EPISODE 1289

[INTRODUCTION]

[0:00:00.3] JM: For some data problems, you may be more concerned with the state of data at a particular point; a ticket is booked or it's, not how many poetry submissions were made to the contest. This is relational data. For other problems, you're concerned with the change in data over time. Solar energy consumption, for example, or price behavior. This is time series data.

TimescaleDB resembles a traditional PostgresSQL database, but is supercharged for time series database. TimescaleDB has queries that are 10x faster, is optimized for time series and advanced and time series analytics, has automatic continuous aggregations, columnar storage, and uses algorithms and memory efficient structures to compress your data, so you can store more at a much cheaper price. Every new domain in databases has custom built databases for that domain. Time series databases is no different.

There are several different time series databases that make different trade-offs in how they do storage, how they do memory management. The ones that are best in class are best in class and TimescaleDB is one of those databases. If you have questions that are not time series dependent, you can still treat TimescaleDB as an efficient and cost-effective relational database. It is in some sense, a multi-model database.

In this episode, we talked to Mike Freedman, the Co-Founder and CTO of TimescaleDB, and I hope you enjoy it.

Our first book is coming soon. *Move Fast* is a book about how Facebook builds software. It comes out July 6th. It's something we're pretty proud of. We've spent about two and a half years on this book. It's been a great exploration of how one of the most successful companies in the world build software.

In the process of writing *Move Fast*, I was reinforced with regard to the idea that I want to build a software company. I have a new idea that I'm starting to build. The difference between this company and the previous software companies that I've started is I need to let go of some of the

responsibilities of Software Engineering Daily. We're going to be starting to transition to having more voices on Software Engineering Daily.
In the long run, I think this will be much better for the business, because we'll have a deeper, more diverse voice about what the world of software entails. If you are interested in becoming a host, please email me, jeff@softwareengineeringdaily.com. This is a paid opportunity. It's also a great opportunity for learning and access and growing your personal brand. Speaking of personal brand, we are starting a YouTube channel as well. We'll start to air choice interviews that we've done in person at a studio. These are high-quality videos that we're going to be uploading to YouTube. You can subscribe to those videos at YouTube and find the Software Daily YouTube channel.

Thank you for listening. Thank you for reading. I hope you check out *Move Fast,* and very soon, thanks for watching Software Daily.

[INTERVIEW]

[00:03:18] JM: Mike, welcome to the show.

[00:03:20] MF: Thanks for having me.

[00:03:21] JM: You started Timescale in around 2014. I want to get a sense for why 2014 was a time when a new category of database needed to be created, because this was around the same time other time series databases also got created.

[00:03:40] MF: Yeah. We originally created Timescale, really from our own need. Around that time, 2014-2015, my co-founder and I, Ajay Kulkarni, who we go back many years, we resynced up and we started thinking about it was a good time for both of us to think about what the next challenges are that we want to tackle. It seemed to us that there was this emerging trend of now, people talk about the digitization, or digital transformation. It feels like somewhat of an analyst term, but I think, it's really responsive of what's happening, in that if you think about the large, big IT revolution, it was about changing the back office. What was used to be on paper was now in computers.

What we saw was somewhat the same thing happened to basically, every industry, from heavy industry, to shipping, to logistics, to manufacturing, both discrete and continuous and home IoT. Sometimes this gets blurred under IoT, but we also think about it more broadly as operational technology, those which are not necessarily bits, but atoms. A big part of that was actually collecting data of what those systems were doing. It's about sensors and data and whatnot.

When we do Initially looked at this problem, we were thinking about a type of data platform we would want to build, to make it easy to collect and store and analyze that type of data. I think that's a way that we're slightly different, or why our – what we ultimately built as our database ended up being fairly different than a lot of other so-called time series databases. That's because many of them arose out of IT monitoring, where they were trying to collect metrics from servers, where we were originally thinking about collecting data more broadly from all these type of applications and devices around your world.

When we started building it, it was originally focusing mostly on IoT. We quickly ran into this problem that the existing databases out there and the time series databases out there were not really designed for our problems. They were often much more limited, because they were focusing on this narrow infrastructure monitoring problem, where the data maybe wasn't as important. It was only a very specific type. Let's say, they stored only floats. They didn't have to have extra metadata that they wanted to enrich their data to better understand what was going on, like through joins.

After, basically working on this platform for about a year, we somewhat came to the conclusion that we actually need to build somewhat of our own time series database that was focusing on this more broad type of problem, and so that's what we do. That's what led the development of what became Timescale.

[00:06:36] JM: Today, what are the most common applications of a time series database?

[00:06:41] MF: Like and speak mostly about obviously, TimescaleDB, rather than – as I was alluding to before, a lot of the other time series databases are much more narrowly focused on IT monitoring, or observability. We really see our use cases across the field. We certainly see cases of observability. In fact, we have subsequently built actually a separate product on top of

Timescale called Promp scale, that is really used for initially Prometheus metrics, but more broadly, to make it easier to store observability data with TimescaleDB.

We see still a lot of IoT. We see a lot of logistics. We see financial data and crypto data. We see event sourcing. We see product and user analytics. We see people collecting data about how users are using their SaaS platforms. We see gaming analytics, where companies are collecting information about how people's virtual avatars are actually playing within the games. We see music analytics. We like to think of the old way, used to find the pop stars, you went down to the smoky club. Now you collect SoundCloud and Spotify streams, and you use that to identify who the next breakout artist is going to be.

All of these are example of time series data. It's really what's so exciting to us as is it's such a broad use case, so horizontal, because basically, it's all about collecting data at the finest granularity you can.

[00:08:19] JM: You mentioned an interesting use case there, the event sourcing use case. I'm not super familiar with event sourcing, but I often hear it in conversations about Kafka. I often hear about people writing lots of events to Kafka, and then doing things with those events. Can you tell me more about the event sourcing use case?

[00:08:42] MF: Yeah. Kafka was originally started as, in some sense, a really reliable way to do high-speed buffering, or queuing between different – it somewhat became the glue that was connecting a lot of your microservices. Obviously, over time, they started adding some more limited capabilities around KSQL, around trying to do simple analytics on Kafka. I think fundamentally, even when you add those type of analytics, stream engines like Kafka are about building up a queue of data somewhat linearly. Then with query, you're doing a linear pass across the data.

When databases obviously, were a lot also about reliably storing data, but then build a lot of data structures and reorganize the data, and have a lot of intelligence about how they can intelligently process the data and join the data and optimize all that query that's 30 years of query planning work to make those types of queries much faster. When we think about event

sourcing, it's that we have – I think there's a continuum again, about what time series data means.

Sometimes that's a continuous stream of readings coming off of sensors. Those sensors could be servers. Those sensors could be machinery. They could also have discrete events. It could be, when we run TimescaleDB with our own Kubernetes infrastructure, the discrete events are actually Kubernetes events that are happening in our infrastructure. It could be in a gaming example, when people, when avatars perform different things in the game, with a website. It could be when a user is walking through – is navigating website to when they go to different pages as part of your session.

Those are all effectively, there's a timestamp, there's a thing that happened, there's potentially additional metadata happening. That all becomes part of the same data set that you want to store in something like a time series database that you could then process and analyze in various ways.

[00:10:59] JM: What is the hardest part of operating a database company?

[00:11:04] MF: I think for myself with a technical background, it's always knowing what not to work on. We see this domain as so large, in the sense that it's touching every company that has data, if they start collecting that data at the finest granularity, are going to be collecting time series data. Really, the opportunity is trying to figure out, we have a product that we believe is so widely applicable. The hardest thing is almost what – now, saying what not to do, because all of it's so exciting.

[00:11:40] JM: What's an example of a time you said no?

[00:11:43] MF: Well, I think there's different types of use cases you want to focus on. The trade-offs are always, how much certain technical needs are specific to one, versus another. I'll give you one concrete example. If you look at financial data, and we see a lot of FinTech and crypto, and even some of these DeFi companies starting to use Timescale pretty seriously.

If you look at, for example, traditional hedge funds, or other type of quant research, they do something that they call back-testing their algorithms a lot. That is they want to take the algorithms they have now and then decide how would this have run a year ago against the current state of the market, to basically see how well their algorithm would have worked compared to how the market actually operated.

One of the interesting things about in the financial world is that you'll actually see market corrections. You'll see that you thought that the earnings report of a company was this on January 1st, but actually come February 1st, there was a correction to what it actually was back in January 1st. When you actually do your back-testing, you have to have a knowledge of, well, when I'm running this, what was my knowledge at the time? because the state of the world, what you do about January 1st data on January 15th, was actually different from what you knew what it was on February 15th.

In that particular niche area, they call this often, they refer to this as bitemporal data, that the data has two different types of timestamps; the event time, and in some sense, your awareness time, or your system time. You build a lot of knowledge about your temporal understanding of it at the time. Now, this is a fascinating area. I think, there's lots of interesting stuff you could do here and we have some support for this type of analysis, but we could go a lot deeper. It's just that the trade-off between how big a need for there is there here, versus other types of time series capabilities we would want to spend our engineering and product efforts on.

[00:14:02] JM: Tell me about the initial architecture for TimescaleDB. You're based off of PostgresSQL. What was the reasoning around that decision?

[00:14:14] MF: I think, as you point out, Timescale is actually implemented as an extension on PostgresSQL. Starting maybe 10 or 15 years ago, PostgresSQL started exposing low-level hooks throughout its code base. This is not a plugin where you're running a little JavaScript code. We have function pointers into – we get function hooks into the C. PostgresSQL is written in C, and so TimescaleDB is, for the most part written in C. We have hooks throughout the code base at the planner, at sometimes in the storage, at the execution nodes. We are able to insert ourselves and do Lot of optimizations as part of the same process.

You could ask the question of why not just implement a new database from scratch? Why build it on top of PostgresSQL? I think this really gets to that, we always viewed ourselves as, and we hear this from our users and community all the time that we are – they are storing critical data inside TimescaleDB, and they need it to, A, work and be reliable. They also need it to be – they have a lot of use case requirements. It's not this, again, narrow thing where you're collecting one metrics, and all you're asking to do is figure out the min-max average of a certain metric.

You want to do fancy analysis. You want to do joins. You want to do sub queries. You want to do correlations. You want to have views. You want the operational maturity of a database. You want transactions, backup, and restore, and all of the replication and all of the above. Some people say, it takes maybe 10 years, at least, to build a reliable database. We thought this was a great way in order to immediately gain that level of reliability, we ourselves are huge fans of PostgresSQL. It has such a great community. It also has such a large ecosystem.

The idea is that effectively, that entire ecosystem would work from us on day one. That means, all of the tooling, all of the ORMs, all of your libraries would just work. If we support full SQL, not SQL-ish. If you know how to use SQL, you could start using – and if your tools speak SQL, if you're running Tableau, if you're running Power BI if you're running Grafana, if you're running Superset, those all just start working on day one.

Now, the second part of it is, well, what does that mean to build a time series database on top of PostgresSQL, which clearly was designed more as a traditional transactional database, OLTP engine? Sometimes they talk about you think about this architecturally. What I mean by that is you somewhat think about what your workloads look like and what that would mean from a software architecture. Maybe I'll give you a very concrete example. Starting maybe 10 or 15 years ago, if you look at traditional databases, you started seeing the growth of what people commonly now called as log structured merge trees, LSMs.

This is a data structure that goes back to the mid-90s, but I think you first saw Google, Jeff Dean and Sanjay Ghemawat built something called LevelDB. The whole idea of an LSM tree was, if you look at a workload that has a lot of updates, so with a lot of e-commerce applications, with a lot of social networks, you're constantly updating things. Traditional database, if you think about a disk, if you're doing a lot of in-place updates, and these updates

are randomly distributed across all of your user IDs, this means that you're going to cause your disk to do a lot of random writes on hard drives, that's particularly bad. You need to move the disk.

Even on SSDs, it doesn't do great, because SSDs still do a lot better to have sequential writes than random writes, the way the internals of SSDs work. You started seeing this new type of database architecture called LSM trees emerge, because people wanted to build databases that had a lot faster updates. On time series databases, on the other hand, don't typically have this type of workloads.

If you think of a stream of new observations, with the timestamp, these are typically about what's happening now. It's typically about a stream of inserts that are about this stock price now, this stock price now, this stock price now, or different, or a 100, or a 1,000, or a 100,000 different sensors all about what the recording right now.

If you think about how you would then design the internals of your database and the data structures, when most of your rights are insert heavy, and particularly about the latest time interval, then what that would mean is the somewhat internal structure of your data should reflect that. You should optimize your insert path to make it super-efficient to perform inserts on the latest time interval. It doesn't have to be perfectly in order, but it mostly is about what's happening recently, as opposed to what's happening a year ago.

That said, Timescale absolutely allows you to backfill data and perform updates, or deletes to older data. It's just from a performance perspective, keeps all the recent stuff in memory and builds more efficient data structure to allow you to insert at much higher rates. For example, on a single machine, if you're collecting a stream of records, eat for several, let's say 10 metrics, you'll be able to collect even once 2 million metrics per second on a single, pretty standard machine.

Then we see this again and again, the way we think about architecting Timescale is, is really thinking about what the workload looks like, that people often care about recent data. The way they want to manage their data changes as that data ages. They might want to optimize for even fast queries for the recent stuff. They might want to start reorganizing their data as ages.

They might want to start automated automatically aggregating the data as it ages, and dropping the raw data for the very old stuff to save space. All of these things are what you'd want in a good time series database, when it's not what you want from either a traditional OLTP database, nor if you have a traditional data warehouse, or an analytical database, which doesn't think of this operational view of time series so central to it.

[00:21:26] JM: Can you tell me a little bit more about the pros and cons of building as a PostgresSQL extension? What are the pros and cons of having that underlying system that you're building on top of?

[00:21:41] MF: Yeah. I think, there's two halves of that. One is, what were all the fun and hard engineering we had to do from day one? Two, what are perhaps, some of the ongoing limitations, or complexities and how are we addressing that going forward? One of the fun things we've been able to do is, I think, really, have been lucky enough to have a really amazing engineering team. We've been able to do a lot of very creative and advanced stuff, even in our little sandbox, which is PostgresSQL, in ways that many people didn't think you could do. One concrete example.

Many of your listeners probably know that PostgresSQL is a row-oriented database. If you store a row of data, let's say with 10 metrics, it stores all of those 10 columns contiguously on disk. That is great for a lot of workloads, where you're trying to pull out data row by row. It's also good if you're trying to within one query, access many of the rows. Not necessarily all of them, but many of them. There are analytical workloads, where you might only be interested in one particular column, and you might actually go back for a year's worth of time.

In a traditional row-oriented database, you would have to, in this case, there's a billion rows over that year, you have to read those billion rows from disk and then pull out the single column that you want. Now, what came out starting a couple – maybe in the last two decades is this notion of column-oriented databases, where a lot of times in the data warehousing world, when all of your queries are these large column scans, because you don't even have indexes, then you would actually write your data, not every – the row being co-located on disk, but you'd write them on disk as large columns, and then you'd compress them.

The downside of these is, if you actually ever wanted to access rows, they were inefficient. Column-oriented databases were really good at large column scans. Row-oriented databases were good when you want to just pull out individual rows. Turns out in time series workloads, you have both. You often get what you call wide shallow queries, and then deep narrow queries. Wide shallow queries, you might be saying, “I want to know all information about these set of devices, or my stock portfolio from the last week.” While a deep narrow query would be, “Tell me about this particular device, or tell me about this particular ticker symbol over the year.” What's interesting is that row-oriented databases are better for the former. Column-oriented databases or better for the latter. In the time series, you have both, but again, they often have the flavor that recent data is typically better accessed in lower order form, when long historical scans are better oriented in column-oriented form. In fact, that's what we do in Timescale.

What we do is we actually have a background engine that internally and asynchronously transparent to the user, converts our representation from row-oriented after the data hits a certain age to column-oriented, so you could enable those type of long scans. We do a lot of advanced compression. Now, when people looked at this, they said, “Well, how do you store somewhat these columnar data inside PostgresSQL? Isn't PostgresSQL itself have this row-ordered nature at its lowest storage level?

While that's true, we were in some sense, able to do a lot of clever engineering, where while at some level, we're storing these compressed. We basically are building and storing what are compressed columns, accumulate a 1,000 rows of data, according to a certain keys that are accessed together a lot. Flipped them on their heads. They're at columns, and now store those as highly compressed arrays in this underlying engine.

In fact, one of the neat things is, because we do this, we could often – and what we do is we apply different compression algorithms based on the type of data it is. We use one form of compression for timestamps. We use another type of compression for floats, another type for integers. We use different compression algorithms for strings, whether or not they're strings with high entropy or low entropy. It really allows you to really do this heavy optimization.

While yes, we had to, at some sense, work around the fact a little bit that PostgresSQL as lowest level, until recently, had everything in row-oriented form. Sometimes, I think that's fun

when you're an engineer that challenges lead to new opportunities. We basically have able to build something that is competitive with most native columnar stores. Yet, it still builds on PostgresSQL's reliability. Everything's done transactionally. All existing tooling works, this is all transparent to the users, so sometimes you get big wins.

I would say that, regarding your question, what are some limitations, is that as many other engines, there's always places where you're looking to improve things. About a year ago, we launched the horizontal scale out version of Timescale in Timescale 2.0. This obviously, that's not what PostgresSQL was designed for, so we are basically building a distributed, horizontally scalable database on top of PostgresSQL.

There is some aspects of PostgresSQL, I talked about the various ways we're dealing with storage engines. The latest versions of PostgresSQL and PostgresSQL 13 and 14, allow you to make better low-level changes to the PostgresSQL storage methods. They have this new interface called table access methods. This is all ways that we could continue to innovate on top of PostgresSQL. I think, one of the really great things has been as PostgresSQL and as the community continues to make contributions, and it's such a active, vibrant community, that also benefits everybody who's using Timescale.

One of the things that we have been proud of in the last year is we've – as we have grown to mature, we just brought on really people, whose job it is to have full-time upstream contributions to PostgresSQL, both in terms of code review, and commit fests and contributions themselves. That's been a great way that we've now started to more heavily give back to the community in a more direct sense.

[00:28:27] JM: The interface for a PostgresSQL extension, what exactly does that expose to you? How deep into the PostgresSQL internals can you reach and change?

[00:28:41] MF: It gets pretty deep. I wouldn't really call it a – Sometimes when you think of an extension, you think of a really clear barrier, where you think of an OS extractions, where you can make system call and here's your well-defined system call interface. I think, one of the ways that PostgresSQL allows you to build is that you have various hooks inside the code base at various parts of the most DML, or DDL changes. When calls like create tables, or alter tables

come in, we could hook in there. When inserts come in, when upstarts come in, when queries come in, we could hook in the planner, we could hook in the execution nodes, we could write custom execution nodes that go into the plan. When a query starts getting executed, our code kicks in.

There's a lot of places we could hook into the internals. We obviously do that in a careful way, in best thinking about how we could achieve the goals, while obviously, still maintaining the safety of transactions and memory safety and whatnot. We are writing low-level code in C that is part of every query, or insert that you're doing against hyper tables, against Timescale's notion of a partitioned table.

[00:30:12] JM: Given that you're a professor at Princeton as well, as an entrepreneur, I'm curious, are there some outstanding theoretical problems that you've encountered as an entrepreneur, like given infinite time? You might you might explore from a theoretical perspective.

[00:30:35] MF: Yeah. I should say, my research background, my academic background is actually more in distributed systems and storage systems, than historically, in databases. I know for a lot of developers, these are heavily the same thing. For whatever reason, at least, in the academic communities are pretty distinct communities. I think, that's also one way why I'm so excited with Timescale 2.0, now that it's actually a distributed database, and we could – when you look at what we can do on 2.0 plus the compression, we basically are able to now hit the petabyte scale, where I think it gets especially interesting for what people could start doing.

More broadly, wearing my academic hat for a minute, the way I typically look at problems, like what turned out to be good research problems, and what drive systems problems is that historically, there have always been two areas. One is, as systems, or database researchers, you get motivated by new application workloads. In some sense, these are things that changed in the world that now, what you are building before no longer applies to these new workloads.

The other place where you get a lot of innovation was when the hardware and the device people change things underneath you. If you think about a lot of the work that was happening with

databases when they were designed for hard drives, for spinning Rust, when we moved to SSDs, we started thinking about things differently.

Now, actually, some students that at Princeton are looking about what does it mean to build databases, where you're storing all of your data on new types of NVMe type devices, or even when you use a blend of NVMe, like optane drives with cheap SSDs, like QLC. What does that mean for trade-offs between performance and cost, when you actually are leveraging heterogeneous storage?

We actually already see this in the commercial world as well. You just probably seen a little bit coarser granularity. For example, with Timescale, and other databases, you have the notion of data tearing, where you want to – as data ages, again, this is a very time-centric view of the world. As data ages, the most fresh data is usually in memory, the medium age data is on something like SSD, and the oldest data gets moved to slower storage that maybe is less performant, but still accessible. Again, you're doing this cost benefit trade-offs.

Continue with these devices, I think there's obviously, interesting questions about how you can over time combine traditional CPUs with GPUs to accelerate individual queries. I think, over if you look forward five or 10 years, I think you're going to continue to see this type of innovation, where we start looking at all the various different types of heterogeneous resources we have at hand, different types of disks, even different types of compute devices, and then start making them accessible. I think, the cool thing about having cloud services is you want to remove all these complexity from your developer.

That's really been a core part of our design from Timescale from the beginning. We give you what looks like a single database table. We call it a hyper table, but you do a SQL, you insert into it, you query it, it just looks like a table. Yet, that table could stretch across multiple different types of disk technologies. That table can stretch across multiple servers. You can treat that data as it ages differently. You can join your row data with your columnar compressed data, and all that is transparent to the user.

I think, and I know I'm jumping around from academia back to industry, if you will. I think, the exciting part is, how do we continue to build databases and other type of server infrastructure

that makes it really easy for developers? Because they don't have to think about this. They don't have to think as much about performance optimizations. They don't have to think about operational aspects. It just works and it's just easy and fast.

Yet, you could give them the benefit of this performance cost trade-offs of all the various types of existing and emerging technology that's really coming. I think, there's lots of interesting opportunities there both academically, but also to make it real and to put it in developers hands. I think, that's a little bit what we're doing with Timescale, both of our database, but also with our cloud service that, again, abstracts all this complexity away from users.

[00:35:39] JM: As the company has gone from an earlier stage infrastructure company to a really advanced and larger scale infrastructure company, how have the sales challenges and customer success challenges changed?

[00:35:56] MF: I think that an important aspect is that we do think differently about the database we're building for our community, and also, the service that we're building commercially. I think, there's two things that are interesting. One thing that you didn't exactly ask, but I thought you were going to, is that from an engineering perspective, we ended up – we obviously, from the beginning, we're building a database. We had a lot of – you had a bunch of great engineers that were really fascinated by building databases. It was a low-level. All the development is in C, for the most part. I say for the most part, because now we have some cool stuff going on in Rust.

When we decided that to really then view our, if you will, commercial service, our go-to market was really to enable cloud consumption. We ended up building a wholly separate, a completely new engineering team in order to do that. Because I think, obviously, the skill sets for building a cloud service is very different from building the underlying database going so.

Now, the second thing which we did was, we really embraced the notion of making our database completely free to the community, and we're going to focus on our go-to market as providing a fully managed cloud service. What that means is we adopted licensing that basically meant, it's the core of our database is Apache Juice, a fully open source. Then, a number of the advanced features are somewhat akin to the same playbook that you see a number of the other software

vendors, like Elastic, Confluent, Redis, Cockroach, some extent, Mongo take, which is that it's free for people to use. It's all on GitHub.

There's nothing that's enterprise, or close source, but there are a few restrictions in particularly, we call the Timescale license. Particularly, you can't basically, run TimescaleDB as a service. You can basically, commercially offer TimescaleDB as a service. The extent that Amazon can't run that version of TimescaleDB as part of RDS, or Azure can't offer that as part of Azure PostgresSQL. Although they do, they can and actually, Azure does offer the Apache 2 version of Timescale as part of Azure PostgresSQL.

What that has allowed us to do is really double down on our community. Every database feature we build is free for anybody, except Amazon, or Azure, or Google to use. In fact, many do embedded as their own SaaS service, and make it available to their own users. In doing so, this is really – so your questions are how we thought about sales? How has it allowed us to scale?

I think, that has been an important decision. Because in a lot of infrastructure companies, product is constantly faced with the question of, “I have this cool feature. Do I make it a paid-only enterprise feature? Or do I make it a free feature and drive the community?” The great thing about the way we decided to do this, and in fact that we said, our go-to market is to provide a managed service, if you want to self-manage Timescale, you can do so and it's completely free.

It has allowed us to really every database feature that we build, makes it a better product for the community. We don't say, “Hey, this is only in our enterprise version.” On the flip side, we can – there is a healthy portion, I think, of the developer community, who are increasingly saying, “Hey, I don't want to manage software anymore.” Particularly, databases where it's not a stateless service that you're just going to spin up and down in Kubernetes. That’s often, your really critical business data, or your critical information and you want to make sure it's reliable and highly available.

They say, “I want to actually turn to a managed service to take that weight off of our engineering and ops teams.” For those, we could say, “Hey, we're going to provide you now a really great

cloud service. It's going to have all the great features of TimescaleDB, as well as over time, more of a platform or experience to really interact with your time series data in a powerful way."

In doing this, I mean, you talked a little – you asked him before about what it means for sales and what it means for customer success. We've really taken a developer-first view of the market, where be someone have what you call a bottoms-up approach. People show up on our website. They sign up. There's a free trial. Within one minute, they have a database running in the cloud, and they can immediately start using it.

In fact, we have people who have happily converted and have been users for years who have never talked to us. We are always very happy to be there and we obviously have a great support team all around the world who can work with people. It's really enabling developers to basically, be able to interact with you as they want to. Not as your sales team wants to interact with the developer.
I think, that has really been a core part of our strategy, to help people, help people in the community, help developers, help customers, but do it on their terms, and do it from a very bottoms-up approach.

[00:41:40] JM: I would be remiss if I didn't ask the question, because I'm sure people listening are curious. What are the points of comparison between Timescale and the other prominent time series databases?

[00:41:58] MF: Yeah. I think, one of the biggest differences between time scale and effectively, all of the other time series databases, and if you look at our name, we often refer to ourselves as the relational database for time series, and not just as a time series database. Because if you look at, for example, what some other people refer to as time series databases, it's things like, if you see some lists, things like, obviously, influx, which many people have heard of Prometheus, Graphite, open TSDB, Amazon Timestream now and other things.

Many of these like, let's say Prometheus, or Graphite, or I think there's M3DB, these were all really designed for IT metrics. They store floats and that's basically it. If you want to do joins, good luck. If you want to do something, I mean, Prometheus obviously supports PromQL. Graphite has an HTTP interface. Influx keeps changing, and it starts something InfluxQL, which

was SQL-ish. It dropped that. It moved to flux. They're now maybe doing something SQL-ish again, but they also have this new IOX engine, which is the fourth rewrite.

I think that what people get with Timescale is really the ability to store both their time series data, as well as their relational data. That turned out to be really, really powerful. In fact, if you were looking one of the common deployments that in fact, influx sales engineers recommend is to deploy in flux with PostgresSQL, or in flux with Mongo, because you often have data, metadata that you actually can't store inside influx's tag model. Even if you tried to shoehorn it, its carnality would quickly blow up and it would become very inefficient and you would run out of memory.

The answer is, well, if you want to do these type of queries, deploy this time series database, and deploy a relational database, and then write application code that joins them. Timescale, the answer is, well, just you have a hyper table with your time series data, you have a relational data table with your other data. You could build foreign keys between them if you want, and you do a join at query time. That is operationally much simpler. It's much faster developer – it's much better developer experience and much better developer productivity.

If you remember, when we started, we talked about in 2015, we were building this IoT platform, and we were frustrated. That was one of the frustrations we had. We were using one of these time series databases and we want to do this query. We realized that the additional information we wanted was in our PostgresSQL. It wasn't in our time series database. It took us another four weeks to actually go through the engineering sprints, in order to roll out the ability to do that one query. On TimescaleDB, it would be writing a new join, and it would be out in an hour.

I think, the extent to which Timescale is really a relational database for time series, or is PostgresSQL with superpowers is really, I think, one of the reasons why people are finding such great ease of use and developer productivity, compared to a lot of the other much more narrow time series databases.

[00:45:29] JM: If you were not working on TimescaleDB, and you had to start a company, what would you start a company around?

[00:45:37] MF: I'd have to think about that a little bit more. We're so excited about, I really feel we're only starting in this journey of what we see as the future of data. I don't have a good answer for you today. I mean, it's funny, because obviously, as you point out, I'm also an academic. I think, there are always good technical questions that you come across, and certainly, building our own timescale cloud, we see a lot of – I see, there's a lot of interesting stuff that's happening in cloud operations. I think, for example, we actually have some research that's starting to ask the question, if you look at most distributed systems, they were databases, or primary backup, even things like Paxos, where you have multi-node replication.

You're all doing recovery of the application space, when if you think about trends that are happening in the cloud, the decoupling of compute and storage, which you see, for example, in stateless sets and Kubernetes, which is enabled by network attached disks and network attached storage that the clouds are enabling, which by the way, we take advantage of Timescale cloud to build, decoupled compute and storage and for our database. I think, this actually leads to interesting questions about rethinking, for example, how high availability works at a much more cost-effective way.
Because today, for private backup, you often would pay 2x, because you need to run two servers. While if you could take advantage of some of this decoupling in the cloud, then it could get you almost as good at a tiny fraction of the cost and we're looking at some research on how to drive down even recovery time by when you have false positives of failures, being able to start rollout and rollback if it is false positives. Anyway, details. I think, there's a big difference between what are interesting technical problems and what are good commercial opportunities. There's demand. There's large market demand, and so forth. I'm not sure I have an answer, because I've just been so focused on building and scaling Timescale.

[00:47:55] JM: An area that I consider Timescale, somewhat adjacent to is that of data engineering. Obviously, you're not directly a data engineering product. You're more of database. You're heavily used by operations people. You're used by a wide variety of applications. You're going to be a source for a lot of data engineering applications. I wonder from that vantage point, what have you seen around the world of data engineering? How are data engineers interfacing with Timescale?

[00:48:27] MF: Just to be clear, when you're talking about data engineering, you mean, the people who are building the pipelines, who often serve data science or machine learning pipelines?

[00:48:35] JM: Yes.

[00:48:36] MF: Okay. Cool. Yeah, so we absolutely see Timescale as part of data science pipelines. Interestingly, in actually a couple different ways. First is somewhat, what you'd expect that companies are actually storing their data and Timescale becomes the both source and sync of that data. They build upstream ingest pipelines. That data comes in. It gets stored in Timescale. Then you have various data scientists who are going to query that data to run various models and training and an inference and whatnot.

We see Timescale being used both on the training side, as well as the inference side as a real-time serving layer for their data models. The other place that we – and actually, let me add on. Again, this comes from part of Timescale's benefit from building on the operational maturity of PostgresSQL, in that Timescale allows you to have multiple read replicas attached to your primary database. Those read replicas can be synchronous. That is, they can be exact copies of your primary, such that before any data is committed to your primary, it's safely and reliably stored on the replicas. In case there's any failures to the primary, you always have a spare copy.

hose read replicas can also be asynchronous. Transactions will commit when it's just written to the primary. Then, they are asynchronously copied to the replicas. Usually pretty quickly, but that avoids, let's say, a failure of the replica, or all of a sudden, a burst of new ingest that allows those inserts to return more quickly. The advantage also asynchronous is what often teams do, is they actually use those asynchronous replicas, those read replicas to basically, be used by their data science teams. Data science teams can start writing crazy queries against their read replicas, and you could have greater confidence that your primary database, which is the operational core of your application isn't going to be disrupted by all of the craziness that your data scientists are doing.

Now obviously, some people might want to further decouple that through other types of ETL jobs, or Kafka, or whatever. This has been a big win for teams to also, again, to deploy things faster, easier, to quickly be able to spin up a read replica and hand it off to a data science team.

The other interesting things we've actually seen Timescale deployed on is actually, we've seen it being used as to monitor the quality of data pipelines. Imagine that you've trained a model, you've deployed a model into production, one thing that you're doing is as you're doing queries against your model, as you're performing inferences, you're also basically, measuring the quality of those of those inferences.

What you start doing is you could start seeing as, “Hey, does my model over time seem to actually no longer achieve the same type of accuracy that it used to?” Possibly, because the – if I could talk a little machine learning. If the underlying distributions that what you trained it on changed, if they're non-stationary, what you're going to want to do is retrain your model given the new data distributions. What we've seen people do is, in applications where sometimes your underlying data does change, the distributions change, they want to determine when they should rechange it. Actually, Timescale plays a role in actually monitoring the quality of their inferences to basically, tell data engineers when they need to start retraining their models.

[00:52:52] JM: That's a cool application. Well, as we begin to wind down, give me your perspective on the future. What is next for Timescale? What are the problems in the next five years that you're going to be focused on?

[00:53:06] MF: I think, the biggest thing, obviously, is we spent our first couple years obviously, building our database. We have a lot of really exciting things still in the pipeline. A big thing last year, we obviously launched our horizontally scalable version of TimescaleDB. There's a lot of stuff that I think we're still building to continue to make it – we see, fast, easy, cost-effective and worry-free. We want our database to be super easy to use continue, to focus on developer experience. We want it to always be highly performant. We're continuing to have various, both short-term and longer-term R&D projects around making it more performant.

We want it to be cost effective. Again, I talked about these different types of how to do compression, how to take advantage of heterogeneous storage, to always make it super-

efficient for users, and to make it worry-free. We really want to build a highly reliable database that developers don't have to worry about.

When we kicked off Timescale, I think the title of my co-founder's blog post was When Boring is Awesome. That is, you want your database in some sense to be boring, because you don't want it to wake you up at 3 a.m. with a problem. You also want it to be awesome, because you never have to worry about it not giving you the type of scale and performance that you need.

The second part of that is really building the cloud experience to enable that still easy, fast, cost-effective and worry-free. We think that the modern developer wants an experience that looks much more like a service that they just get an end point, whether or not that end point is SQL, or even something else. I mentioned earlier, I talked about our product Prom scale, which enables Prometheus with Timescale. There, you can get SQL, or PromQL, or other interface in the future. We want to make it that they have this amazing time series experience. They don't worry about scale. They don't worry about performance. They don't worry about as much about costs and it just works.

It can really allow the developer now to focus on the application they're trying to build. In many things when trying to do all these, it will – I think we've done a lot of great things the last couple years, but I think there's a lot of great things still to do.

[00:55:31] JM: Mike, thanks for coming on the show. It's been a real pleasure talking to you.

[00:55:35] MF: Thanks a lot, Jeff. I had fun.

[END]