ALWAYS-ON TIME-SERIES DATABASE

# Keeping Up where there's no way to Catch Up

A DISCUSSION WITH THEO SCHLOSSNAGLE, JUSTIN SHEEHY, AND CHRIS MCCUBBIN

*In all likelihood, you've never given so much as a thought to what it might take to produce your own database. And you'll probably never find yourself in a situation where you need to do anything of the sort.*

*But, if only as a thought exercise, consider this for a moment: What if, as a core business requirement, you found you needed to provide for the capture of data from disconnected operations, such that updates might be made by different parties at the same time—or in overlapping time—without conflicts? And what if your service called for you to receive massive volumes of data almost continuously throughout the day, such that you couldn't really afford to interrupt data ingest at any point for fear of finding yourself so far behind present state that there would be almost no way to catch up? Given all that, are there any commercially available databases out there you could use to meet those requirements?*

*Right. So, where would that leave you? And what would you do then? We wanted to explore these questions with Theo Schlossnagle, who did, in fact, build his own time-series database. As the founder and CTO of Circonus, an organization that performs telemetry analysis on an*

*already large and exponentially growing number of IoT (Internet of Things) devices, Schlossnagle had good reason to make that investment.*

*Justin Sheehy, the chief architect of global performance and operations for Akamai, asks Schlossnagle about the thinking behind that effort and some of the key decisions made in the course of building the database, as well as what has been learned along the way. On behalf of ACM, Chris McCubbin, a senior applied scientist with AWS (Amazon Web Services), contributes to the discussion.*

JUSTIN SHEEHY As someone who once made the dubious decision to write my own database, I know it *can* prove to be the right thing to do, but—for most companies—I don't think it turns out that way. This isn't just a business question, but one that also has some interesting engineering dimensions to it. So, Theo, why did you feel the need to write your own time-series database?

THEO SCHLOSSNAGLE There were a number of reasons. For one, almost all the data that flows into our telemetry-analysis platform comes in the form of numbers over time. We've witnessed more than exponential growth in the volume and breadth of that data. In fact, by our estimate, we've seen an increase by a factor of about $1x10^{12}$ over the past decade. Obviously, compute platforms haven't kept pace with that. Nor have storage costs dropped by a factor of $1x10^{12}$. Which is to say the rate of data growth we've experienced has been way out of line with the economic realities of what it takes to store and analyze all that data.

So, the leading reason we decided to create our own database had to do with simple economics. Basically, in

the end, you can work around any problem but money. It seemed that, by restricting the problem space, we could have a cheaper, faster solution that would end up being more maintainable over time.

JS Did you consider any open-source databases? If so, did you find any that seemed almost adequate for your purposes, only to reject them for some interesting reason? Also, I'm curious whether you came upon any innovations while looking around that you found intriguing... or, for that matter, anything you really wanted to avoid?

TS I've been very influenced by DynamoDB and Cassandra and some other consistent hashing databases like Riak. As inspiring as I've found those designs to be, I've also become very frustrated by how their approach to consistent hashing tends to limit what you can do with constrained datasets.

What we wanted was a topology that looked similar to consistent hashing databases like DynamoDB or Riak or Cassandra, but we also wanted to make some minor adjustments, and we wanted all of the data types to be CRDTs [conflict-free replicated data types]. We ended up building a CRDT-exclusive database. That radically changes what is possible, specifically around how you make progress writing to the database.

There are a few other nuances. For one thing, most consistent hashing systems use the concept of vBuckets in the rings where, say, you have 15 hosts and 64 virtual buckets that data falls into, with the hosts ultimately negotiating to determine which of them owns which bucket.

With our system, we wanted to remove that sort

of gross granularity. So, we actually use SHA-256 as our hashing scheme, and we employ $2^{256}$ vBuckets. As a consequence, where each data point falls is driven by where the node and the ring fall instead of by vBucket ownership.

This allows us to break into what we call a "two-sided ring." In a typical consistent hashing ring, you have a set of hosts, and each of those has multiple representations all around the ring. What we did instead was to allow the ring to be split in half, with the first 180 degrees of the ring—the first pi of the ring—containing half of the nodes, and the other pi containing the remaining nodes.

Then, to assign data to the nodes, we used what we call a "skip walk." Basically, that flips pi radians back and forth across the ring, thus guaranteeing that you alternate from one side of the ring to the other, which turns out to be pretty advantageous when it comes to putting half of your ring onto one AZ [availability zone] and the other half onto another—or into one region or another, or into one cloud or another.

An interesting aspect of using CRDTs—rather than requiring consensus to make progress—is that it lets you, say, put half of your ring on an oil rig and the other half in the Azure cloud, and then have everything synchronize correctly whenever the VSAT [very small aperture terminal] link is working as it should. That way you can have all the same features and functionality and guarantees even when they're disconnected.

JS I've also done some work with CRDTs and find them interesting in that you have data types that can be updated by multiple parties—possibly on multiple computers, at

the same time or in overlapping time—without conflicts. Since updates are automatically resolved, you don't need isolation in the traditional database sense to ensure that only one party at a time is able to perform a transaction on a given data structure. In fact, this can happen arbitrarily, and it still will all sort out.

Also, there are different ways to solve this, whether through commutativity or convergent operations—bearing in mind that a lot of research has been done on these over the past 10-plus years. So, you can have some of the benefits of consensus without forbidding more than one party to act on the same data at any one time. Which is why I consider this to be an exciting area of research and implementation.

CHRIS McCUBBIN How does this apply to the data types and operations that are best suited for time-series databases? Has that even been the focus of CRDT research to date?

TS Much of the CRDT research so far has focused on disconnected operations, and, certainly, that isn't the first thing that comes to mind when you think about time-series databases. But the real advantage of the CRDT approach is that, if you can limit your entire operation set to CRDTs, you can forego consensus algorithms such as Paxos, Multi-Paxos, Fast Paxos, Raft, Extended Virtual Synchrony, and anything else along those lines. Which is not to disparage any of those algorithms. It's just that they come with a lot of unnecessary baggage once the ability to make progress without quorum can be realized.

There are a couple of reasons why this makes CRDTs incredibly appealing for time-series databases. One is that most time-series databases—especially those that

work at the volume ours does—take data from machines, not people. The networks that connect those machines are generally wide and what I'd describe as "always on and operational." This means that, if you have any sort of interruption of service on the ingest side of your database, every moment you're down is a moment where it's going to become all the more difficult to recover state.

So, if I have an outage where I lose quorum in my database for an hour, it's not like I'll be able just to pick up right away once service resumes. First, I'll need to process the last hour of data, since the burden on the ingest side of the database continues to accumulate over time, regardless of the system's availability. Which is to say, there's an incredibly strong impetus to build time-series databases for always-on data ingest since otherwise—in the event of disruptions of service or catastrophic failures—you'll find, upon resumption of service, your state will be so far behind present that there will simply be no way to catch up.

JS It sounds like, for thoughtful reasons, you traded one very hard problem for another—by which I mean you got out from under the issues related to consensus algorithms. I've learned, however, that many so-called CRDT implementations don't actually live up to that billing. I'm curious about how you got to where you could feel confident your data-structure implementations truly qualified as CRDTs.

TS It's certainly the case that a lot of CRDTs are really complicated, especially in those instances where they represent some sort of complex interrelated state. A classic example would be the CRDTs used for document

editing, string interjection, and that sort of thing. But the vast majority of machine-generated data is of the write-once, delete-never, update-never, append-only variety. That's the type of data yielded by the idempotent transactions that occur when a device measures what something looked like at one particular point in time. It's this element of idempotency in machine-generated data that really lends itself to the use of simplistic CRDTs.

In our case, conflict resolution is the primary goal since, for time-series data, there can be only one measurement that corresponds to a particular timestamp from one specific sensor. To make sure of that, we use a pretty simplistic architecture to ensure that the largest absolute value for any measurement will win. If, for some reason, a device should supply us with two samples for the same point in time, our system will converge on the largest absolute value. We also have a generational counter that we use out of band. This is all just to provide for simplistic conflict resolution.

**Conflict resolution is the primary goal since there can be only one measurement that corresponds to a particular timestamp from one specific sensor.**

With all that said, in the course of a year, when we might have a million trillion samples coming in, we'll generally end up with zero instances where conflict resolution is required simply because that's not the sort of data machines generate.

*As might be expected, Schlossnagle and his team didn't start from scratch when it came to designing and developing their time-series database. Instead, they looked to see what frameworks might be used and which libraries could be borrowed from.*

*Up front, they determined what was most crucial and which tradeoffs they would be willing to make. For example, they knew they would need to treat forward and backward compatibility as a fundamental requirement since they couldn't afford to have anything disrupt data ingestion.*

*They also understood that the actual matter of writing data to raw devices would be one of the hardest things to get right. So, they designed their database such that there are only a few places where it actually writes to disk itself. Instead, a number of existing embedded database technologies are leveraged, with optimized paths within the system having been engineered to take advantage of each of those database technologies while also working around their respective weaknesses.*

JS It's clear you viewed CRDTs as a way to address one of your foremost design constraints: the need to provide for always-on data ingest. Were there any other constraints on the system that impacted your design?

TS We've learned firsthand that whenever you have a system that runs across multiple clusters, you don't want to have certain nodes that are more critical than all the others since that can lead to operational problems. A classic problem here is something like NameNodes in Hadoop's infrastructure or, basically, any sort of special metadata node that needs to be more available than all the other nodes. That's something we definitely wanted to avoid if only to eliminate single points of failure. That very much informed our design.

Also, while we focused on the economies of scale since we were taking in some really high-volume telemetry

data, one challenge we didn't think about early enough, I'd say, was how we might later manage to find things among all that data. That is, if you have a million trillion samples coming into your system over the course of a year, how are you going to find something in all that? How are you even going to be able to navigate all that data?

For example, in the IoT realm, if you're taking in measurement data from 10 billion sensors, how are you then going to isolate the data that was obtained from certain sensors based on a metadata search across a distributed system? I don't think that would normally pose a particularly hard computing challenge, but because it's not something we designed for up front, it certainly led to a lot of pain and suffering during our implementation.

JS You rarely hear people talk about how they needed to change their approach based on what they learned that was at odds with their initial assumptions or exposed something that hadn't been provided for—like the very thing you're talking about here. If I understand you correctly, you initially didn't include exploratory querying for some data you considered to be of lesser importance, only to realize later it actually was significant.

TS That's a fair characterization. To be more specific, let's just say that, with any telemetry-based time-series system, you're going to have a string of measurements attached to some sensor. Say you're measuring CPU usage on a server. For that case, you would have some unique identifier for a CPU linked to some number of measurements—whether taken every second, every minute, or every tenth of a second—that express how that CPU is actually being used.

We found you can have as many as 100 million or even a billion of these strings within a single system, meaning it can be very difficult to find one particular string and explore it. As a stand-alone computer science problem, that wouldn't be all that difficult to solve.

What complicates matters is that the metadata around these strings keeps changing over time. A great example has to do with container technology. Let's say you attach your CPU-usage data to a container you're running under Kubernetes, and then you decide to do 40 launches a day over the course of a year. This means that, where you previously had 100 million streams, you now have 14,600 times as many [365 days x 40 launches]—so, you've got a real cardinality challenge on your hands.

JS Given your always-on, always-ingesting system and the obvious need to protect all the data you're storing for your customers, I'm curious about how you deal with version upgrades. When you're planning to change some deeply ingrained design element in your system, it's not like you can count on a clean restart. I've dealt with this concern myself a few times, so I know how much it can affect your engineering choices.

TS I think you'll find a lot of parallels throughout database computing in general. In our case, because we know the challenges of perfect forward compatibility, we try to make sure each upgrade between versions is just as seamless as possible so we don't end up needing to rebuild the whole system. But the even more important concern is that we really can't afford to have any disruption of service on ingestion, which means we need to treat forward and

backward compatibility as an absolutely fundamental requirement. Of course, you could say much the same thing about any database that stores a lot of data and has a wide user base.

Postgres is a great example of a database that has really struggled with this challenge in the sense that, whenever you make an on-disk table format change for a 30-TB database, you can count on some sort of prolonged outage unless you can perform some serious magic through replication and that sort of thing. Still, I think Postgres over the past few years has become much better at this as it has come to realize just how large databases are getting to be and how long these outages can last whenever people make changes that break forward compatibility.

JS And it's not only storage you need to worry about with forward and backward compatibility. These same concerns apply to all the nodes you're running in your system since you can't have all of them change versions at exactly the same moment. Also, in my experience, the compatibility issue proves to be quite a bit more difficult from an engineering perspective, since then you're more or less living it all the time—not just when you need to upgrade your storage.

TS This is definitely about more than just on-disk formats and capabilities. Protocol compatibility also needs to be taken into account, specifically with regard to replication and querying and that sort of thing. There are some frameworks such as Google Protobuf [Protocol Buffers] and gRPC that include some advances that provide for this. We use FlatBuffers, which, ironically, also happens to

come from Google. All of these frameworks help future-proof for compatibility. So, you can serialize data and add new fields, but people who have been around for a long time will still be able to read the data without knowing a thing about all those new fields. And, just so, new people will be able to read the old data even though it's missing those fields. This definitely helps ease many of the implementation concerns. But the design concerns remain, so I think you need to approach it that way. In fact, I'd love to learn about any best practices emerging now in industry that address how to design for this forward-compatibility challenge.

Another reservation I have is that these frameworks, for all the advantages they offer, tend to be very language specific. If there's a tool you can use to good effect in, say, Erlang, it's unlikely to be of much help to anyone who works in Go, just as a tool written for Rust isn't necessarily going to do much for people who have to work in a C environment. When you consider that there are more than just a handful of commonly used production-system languages out there, it becomes easy to see how this can make things pretty tricky.

JS I completely agree, but let's get back to what in particular drove the development of *your* time-series database. I'm especially interested in learning about any preexisting things you managed to leverage. We already talked a little about FlatBuffers, but I imagine there were some other pieces of software, or even hardware, you were able to take advantage of.

TS Let me first say that FlatBuffers provides an especially good example in the sense that it provides for the

reuse of a ready-made solution for serialization and de-serialization, which has got to be one of the least glamorous tasks for any engineer. What's even more important, though, is that, by providing a nice toolkit around all that, FlatBuffers also delivers important backward-compatibility and endian guarantees for the network, which are invaluable. This also applies to Protobuf and Cap'n Proto.

I will say, though, that we're not equally enamored of every one of these frameworks. In particular, Avro and Thrift, with their blatant insistence on ignoring unsigned types, have proved to be quite difficult to use in practice. We ended up deciding to focus only on those serialization/de-serialization solutions that actually understand common systems types.

Beyond this, we rely on a large number of libraries. In fact, most of the data structures we use come from open-source libraries. Concurrency Kit is a great example that provides a set of basic data structures and primitives, which really helps in producing non-actor-based, high-concurrency, high-performance systems.

Then there's the matter of storing things on disk, which is always an interesting challenge. One reason people say you should never write your own database is because it is difficult to write something to disk while making sure it's safe and actually located where you think it is. We designed our system such that we have only a few places where we write to disk ourselves. Most everywhere else, we rely on existing embedded database systems that, over time, we've learned to make as pluggable as possible. Today we use four internal embedded database technologies

altogether, with the two most popular being LMDB [Lightning Memory-mapped Database] and RocksDB, a Facebook derivative of LevelDB.

JS When you mention storing things directly on disk, I think of all that has been said about how the common choice to sit on top of a file system comes with lots of conflicts that can keep you from designing your database correctly. Yet, I know you made a conscious decision to go with one specific type of file-system technology. What drove that choice, and how do you feel about it now?

**Bit rot is real, especially when you're working with large-scale systems.**

TS There absolutely is a performance penalty to be paid when you're operating on top of a file system. Most of that relates to baggage that doesn't help when you're running a database. With that said, there still are some significant data-integrity issues to be solved whenever you're looking at writing data to raw devices. The bottom line is: I can write something to a raw device. I can sync it there. I can read it back to make sure it's correct. But then I can go back to read it later, and it will no longer be correct. Bit rot is real, especially when you're working with large-scale systems. If you're writing an exabyte of data out, I can guarantee it's not all coming back. There are questions about how you deal with that.

Our choice to use ZFS was really about delivering value in a timely manner. That is, ZFS gave us growable storage volumes; the ability to add more disks seamlessly during operation; built-in compression; some safety guarantees around snapshot protection; checksumming; and provisions for the autorecovery of data. The ability to get all of that in one fell swoop made it worthwhile to take the performance penalty of using a file system.

The other part of this, of course, is that we would have had to build much of that ourselves, anyway. Could we have built that to achieve better performance? Probably, since we could have dispensed with a lot of unnecessary baggage, such as Posix compliance, which is something ZFS provides for. But that probably also would have required six or seven years of product development. Instead, we were able to get what we needed right out of the gate. It came at the price of a performance penalty, which we were willing to pay.

JS Another consideration, which I'm sure you took into account, is that ZFS has a complicated history in the public eye, with many people having serious doubts about its legal status. Did you run into any difficulty with your customers around your decision to go with ZFS?

TS Success is always defined in the court of public opinion. So, yes, I'd say the ZFS gamble was a risky proposition from the very start. Over time it proved to be a safe market choice due to the adoption of OpenZFS for Linux. Still, I have a feeling that if ZFS had not been made easily available on Linux, we would have needed to re-platform owing to a widespread reticence to deploy ZFS.

In 2016, we had some serious discussions about whether we could move forward with ZFS, given that the majority of our customers deploy on Linux. We hung in there and delayed that decision long enough for OpenZFS on Linux to come through and legitimize our choice. But there was a time when we were close to abandoning ZFS.

JS I fully understand your decision, but I continue to be perplexed about why you chose to go with four embedded database technologies. I assume they don't sit on top of

the file system; you instead give them direct device access, right?

TS No, our embedded databases also sit atop ZFS.

JS How do four embedded database technologies prove useful to you?

TS At root, the answer is: A single generalized technology rarely fits a problem perfectly; there's always compromise. So, why might I hesitate to use LMDB? It turns out that writing to LMDB can be significantly slower at scale than writing to the end of a log file, which is how LSM [log-structured merge]-style databases like Rocks work. But then, reading from a database like RocksDB is significantly slower than reading from a B+ tree database like LMDB. You must take all these tradeoffs and concessions into account when you're building a large-scale database system that spans years' worth of data and encompasses a whole range of access patterns and user requirements.

Ultimately, we chose to optimize various paths within our system so we could take advantage of the strengths of each of these database technologies while working around their weaknesses. For example, as you might imagine, we write all the data that comes in from the outside world into an LSM architecture (RocksDB) since that doesn't present us with any inherent performance constraints. You just write the data to a file, and, as long as you can sort things fast enough, you can keep up. Still, given that these databases can grow substantially over time, you need to keep an eye on that. I mean, if you have a 30-TB RocksDB database, you're going to be in a world of hurt.

We have a number of techniques to stay on top of this. Many of them have to do with time-sharding the

data. We'll have a Rocks database that represents this week's data. Then, as the week closes up, we'll open a new database. Beyond that, after the previous week's database has remained unmodified for a while, we'll ETL [extract, transform, load] it into another format in LMDB that better services read queries—meaning we *re*-optimize it.

In the end, the Rocks database we use to handle ingest is a key-value store, but those values then are stored in a very column-oriented manner. We also glue all that together so you can read from the write side and occasionally write to the read side. In the end, using and blending these two different techniques allows us to optimize for our predicted workloads.

*Given the volumes of data the Circonus system needs to handle and process, optimization is critical. Indeed, the database contains thousands of tunable parameters to allow for that. Making the best use of those capabilities, however, requires extraordinary visibility into all aspects of system performance. Toward that end, Schlossnagle says HDR [high dynamic range] log-linear quantized histograms are used to "track everything" over time and even conduct experiments to find out how performance might be optimized by changing certain isolated tunings.*

*Still, does the team have any regrets about how the system was built? Just a few.*

JS Whenever you're building a database for others to use, there's a tension between just how configurable or tunable you want to make it. At one end of that spectrum is the

option of making almost every variable as user accessible as possible. The other extreme is to make everything as turnkey as possible, such that everything works pretty well and it's hard for users to accidentally break things. Obviously, the spectrum isn't nearly as linear as that, but there's this tension just the same. I'm curious to learn about the approach you took to sort that out. Even more, I'd like to hear about what you learned in that process and what adjustments you then found you needed to make.

TS My own experience, having done it both ways, suggests there's no right answer. I will say, though, that this notion of autotuning, self-configuring software that always works is pretty much a pipe dream unless your use case happens to be really simple. Even if you do choose to let every single configuration parameter or setting be tunable, it's practically impossible to make it such that all those tuning combinations will be valid. You really could wreck your system if you're not careful.

And yet, we probably have between 5,000 and 10,000 tunable parameters inside the system that we can configure online. In fact, the vast majority of those are only internally documented. By way of Tier 2/Tier 3 support, we're able to investigate systems, speculate as to what might be causing some particular problem, and then try to hot-patch it. The feedback from that then informs the default-handling parameters for the software from that time on.

We also have some self-adjusting systems—mostly around concurrency control, throttling, backoffs, and that sort of thing—which more or less just measure use patterns and then self-tune accordingly. These are limited

to those cases where we have extensive real-world experience, have seen the patterns before, and so have the wherewithal to build suitable models.

JS I have to say that having up to 10,000 levers does seem to give you a *lot* of power. But how do your support folks figure out which one to touch? That is, what did you do to provide the live inspection capabilities people could use to really *understand* the system while it's running?

TS That's something I can talk a lot about since one of the really interesting parts of our technology has to do with our use of high-definition histograms. We have an open-source implementation of HDR [high dynamic range] log-linear quantized histograms, called circllhist, that's both very fast and memory efficient. With the help of that, we're able to track the performance of function boundaries for every single IOP [input/output operation], database operation, time in queue, and the amount of time required to hand off to another core in the system. This is something we use to track *everything* on the system, including every single RocksDB or LMDB `get()` or `put()`. The latency of each one of those is tracked in terms of nanoseconds. Within a little web console, we're able to review any of those histograms and watch them change over time. Also, since this happens to be a time-series database, we can then put that data back in the database and connect our tooling to it to get time-series graphs for the 99.9$^{th}$ percentile of latencies for writing numeric data, for example.

Once the performance characteristics of the part you're looking to troubleshoot or optimize have been captured, you've got what you need to perform controlled

experiments where you can change one tuning at a time. This gives you a way to gather direct feedback by changing just one parameter and then tracking how that changes the performance of the system, while also looking for any unanticipated regressions in other parts of the system. I should add this all comes as part of an open-source framework [https://github.com/circonus-labs/libmtev].

JS It sounds like this has really paid off for you and that this whole undertaking has yielded some impressive returns. But I wonder, if you were just starting to build your time-series database today, is there anything you would do in a substantially different way?

TS Absolutely! There are lots of things we would approach differently. The system we're talking about here is nine years old now, so plenty of innovation has been introduced since then that we could leverage. Also, under the heading of "Hindsight is 20/20," I wish I'd selected some different data structures that would have transitioned better from the in-memory data world to the on-disk data one—particularly in-memory indexes and in-memory caches where, at a certain volume, you actually do want to take a cache-style approach, but you also want it to be on disk for availability reasons.

And you really want to be able to treat that as semipermanent. Just think in terms of a 130-gig in-memory adaptive radix tree, for example. Well, it turns out that building a 130-gig ART is nontrivial. It would be nice if I could have those data structures map seamlessly to on-disk data structures. Of course, those are data structures that would have had little practical purpose in 2011 since they would have been too slow without technology such as

NVMe [Non-Volatile Memory Express] supporting them. Still, making those data structures memory-independent pointer invariant—would have been a really good investment. In fact, we're in the middle of that now.

Probably the biggest change I would make at this point—looking back over all the bugs that have surfaced in our product over time—is that I wouldn't write it in C and C++. Instead, if Rust had been around at the time, that's what I would have used. It would have been pretty fantastic to write the system that way since Rust, by introducing the borrow checker and ownership models of memory, has essentially managed to design away most of the issues that have caused faults in our software.

But now, I'd have to say that ship has already sailed; retooling our platform on Rust and reeducating our team at this point would be an intractable proposition. Still, I continue to see this as a missed opportunity because I think a Rust-based system would have served us better for many of the use cases we've encountered over the past few years.



CONTENTS