a long time. Big Table [23], LevelDB [13], Cassandra [24] and HBase [18] are a few examples. To our knowledge, most of the optimization techniques we introduced in this paper are novel and not present in previous systems. MyRocks also differs from other systems as space capacity efficiency is the primary optimization goal.

Spanner [25] is a SQL database based on LSM-tree, but it is a SQL database built from ground up while MyRocks has a clear goal of matching performance of existing systems and keeping the database behavior compatible.

Several database services built their SQL databases using RocksDB as a storage engine, such as CockroachDB [26], Yugabytes [27], and TiDB [28]. Those systems built SQL and distributed capability from scratch, while one important goal of MyRocks is to keep those layers intact by continuing using MySQL.

There are other projects that create or extend MySQL storage engines, while keeping it transparent to database users and administrators. Amazon Aurora [29], TokuDB [30] and PolarDB [31] are a few examples. The MyRocks solution differs from these solutions for (1) MyRocks uses LSM-tree; (2) an existing key/value store library, RocksDB, is used rather than implementing a new one.

Some works, e.g. [32], introduce database migration system in the context of multi-tenant databases in cloud. Others shared experience on large scale migration of their databases, e.g. [33][34]. While their works focused on data integrity for the migrated data itself and performance tuning, we focused on detecting performance regression, data correctness bugs and query incompatibility caused by storage engine implementation as early as possible.

Regarding saving DRAM for bloom filter in LSM-trees, [35] and [36] proposed more adaptive and general approaches. While RocksDB uses prefix bloom filter to filter out short range queries, [37] proposed a general range filter for LSM-trees.

8. CONCLUSION AND FUTURE WORKS

This paper introduces UDB, our largest OLTP database for handling social activities at Facebook. We placed a high priority on continually increasing efficiency, which led to the development of a LSM-tree database that is more space and write optimized than the B+Tree database, InnoDB. We created MyRocks, a MySQL storage engine, on top of RocksDB, a key/value store library. MyRocks made our production database migration from InnoDB significantly easier, since both are MySQL storage engines. Leveraging MySQL features, MyRocks and InnoDB instances could replicate from each other. No significant client changes were needed. The LSM-tree database was known to be space and write optimized, but the downside was more expensive reads. While MyRocks addresses two major bottlenecks of the systems, we faced several challenges, including CPU efficiencies. Significant optimizations in RocksDB, such as hybrid compression algorithms and flexible bloom filter, addressed these issues. Our MyRocks mixed read and write workloads in UDB were eventually more CPU efficient than InnoDB's. Our success with UDB led to the HBase to MyRocks migration in Facebook Messenger.

Simplifying MyRocks performance tuning so it can be used without in depth knowledge is our next milestone. While configuring the prefix bloom filter, reverse key comparators and skipping last level bloom filters, we have managed to match the performance of InnoDB, it required significant effort. We plan to allow RocksDB to adaptively tune itself dynamically.

9. ACKNOWLEDGMENTS

While we cannot list all our contributors to the MyRocks and RocksDB engineering projects, we would like to thank everyone who supported us and note a few special contributors. Sergey Petrunya at MariaDB worked with us from very early stage of MyRocks and LevelDB Storage Engines. Sergey developed much of MyRocks storage engine implementation, including comparators, secondary index support, optimizer statistics, index condition pushdown, and batched key access. We would also like to thank Google for releasing LevelDB as an open source LSM-tree database, Dhruba Borthakur for creating RocksDB from LevelDB, and Mark Callaghan for generous mentoring and great advice. And finally, many thanks to our former and current colleagues on the MySQL Software Engineering, RocksDB Software Engineering, and MySQL Production Engineering teams at Facebook.

10. REFERENCES

[1] M. Athanassoulis, M. S. Kester, L. M. Maas, R. I. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In Proceedings of the International Conference on Extending Database Technology (EDBT) Conference, 2016

[2] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (June 1996), 351–385.

[3] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: how facebook serves the social graph. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). Association for Computing Machinery, New York, NY, USA, 791–792.

[4] Facebook's MySQL extensions. https://github.com/facebook/mysql-5.6

[5] Data centers year in review. Facebook Engineering. https://engineering.fb.com/data-center-engineering/data-centers-2018/.

[6] Sharma, Y., Ajoux, P., Ang, P., Callies, D., Choudhary, A., Demailly, L., Fersch, T., Guz, L.A., Kotulski, A., Kulkarni, S. and Kumar, S., 2015. Wormhole: Reliable pub-sub to support geo-replicated internet services. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15) (pp. 351-366).

[7] Flashcache https://www.facebook.com/notes/mysql-at-facebook/releasing-flashcache/388112370932/

[8] MySQL Glossary for Covering Index https://dev.mysql.com/doc/refman/5.6/en/glossary.html#glos_covering_index

[9] RocksDB. https://github.com/facebook/rocksdb

[10] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. 2019. Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 977–991.

[11] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 1087–1098.

[12] Arun Sharma. Dragon: A distributed graph query engine. https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/

[13] Ghemawat, S. and Dean, J., 2011. LevelDB. https://github.com/google/leveldb

[14] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strumm. Optimizing space amplification in RocksDB. In CIDR, volume 3, page 3, 2017.

[15] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). Association for Computing Machinery, New York, NY, USA, 1185–1196.

[16] Tasha Frankie, Gordon Hughes, and Ken Kreutz-Delgado. 2012. A mathematical model of the trim command in NAND-flash SSDs. In Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12). Association for Computing Machinery, New York, NY, USA, 59–64.

[17] MySQL InnoDB Undo Logs https://dev.mysql.com/doc/refman/5.6/en/innodb-undo-logs.html

[18] George, Lars. HBase: the definitive guide: random access to your planet-size data. " O'Reilly Media, Inc.", 2011.

[19] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS under HBase: a facebook messages case study. In Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST'14). USENIX Association, USA, 199–212.

[20] Xiang Li, Thomas Georgiou. Migrating Messenger storage to optimize performance https://engineering.fb.com/core-data/migrating-messenger-storage-to-optimize-performance/

[21] Evans, J. 2006, A Scalable Concurrent malloc(3) Implementation for FreeBSD

[22] Stonebraker, M. 1981. Operating System Support for Database Management. Communications of the ACM 24(7): 412-418

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages.

[24] Lakshman, A. and Malik, P., 2010. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), pp.35-40.

[25] Bacon, D.F., Bales, N., Bruno, N., Cooper, B.F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E. and Lloyd, A., 2017, May. Spanner: Becoming a SQL system. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 331-343).

[26] Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R. and Bardea, P., 2020, June. CockroachDB: The Resilient Geo-Distributed SQL Database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (pp. 1493-1509).

[27] Yugabyte, Inc. The Leading High-Performance Distributed SQL Database. https://www.yugabyte.com/. Accessed: 2020-02-09.

[28] PingCAP. Tackling MySQL Scalability with TiDB: the most actively developed open source NewSQL database on GitHub. https://pingcap.com/. Accessed: 2020-02-09.

[29] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T. and Bao, X., 2017, May. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 1041-1052).

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

[31] Feifei Li. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *PVLDB*, 12(12): 2263 - 2272, 2019.
DOI: https://doi.org/10.14778/3352063.3352141

[32] Elmore, A.J., Das, S., Agrawal, D. and El Abbadi, A., 2011, June. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (pp. 301-312).

[33] Netflix Technology Blog. Netflix Billing Migration to AWS — Part III. https://netflixtechblog.com/netflix-billing-migration-to-aws-part-iii-7d94ab9d1f59

[34] Migrating from AWS RDS MySQL to AWS Aurora Serverless MySQL Database. https://www.adelatech.com/migrating-from-aws-rds-mysql-to-aws-aurora-serverless-mysql-database/

[35] Dayan, N., Athanassoulis, M. and Idreos, S., 2017, May. Monkey: Optimal navigable key-value store. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 79-94).

[36] Zhang, Y., Li, Y., Guo, F., Li, C. and Xu, Y., 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based {KV} Stores. In 10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18).

[37] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 323–336.