EPISODE 762

[INTRODUCTION]

[00:00:00] JM: RocksDB is a storage engine based on the log-structured merge-tree data structure. RocksDB was developed at Facebook to provide a tool for embedded databases. The code for RocksDB is a fork of LevelDB, an embedded database built by Google for the Chrome browser. Every database has a storage engine. The storage engine is the low-level data structure that manages the data in the database. RocksDB is widely used in database applications where a log-structured merge-tree is preferable to a B-tree. These tend to be write-heavy workloads.

In past shows we have explored applications of RocksDB in our coverage of databases like TiDB, data-intensive applications like Smite, and data platforms like Rockset. In today's episode, Dhruba Borthakur and Igor Canadi join for a deep-dive into how RocksDB works. Dhruba was the original creator of RocksDB, and Igor is a former Facebook engineer who worked on RocksDB in its early days. Both Dhruba and Igor now work at Rockset.

We talk about the log-structured merge-tree in detail and discuss why an LSM has higher write throughput than storage engines based on a B-tree, and we also evaluate some of the use cases for RocksDB. This was a technical episode, but we cover the topic of RocksDB in a great amount of detail, from high-level to low-level. I hope it's useful to anybody that is considering using RocksDB in an application.

[SPONSOR MESSAGE]

[00:01:53] JM: Triplebyte fast-tracks your path to a great new career. Take the Triplebyte quiz and interview and then skip straight to final interview opportunities with over 450 top tech companies, such as Dropbox, Asana and Reddit. After you're in the Triplebyte system, you stay there, saving you tons of time and energy.

We ran an experiment earlier this year and Software Engineering Daily listeners who have taken the test are three times more likely to be in their top bracket of quiz scores. So take the quiz

yourself anytime even just for fun at triplebyte.com/sedaily. It's free for engineers, and as you make it through the process, Triplebyte will even cover the cost of your flights and hotels for final interviews at the hiring companies. That's pretty sweet.

Triplebyte helps engineers identify high-growth opportunities, get a foot in the door and negotiate multiple offers. I recommend checking out triplebyte.com/sedaily, because going through the hiring process is really painful and really time-consuming. So Triplebyte saves you a lot of time. I'm a big fan of what they're doing over there and they're also doing a lot of research. You can check out the Triplebyte blog. You can check out some of the episodes we've done with Triplebyte founders. It's just a fascinating company and I think they're doing something that's really useful to engineers. So check out Triplebyte. That's T-R-I-P-L-E-B-Y-T-E.com/sedaily. Triplebyte. Byte as in 8 bits.

Thanks to Triplebyte, and check it out.

[INTERVIEW]

[00:03:43] JM: Igor and Dhruba, you guys are both engineers at Rockset. Welcome to Software Engineering Daily.

[00:03:47] DB: Thank you.

[00:03:48] IC: Thanks for having us.

[00:03:49] JM: So we did a show about Rockset and data engineering and some modern data infrastructure problems. I wanted to do another show that dove deeper into RockDB, which is a lower level infrastructure primitive that is a part of Rockset. It's a part of the company that you're building. So I'd like to go through some various aspects of RocksDB.

At a high-level, can you just give a little bit of history and how RocksDB got started?

[00:04:17] DB: RocksDB is an LSM engine. It's a data storage engine, which means it's kind of the base part of any database management system. As far as the history is concerned, I think

we started this project around 2011. That was a time when I was at Facebook and Igor was at Facebook, both of us were Facebook engineers. We were looking at databases at Facebook, which are moving from disk subsystems to flash. In flash storage, random writes are not a good thing in general, because it wears out the flash. So we're looking for an LSM engine.

One of the nice small elegantly written LSM engines that I found was LevelDB. It was a project from Google. It was a great piece of code. It was very well-understandable very quickly. The disadvantage of that piece was that it was mostly written for Chromium browsers. So we had to do a lot of things to – We basically took the basic LSM engine from LevelDB and created RocksDB out of it.

Then maybe the first year at Facebook it got used in maybe four or five use cases. I can describe those to you later, but those use cases were all on the server-side and we realized that we needed an LSM engine that can power big server-side database.

[00:05:32] JM: Talk a little bit more about why you needed a new type of database for server-side applications, because I mean we have a bunch of server-side databases that have been around forever. We have MongoDB, MySQL. What was the reason for needing a new database?

[00:05:51] IC: The time at Facebook, there was a lot of applications that wanted to have basically their binaries running close to the data. For example, there was newsfeed, there were some ads backends. A lot of the ranking frameworks that really made decisions based on huge amounts of data and then didn't want to go over the network.

Each of those teams built a separate engine, obviously for their own use case, and then when they improved their engine, only their application went faster. So I think when Dhruba and I were at RocksDB project, the cool part that we achieved there was now we had the common infrastructure where all of those applications that wanted to have data close to their binaries on the same host could use common infrastructure like RocksDB. So when the team made let's say performance improvements or some usability improvements, all teams that were based on that architecture benefited.

[00:06:40] JM: Would you call this an embedded database?

[00:06:44] IC: That's exactly right. So embedded database in terms of when you use RocksDB, there's no networking involved. It's just library for you to store data on the same host that your application is running on.

[00:06:54] JM: You talked about some applications there, newsfeed, some ads, backends. These are applications that were novel at Facebook. These were – The social networks unlocked some new types of applications. Can you describe why a new database was required for these kinds of applications that were perhaps closely tied to social networking?

[00:07:16] DB: Yeah, absolutely. So the first use case I still remember is Facebook has something called a graph database, and they needed to build a huge secondary index on this graph database, which means that the graph database, let's say, is in MySQL, and then you need to build indexes on all the different columns of the table, and we used to run a project called Dragon, which was the first project that uses RocksDB inside Facebook.

The first use case again is to build a huge secondary index on the large graph database. This is similar to what we are doing here at Rockset, but we can talk about that later. Yet another use case at Facebook was spam detection. RocksDB was used for building some of the core spam detection software that Facebook was running at the time. Then I think the third use case was more about recommendation. So things like how you can find coefficients or like similarity coefficients between two different Facebook users.

So there is a big process that runs, which finds how close are you to any other person in your friend list. There's a coefficient that gets calculated, and that was built using a RocksDB backend database. So these were like two different types of applications where RocksDB excels in.

[00:08:28] JM: When I'm thinking about these applications, they sound very different than something like a server that hosts my user data, or my login data, or my password data. These systems that have to be much more consistent, perhaps more stringent requirements on what the data actually is and it's availability. If you have something like newsfeed, what my newsfeed

is could be many different things. What ad get served to me could be many different things. Is there something there about the fact that there is less strong consistency requirements or – I don't know. Tell me more about the theme between those applications.

[00:09:10] IC: So one theme that we discovered was so-called log tailor architecture, where basically what you have is you have a stream of data coming in, for example, for a newsfeed, stream of all events that are happening with your friends. Then you have the dialer for that log, which applies those updates in real-time and builds basically some kind of an index of that data and you usually do that in a sharded way. For example, for a newsfeed thing, they had obviously a couple of shards, then you have an aggregator that responds to newsfeed queries where it's a one single machine that talks to basically many different leaves, yet host that data that they tail from the log. Then when the aggregator, the request comes in, it distributes the request a bunch of different leaves that then do some copulations locally.

For example, some ranking based on huge amounts of data. That's way it has to be sharded. They send top, say ten, results to the aggregator that now has 10 times number of shards results and then picks the best 10 results to show to the user.

So the cool part there is you have basically distributed query engine that can run on huge datasets that's updated in real-time.

[00:10:15] DB: So yeah, I think just to add to your comments. So RocksDB is very good in write-heavy workload as well as low-latency read workload. So the workload for reads are mostly like point queries and short range scans, and these are very useful for these applications, like newsfeed that Igor mentioned, where a person needs quick latencies on a huge large dataset. So take for example if you have the ability to process as part of the query a few gigabytes of data in a few milliseconds and give responses to the user. That is the kind of workload that RocksDB excels in. So that's the difference from MongoDB and other systems also as well.

[00:10:51] JM: We've covered the applications to some degree. Let's talk a little bit more about the lower level architecture. As you said, RocksDB is built on a data structure called a log-structured merge-tree, or an LSM tree, and this works by having different levels of storage that

got compacted overtime. So there's a layer of recent writes that sit in-memory and those database writes are sitting in-memory, they get compacted into small files that are sorted, then the small files can get compacted into larger sorted files and you can have this hierarchy of larger and larger files that are sorted. You can tell me if any of that is wrong, but describe the log-structured merge-tree data structure in more detail.

[00:11:39] IC: What you said is exactly right. So what we do is we apply the writes are buffered into in-memory write buffer, which in the implementation that we use and LevelDB uses and some others is a skip list, although it doesn't have to be. It just has to be sorted. Once the skip list is full, you don't want to run out of memory. You flush the meme table into a file on disk, and then obviously after sometime those files accumulate and then you do so-called compaction.

Now, there are many different strategies for compaction and originally there's a Cassandra compaction, LevelDB compaction, and RockDB basically, the cool part is we actually enable many different compactions. So by default, there's a LevelDB, which what it means is there're riles organized on levels. So once the mem table is flushed into level zero, and the level zero is obviously the newest writes, the newest updates. Then overtime it gets pushed into level one using compaction level two and so on. Compaction process in a leveled compaction processing is you take some files from let's say level three, some files from level four, merge them together and the output gets written into level four.

Now on the read side, what happens is on each of those levels, files are partitioned, so the key range is partitioned. So when let's say you want to key b, you know exactly for particular level which file contains the key b. So then when you do a read, for each level, you need to do only one read, because you know exactly where that key can reside. Then there's also cool stuff built on top of those files, which is called bloom filters, where if you can prove with the bloom filter that the key cannot be there possibly, then you can avoid I/O.

[00:13:15] DB: Yeah, just to add to Igor's comments. As far as compaction is concerned, I generally get questions of the type like what is different? Why is compaction different in RocksDB versus LevelDB and Cassandra and other places?

So what happens is that by default, the compaction strategy in RocksDB is level compaction, which means that the files that restore on disk, they're partitioned by key ranges. Whereas the default for HBase and Cassandra, there the compaction strategy is that file ranges are portioned by time, the default strategy. You can always put a non-default one later.

So RocksDB has the default one where files are partitioned by key ranges, but you also have a universal compaction style in RocksDB where you can say that I want actually to partition files by time ranges, not just by key ranges. So that's one of the basic differences between these two different – Or three different LSM engines that are out there.

[00:14:09] JM: Different databases have different data structures that underlie the writes and the reads that are serving the queries that are hosting the data that is actually in the database, and I think it would be worth talking a little bit about LevelDB as an example of using the LSM data structure. As you said, the background here at LevelDB came out of Google. This was an LSM-based database made for the Chrome browser. It's a database that's in the Chrome browser that is using this LSM data structured that we just described. Can you explain why LevelDB, why the LSM data structured was useful for the Chrome browser?

[00:14:56] DB: Yeah. No, absolutely. So one line of thinking is that how do you design a storage engine where you handle a lot of writes, a lot of random writes? If you have a – Traditionally storage engines have been using the B-tree structure, which means that when a write comes in, you go find the page on the storage engine where – Or the storage where the key is supposed to reside, read that page then make modifications and write it back to that page.

So now every page – So if you write 100 keys, it's likely that you probably modify 100 pages on the disk. Whereas when you come to in-memory data systems, where all your data is in RAM, for example, you are very cognizant of say cache line, how your data is cached using the CPU caches, where if you update a little bit of the data in the CPU cash, you need to invalidated those CPU caches into other CPUs in your multi-threaded machine, or even if you look for SSDs, they're also – If you do random writes to a page, the way it works on an SSD is that it reads the page, makes the modification to that page and then writes it to a new place in the page. So random writes cause a lot of trouble when you are running in — pure in-memory databases or on SSD-based databases.

So this is why the LSM storage engine is popular. So LSM storage engine, what it does is that when a write comes in, it doesn't go write it to overwrite the original place in the storage. It goes and writes this – Accumulates some writes for a little bit of time and then bulk writes all those accumulated writes into a new place on the storage engine. This is very compatible and works elegantly with flash and in-memory data storage engines.

Then the other thing is that for the LSM design itself, the original paper was actually built – I think it came out in 1995 or 1996, but it wasn't –

[00:16:45] JM: Actually, the first paper about the data structure.

[00:16:47] DB: That's right. The first paper, I think Patrick O'Neil created this paper. It was just a paper. It wasn't really in vogue till flash storage became really popular. I think 2005 or 2006 is when I heard more about LSM engine and Bigtable. So Bigtable also uses the LSM engine. That is mostly on disk-based systems still, but LSM is good for disk-based systems if you have a lot of writes, because disks cannot do a lot of random writes again. Yeah, I think 2005 is when it started to become popular again.

[00:17:20] JM: Yeah, LSM was kind of an obscure data structure in the mid-90s, and then it got popularized. But I think what he talked about in that paper, if I – I think I read the abstract and he was talking about this issue of the tradeoffs between how you can do random writes versus sequential writes, I think. Can you talk about those terms in a little bit more detail, random writes versus sequential writes? What does that actually mean?

[00:17:49] IC: So let's say you're writing out 10 megabytes to a storage, and let's say you decide to write sequentially whether on SSDs or on disks. That operation will be very fast. But let's say now you tried to write 10 megabytes but chunk it out into let's say kilobyte increments randomly on the disk or randomly on SSDs. On the disk, that will take very, very long time if you try to write on hard disks. On SSDs, it will take a bit faster, but it will wear out the SSD more quickly because of what Dhruba said, where when you do even a small write on an SSD, it actually does a bigger write, because it explodes into page size chunks.

So that way both the hard disks, sequential are so much faster because how hey spin. Even on flash devices, sequential writes wear out the SSD less. Then in-memory, it also makes sense. Dhruba said, if you're talking about very low-latency environments, it makes sense because then your cache is more pure. You don't invalidate your cache line that frequently.

In all scenarios, if you can avoid random writes, especially small random writes, the better. Then the tradeoff we didn't about about LSM is that your reads are slower, because now think about you have more files where your key could potential reside, whereas on B-trees, you know exactly the page that you have to read. So that's a tradeoffs, and the tradeoff is much worse off on hard disks. That's why B-trees were so popular than hard disks. But now flash devices, most good flash devices have a lot of random I/Os that they can do on the read side. So it's not a big of a problem.

[SPONSOR MESSAGE]

[00:19:31] JM: Clubhouse is a project management platform built for software development. Clubhouse has a beautiful intuitive interface that's made for simpler workflows or complex projects. Whether you are a software engineer or a project manager or the CEO, Clubhouse lets you collaborate with the rest of the product team. If you're an engineering-heavy organization, you will love the integrations with GitHub, and Slack, and Sentry, and the other tools that you're feeding into your issue tracking and project management.

To try out Clubhouse free for two months, go to clubhouse.io/sedaily, and it's easy for people on any team to focus in on their work on a specific task or project in Clubhouse while also being able to zoom out and see how that work is contributing towards the bigger picture. This encourages cross-functional collaboration. That's why companies like Elastic and Splice and Nubank all use Clubhouse. Those companies have all been on Software Engineering Daily and we know they are competent engineering organizations.

Stay focused on your project and stay in touch with your team. Try out Clubhouse by boing to clubhouse.io/sedaily and get two months free. Thank you, Clubhouse, for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:21:02] JM: Let's talk about a read and a write in a log-structured merge-tree data structure. Again, you have these different levels. You have the n-memory level. You have some smaller file levels near the top and some larger file levels near the bottom. Can you describe what happens during a write to this data system and a read from this data system?

[00:21:28] IC: Sure. So on the write, basically what happens is you just append the write to your in-memory data structure skip list and you're done. Obviously, you add it also the write that had logged just in case you lose your memory data structure. So you can recover your state. But in most cases, this is very simple and very fast operation.

In the background, what happens is that write is propagated using a process called flash, which takes this in-memory buffer and puts it on disk. Then also process of compaction that we talked about earlier where you take a couple of files and marge them together and write out the results and delete old files. That's the write path.

Then for the read path, there are two different read paths. One is for range queries, another one is for point lookups. Point lookups are very well-optimized, because as I said before, there's a bloom filter. When you have a point lookup, first you go obviously to the in-memory buffer, because that's where if the write happened recently, that is where the key resides. If you find it there, you're done. If you don't find it there, then you go into the level zero, which is the newest writes after the in-memory write buffer. You look it up there. If you don't find it there, you go to level one, level two, level three. Obviously, this sounds like a lot of I/Os, but with bloom filters, most of those I/Os are filtered out.

For the range scans though, especially short range scans, bloom filters, traditional bloom filters don't work. So you actually have to go to all the files that could potentially contain this particular key. For example, range scan means for particular key, find me that key and all other keys in sorted order after that key. That happens a lot into index scans and traditional databases and/or where you just want to get a bunch of results for a query.

What happens is you basically have a marker in each file and in a mem table, in memory write buffer. You find the marker that's some kind of binary search over your dataset and then you put all those markers in a heap to find the smaller marker. Then you start form there. Once you advance that one, you just advance them all as you go along, and that's how RocksDB provides a sorted view of the data. That is a very important building block for building databases.

[00:23:33] DB: Yeah, I'd like to add one more thing there. For short range scans, one of the thing that we did when we were indexing the Facebook graph database was that we built something called prefix bloom in RocksDB. So this was a new concept. So in the Facebook database, what happens is that those short range scans mostly happen within a specific user ID or within a specific short period of time, those scans that happen in a database.

Like Igor said, if we had a bloom for every key in the database, then it doesn't help short range scans. So what we did was that we created this concept called a prefix bloom, which means that you can have a bloom on a prefix of the keys. So if your key is very large but the first portions of your key is the user ID, then you could have a prefix – then you can have a bloom on the prefix of the key. So when a range scan comes with a certain prefix, you can actually avoid reading the files which don't even have that prefix. So that helped in some of the short range scans for querying the inverted index on a social graph.

[00:24:37] JM: Does data structure here that Igor mentioned called a bloom filter. So if I recall, bloom filter is that you can give it a key and it will tell you if the key might be in a dataset. Is that what it does?

[00:24:52] DB: Yeah. Given a – You can have a configurable number of bits for every bloom that we create for every key. So what happens is that when a query comes in, if we have to look into a file to see if this key is in the file, we first look at the bloom. If the bloom says this key is not in the file, then we can avoid reading the file completely. So that part is 100% accurate, but sometimes the bloom might say that, "No. The key is in this file." So we go look into the file, but maybe the key is not there. So then that read was kind of wasted, but it's a way basically for you to reduce the number of reads that you have to do to find your data.

[00:25:28] JM: Yes. So if I understand, this data structure is useful here because you've got these different levels which it's referring to the same dataset, but they have different ages basically. The mem-pool at the top is the most up-to-date age, but it's not in order. Then some of them, they get cleared out of the mem-pool and then they get stored in the levels beyond that. Then at each of those levels, you look at a bloom filter first and say, "Okay, should I look in the actual file here?" If not, you can proceed to a lower level and look at these increasingly compacted datasets so that you can eventually find the level of compaction at which your key is actually going to be found.

[00:26:18] IC: Exactly. Exactly.

[00:26:19] JM: Okay.

[00:26:19] DB: Yeah, that makes sense. I think like the LSM paper I think becomes very practical only in the presence of blooms. So although the LSM paper came out in 1996, it's combination of that technology along with the fact that there is something called these bloom filters. Bloom filters came out of the network world I think. The first thing was about like some [inaudible 00:26:42] encoding that kind of network research.

But then when it got applied to the LSM engine, this became a really useful technology, where LSM you can combine with bloom filters. Otherwise, every read in LSM would be like very high-cost, because you'd have to look at every file. So I just wanted to make the point that LSM engines became really impractical only in the existence of bloom filters.

[00:27:04] JM: Just to emphasize what we're even talking about here, RocksDB or LevelDB, these LSM-based structures, the different levels I think there's an analogy to be drawn between what is commonly called a cache hierarchy. Would you say that's accurate? Is that a good analogy? A cache hierarchy versus these levels in LevelDB or RocksDB?

[00:27:27] DB: Yeah. I mean, the data inside the RocksDB storage engine, they're kind of organized in the form of levels. So it's not really caching, but it is kind of persistent, but the same key could exist in different files inside the RocksDB storage engine. So let's say you write a key and after 10 minutes you write the same key again. Now, both versions of the key might

exist in the storage engine. But when you read, you always find the latest one. The RocksDB code ensures that it always returns you the latest version of the data. Yeah, you can think about it as maybe some kind of caching or duplicate data internally, but that's how the storage engine is configured.

[00:28:08] IC: I think the difference between caching and RocksDB levels or LevelDB levels is that in caching, you want hotter keys or keys that you just recently accessed are in memory. Versus in RocksDB levels, keys that you recently written are in a higher level. So we do have – So that's in leveled structure. But we do have on top of level structure, we have a block cache, which then also does caching based on the keys you read most recently.

[00:28:35] JM: But there is the most recent writes are going to be faster to access than a write that happened a longtime ago.

[00:28:44] DB: This again depends on your compaction strategy like I mentioned. So if your keys are portioned by – If your data is partitioned by key rangers, then it doesn't really matter whether you wrote the data a long time back or recently. Whereas if you partition your keys by time ranges, then it will be a different answer for that question.

Also in the process of blooms, I think some of these are alleviated, because blooms kind of reduce 99% of your reads. Again, the performance is different on different hardware. So the focus of RocksDB project was always to keep – The focus was entirely on performance. It is a very lightweight C++ language, and the focus is always been on performance and it is very pluggable, which means that you can run it on different hardware systems by giving different configurations. You'd have probably 100 tunables that you can tune, which lets you extract the last mileage out of the hardware, and your hardware might be very different from the hardware I run it on. I know people run RocksDB on disks, as well as SSDs, as well as in-memory systems.

[00:29:46] IC: Anything else on performance?

[00:29:47] DB: I'm trying to think. So as far as the performance is concerned, given performance on different hardware is very difficult in general, because you tune it differently. So we have made the RocksDB's engine very pluggable, which means that you can have your own

mem-table format. So you can have your own SSD file formats. You can plug in your own code. You can have what other configurables out there.

[00:30:10] IC: Many, many others.

[00:30:11] DB: Yeah, there are many, many other configurable pieces where you can plug in your own code. Oh! There is another interesting thing, things like compaction figures. So there is a use case in the early days at Facebook where we wanted to delete data after 30 days or after 15 days. So you can specify TTL or time to leave for every piece of data. Typically, other storage engines. So you'd have to write some application code to scan your database and delete records which are older than 30 days.

But in RocksDB, there is feature called compaction filter. So you can set this in the RocksDB storage engine. So when RocksDB compacts data into background, it automatically deletes data that is 15 days old. You see what I'm saying. So you don't have to write too much application code. You just plugin things that are pluggable in RocksDB and configure it correctly.

There is another thing in RocksDB called the merge operator. Merge operator is used to create kind of virtual – So RocksDB is a key-value store. But supposed you wanted to give the abstraction of a list, like a name of a list with a list of values inside the list, like a Redis list for example. So it's easy to build using RocksDB, because you can use merge operators.

So every write, you don't have to read the list and append it to it and then write it back. You just write it to another place in RocksDB, and you configure something and RocksDB telling him that, "Hey, these are actually part of the same list," and the application can specify some code to be run as part of the merge operation to merge these things in the background and create the list. So it avoids random writes again.

Another example I remember was accounting service. So accounting service needs to read the whole value from the storage engine, add one to it and write it back. But this one is very cheap to do in RocksDB by using something called merge filters. Where the only thing that the application writes is +1, +1, +1, +1 kind of things into the storage engine, and the storage engine when you do a read, it automatically knows how to get all those plus one's and make it –

Let's say you added five of those and it gives you result five. It gives you the list abstraction or the counter abstraction and by avoiding a lot of read modify writes.

[00:32:13] JM: So that's important, because at its core, RocksDB is just a key value store which would just be like, "Okay, the only thing I can do with that is like retrieve one value associated with one key," which is like not very useful if I want to like give me all the users in this database.

[00:32:31] DB: Yeah, absolutely. Take for example supposed you have a service, which is counting the number of clicks that somebody id doing. So let's say you are a user, I want to count how many times you clicked on this object. So one option would be to read the old value from the storage engine, add one to it and then write it back saying you clicked it one more time, one more time. So every operation is a read and then a write.

Whereas if you use the RocksDB merge operation, your application is not going to do a read modify. It is going to just say add one more to that counter. Add one more to that counter. Then when your application reads the value of the counter, it magically gets the value summed up from all the plus ones that you have been putting to the database.

So this helps avoiding a lot of read requests on the database. It's also good for essentially things like evens. So here I'm talking about these evens, right? You're clicking on some pieces of your app or device, and evens are getting generated and somebody is counting the number of evens. So using the merge operator, you can actually make these even countings quite less resource – Fewer resource can do more things compared to other storage engines.

[00:33:35] JM: So we talked about basically the core of the database engine or the storage engine, which is just this is a really good system for storing key value pairs. Then you just talked about some of the things that were written on top of that, written in addition to that core storage engine where you have things like being able to do aggregations easily or define abstraction of a list more easily at a low level so that the higher level lists can use those low level abstractions.

When we're talking at the other tunability aspects, because you mentioned some tunability aspects, are we mostly tuning the compaction policies, which are basically equivalent to tuning

like the latency of our reads and writes, or is there also tunable consistency or the reads always going to be consistent in RocksDB?

[00:34:29] IC: So RocksDB offers MVCC consistency by default.

[00:34:32] JM: Multi-version concurrency control.

[00:34:34] IC: Correct. That's correct. So what it offers you to do is to create a snapshot, for example, in some point in time, and then execute all reads you want from that snapshot, whether range scans or point lookups. Any new writes that happen will not obviously affect the particular snapshot, and that is true for also when you create an iterator. Iterator is an interface you use to do range scans.

Iterator is also executed on a consistent view of the data. That is done by basically every single write to RocksDB. It has a sequence number, increasing sequence number. Then when you create a snapshot, you just remember, "Okay, what is my sequence number?" When you do read, you just ignore any writes that happened that are past the sequence numbers you have.

Obviously, during compaction, you have to know which snapshots are live so you don't remove the – Even though a key might be old, there might be new update. If there's a snapshot within those two keys, you don't want to remove the old key. Otherwise, you will lose MVCC guarantees.

Then on top of MVCC, what RocksDB also provides is a layer on top, not in the core engine, is optimistic transaction utility and pessimistic transaction utilities. I think MyRocks projects, which is MySQL built on top of RocksDB storage engine uses those utilities to provide different consistency level for MySQL.

[00:35:57] DB: Yeah. Just to carry on that thought, so RocksDB is part of the MySQL ecosystem now, because people are going to use MyRocks. Basically, you use an SQL interface to a RocksDB storage engine inside it, but you can also use – There's also a project called MongoRocks, where you could MongoDB on a RocksDB backend. Earlier, Igor was one of the creators of that project.

[00:36:19] JM: Oh, wow!

[00:36:19] DB: MongoRocks. Then recently, you might also have heard about the Rocksandra Project, which means that Cassandra – You can run a distributed version of Cassandra software with RocksDB as the backend, and I think the Facebook guys – The engineering team has produced some benchmarks results on how better Rocksandra is compared to a standard storage engine.

Again, a lot of these RocksDB code is being powered by a great team at Facebook. There are probably five or six, seven, eight great developers. We work closely with them. We used to be part of that team earlier. Now we are not, but we still work closely with them. There're been a lot of contributions from the open source community. So the community is also pretty strong. There are a lot of contributions, and your pool requests get committed to RocksDB code quite quickly. It doesn't get stuck in no man's land for long periods of time.

[00:37:09] JM: Okay. Well, one more question on the core discussion around LSM storage engineers and RocksDB in particular, especially given that MyRocks represents a different storage engine from MySQL begs the question; what are the other storage engines for these kind – You talked about Mongo or MongoRocks or whatever it's called. Mongo, the typical storage engine I believe that it's using is WiredTiger. I don't know what the typical MySQL storage engine is, but it's not RocksDB.

So how did these different storage engines tradeoff from one another and what makes RocksDB a desirable choice for using as a storage engine that's underlying MySQL or MongoDB? By the way, for people listening who are a little confused, the underlying storage engine means this is just going to change a certain latency or perhaps consistency. But your interface into the database is going to be the same. You're still making the same kind of Mongo query, same kind of SQL queries.

[00:38:12] IC: So let me tell a story about MyRocks, which is a project out of Facebook. So Facebook used to use MySQL with InnoDB, which is a B-tree based storage engine, and what they found is on InnoDB, they had a lot of random writes. Let's say they configured a thing with

4-kilobyte pages and then they have 100-byte write. So this 100-byte write to 4-kilobyte pages means now for 100-byte write you have to write out 4 kilobytes. That is write amplification. So how much more data you have to write of think 40?

They saw that for B-tree based architectures, the write-read was much, much higher than for LSM. Then the second part was compression InnoDB was also not as good as compression RocksDB, and RocksDB has this unfair advantage of when you write the file, you never change it and you write it in one big shot. So you can do a lot of cool compression optimizations on a whole 64 megaytes, or 128 megabytes as you write out. So your compression would be much, much more tighter.

Those two aspects of it, less write amplifications, so less writing all to flash, less wearing out of the flash device and a better compression was the motivation for project MyRocks, and the results were, to me, very surprising where even very early on in the project, it was shown that I think MyRocks uses only 50% of the space of MySQL InnoDB, and then I think write amplification was shown to be 10 times better with MyRocks. Obviously, when the teams at Facebook saw those gains, those were incredible cost savings for the infrastructure side and then they invested a lot of resources into making that happen both on the RocksDB side, but even more on the actual MyRocks side in terms of combining the two and deploying the production. There're many, many very smart people working on that project.

[00:40:02] JM: Dhruba, anything you want to add?

[00:40:03] DB: I think my take from that MyRocks project that became quite successful at Facebook was that people wanted to use the feature and the performance of RocksDB, but they needed an SQL interface, because that's an interface that is quite useful for an application developer to write stuff in, sort of writing to a very custom API.

So the two things in my mind was they wanted to leverage the power of the performance features and they wanted to use SQL or SQL-ish kind of language to be able to operate on it. Then the third one obviously was that the MySQL database at Facebook is distributed in nature. So they wanted a distributed system to be able to store a lot of data. So these three things in my

mind is what that system kind of shows the power of the system. As a side note, that is kind of the same system that we are building here at Rockset.

At Rockset we have an SQL interface to a backend RocksDB databases. So you can run SQL on large amounts of data or just store it in RocksDB. Then also at Rockset we have a hosted service, which means that you don't really need to care about loading the library, compiling the library, installing it on your machine. It's installed for you. You can just run it using a known language that people are quite familiar with.

[SPONSOR MESSAGE]

[00:41:24] JM: OpenShift is a Kubernetes platform from Red Hat. OpenShift takes the Kubernetes container orchestration system and adds features that let you build software more quickly. OpenShift includes service discovery, CI/CD built-in monitoring and health management, and scalability. With OpenShift, you can avoid being locked into any of the particular large cloud providers. You can move your workloads easily between public and private cloud infrastructure as well as your own on-prim hardware.

OpenShift from Red Hat gives you Kubernetes without the complication. Security, log management, container networking, configuration management, you can focus on your application instead of complex Kubernetes issues.

OpenShift is open source technology built to enable everyone to launch their big ideas. Whether you're an engineer at a large enterprise, or a developer getting your startup off the ground, you can check out OpenShift from Red Hat by going to softwareengineeringdaily.com/redhat. That's softwareengineeringdaily.com/redhat.

I remember the earliest shows I did about Kubernetes and trying to understand its potential and what it was for, and I remember people saying that this is a platform for building platforms. So Kubernetes was not meant to be used from raw Kubernetes to have a platform as a service. It was meant as a lower level infrastructure piece to build platforms as a service on top of, which is why OpenShift came into manifestation.

So you could check it out by going to softwareengineeringdaily.com/redhat and find out about OpenShift.

[INTERVIEW CONTINUED]

[00:43:32] JM: Okay. I think we've given people quite a detailed introduction to RocksDB, and I'd like to now talk about running it in the cloud. So you guys are here at Rockset, and Rockset is building a data platform that we've talked about in a previous episode with Venkat. But when we talk more abstractly about this RocksDB, getting RocksDB running in the cloud whether we are building a MySQL database on tops of RocksDB or building Mongo on top of RocksDB or building whatever we want no top of RocksDB, there are things that we need to keep in mind. So when you take RocksDB and build a system around it in the cloud, what are the requirements?

[00:44:22] DB: So the first requirement was that if you store your data in RocksDB, the data gets stored on the storage system that you configured, it could be an SSD, it could be a disk device, but then if that machine dies or the machine goes bad and you cannot access the disk anymore, then you kind of lose your data. So what it means is that for any kind of hardware files, you might lose your data.

In the cloud, the constraints that we worked with are that you need the ability to spin up new machines when there's more work and you need to be able to spin down and give up all your machines when there's no work to be done. That's kind of an ideal hardware elastic system that you can build on the cloud.

So if you want to run RocksDB on that system, if you give up your machine, you'll lose all your data, because your SSDs and storage systems are gone. So we invented this project called RocksDB cloud. It's an open source project. Again, you can look it up in GitHub. It's called RocksDB Cloud. So what it does, it basically gives a durable layer to the RocksDB storage engine. So it has the same API as RocksDB, but then when you shut down your machine, your data is actually automatically backed up into cloud storage, which means it could be S3, it could be Azure storage, it could be Google Cloud Storage.

Then when you spin up a new machine at a different point in time, you can point him back to the cloud storage and say, “Hey, get me my database back.” Then your RocksDB database appears as if it’s a normal RocksDB database.

We have designed it in such a way that although the durability on cloud storage, the performance impact is negligible for reads just because or some configurations that we have. Then for the writes, those anyway happen asynchronously in the background. They automatically – SSD files automatically get flashed to the cloud storage, and that’s how you get durability for RocksDB. So this is the RocksDB cloud project. Did I miss anything in explaining the other features?

[00:46:23] JM: So as you’re pointing out here, RocksDB itself is a single node abstraction. This is not something that has fault tolerance built in. If you’re building a database, you want some fail over and there are different options for you can configure that fail over. RocksDB cloud presents a failover system, a durability system that is kind of conscious of cloud abstractions. It’s not like something where you have to take into account like two-phase commit, or three phase commit, or something. It’s like you’re taking advantage of cloud abstractions, like Amazon S3.

[00:47:04] IC: That’s correct. So one cool piece of – You can think of it as basically continuous backup. Every time your file gets created on disk, it’s also in the background put on to the cloud. Then if you have continuous backup that’s almost up-to-date obviously without in-memory, you can also spin up more machines in the same dataset.

So let’s say you have one RocksDB database that is getting writes and you’re getting continuous backup to S3 to the cloud storage, at the same time you can say, “Okay, I want my second, third replica pulled out from this, let’s say, check-pointed state, because I need to answer more queries. I need more throughput at that particular point of time,” and then the second machine gets up, pulls the data from the cloud and now it can execute queries basically on the copy of the check-pointed state.

The other alternative of failover and of this kind of architecture is replication, where you have two nodes, and then first node is a master. It kind of receives writes and then sends writes to the

secondary. But then what happens if your secondary goes away and it comes back up, there're a lot of things you have to kind catch up on. In our system, there's a checkpoint in S3. You just have to pull it all in. Basically, instead of tailing the log, like you do in a replication, you just get a state from S3.

[00:48:22] JM: So in the cloud, stuff fails all the time. Nodes die. When you're trying to get RocksDB to run in spite of these Byzantine cloud failures that can occur, you need to have failover strategies. Can you describe some of the common failures that can occur in the cloud and the recovery strategy for RocksDB cloud?

[00:48:44] DB: Sure, yeah. Absolutely. Some of them are failures, but more of them is what we call cloud native architecture. So what the cloud native architecture entails is that we want the ability for a backend system to use more resources when there is work to do and then give up resources when there's no work to be done. Do you see what I'm saying? So take for example you are running a system on four machines and now there are no more requests or no more writes, you might to shrink it to one machine because there's not much work to be done. This is what I mean by more than machines failing. It's more like giving up machines when you don't need to. You voluntarily give up machines. So this is not true on a non-cloud environment, because in a non-cloud environment where you buy hundred machines, your goal is to use those hundred machines as effectively as possible, right?

So RocksDB cloud is very cloud native in the sense like take for example if you build a distributed data processing system on RocksDB cloud, if there's more work to be done, then there'll be more instances of RocksDB cloud that you spin up because the data is on the cloud storage, and we have these features called zero copy clones that each of our nodes can make. Then when there's no work to be done, we can shut down that machine and give it up without losing all the data, because the data is still in the cloud, which means it's either in Azure or in S3 storage.

So this is what I mean by saying cloud native architecture. If you compare it with other database systems, like take for example if you compare MongoDB or Elasticsearch or even Hadoop. There, the strategy is more like replication. So if you have a piece of data replicated three ways so that you can handle hardware failures.

For RocksDB cloud and at Rockset, we don't do replication for giving you durability of data. We do replication only if there is a requirement to serve your queries for performance reasons. But if there is not much queries, then we store the data in S3 and there's no need for us to make three-way replications. So think about the cost effectiveness of the system. If you compare it with Hadoop or Elasticsearch or MongoDB, there are three replicas of live data. Whereas in RocksDB cloud and Rockset, we have only one replica if nobody is using that database. So we already like two or three times more cost-effective for that same workload. This is what we internally refer to as a cloud native architecture for data processing engines.

[00:51:04] JM: Okay. I'd like to get into Rockset in the remaining time, the company that you guys are building. Can you give me your perspective on the company how it got started, because I know you were both at Facebook and you're working on RocksDB and then something happened, things happened. You got traction with this project and you had ideas for products. Then next thing you know you're running a company. Can you describe what happened in between the point at which you started RocksDB and what made you want to start a company around it?

[00:51:39] IC: Yeah, and that's a good question. So I have been working on the RocksDB project for a while at Facebook before I left, and what I saw was that the RocksDB project was successful, because the project tried to leverage a new piece of hardware called flash devices.

So there were many few other data storage engines which was leveraging the power of the SSDs to give better performance for storage engines. But like maybe in the last two or three years, I see this trend where a lot of data processing software is moving to the cloud. In my mind, the cloud is a different hardware design, where if you build software that runs only in the cloud, you are probably 5 to 10X better and give better features to an application, because you paid only for the cloud.

So my focus at that time was saying that how can I build a software infrastructure for data processing on the cloud that is very disruptive to existing pieces of software, because it's not a port of existing software to the cloud. It's software that we build natively for the cloud, which is what I meant by the cloud native design, where hardware elasticity is one of the core

competence of this system and the ability that the cloud has so much different pieces of hardware.

At that time we talked that, “Hey, this is a great project to be done, because I think we can add a lot of value to existing people who use data,” because if they move the cloud Rockset storage engine or the Rockset processing engine, they’d be able to do stuff that they cannot really do now using other systems. So that was our motivation saying that, “Hey, we can build something that adds a lot of value to a lot of backend application,” because the amount of data is growing and there wasn’t a great way to process all these in the cloud at a cost effective way.

This is very similar to when Hadoop started, because I was also one of the founding engineers of the Hadoop file system, and that time the Hadoop file system was the only system which could store petabytes of data. There’s no other system which would store petabytes of data. Now at Rockset it feels like if we build this cloud native architecture, it will be one of the softer powerhouses which can allow a customer to do something that they cannot really do without our system. So that’s kind of the mission or the journey for our company.

[00:54:00] JM: So things like what? What would you like to enable that people can do today?

[00:54:04] DB: Yeah. Take for example if you’re trying to gather insights into your data, like take for example the thing that I keep playing in my mind is like how we can have a query which can process maybe 60 gigabytes of data in 60 milliseconds? No other systems can do this right now, but because we’ve built Rockset only on the cloud, we can spin up enough hardware. When the queries come in, we can provision you to be able to give that horsepower.

So when you have this horsepower, you can create a lot of new applications. Most of these are applications that need to find insights. Like take for example, I think Venkat mentioned this to you earlier, is that take for example you walk to a car dealership and they can show you the most relevant cars that you are interested in buying. That would be tremendous value to you as well as the dealership who is going to be using that software.

Right now they can't really do it because they have a lot of data, but it takes a lot of time for them to process it and you might walk in suddenly into a dealership, they don't know that you are going to be there on that particular day. Those are the kind of apps we want to enable.

Do you want add any examples of any other apps? Even like things like take for example your food delivery system. You might be using some of these popular food delivery systems. There are certain software that they use to figure out what is the best route for the bike rider to go and deliver food to your house.

So if you get your food much faster than what you – More efficiently or – much faster than what you're kindly getting, that needs to process a lot of data in real-time and give the recommendations right there and there. So it can impact you in many different ways. So this is –

[00:55:35] JM: What does that have to do with RocksDB though? That sounds like it just – I mean, if you wire together better in-memory systems and –

[00:55:42] DB: I think there are two different things. One is that how can you build cloud native systems? Let me give a simple example. It's a hypothetical example. Let's say there's a food delivery system that delivers food to your house. There's a road that the person is going to take. Now, you are not a person who is ordering your food regularly. So suddenly one customer comes in and says, "I need to order food. Give me to as quickly as possible."

So now the delivery company needs to be in real-time, find out what is the best route to take to your house. Is it going to be this road or is it going to be that road today? So you cannot pre-calculate these things. Do you see what I'm saying? Because there might be some construction going on, maybe the traffic light is broken over there. So we can get sensors.

So these things means that you might want to do a lot of work where there's a request, and then when there's no work to be done, you need to cut down your costs. This is what I meant by cloud native architecture.

[00:56:35] JM: Serverless.

[00:56:36] DB: Serverless and cloud native. Exactly, yeah.

[00:56:39] JM: [inaudible 00:56:39]

[00:56:40] DB: So none of the existing data processing systems do scale down well. They all do scale up, but for Rockset, we try to make sure that we can do both. So take for example this example that I gave you. If you want to do this on an on-prem system, let's say you need 60 machines 24x7, right? Again, your demand don't happen all throughout the day. Whereas if you do it in Rockset, it's possible when a period of high-demand, we use hundred machines, but when there's not much demand, we probably use only one machine. So that's how you save your cost and you make this cost effective to be able to do it. You cannot really do it on a non-serverless and on-prem system.

[00:57:18] IC: So cloud elasticity is what powers Rockset, but there's also a lot of stuff built on top of RocksDB cloud. So what we provide users is – And I think Venkat talked a little bit about that in the previous episode, is you can ingest any kind of data. That way we enable that is we built new SQL engine, distributed SQL engine that actually doesn't rely on the schema of the data. That was a challenging project. It took us a couple of months or even a year to get there, but basically we stored data in RocksDB, RocksDB cloud, we indexed it in many different ways using emergent index and columnar index. Then when the query comes, most of the traditional query optimizers and query execution engines actually have schema as a first primitive.

So if you say let's say foo equals a string, and foo is an integer, it says error. Foo is zero results. We cannot do such a thing. So we had to build a new query engine that doesn't rely on the schema. So if you say foo equals five, and foo is a string, it still tries to get — executes the query and only when it gets to RocksDB and says, "Hey, I need this particular document for which foo is equal to five." Then it says, "Okay. No, all my foo is stringed. There is no such document," or maybe one four is an integer into all other strings. So it finds that foo.

So we build that distributed query engine and what it allows people to do is just load it into Rockset in whatever format it's in. We can now adapt to the schema, and then the query engine doesn't care about the schema. So you can run any kind of SQL command and it runs even though there might be a new field, let's say there's a new document coming with the field that

the system hasn't seen before, a couple of milliseconds later you can already query for that field. There's no schema update command or like alter table — things in that.

So what that allows our customers to do is just extreme simplicity. There's no schema to configure. There's no indexes to build. As Dhruba said, there's no service to manage. Just give us the data and give us the queries and that's it.

[00:59:10] JM: Where do you feel like you're at today relative to where you'd like to be in terms of what you can enable for the customer? Are there any kinds of cost reductions or cloud abstractions that you're waiting for that will allow you to achieve what you want to, or do you feel like Rockset today has really done something groundbreaking that certain data customers can't get anywhere else?

[00:59:37] DB: Yeah. I can think about one or two things. One of them is that Rockset makes it very easy for a customer to get a lot of these evens, which are semi-structured. Like Igor said, some of them, sometimes the feel is an integer, sometimes the feel is a string.

[00:59:51] JM: Right. Ease of use and the automatic indexing is one thing that stood out when I was talking to Venkat is like just the process of ingesting all the data no matter how it's formatted and putting in in a way that's really easy to manipulate as a developer. Me thinking as a developer just like what's my appetite like, that's sounds great. The fact that I can work with that stuff so easily is basically cutting down on data claiming, and everybody hates data claiming. By the way, you can do that, because by taking advantage of the decrease in costs of the cloud.

[01:00:24] DB: Exactly. Yeah, we leverage the cost efficiency of the cloud, which is why we can do these things now. We also use RocksDB quite efficiently so that we can compete with other storage engines or we can be much more cost effective compared to other storage engines.

[01:00:37] IC: I think for Rockset, the roadmap is quite busy. Obviously, we are not even close to what our vision wants us to enable. So elasticity works well today, but we are not yet at the point where like we can execute huge, huge number of big queries on like petabytes of data and in short amount of time. So there are a lot of things on performance side that you want to work

on. Our query engine is still new even though it works pretty well so far, but there's so much more we can do there enabling more SQL functionality. So we are a fairly young company, I think two years now, two years and a couple of months, and databases take a while to build and to actually achieve like crazy good performance.

I think leveraging some of the instance we have about RocksDB and about cloud native and some cool things we built in the engine layer, the query engine layer, I think even today we can provide some interesting aspects to our customers, even though obviously there's a lot more work going on, a lot more ideas jumping around that we want to leverage to make it even faster to query data, even faster to index data and even cheaper for our customers by leveraging elasticity even more.

[01:01:45] JM: You guys launched at a Stealth, was it four months ago? Three or four months ago?

[01:01:49] DB: We launched November 1st, 2018.

[01:01:51] JM: How's the experience been since then?

[01:01:55] DB: Yeah, I think it's a very great experience. What has happened is that before the launch we were working with few select set of larger size users, but after the launch, we have a long, long list of different size users. Some of these are small, some of these are big. Some of them are heavyweight, some of them are different kind of workloads. So it's been really fun to look at all these different kind of workloads and see, "Hey, all of these could actually get value if they use Rockset."

[01:02:20] IC: Yeah, it's been fun. I mean, talking to customers, seeing wow in their faces when they see how, "Hey, I just load the data. How can you query it immediately?" "Hey, I didn't have to configure anything. How does this even work?" I think it's been tremendous and very, very fun journey so far.

[01:02:36] DB: Some of our users were kind of overwhelmed with the pipelines that they used around earlier. Some of our users are, “Oh! We have small set of data, just like we try Rockset.” There are all different kinds of –

[01:02:51] JM: The pipelines, so you just replaced their pipeline was just like ingest it?

[01:02:55] DB: It is not a question of replacing all the pipelines, but a major chunk of the pipelines we can reduce, because we index all the fields in your even database or in your dataset. Some customers – Actually, I remember talking to one user who says that, “Hey, I’m actually impressed that I don’t have to use this command called create index database anymore. That command is gone for me.” I said, “Yeah, yeah. That’s good.

[01:03:19] JM: That is great. Yeah, that seems to be like one of the big trends that you guys have taken advantage of, is the fact that if you just index everything, take advantage of the lower cost of the cloud, you can make databases a lot easier to work with.

[01:03:30] DB: Yes.

[01:03:30] IC: Yes.

[01:03:31] JM: Cool. Well, you guys, this has been an awesome show. I really enjoyed talking to you about the depths of RocksDB and the exciting applications that you’re building on top of it with Rockset.

[01:03:42] IC: Thank you for hosting us.

[01:03:43] JM: Okay. Thank you very much.

[END OF INTERVIEW]

[01:03:47] JM: DigitalOcean is a reliable, easy to use cloud provider. I’ve used DigitalOcean for years whenever I want to get an application off the ground quickly, and I’ve always loved the focus on user experience, the great documentation and the simple user interface. More and

more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.
[END]

