EPISODE 897

[INTRODUCTION]

[00:00:00] JM: A time series database is optimized for the storage of high volumes of sequential data across time. Time series databases are often organized as columnar data stores that can write large amounts of data quickly. These systems can sometimes tolerate data loss, because the data are being gathered very quickly and that data is usually used for monitoring and other applications that require aggregated datasets rather than high-fidelity, highly important individual transactions.

The demand for time series databases has grown over the last decade with the rise of mobile devices and the decreasing cost of cloud storage. There's been an increase in the number of systems that require monitoring and some of those systems produce an incredibly large amount of data requiring compression, downsampling and garbage collection.

Rob Skillington is an engineer at Uber where he helps create M3DB, a time series database. In a previous show, Rob described the basics of M3DB and how it helps Uber with storing large volumes of data from Prometheus, which is a monitoring system. In today's show, we discuss the field of time series databases and Rob's approach to building M3.

[SPONSOR MESSAGE]

[00:01:27] JM: Apache Kafka has changed the world of data infrastructure, and Kafka Summit is the place to learn about new design patterns and engineering practices in the world of Kafka. Kafka Summit returns to San Francisco, September 30th through October 1st, 2019. Kafka Summit has sold out in New York and London, and the San Francisco event is likely to be just as popular.

Listeners of Software Engineering Daily can get 25% off their ticket to Kafka Summit by entering promo code SED. With the promo code, Kafka Summit is only about $900 to attend, and if that's still too expensive, you can consider asking your company or your manager to pay for your ticket.

Kafka Summit is an educational experience with top engineers from places like Netflix, Microsoft, Lyft and Tesla. At Kafka Summit, you can meet with experts who will help you address your toughest Apache Kafka and event streaming questions, or you can start to learn the basics of how to deploy and operate Apache Kafka. There are also hands-on beginner and advanced training courses available, as well as certification.

Join the Kafka Summit, September 30th through October 1st, 2019 and get 25% off your ticket by using promo code SED. I plan on attending Kafka Summit, and I hope to see you there.

[INTERVIEW]

[00:03:06] JM: Rob Skillington, welcome back to Software Engineering Daily.

[00:03:09] RS: Thank you, Jeff. It's great to be here again. Thanks for having me.

[00:03:11] JM: Absolutely. Your last episode was wonderful, and listeners can check that out. It's about Uber and infrastructure there and specifically M3DB and scaling Prometheus. Today we're going to talk about time series databases more specifically. Time series databases are specifically built for time series. Why do we need domain-specific databases for time series data?

[00:03:37] RS: Yeah, that's a great question. So time series databases, it's interesting. They're not actually extremely well-defined. Most definitions of a time series database will describe them as a key-value store like database that allows for appending values associated with a given timestamp, and that they are intended for efficient retrieval of values for a key over a given range of time.

So, that's the primary concept of why they even exist. So you can actually represent that data model with plenty of other databases. They are mainly used for relatively high-volume or other types of special use cases where a traditional database doesn't do a very good job at either cost-wise or performance-wise for storing and retrieving values in an on OLAP kind of like style workload.

[00:04:34] JM: Could you give a few examples of those types of high-volume data applications?

[00:04:39] RS: Most definitely. KDB, which is one of the oldest and fastest time series databases around was released in 2003 for financial institutions primarily. That is definitely a powerhouse and used by the financial institutions for storing a lot of market data very quickly and being able to analyze at a very high-resolution different parts of the market and the data they're collecting about the market in real-time. But also, which is obviously used for high-frequency trading and other types of use cases.

So that's kind of historically where a lot of like the oldest and probably most production grade time series database started. However, monitoring is another extremely high-volume use case which collects a lot of data around specific types of things that you're tracking, again, overtime. A lot of those values are changing in real-time either at very high-resolution. So down to sub-second or more granular resolution for higher level insights at, say, by 10, 15, 30, 60-second resolution.

There're plenty of others as well in the internet of things. There're a significant amount of different types of key attributes that people like to monitor. So for instance things like battery level or energy efficiency, for instance, like on your console. There's probably some, say, like the console that you're using to play games or watch movies at home. Some of the data about the energy efficiency maybe shipped back to the manufacturer and then they can aggregate, basically analyze if due to different software updates or different firmware or just different like types of hardware and how that's performing overtime essentially.

So there's a whole bunch of these types of use cases, but those are some of the more popular ones.

[00:06:39] JM: When we think about an application like Uber, there is a high-volume of transactions that are going on across Uber, and those transactions, maybe they're handled by MongoDB or they're handled by a relational database, like Postgres. These are transactions like a user summoning a ride or a user paying for something or a user ending their ride. For each of these things, there is a network hop. There's a write to a database. What are the parts of the relational database infrastructure, the NoSQL infrastructure, that fall over when we go from talking about these transactional use cases to these use cases where we're gathering higher volumes of data?

[00:07:27] RS: Yeah, great question. So, basically, time series databases can be built on things that we like to call row-oriented databases or they can be built on column-based databases. Column-based databases support basically higher compression on the values and they also locate the values next to a specific key right next to each other both in-memory and on disk.

For row-oriented database, and sometimes series databases built on row-oriented databases, like TimescaleDB, which is built on Postgres. Other time series stores actually don't even implement the storage layer themselves and they use like Cassandra, DynamoDB or other similar kind of offerings as backends, and those are all row-oriented. Now, the difference there is it's basically – The querying for them is essentially scan-based in a row storage solution.

Then for column-based, once you find the key, it's a matter of basically jumping to the range in the column that's stored next to that key and selecting the values. But once you find the start of that range, it's very quickly to grab the consequent values rather than traversing the rows big row – In a row-oriented engine.

So that's one reason why on the query side why essentially you may end up seeing much better performance from column-based time series databases and just time series databases in general. Then on the value, like writing these values into the database, which is kind of I think more what you're asking about. Yeah, traditional OLTP-like database such as like MySQL or Postgres, which is, as you mentioned, usually offers transactions for online processing.

They tend to essentially offer a whole lot of features that provide high-level of consistency. They offer a bunch of different indexing techniques. They're not very opinionated about any one thing. Although I'm sure, respects they are to the feature set. But they're more around offering a very strict and reliable set of features around some of the data that becomes your source of truth for your business.

So they're not going to optimize for high-bulk inserts. They're not going to optimize for inserting streams of values. Consequently, a lot of the time they just can't ride as many samples per second or values per second for time series-oriented workloads. That at a high-level, at high-scale, can end up meaning the difference – The cost difference can end up being very, very high.

[00:10:17] JM: You're alluding to differences between column-oriented databases and row-oriented databases. We've talked about this on some previous episodes. You're also alluding to the fact that perhaps the data is more sensitive in the context of a relational or a transactional data store, the kinds of things that you're using, Postgres, or you're using MongoDB for. You really can't afford data loss. In the case of a user on Uber, for example, if they're summoning a ride or if they're ending a ride, that's a really important database transaction that has to occur, whereas in a time series database, you may be able to afford some data loss. You may not need every time data point. Are there any other tradeoffs or notable considerations we can make as we are moving to the domain of time series from the transactional workload?

[00:11:16] RS: Most definitely. Actually, data loss is something that you ideally never put up with a database that you're using. Having said that, obviously, there are features you do want to tradeoff. Honestly, any database that you use that has a single replica is likely to possibly have data loss if you're not doing synchronous replication. But getting more back to the point of what other things we can tradeoff of is primarily it comes down to things like consistency.

So, when you support transactions and the ability to make sure that several things must occur and you want to atomically update maybe multiple rows, or at least like some values within a single row, that means that you need to basically have some version of MVCC, just multi-version concurrency control.

So may want to use a database that – Databases like MySQL and Postgres and other row-oriented transactional databases offer multi-version concurrency control, which means that you can get a single view of the data even though many other transactions or edits of mutations may be happening at the same time, but you are guaranteed at least that the data that you're reading and writing within a single transactional can be looking at a single set of data that is guaranteed to not change between the transaction starting and the transaction ending, or at least not meaningfully in the way for your transaction that you are running.

So, a lot of those really powerful features really are needed for certain different types of workloads, and time series database definitely, a lot of the time, will tradeoff consistency, because consistency, especially with replication involved, either involves a single write master or it involves essentially some consensus protocol, like Paxos or Raft or something like that to make sure that all replicas arrive at the same agreed upon value for that write. So, that's a huge one.

Then there's other things honestly that for even more high-volume use cases where it's acceptable for data to show up. So, a little bit later, after you write it. So if you don't need the ability to read your own writes immediately after a write is complete, then you can put up with the database asynchronously making that value accessible to your query layer. That means there's a ton of different batching that it can do. So it can essentially decide that I'm going to put a whole bunch of values together and then make them appear to queries all the same time. But a side effect of that is essentially there is a small delay between an insert finishing and an insert showing up, and the ability to be able to do that is pretty valuable, because it really does unlock a next level of high-insertion volume.

When it's combined with tactics to write that data durably to the journal, before, it batches that together with other writes. It means that it can be durable as well. So you can reply to the client knowing that you've basically written that as part of a batch to the disk, but haven't indexed it yet and haven't made it show up to the query layer just yet.

[00:14:45] JM: So there're a number of consistency loosenesses that we have available to us. We may not have to have consistent write properties. The writes in the database may not have to necessarily appear in order. The reads to that database may not necessarily have to appear in order. Because oftentimes these databases are read as aggregates, and you think about the end user, the end user is probably consuming a dashboard. They're looking at a graph. If there's a data point missing or a data point that's slightly out of order, it's probably not a big deal, because you're going to have the graph doing interpolation and it's probably zoomed out sufficiently that it just gets smoothed out.

So that said, I'd like to go a little bit deeper on the ingest path for this data, because if we think about the kind of asymptotic ideal of how much data an application like Uber would be consuming. If you're riding around in a car with somebody that's essentially a stranger, ideally, this is a situation where pretty much you want to have like constant surveillance. You want your phone to be communicating with the backend servers as aggressively as possible. In an ideal world, you would have down to the smallest sliver of time. You would have your phone heartbeating to the backend servers and having this data written to Uber, to Uber's backend. Obviously that's not possible.

So there's some batching that's taking place on your mobile client. That data is being sent to Uber's backend and then it's being downsampled as we discussed in that previous episode. Then that downsampled data might be written to a database like M3DB, or maybe there's some intermediary, like a Kafka system in the loop.

Before we get into the internals of time series database architecture, let's talk a little bit more contextually about the ingest path. When we talk about data that's often coming from a user's mobile, it's being ingested by some kind of backend set of systems and then eventually it's making its way to the time series database. Put this time series database architecture in context.

[00:17:05] RS: Yeah. So, that's a great question. What we find with the location data that comes from a trip, yeah, we do collect it up to basically two second resolution from both applications that are running if we can during the length of a trip. What we end up finding is that this generates a lot of data and also the latency to which it gets into that database is essentially can be dependent on what type of application you're building on top of that.

For Uber, which has features like share your ride and other types of location-based features that basically want to show up to the second where your trip is or where a driver nearby is. Writing those values with low latency and making them available to the query layer at low latencies is relatively important. So a lot of the time Uber tries to avoid actually putting really too much of a buffer in between the data point arriving from the mobile phone in the data center and it being inserted into a time series database.

So it's funny that we're talking about this, because M3DB is basically taking over that and storage of this data, location-based data for these types of trips. What we've noticed is, yeah, that basically because we have – We basically have in-built buffering and batching support into the client itself, which helps improve the latency – Well, traditionally when you increase batch size, it actually decreases the ability to give you low-latency because you have to wait for a few values to arrive.

So we have this interesting mechanism whereas data points from mobiles phones are showing up to application service that write the to the database, we wait at least a minimum of five milliseconds before we even start sending the data to M3DB. This helps essentially for the high-write volume case, you always are guaranteed to get pretty full batches of data. But for the low-volume case, it does hurt the P99s, because you're stuck with a 5-millisecond latency buffer essentially on the minimum end rather than being able to go like sub-millisecond if you turn that feature off.

So, yeah, there're all these sorts of interesting things that happen there. But I will say that a data can also be written to Kafka in parallel. So if you want to have the ability to replay some of that data, you can then also basically start consuming off of the Kafka feed to replay that data into the database, and that's useful, say, like for whatever reason. You have a problem between your application layer talking to the database. So Kafka can act as basically a nice intermediary buffer of data that should always be available hopefully regardless of whether your connectivity to the database is up or not.

[SPONSOR MESSAGE]

[00:20:09] JM: Buildkite is a CI/CD platform for running scalable and secure continuous integration pipelines. Buildkite helps you keep your builds fast and reliable even as they grow large. Builtkite's web UI and APIs are fully-managed, well-documented and backed by great support and SLAs. Teams can easily set up and maintain their own build pipelines and get help directly from Buildkite support.

Build configurations are checked into source control and it works with GitHub and GitHub enterprise, GitLab and Slack workflows. There's also support for webhooks, GraphQL and plugins, letting you extend Buildkite in new ways. The Buildkite agent is open source, written in Go and you run it within your infrastructure. It's under your control, so you can be sure that the source code and the secrets don't leave your infrastructure.

There's an AWS cloud formation stack to get your started and it auto-scales from zero to hundreds of agents, or you can deploy it to a Kubernetes cluster, a cloud provider, bare metal hardware or a cluster of MacOS machines.

Visit buildkite.com/sedaily to learn more and see how Shopify used Buildkite as they scaled from 300 to 1,200 engineers. They migrated between cloud providers and they kept their build times under 5 minutes. Check it out at buildkite.com/sedaily.

Thanks to Buildkite for being a new sponsor of Software Engineering Daily, and it's always nice to see new CI/CD platforms such as Buildkite.

[INTERVIEW CONTINUED]

[00:22:06] JM: Let's start to get into the architecture of a time series database in more detail. Now, first of all, you created your own time series database, and when you started M3DB, there were other time series databases. We talked about this a little bit last time, and I think that the motivation for building your own time series database was the fact that the other time series databases were not scalable enough for Uber, and that's kind of a curious idea, because – I mean, obviously, not every database can or should scale to support a company like Uber.

We've done shows on Facebook recently, and Facebook builds its own infrastructure, like cloud infrastructure simply doesn't work for Facebook. It's too big of a scale. Can you give some larger context for why these hyper-scale companies sometimes cannot be compatible with off-the-shelf infrastructure like the time series databases that were available at the beginning of you starting to work on M3DB?

[00:23:19] RS: Yeah, most definitely. I think you'll find that most people, when I tell them that we ended up writing a time series database kind of take a very strange, "Look at you," the second you mention that to them. Rightly so, I think that it's not a small fete. It's a super large investment, and ultimately, especially for people that haven't written a database before, it's something that you're doing that you clearly have not done a Ph.D. in or you don't have research and education behind it. So it's always a questionable thing to do.

Having said that, I think that you'll find that some of the time series databases that have been built, and specifically at Facebook they built Gorilla, which was a purely in-memory time series database. Netflix, they built Atlas, which again was pretty much a purely in-memory time series database. A lot of these are custom-built for the application that they're supporting, not necessarily basically building a database to be a generic database to be able to be used by almost any application.

So, that is primarily I think a reason why some of these companies you find will write their own time series database or come out with a similar custom solution that is not available, like cloud vendors, or even vendors in the time series database. Well, I definitely have mentioned a few before, like Kdb, say, enterprise time series database. So is InfluxDB. Unless you are using a single node in InfluxDB, that is an enterprise database. TimescaleDB, QuasarDB are also enterprise databases.

So, essentially, there is definitely a lot of reasons why you might do that, but they tend to end up being purely around cost efficiency and scale. Most companies can use a lot of out-of-the-box offerings out there today without having to build their own, because they're probably not quite yet at that relationship where they're trying to save on the infrastructure side of things. But I will say that it really does come down to your business model as well. So, if your business model is very high-margin, then you can get away with using very expensive infrastructure. Probably, the more lower margin you are, the more lane you need to be.

But what ends up happening is that monitoring data, especially, tends to be a function of how many engineers you employ. Not necessarily how big your systems are or how much money you make or anything like that. So, it does tend to be that at the bigger engineering organizations, you have a much higher volume of monitoring data. A lot of the time it's not just being used for purely monitoring. It's for research and development, obviously as well. So it's more like application insights, and a lot of people will even use it for tracking how an experiment is performing in their application or how certain parts of the business are actually operating in real-time due to how their application is performing.

So I will say that, essentially, a lot of those factors combine to the point where people say, "Okay, I know I shouldn't write a database, but given how much data I'm collecting and the margins of my business and how many engineers I have, it will make a lot of sense and enable us to do either more things or the same amount of things much more achievelly."

You'll actually notice here that Facebook and Netflix and I think even Monarch at the beginning, I'm not sure, what it is like now. But most of their time series databases are purely in-memory, which is a lot easier to get started. So that can also help out as well, because it means essentially you don't have to be an extremely well-versed database engineer to implement an in-memory database a lot of the time. So I think that's why you'll see as well that a lot of them start out as in-memory time series databases at these companies and then either stay that way and move the rest of the data to HTFS or some other long-term store or they start to actually make that time series database both in-memory and disk-based.

[00:27:51] JM: Could you speak broadly about the tradeoff between storing something in-memory versus storing something on disk in a database architecture? I know this is kind of a computer science fundamental sort of thing, but maybe you could just explain it and then put it in the context of a database architecture. Because I feel like a lot of times in the conversations I have with different database companies, the tradeoff between in-memory and disk and the sort of flush to disk model, the batching of in-memory to disk can be really important and it can be kind of hard to understand.

[00:28:31] RS: Yeah, most definitely. I think it's an extremely interesting time, honestly, for databases. I think that there is a lot of hardware that is basically becoming cheaper and cheaper, that has more memory available. So what traditionally seemed like a really crazy thing, like keeping hours of data in-memory before writing that as one block to disk seems possible as long as you have a durable journal or commit log of the data that you're saving. Cassandra is a prime example of this with its – Basically, it's journal and then SS table-like structure that will flush a whole bunch of values together.

So there's two ways to look at this. On the write path, it's all about optimizing how frequently you combine files together on disk or essentially how much overhead do you pay for organizing the data so that it can be queried efficiently. Basically, the more memory you use, the less you have to write it to disk and then read it back from disk and write it back to disk again combined with another dataset.

However, the more memory you use, then obviously the more dangerous it becomes operating your database, because once you pretty that memory bubble, you have to reboot up the whole database and then basically find some way to take all that data that was in the journal and then actually write it out as an SS table or some kind of optimized for query structure on the disk again after crashing. So on the write path it's kind of this tradeoff of using more memory will mean less writes to disk and then as a data structure that can be queried efficiently. Again, come with some dangers.

Then on the read side, this is where it's obviously very interesting. There're tons of solutions out there like Apache Spark. Presto is kind of getting into this area where they're not even a database. They're just purely in-memory query execution engines that can scale horizontally. So they can take a whole bunch of data out of other databases and just perform really tight amounts of aggregate queries and really complex joins because all the data is in-memory and the access is very quick.

Now, you see some databases like MemSQL is kind of doing the same thing, but they're obviously like the storage layer. Using more memory basically for the read and the query patterns just essentially performs much more interesting queries at much higher efficiency in terms of performance of speed and things that are kind of access from disks and things like that.

So I think the write path is very – It's almost like a more simple case of like, "Well, it's really just – It can help avoid compactions and stuff like that depending on how much memory you want to set aside." Then on the query side, it's really just about, "Well, how interesting and how many crazy things do you want to throw against this database with your queries?"

[00:31:33] JM: Let's talk about the instantiation of an M3DB. What happens when I stand up an instance of M3DB? What do I need to do and what is that spin up process look like?

[00:31:47] RS: Yeah, great question. This obviously varies quite significantly with different databases. I think the longer that a database has been around or the more simpler it is in architecture, the easier it is to stand up and start using essentially. That's primarily due to the fact that it's either simpler, just nature, or it's had a whole bunch of engineering work to make it look like it's simpler in nature to operate, which I think is very important. Software today can be used in a relatively straightforward manner by the masses, that is. I mean, you'll find some shops that spend countless hours setting up a very complicated architecture, because it makes sense based on what they're trying to do.

So M3 and M3DB, it's a little bit of a compromise here, because it isn't the – Well, simplest database. It is similar to Cassandra and other distributed databases. It does want to allow you to add or remove machines and handle all the data retransmission between the nodes by itself. So, for instance, even really high scalable databases like ClickHouse won't do this for you. You can basically send data to different nodes and it has a way to fetch all the data from all the machines but it won't actually retransmit data when you add or remove machines to the cluster.

So, within M3 and M3DB, what it looks like when you set it up is you either use etcd either embedded in the M3DB nodes itself. If you do that, you have the concept of seed nodes. Seed nodes are essentially similar to Cassandra, these database nodes that all other nodes know the address of, and so they can discover the rest of the cluster by talking to these seed nodes.

So, what that looks like is a little bit different if you set up a dedicated etcd cluster, which is essentially just like another set of database nodes purely for storing M3DB's membership and topology information. So, etcd and M3DB is used for essentially making sure which database machines own which parts of the partition range and then managing the state transitions when machines are added or removed from that set of distributed database nodes.

So, it's very important that that data is highly consistent, because if it's not, then database nodes can disagree about the partition range that they own and you'll get a very different answer for what your data looks like depending on which database node you query if they ever start to not be aligned. So I'm sure you'll find different Cassandra users that have come across this problem where some of the Cassandra nodes in the gossip ring. The way that Cassandra shares topology information is via peer-to-peer protocol exchanging membership information with other machines in the distributed database cluster.

Soon, operators of Cassandra have found that, essentially, sometimes it tends to not be essentially unaligned and can sometimes fall into a state where it never recovers. So then issuing a query to different nodes may give you back different results, because they thought that they owned a different set of the partition.

So that is probably the biggest thing with M3DB. It's basically setting up an etcd cluster either whether that's embedded or dedicated. Then once you have that going, then you kind of just basically add machines as you start to need to scale up or scale down your instances. We do have a Kubernetes operator that we use pretty frequently.

That is actually helpful, because it kind of manages at least performing cluster operations based on the desired amount of instances that you would like in your cluster. Then you can declaratively edit the definition of your cluster with respect to number of nodes and the availability zones that it should be split between. Then the Kubernetes operator basically makes calls to the database APIs for you essentially to basically add or remove machines as required.

[00:36:09] JM: Once I have M3DB set up, I can issue writes to it. I can make reads from it. Talk about the reads and write path for M3DB specifically.

[00:36:22] RS: Yeah, most definitely. This is one thing that I love to talk about, because it's kind of an evolving thing. So, most time series databases don't have what we call a full text reverse index on top of their time series metadata. So Prometheus and M3DB both have this, and I believe that InfluxDB has this as well. Basically what it allows you to do is to do multi-dimensional lookups on your time series metadata. For instance, for metrics, and Prometheus metric's means like labels on your metrics.

Then, essentially, once you specify multiple of these filters, it will find all the matching dimension values individually and then essentially find all the underlying time series and match those dimension or label values and intersect all of those sets together to give you one list of time series that match all of your filters. Then you can perform aggregate functions on that and other interesting things.

Basically, most time series databases don't really support this type of multidimensional full text search. So with M3DB, once you start writing in metrics, or you can actually write protobuff messages to M3DB now. But regardless, once you start writing those time series in and you associate them with different tag values as we like to call them, which are essentially just dimensions and dimension values, you can then on the query side basically look up. You specify a list of filters that will basically be unions together to compromise the set of time series that are returned to you.

Of course, if you use a query language like PromQL on top of that, which you can if you use an M3 query or an M3 coordinator service that sits in front of M3DB, that basically allows you to perform functions on top of these time series that come back. So that's really useful if you want to do summing or perform histogram calculations based on buckets of data that you wrote to your database in a histogram fashion and other kind of complex time series statistical functions on top of that data.

So that's kind of what it looks like, and I think one thing that's kind of new is kind of the combination of Apache Lucene and Elasticsearch type features, which are basically providing full text search on multi-dimensions being added to databases natively right now, which I think is definitely a pattern that we've only seen to start happening probably within the last few years.

[SPONSOR MESSAGE]

[00:39:19] JM: Cruise is a San Francisco based company building a fully electric, self-driving car service. Building self-driving cars is complex involving problems up and down the stack. From hardware to software, from navigation to computer vision. We are at the beginning of the self-driving car industry and Cruise is a leading company in the space.

Join the team at Cruise by going to getcruise.com/careers. That's G-E-T-C-R-U-I-S-E.com/careers. Cruise is a place where you can build on your existing skills while developing new skills and experiences that are pioneering the future of industry. There are opportunities for backend engineers, frontend developers, machine learning programmers and many more positions.

At Cruise you will be surrounded by talented, driven engineers all while helping make cities safer and cleaner. Apply to work at Cruise by going to getcruise.com/careers. That's getcruise.com/careers.

Thank you to Cruise for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:40:43] JM: Can you talk about how compression fits into the architecture of performing write to M3DB? Do you compress data as it comes in?

[00:40:55] RS: Most definitely. Yeah, different databases have very different approaches to this. Some rely on the file system to do all of the compression for them or they use custom compression for each type of fields that they're storing. For instance, a lot of monitoring solutions as like to use a variant of a Flight 64 data compression algorithm called Ts.ED. Ts.ED is a Flight 64 compression algorithm first published in a VLDB paper about Facebook's in-memory time series database, Guerilla, which we kind of mentioned before.

If you Google Guerilla Ts.ED PDF, you'll find the paper I'm talking about. Essentially, this algorithm is heavily based on delta of delta on the values themselves that are written to the database and it's also very good at doing delta of delta on the time stamps. So, essentially, if you're writing in at the same frequency for given time series, the delta of delta on the time stamp, basically, it becomes zero because you're writing in at the exact same time and you can encode basically with a single zero bit the fact that you have a new value and ensured up the exact same distance as the last one relative to the last one's distance with its last data point. So that's quite important on achieving very high compression rates.

Why you might want to do that is that you basically can store much less in-memory and much less on disk if you don't support any type of compression on all your data. So, this is something that's highly desirable especially in larger dataset, because it essentially means that you basically just pay a lot less, but also you use a lot less memory. So that means your database engine in general is doing a lot less work, because as we've mentioned before with compactions, it means you can buffer a lot more data in-memory before you need to flush out a representation of that data to disk again as long as you're persisting that to disk in a journal for temporarily keeping that data around for durability reasons.

So some time series databases like IRONdb from Circonus and TimescaleDB relying on ZedFS, which is for compression, and ZedFS is a file system. I those come out to – I've heard numbers quoted in like the 3 bytes per data point range, while others like Prometheus, M3DB and InfluxDB use Ts.ED or variant of their own Ts.ED algorithm. For instance, M3DB uses a variant called M3 Ts.ED, which is actually not 100% lossless. It trades off accuracy after a configurable amount of number of decimal values are used.

So for monitoring workloads, this is acceptable. You may not care about after 8 or 10 decimal values how accurate the precision of the Flight 64 data you're riding in comes back. So, for protobuff storage, this extension is not enabled by default in M3DB because you probably want to read what you wrote. For monitoring workloads, that means basically an extra 20% to 40% of compression we can eke out on top of the default Ts.ED algorithm, because once you basically make a Flight 64 piece of data look like an int, which is just being offset by the number of decimal values that you're storing, you can use much less bits to represent the delta of delta on that value.

So, that's kind of how the different databases support compression. For reference, a default data point in the metric universe basically takes 16 bytes to represent in raw format. So that's 8 bytes for the timestamp as a UN64. Then 8 bytes for a float 64 data point value. So that together it basically means that you're setting aside 16 bytes for every data point.

I mentioned that IRONdb and TimescaleDB use ZedFS, which has been said to roughly come out to around 3. something data points per value. That's a pretty significant game. Then M3DB and other databases, which is Ts.ED and Ts.ED variants is able to improve on that down to like

1.2 to 1.4 bytes per sample, which ends up equaling almost like roughly 11x compression on your data.

Again, this just allows for much more dense machines and dense database nodes. Because of compaction can mean that it can use less memory and essentially compact at a lot lower frequency than a traditional database that isn't using compression.

[00:45:44] JM: Beautiful. One prominent feature of M3DB that you've talked about is its ability to store high-cardinality data. Can you explain what that means in more detail and how that impacts the architecture of the database?

[00:45:59] RS: Yeah, great question. So I've found that a lot of people have very different opinions on what is high-cardinality data. For instance –

[00:46:08] JM: By the way, it might be worth defining that term cardinality.

[00:46:11] RS: Yep, that's a great point. So, when we talk about cardinality, we may mean a few different things, but really what it comes down to at least in the monitoring space is talking about the unique number of entities that you are either tracking or the granularity to which you are bucketing things together to look at. So you can imagine like the highest cardinality in some sense is just looking at a stream of raw events, because if you were to group all like events together and aggregate them even if you just want to sum, say, one of their fields together.

If you're using raw events, and raw events has something like the user I.D. as one of the fields in them. When you try to group them together, they all look like distinct buckets, which means that essentially the cardinality which is, again, like the number of unique views on the data ends up being equal to the same number of raw events that you put into your collection. So that's the most highest cardinality you can think of.

However, when you kind of like just collect raw events, it means that you basically end up storing way more data than you ever would if you actually start to collect and group some of these events together. So, that's why at a certain level of scale, it makes sense to use something that's not just storing raw events for you, especially if you want to do quick and relatively fast queries on this data either for monitoring or maybe you're doing it for routing. Perhaps you're building some analytics for some of the users of an application that you're building. There're many different reasons why you might want a faster view over a set of data rather than just constantly scanning or filtering on raw events.

Basically, what that cardinality comes back to is like how many dimensions are you going to keep on the buckets that you are going to be aggregating on when you look at the data that you're collecting from a set of events. For metrics, this essentially is always associated with, A, how many unique values you have in a given label, and then also the combination of those unique values in a label multiplied by every other label that's used on that metric.

For instance, if I have a metric that basically has the name of the host in it as well as the name of a book title. For instance, if I'm selling books, I guess. Then, essentially, the cardinality of that data will be the number of hosts that is unique that will process most of these queries for your books multiplied by the number of books that you have. So if you have 10,000 servers and you have 10 books you're selling, then that's a cardinality of a 100,000.

So this is why a lot of time you essentially don't tend to see things like customer I.D. or request I.D. put in metrics, because metrics are meant to be more volume metric views of your data, not just raw events, and that again is helpful for some of the reasons I mentioned before.

With M3DB, the support for high-cardinality came from the fact that even when taught and educated engineers as best as possible to leave data that they weren't going to slice and dice by in their queries, even in that time, there was still such a combinatorial nature of the data they were storing because of just regular dimensions that we had at the organization, which is things like the city. Things like the hexagon that your trip started in. Things like the iOS app version you're using. Things like the Android app version you're using and all sorts of other things that it made a whole lot of sense to keep the dimensionality for basically looking at this data.

You'll notice that in tracing systems and other systems that basically also like to store high-cardinality data, what tends to be – One of the approaches to combat this is you bucket it with the smallest amount of buckets as I mentioned to kind of reduce the cardinality and then you keep basically a sample of like the request ID or a customer ID next to one of these buckets of each type so that you can then go and investigate like a specific instance of a drop of a data that fell into one of these buckets. I can kind of explain a little bit more how that's used with exemplars and what that looks like in tracing, but you kind of get the general concept with that explanation.

[00:51:01] JM: Yeah, I think we should zoom out at this point and put things in a little bit more context. There're probably people listening who are curious about time series databases right now. They're evaluating whether they should use a time series database for their system. You've kind of answered who should be using a time series database in an earlier response.

My sense is that there are people who should be using M3DB. There are other people who perhaps should be using a different time series database for their application. My sense is that this is a growing field and there are differences in what people should or should not be using and what they're going to be optimizing for in these different systems. Can you give some selection criteria for the listeners who are thinking about which time series database to choose for their application?

[00:51:50] RS: Yeah, great question. I mean, I think we've been talking about mostly open source time series databases. So that's really where a lot of my advice is going to fall into. I will say at least for monitoring, that you really can't go wrong starting off with Prometheus. Kind of like Postgres and MySQL, it is essentially a single machine that Prometheus is isolated to. It's unaware of its other machines.

You can make it aware of other machines and also archive some of the data that it's storing using an open source project like Thanos. Thanos essentially clusters the Prometheus instances so that they are aware of each other and then they can query each other to fulfill data that may span the different instances and it backs up some of the data to S3 for a kind of like a colder storage-like experience when you want to access more historical data.

So, I think starting with Prometheus makes a lot of sense. Basically, whether you want to store – Basically store data that is going to exceed a single Prometheus instance or you want to basically access data that's being collected in multiple regions and multiple availability zones and have them isolated to each other but still provide access to a single query layer on top of all of them. That's when you want to use something a little bit more powerful.

So, Thanks is one solution I mentioned there. Again, it's kind of more like clustering Prometheus to be aware of each other and use S3 as kind of like a backup. Then you can actually query, but with obviously elevated response times that might come with that solution.

Then M3DB is more – Basically, if you want to essentially store this data and you know that the data will either increase overtime or you want to be able to access most of that at pretty low-latency at least across all the regions and for some level of historical look back, because even on a single Prometheus instance, for instance, you are really restricted by the size of the disk. Very high write volumes. That might not be actually a lot of space for you, but it depends on your workload.

So, I really think that you should start off with Prometheus for monitoring and then evaluate the other solutions that are out there based on your needs. Then there's a few out there now, some of them that are open source, some of them that aren't. InfluxDB is open source, but only open source for a single machine. So you can't use that cluster version of InfluxDB, for instance, for storing, monitoring data unless you're paying for the enterprise edition.

Obviously, I'll also kind of recommend that you look at some of the cloud vendors out there as well. You need to weigh up how much they're giving you for how much is going to cost you to use them. Again, depending on your business, if you're a very high-margin business, extremely high-margin, that that might be totally fine. But as you're seeing with Kubernetes and other dynamic workloads, there is an increasing number of basic dimensions that you're trying to query and store metrics data in your system.

So pod names are basically cycling around constantly. There's essentially all sorts of different tags that could be added based on what name space you're running in, what service, microservice name is running inside the Kubernetes cluster and a whole bunch of other things that basically is increasing the regular level of cardinality in these systems these days. So, that's why I think the efficiency and the cost model is becoming more and more important in the modern day, because, honestly, people are just trying to do a lot more with their systems and they want more out of their monitoring and observability.

So, yeah, I think you'll see this ongoing evolution of – It may makes sense at different points in time depending on how advanced a cloud vendor is versus how advanced a purely monitoring-focused vendor is, versus how well the open source solutions today are performing and how easy they are to use. So I think it's a mix of all of these things and I think it's something that's changing very rapidly out there.

[00:56:06] JM: Rob Skillington, thank you for coming on Software Engineering Daily. It's been a pleasure talking to you once again.

[00:56:10] RS: Yeah, it's been great, Jeff. Thanks for having me and I really appreciate you taking the time.

[END OF INTERVIEW]

[00:56:19] JM: Podsheets is open source podcast hosting platform. We are building Podsheets with the learnings from Software Engineering Daily, and our goal is to be the best place to host and monetize your podcast.

If you've been thinking about starting a podcast, check out podsheets.com. We believe the best solution to podcasting will be open source, and we had a previous episode of Software Engineering Daily where we discussed the open source vision for Podsheets.

We're in the early days of podcasting, and there's never been a better time to start a podcast. We will help you through the hurdles of starting a podcast on Podsheets. We're already working on tools to help you with the complex process of finding advertisers for your podcast and working with the ads in your podcast. These are problems that we have encountered in Software Engineering Daily. We know them intimately, and we would love to help you get started with your podcast.

You can check out podsheets.com to get started as a podcaster today. Podcasting is as easy as blogging. If you've written a blog post, you can start a podcast. We'll help you through the process, and you can reach us at any time by emailing help at podsheets.com. We also have multiple other ways of getting in touch on Podsheets.

Podsheets is an open source podcast hosting platform, and I hope you start a podcast, because I am still running out of content to listen to. Start a podcast on podsheets.com.

[END]