Data Size and I/O Constraints within RocksDB

Michael Lanthier[1]

Abstract— **RocksDB is a Key Value store that uses Log Structured storage to efficiently store data on disk. Developed by Facebook, their system trades of CPU time for compact storage on disk and employs many compression techniques to ensure disks are being utilized maximally. They are able to achieve good enough read and write metrics for their workload but how does that apply outside of Facebook's environment? Through the use of data sets of varying sizes and system analysis tools this paper benchmarks MyRocks, a MySQL database using RocksDB as a storage engine. It will discuss the performance of MyRocks in the face of different data sizes as well as its underlying I/O constraints on cloud infrastructure.**

I. INTRODUCTION

As data sizes grow, large companies must spend a fortune on hardware costs to house and store their customers data. These high costs have been a motivating factor for companies like Facebook to develop alternative database systems that maintain good enough performance while allowing for better disk efficiency. If data is stored more efficiently on disk, that can reduce hardware expenses.

Traditional database systems tend to use B-Tree style indexes for their data. B-Trees allow for efficient reads and writes but are lacking when it comes to space efficiency. Because of the nature of B-Trees, they tend to not be completely full which results in wasted space on disk. Facebook took notice to this issue and aimed to create a system that more efficiently stores data on disk.

When looking at their own workload, Facebook observed that their production MySQL instances were not being stressed at peak loads. Their CPU utilization remained relatively low even at times of peak traffic. Their instances tended to get relatively few queries because of the way their data is distributed across systems. With their MySQL instances having low CPU utilization and an inefficient storage solution, RocksDB was born.

RocksDB is a key value store that uses log structured merge trees (LSM Trees) as its underlying storage engine. RocksDB stores new incoming data in a memory write buffer. Once the buffer is filled, the data is considered immutable and it is then flushed to disk. All the data that is flushed to disk is stored as Sorted Sequence Table data files. They are stored in sorted order in different levels of the LSM tree. The deeper into the tree you go, the bigger the level size. As data is inserted and removed, new levels are added or old levels are removed and the remaining data on is compacted to shallower levels.

In addition to the use of this LSM tree structure, RocksDB also provides extra optimizations to improve performance. Prefix bloom filters are used to assist with range scans and compression algorithms can be implemented to increase storage efficiency at the cost of write and read performance. Facebook has open-sourced RocksDB as well as their MySQL version, MyRocks, which uses RocksDB as a storage engine.

RocksDB is making the trade off between CPU time and space efficiency. Reading and writing is going to take a little bit longer because data may need to be compressed or uncompressed but in turn disk space is better utilized. Facebook has found that their own systems under their own workloads are able to perform with adequate read and write performance despite this tradeoff.

The goal of this paper is to see how RocksDB performs as the underlying storage layer of MySQL, known as MyRocks. This paper will go over a self designed customer-product workload that aims to emulate basic relational business logic that is loaded onto a MyRocks instance. I will describe the design methodology, testing structure, as well as the effects of different data sizes on the performance of MyRocks.

[1]M. Lanthier is an undergraduate within the Computer Science Engineering Department in the Jack Baskin School of Engineering at the University of California, Santa Cruz.

II. Project Goal

The creation of RocksDB was quite an interesting process. Facebook identified where their systems were not operating optimally and they designed a whole data management system around their workload to maximize their goals. So that begs the question, how applicable is RocksDB outside of their workload? Will they maintain good enough read and write latency to handle business logic? As data size grows and the amount of data stored on a single machine exceeds memory, how will reading and inserting to the LSM tree effect performance. If the data size is just outside of memory how does the occasional I/O operation to disk effect performance. Once the data set is well outside of memory what slowdowns can a user expect? These are all questions that adopters of MyRocks and RocksDB may want to know. This paper aims to take a look at the performance of MyRocks under multiple varying data sizes.

A recent paper published goes into the use cases that RocksDB sees in production at Facebook. While the paper focuses mainly on how the YCSB does not allow Facebook to replicate the types of workloads that their systems experience it does go more into more detail on what a RocksDB instance deal with on a daily basis.

In production at Facebook RocksDB sees quite a variety of workloads. It sits below MySQL and serves high volumes of get and put requests. Additionally it is also used in a system serving metadata for objects such as photos, receiving high read volumes. RocksDB also is an underlying part of their ML and AI platforms, serving counters and statistics for learning algorithms. This results in large amounts of merges. Within Facebook, RocksDB can be applied to a wide variety of workloads. In the original paper introducing RocksDB they also detailed that RocksDB performed well on OLTP style workloads on the systems they used. They ran it against TPC-C and it did very well.

But what kind of systems does RocksDB run on though? They detailed their test configurations in the original paper, "4-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." While I cannot speak to whether systems like this are common place in data centers, if I understand it correctly, Raid 0 is not very commonplace. Raid 0 allows for high performance reads and writes. Why use a high performance storage solution for testing MyRocks? Will the on disk storage solution affect database performance?

While we read the original RocksDB paper in class, the recent benchmark paper was published in late February, after this project had begun. While they are able to show RocksDB is definitely good enough for Facebook's MySQL instances, I will continue onward and ask how does it perform in my use case on varying data sizes and how will it perform on systems unlike their own?

III. System Design

A. Schema Design

The goal with the schema was to design a relational data model that looks and smells like something that could be seen in a real production level system. While I have no production level experience and I have never seen the data model of a production level system, I imagined what one may look like.

The schema is broken up into three tables, $customers$, $products$, and $orders$. The $customers$ table holds information about a customer such as the unique primary key, their name, address, and age. While storing age is significantly less efficient than storing a date, this is just filler data so it was a sacrifice to make data generation more convenient.

The $products$ table holds information about a product that could be sold. It holds the primary key, product_id, as well as other useful information such as name, category, price, quantity, posting date as well as a description.

An $orders$ table is also maintained containing orders made by customers on products. It contains its own primary key, order_id, as well as two foreign keys, customer_id and product_id. Foreign keys are not supported natively by MyRocks so the burden is on the developer to ensure that a customer or product id in the orders table exists in their respective tables as well.

Additionally a secondary index $customer\_index$ is maintained on the $orders$ table. It contains the customer_id as well as the order_id. This allows for efficient range scans of the $orders$ table based off of customer_id. Originally tests were run without this secondary index, this resulted in poor performance for all transactions that requested the orders from a specific customer, especially in the larger data sets. Once the index was created average query time and CPU load was decreased significantly.

B. Code base

The project is written using Java. The two biggest motivating factors were my comfort with JDBC as well as the convenience of being able to run Java byte code on JVMs on different machines without adverse effects. In retrospect Python may have been a better choice as I used JSON fairly frequently when moving data between machines. Since all the systems were running Ubuntu, Bash scripts were also used fairly extensively to automate repetitive tasks such as moving data sets and output data between machines.

C. Data Generation

One of the main goals of the project was to test MyRocks on different sized data sets, mainly sizes that fit in memory, just outside of memory, and well outside of memory. The MyRocks instance that all the test were run against has only 4 Gbs of memory. The motivation behind system size and other details about them is mentioned later in the paper.

In order to run our tests we generated three data set of varying sizes, 3 Gbs, 5 Gbs, and 10 Gbs. The first data set sits just inside memory, the second just outside of memory, and third sits well outside of memory. With these three configurations the goal was to see the effect of having to reading from disk. The smallest data set ideally should have little to no I/O operations which should in turn produce better latency and higher throughput. The two larger data sets are where it gets interesting. How will throughput and latency react with data sizes of varying degrees outside of memory.

But what is the composition of these data sets? In each data set customers and orders were forty percent of the records respectively with products making up the final twenty percent. This was to emulate a large customer base who make a fair amount of orders on a relatively small product size.

In order to generate a data set of a given size, we must figure out how the data is represented within the system itself. Sadly RocksDB and MyRocks provide little information on how much certain data types take up in memory and on disk. We know compression occurs deeper down in the LSM tree but how data such as dates and decimals are stored uncompressed was not available. Therefore, row size estimations were done based on how much space each data type took up in MySQL. Basic addition yields $customers$ taking up roughly 187 bytes, $products$ taking up 1148 bytes, and $orders$ only maintaining 32 byte records. This resulted in each data sets holding 10,600,000, 18,000,000, 35,000,000 records respectively to reach the size necessary for testing.

All the data sets were generated using functions written by myself as well as a library function for the more complex data. Strings were generated by randomly selecting alphanumeric characters which were varied in length to match the fields being generated. Dates and timestamps used a combination of a random date as well as the real time when they were generated. Range scans on dates or time stamps were not used so having similar dates and times was not an issue. Numbers were generated using the Math library within Java. Variable length structures such as $VARCHAR$s were filled up almost completely maybe minus a few characters do add variability to record size.

The most complex piece of data that needed to be generated were the unique identifiers (UIDs) being used as primary keys. I opted to generate my own UIDs for primary keys rather than use a sequence number handled by the database. This seemed more realistic and closer to a real world use case. UIDs were generated using the UUID class in Java. This class is guaranteed to generate unique identifiers for the system the program is running on. It generates a 128 bit unique identifier which I took the lower order bits from and stored them as a Java long. From there The five higher order bits were reserved to preserve uniqueness across machines. The top three bits hold a machine identifier while the next two bits are the thread identifier. This allows for eight machines, each with four threads, to be generating unique identifiers with out conflicts. This allows for easy inserts from multiple machines, something that ended up not becoming necessary in the final stages of testing but something that seemed necessary early on. Every UID is stored as a $BIGINT$ in the database.

All the data was generated into comma separated value(CSV) files. While convenient to work with, this has become problematic when loading bulk data. RocksDB is not very good at loading 10 gbs of unsorted data, resulting in extremely long wait times. RocksDB does support other methods of data ingestion when sorted on primary key and other conditions which allows for quicker data ingestion. Sorting via primary key before exporting to CSV in retrospect would have saved me a significant amount of time when it came to data loading. On one occasion, the database instance failed a health check and was brought down automatically by AWS, interrupting a long running load of the large data set. This had to be restarted and wasted a fair amount of work.

In order to always have data to request, the data generator also outputted a JSON file containing a subset of the existing UIDs for $customers$, $products$, and $orders$. Requesting nodes can immediately start making requests to the database and do not have to resort to expensive scans at the beginning of every test to generate a set of UIDs to query on.

D. *Driver*

In order to properly load test MyRocks and keep track of transaction information, a driver was written from scratch. The driver takes in the JSON emitted by the data generator and spins up a user defined number of threads. Each thread independently runs transactions against the database. The composition of each transaction will be covered in the next section.

Since we use a small set of transactions, the driver uses prepared statements to execute operations. These statements are prepared once the driver starts. The amount of time spent on a transaction is measured in milliseconds using $System.currentTimeMillis()$. This is called once before the first operation begins and then after the final operation finishes and is commited. The difference between the final and start time is the transaction duration. Sadly this does measure network time which is assumed to be roughly even across machines. Every test is run for 60 minutes.

As transactions run, each thread keeps track of the amount of time spent on each transaction. Information about these transactions are aggregated into an object of a class called $TransactionInfo$. This class takes in transaction start and end times and separates them into 1 minute epochs. Each epoch stores the number of transactions that begin in that epoch as well as all their lengths. So all the information about transactions that happen in the first 60 seconds of a test are stored in the first epoch.

In order to ensure that time intervals are roughly the same, epochs are determined relative to the time the database thread starts, not system or global time. Two systems with two clocks set to different times that start the load test at the same global time (I click start on the machines simultaneously) will have their epochs line up assuming that they are counting milliseconds forward at roughly the same rate. Synchronizing the machines to ensure that they all were working on the same interval is out of the scope of this project.

$TransactionInfo$ exports JSON which allows for data to be moved away from the requesting nodes to a central location for analysis. Outside of transaction times, it also calculates 50th, 99th and 99.9th percentiles of slowest transactions. Systematic file naming, bash scripting, and some short Java programs allows for all the JSON files to be easily aggregated together for analysis.

E. *Transactions*

The systems host only a small set of transactions which tend to be read dominated. There are no deletions in this workload. The first transaction occurs about 25% of the time, it gets a customers information based on UID and then gets all of the orders made by that customer. The next transaction occurs about 50% of the time, it gets information about a customer, gets info about 10 random products and then orders one of those products by inserting it into the orders table and then updating the inventory of the product. The final least popular transaction is an update to the products table. Select a random product and update its quantity.

None of these transactions ensure any form of data validation. Negative inventories are certainly possible and orders of sizes of zero may also exist. A date range scan was also originally employed but skewed results because there was no secondary index on the date being searched resulting in costly full table scans.

F. Working Environment

1) Instance Types: All systems were run on Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instances. The single database system was hosted on a c5.large compute optimized instance with 8 Gb of memory, 256 Gb of storage, and a 2 core CPU. The driver was run on a t2.micro instance with 1 Gb of memory, 8 Gb of storage, and a 1 core CPU. Both instances were running Ubuntu 18.04. The database was also tested on a t2.large general purpose instance which ran into performance issues detailed later in the paper.

All three data sets were generated locally on an Ubuntu 18.04 laptop and were stored in AWS S3. All output information to be used for analysis was sent to S3 where it was pulled locally to the same laptop to be aggregated. An overview of the system architecture can be seen in Figure 1.

2) Data Flow: Every test required that the data set being tested be pulled down from S3 onto the database instance. It is then loaded in via the $LOAD\ DATA\ INFILE$ command in MyRocks. All code is compiled on my local Ubuntu laptop, and then uploaded to S3 using a combination of Bash scripting and the AWS command line interface. Once the database is loaded and ready, a startup script then spins up the requesting node. It is fed a startup script which downloads all the required code from S3 and installs system dependencies such as a JVM. Once MyRocks is loaded with the data, $iostat$ is run on the database instance to measure where IO is spent. The requesting node can then be started running 4 threads via a bash script. Each test runs for an hour and the output JSON detailing the transactions are uploaded to S3.

3) MyRocks Configuration: Configuring MyRocks was a task that took much longer than expected. Facebook only officially supports CentOS, an operating system that AWS does not have EC2 instances for. Community instances of CentOS exist but they are either to old or too new to properly run MyRocks without issue. Ubuntu is supported

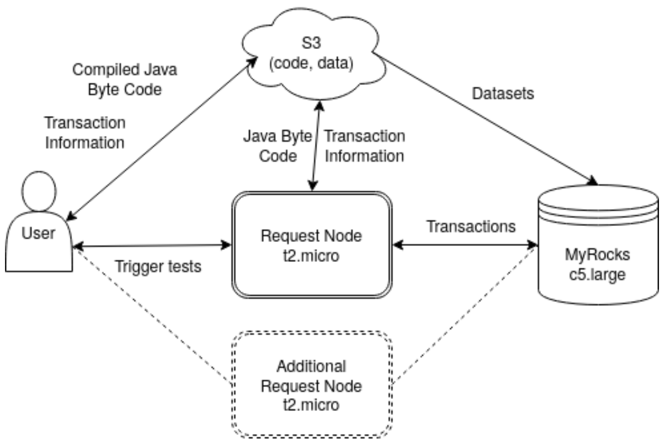

Fig. 1. System Architecture

by best effort and mostly by the community. When building from source, the most recent release did not build out of the box on Ubuntu 18.04. After much debugging I was forced to disable any ability to use LZ4 compression as well as hard code the level at which compression would begin. Also determining which packages to install that were compatible with MyRocks required some trial and error.

After installation, I also ran their full test suite which promptly failed every test. After attempting to debug it and doing some internet research it appears that this is not an uncommon occurrence and MyRocks worked just fine for what I needed it to. After some debugging a subset of the tests succeeded. For some reason the tests tried to test InnoDB as well which does not ship with MyRocks, so it is quite possible these tests cover more than the base installation.

Documentation on MyRocks was available online but remained fairly vague. Most of their online instructions explained the wonderful features of MyRocks and RocksDB but failed to provide resources on which system variables needed to be set and what to set them to do get the performance needed. In a single occasion I had to play the guessing game on acceptable values trying multiple combinations of words, letters, and numbers to try and set a database variable.

Out of fear of breaking everything and running into more issues, the database is running in almost pure default mode. It holds a read committed isolation level and begins compression using ZLIB

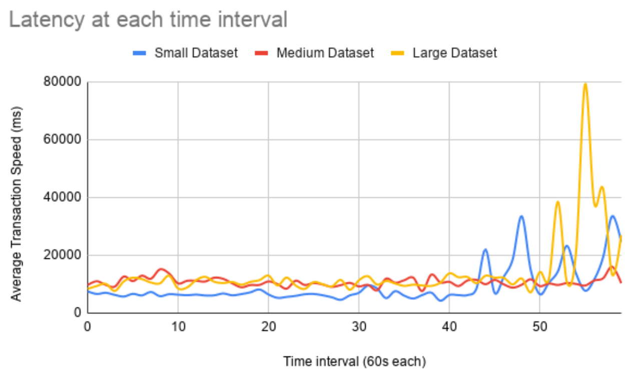

Fig. 2. MyRocks latency against 4 machines, each with 4 threads

at level 4. The only variable enabled was $rocksdb-commit-in-the-middle$ to allow large CSV files to be read in.

4) Instance Selection: Determining which EC2 instances to use took trial and effort. The first run of tests was run with a t2.large hosting MyRocks and 4 requesting nodes on t2.micros each running with 4 threads. This meant that at any time MyRocks could be handling 16 concurrent transactions. Using the Amazon Cloudwatch Logs, which measure instance performance, revealed that the t2.large was pinned at 100% CPU usage for the full duration of the test. This resulted in unexpected performance such as large spikes in latency across all tests. The variable performance can be seen in Figure 2. This system set up resulted in transactions that took 10 seconds or even more which is an unacceptable amount of time for the small workload and fairly simple queries.

In order to look at the effects of data size on MyRocks, completely saturating the CPU seemed to be the wrong approach. To reduce the amount of work the database was doing a secondary index was added to the $orders$ table. This drastically improved transaction latency and decreased CPU load. The number of requesting nodes was also reduced to a single node with four threads. Doing so smoothed out the latency graphs and removed the odd variations happening at the later epochs. The instance type was also changed to a c5.large to stop the system from being CPU constrained.

Another constraint behind instance selection on AWS is pricing. Students are eligible to recieve \$100 of free AWS credit on a yearly basis which was my budget. AWS pricing can be somewhat mysterious, depending on the instances you are running you may be paying extra for higher CPU usage or higher network traffic. This made me err on the size of caution and use smaller cheaper instances.

Data set size was also another issue. As the instances get larger, memory sizes increase drastically. This proved to be an issue as generating and loading data sets into MyRocks would take much longer. This meant less time for meaningful work. A smaller instance size allowed me to generate data sets large enough to reason about database performance while not having to deal with the extra time overhead of generating data for a machine operating with 16 gigabytes of memory or more.

IV. Results

So how did the system perform? Once the index was added and the CPU was no longer pinned at one hundred percent usage, the system performed fairly well. In all workloads transactions were well below sub-second latency. The slowest transactions also remained relatively quick, with the .1% of slowest transactions clocking in at 200ms.

As expected, the smallest data set performed best in terms of latency and throughput. It was able to handle the highest number of transactions and maintain a low latency in every given time interval. It also had good tail latency, the .1% slowest transactions still averaged around 60 milliseconds. If you are able to keep your data set in memory, do it.

A brief look over the data exported by $iostat$ on the MyRocks instance shows that the small data set has a lower amount of disk reads than the other data sets but oddly enough has higher disk writes than the other two data sets. This could be for a multitude of reasons. One theory is that because it had such high throughput, all changes must be made fully durable via the write ahead log. So that means more of the WAL is being flushed to disk which could in turn result in more writes but less read as they always occur in memory.

In terms of CPU performance, this system is not constrained at all. It hung around 50% or 60% idle, well bellow heavy load. The CPU did have to wait for some of the I/O operations it was performing as

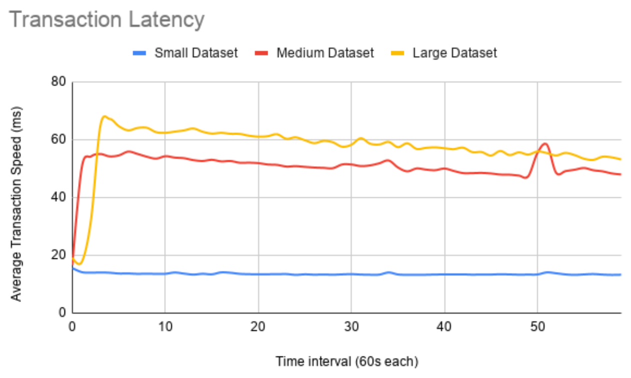

Fig. 3. Latency with a single requesting node with 4 threads

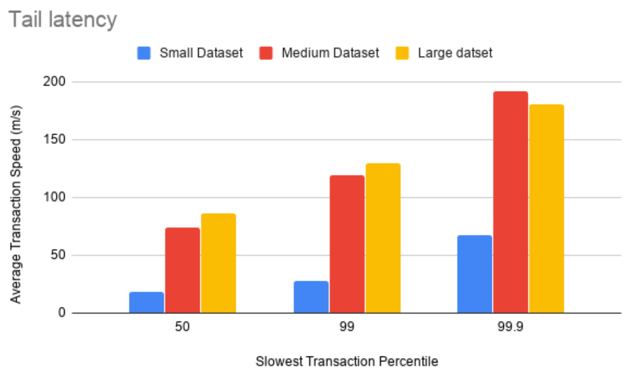

Fig. 5. Tail latency with a single requesting node with 4 threads

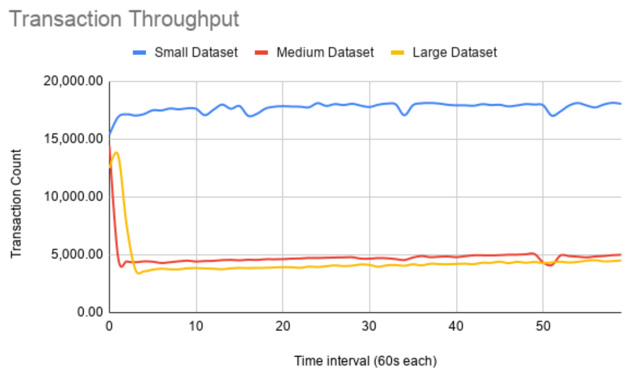

Fig. 4. Throughput with a single requesting node with 4 threads

it averaged roughly 20% to 25% I/O wait time on the CPU. This is the proportion of processes that are waiting for I/O operations to complete. This is a relatively low amount and still allows for good performance.

The medium and large data sets show some interesting results. As expected, they both performed worse than their in memory counterpart. When looking at Figure 3 latency quickly jumps up and sits well above the latency of the small data set. The same can be seen in the steep drop of throughput measured in Figure 4. Interestingly enough both the large and medium data sets start to see decreased latency and increase throughput at the later time intervals. Even though the large data set has double the data of the medium data set, it still performs similarly to its medium sized counterpart.

Tail latency for these data sets was also nearly identical. As seen in Figure 5 the medium data set is slightly faster in the 99th percentile but the large data set actually beats out the medium data set when it comes to the 99.9th percentile. So despite the doubling of data size, the slowest transactions running on appear to take a similar amount of time.

The information provided about CPU load by $iostat$ was quite interesting for both of them. The medium and large data sets had almost identical CPU usage and disk usage. They both were constrained by I/O wait time, the medium data set hovering between 85% and the large data set only a couple of points above it. They both also had similar total CPU load, both data sets only having around 7% to 10% CPU usage left idle.

It is quite interesting how similarly they performed. In the use case constructed here, once data leaves memory the system performance immediately takes a huge hit. As the data set size increase outside of memory though it appears there is slowdown but not a drastic amount of it. Both the large data set had higher latency and lower throughput but not by very much.

How will this scale with more users and data though. Looking at the information provided by $iostat$ shows that one of the biggest constraints on operation appears to be I/O wait time. Both the large and medium tests consistently had a high percentage of instructions waiting on I/O. This bottleneck is a concern as it inhibits the ability for the system to scale. Long I/O operations result in increased latency and this can only get worse as the number of conccurent users increase. One remedy for this could be higher performance storage which may reduce the amount of time spent on disk reads

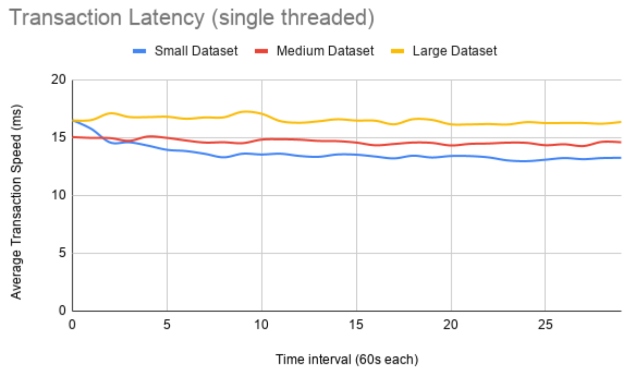

Fig. 6. Transaction latency against a single thread

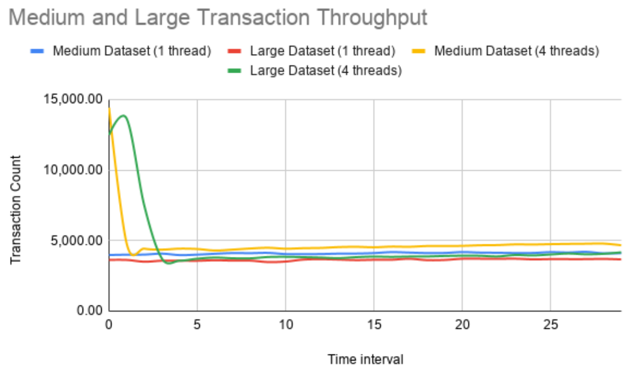

Fig. 8. Medium and Large Data Set Throughput

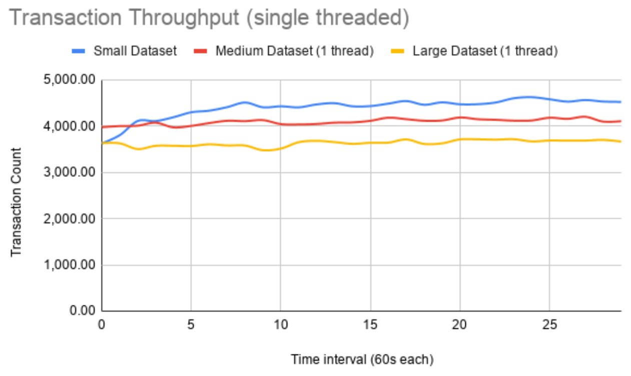

Fig. 7. Transaction throughput against a single thread

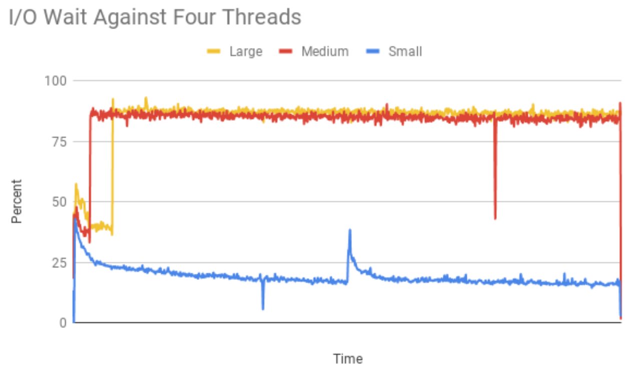

Fig. 9. I/O Wait Times Against Four Threads

and writes. I am unaware of the properties of the storage attached to default EC2 instances so this could be the culprit.

To try to further verify what could be limiting system performance, the tests were re-run with the requesting node using only a single thread. In the interest of time these tests only ran for 30 minutes.

As seen in Figure 3, all three data sets have exceptionally low latency when there is only a single thread operating on the database. While not graphed, tail latency at the 50% level is almost identical with them all separating a bit at the higher percentiles with the large data set performing the worst and small data set performing the best.

Performance across data sets is very similar under the hood. *iostat* shows that the small, medium, and large data sets fluttered around 8%, 10%, and 13% I/O wait percentages respectively. They also continued to remain at extremely low CPU usage, all of them very idle. This can be seen in Figure 10.

Now let us compare this back to the multi threaded example. The small data set is able to continue to perform well. Looking at the *iostat* data we see it is not burdened by high I/O wait times and therefore has a drastic increase in throughput (roughly a 4x increase). But the medium and large data sets tell a very different story. As shown in Figure 8, the medium and large data sets do not scale nearly as well. They both barely increase throughput over their single threaded counterparts. This is because I/O operations are a bottleneck to the system. I/O wait times in the CPU jump from around 10% all the way up to 80% in the medium and large data sets. This results in only an ever so slight increase in throughput. I/O wait time in the four threaded test can be seen in Figure 9.

Facebook's optimizations with RocksDB revolved around the CPU being free to do more

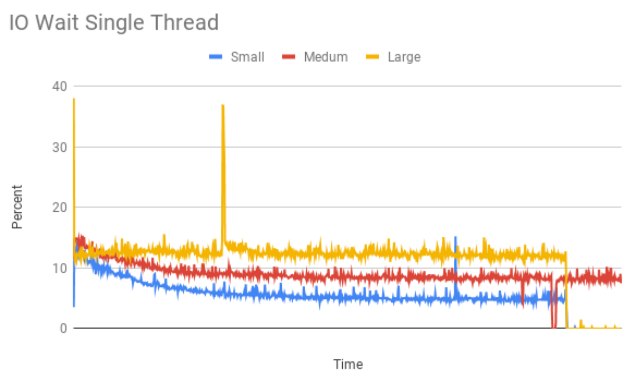

Fig. 10. I/O Wait Times Against Single Thread

compression and work. This extra free CPU compute capacity does not matter if the storage system is still lacking. In these tests it seems like the storage system is the limiting constraint. With higher performance storage it is quite possible that the medium and large data set will perform much better. It is also doubtful that compression kicked in for the large test as it should have dedicated more CPU time to decompression and likely decreased throughput and increased latency at a higher amount than observed in these tests.

V. Conclusion

MyRocks performs well on the three data sets provided here when paired with adequately fast storage and compute. Their is definitely a performance hit, as to be expected, once the data size exits memory sizes. If data sizes remain below the memory limit MyRocks can properly scale with traffic increase. As to be expected, data sizes outside memory are not as fast as those in memory. But once these data sets are outside of memory, while system latency and throughput are similar between medium and large data sets, they scale very poorly compared to small data sets. By looking at the proportion of time the CPU spends doing certain operations, MyRocks is be constrained by I/O waiting times on the the low end AWS EC2 instances. Better storage solutions or increasing memory size are two adequate ways to possibly improve the performance of the MyRocks instance we ran. Having a fast enough storage solution is extremely important as relying on your data set to stay in memory is not a valid option. This explains why Facebook tested on a RAID 0 array of disks enabling exceptional disk performance.

Future work can be done to determine how MyRocks performs against a MySQL database with similar specifications to allow for a benchmark of comparison. Another side of MyRocks that could be further tested is it's batch data loading features. Getting data into MyRocks was extremely slow, how much do their techniques such as sorting via primary key help decreasing the times spent on bulk insertions.

VI. Challenges and Learning

As someone who has never used a database in production or ever seen what a database benchmark actually looks like, this was quite the challenge and learning experience. Here I will break down a few areas that I thought were challenging or fulfilling that do not fit very well into the previous sections of the paper.

A. *Amazon Web Services*

AWS was the cloud provider of choice because it is the cloud provider that I have used in the past. Whether that is a good reason or not, that is the way things work. It was a mostly positive experience. Time was definitely saved due to my ability to automate tasks through their command line interface. I was easily able to move data around, spin up EC2 instances, and do other tasks which made for an efficient testing workflow. To get to that workflow as always was a little bit of a chore. In my experience with AWS I can get about 95% of the systems I'm working with to interact together but there is always that small 5% that lacks documentation that I have to go down the rabbit hole to solve. Thankfully working with their more popular services (S3 and EC2) made finding solutions much easier but it is not unheard of to spend a week trying to solve a seemingly simple issue. One issue I did run into that threw me into a loop for a while was realizing that their monitoring program (Cloudwatch) for EC2 instances for some reason actually did not measure disk reads/writes. Even if they were occurring it continued to flat line at zero. This forced me to redo quite a few tests where I was depending on it for data.

In terms of pricing the AWS credit was extremely helpful. In the two weeks of most intensive testing, I continued to be extremely careful of when the instance was running to reduce costs. This resulted in a $25 total cost for two weeks, but it was projected I would finish around $45 for the month with EC2 and S3 storage costs.

B. MyRocks

MyRocks was the first time I have used an open source system that I had to build and install from the source code. This took about the whole part of two weeks and required a fair bit of outside help. Wrestling with Ubuntu, Facebook's own two different instructions on how to install, and recent changes to the source code that broke the build all ended up taking much longer than expected but was a good exercise in debugging and understanding how a large system is built.

C. Database Performance

While the three data sets that I worked with are relatively small, they are still the largest amount of data that I have put into a database system that I queried on. When sitting in class talking about secondary indexes, I never realized how much of an optimization they would be until I tried doing a full table scan over a 10 Gb data set. This brings further appreciation and understanding of the necessity of the optimizations within databases. As the data sizes grow these operations will become more and more expensive and without proper optimization a database can increase its latency extremely fast.

D. Database Benchmarks

This project has given me so much appreciation for those who benchmark databases and those publish tests of their own systems. There are so many different variables and operations happening it is extremely hard to weed through all the information to accurately test the database. As I was writing transactions, test harnesses, and analysis functions I would quite often realize that in order to isolate exactly what I wanted would require an ungodly amount of work. Real database benchmarks are complex systems in of themselves and I'm constantly impressed with those who work on them.

References

[1] Cao, Zhichao, et al. ”Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook.” 18th USENIX Conference on File and Storage Technologies (FAST 20). 2020.

[2] Dong, Siying, et al. ”Optimizing Space Amplification in RocksDB.” CIDR. Vol. 3. 2017.