**EPISODE 795**

[INTRODUCTION]

**[00:00:00] JM**: Twitter's early engineers faced scalability problems that caused infrastructure failures on a regular basis. Remember the infamous Fail Whale? The Fail Whale could happen as a result of problems in the applications servers, the network, or the database layer. When Twitter was scaling in its early days, the cloud providers were still immature. Engineers did not have access to the auto-scaling cloud infrastructure that's available today. In the early Twitter infrastructure was a combination of open source tools and internally created architectures that were custom build for Twitter's workloads.

Evan Weaver was an early engineer at Twitter and he saw the deficiencies of the data tools that the company had access to. Twitter engineers wanted access to a truly reusable data platform that would fit Twitter's requirements; High-availability, globally replicated and transactionally consistent. By 2012, Evan had left Twitter and started consulting for other technology companies. He found the databases across the industry were lacking the same properties that Twitter wanted, and the ideas for FaunaDB began to percolate.

Around this time, there were two relevant papers about distributed databases that had come out; the Spanner paper from Google and the Calvin paper, a distributed systems paper from Yale. With inspiration from the literature and from his time at Twitter and from his knowledge consulting across the industry, Evan Weaver started FaunaDB. Seven years later, FaunaDB is a fully fledged open source project as well as a database company with a cloud service offering.

FaunaDB is a database for transactional workloads that are online and mission-critical. It's an OLTP database used by companies like NVIDIA, Nextdoor and Capital One. Evan joins the show to talk about his time spent scaling Twitter and the architecture of FaunaDB.

We have a few upcoming events to announce. We have a meet up at Cloudflare on the 6th of April in San Francisco, and we also have a meet up on April 6th, that is a Hackathon at App Academy in San Francisco. On April 3rd, Haseeb Qureshi will be joining me for a conversation

about cryptocurrencies, engineering, his time at Airbnb and life as a professional poker player before he became a software engineer and an investor. That will be a great evening.

You can go to softwareengineeringdaily.com/meetup to find out more about that, and our hackathon can be found at softwareengineeringdaily.com/hackathon. It's both a virtual hackathon and an in-person hackathon, and the prize purse is $5,000. So your coolest projects, your coolest hacks can be entered into win the $5,000 prize purse. You can be working on an open source project, a game, an art project. The goal of the hackathon is to check out and test out a new product that I'm working on called FindCollabs. So you can go to softwareengineeringdaily.com/hackathon or findcollabs.com/hackathon to find out more. Any feedback you have on my new product, FindCollabs, would be much appreciated.

With that, let's get to today's episode.

[SPONSOR MESSAGE]

**[00:03:35] JM**: When I start a new project, I want to move quickly. I want to be able to set up a database fast and I want to be able to change the data model easily. MongoDB was built for this reason. MongoDB is the most popular non-relational database and it's very easy to use. Whether you're working at a startup with millions of users or hacking on your own project, you're probably using MongoDB to manage your objects.

MongoDB Atlas gives you MongoDB as a service simplifying the availability, scalability, backups and everything else that you need out of a database. MongoDB Atlas was built by the team that created MongoDB. They have a decade of experience in running and managing MongoDB.

Try out MongoDB Atlas by going to MongoDB/sedaily and get a free $10 in credit. It's the perfect amount to get going with that side project that you've been meaning to build, or a new microservice within your company that you're spinning up a fresh database for. At mongodb.com/sedaily, you can find out more about MongoDB Atlas and you can also listen back to our episode with Andrew Davidson from MongoDB where we talked about how to run databases in the cloud.

Thanks to MongoDB for being a sponsor of Software Engineering Daily, and you can check out MongoDB Atlas by going to mongodb.com/sedaily and getting $10 in free credit. Thanks to MongoDB for making the database that I use the most and for supporting Software Engineering Daily.

[INTERVIEW]

**[00:05:20] JM**: Evan Weaver, you are the CEO and founder of FaunaDB. Welcome to Software Engineering Daily.

**[00:05:25] EW**: Great to be here.

**[00:05:26] JM**: You were an early infrastructure engineer at Twitter. Tell me about data management in the early days of Twitter.

**[00:05:33] EW**: So I joined in Twitter in 2008. I was employee number 15, and by the time I joined it was a total chaos, and when I left it was partial chaos. So we made a huge progress on the platform and the product and the business side in the four years I was there.

I ended up running what we called the infrastructure team and we managed all the backend distributed storage for the core business objects. So that means tweets, timelines, users, the social graph, image storage, the cache, probably some other storage that I forget. We also worked on runtime performance across the board.

If you remember in the battle days a decade ago, it was basically pre-cloud. APIs were new and weird. This idea that services would be composed was just coming back to the forefront after everyone got super burned by XML and SOA and like kind of the J2EE revolution or devolution. There's a renaissance in data systems in building scalable storage mostly predicated on social media, which Twitter and Facebook, etc., were all in the mix at the time.

My mandate was really to keep throughput going. The way to grow this site was to deliver throughput, because we're essentially redlined all the time. All the machines were running at 100% utilization. We would double the hardware footprint every year. On top of that, we deliver

about 20% increase in software efficiency every month and there was unlimited demand for tweets, especially via the API.

When I started there, I didn't know a lot about scalable databases. I basically had a performance analyst background. I worked CNET Networks before that with the team that went on to found GitHub. I was just trying to keep this side up. But to do that, we evaluated a lot of off-the-shelf data solutions, legacy sequel, like MySQL systems we were already using when I got there as well as up and coming distributed or so-called NoSQL solutions, like Mongo, Cassandra, that kind of thing.

We basically found that nothing worked. So we ended up building these sharding services that managed legacy storage engines, specifically InnoDB and Redis and that kind of thing. But me and my team, which grew overtime, we're always frustrated that there wasn't a general purpose operational data platform that could be able to scale and be reliable and be flexible for new product development. Effectively, that's what Fauna is.

**[00:08:12] JM**: All of those database that were available to Twitter at that time were not sufficient for your workloads. Why not? What was so unprecedented about the Twitter workloads?

**[00:08:24] EW**: It was effectively the scale. Twitter was a soft, still is a soft real-time system, and growth was genuinely exponential for at least the first three or four years of that part of the curve. The premise of the NoSQL revolution – For example, my team Twitter was the first user of Cassandra outside of Facebook. We fixed the build, because it didn't compile. I wrote the first tutorial. We hosted the first meet up.

The hook for Cassandra was that it could scale. You had to give up everything else. You had to give up transactions. You had to give up foreign keys. You had to give up any kind of data integrity basically to follow the original Dynamo model, but it could scale, and scale was really the limiting factor for the growth of these social platforms. Twitter in particular was wildly hardware-constrained and behind the curve in terms of planning for this level of growth, unlike Facebook, which was deploying a lot more hardware per user and initially had been managing

their growth in a more measured way going from school to school effectively following broadband penetration rather than mobile penetration the way Twitter was.

So we had kind of Twitter, Facebook leading the charge and a lot of other teams, companies, startups in the mix. We're looking at the kind of scale they had to deliver, especially deliver cost effectively and having to reject everything off-the-shelf. There were systems that extensively could scale, like MySQL cluster, or Oracle Rack, but either the scalability was very limited. Once you get beyond a dozen nodes, you're done, or the pricing didn't make sense for the advertising-driven business models, that kind of thing.

Essentially, everyone in the market at the time on the social media side of the equation, it was trying to basically buy database, whether that means through open source or otherwise. I had to do a very unusual evaluation where we had to say, "Nothing works. Nothing is even marketed to work." But what systems have the correct underlying architecture to continue to scale for the next 10X, the next 100X period of growth?

**[00:10:31] JM**: You did use Cassandra. Describe what was new about Cassandra when it came out.

**[00:10:37] EW**: Cassandra came out of Facebook. It was written at Facebook by the team that had originally done the first Dynamo implementation for Amazon shopping cart. To be clear, Cassandra and Dynamo took a very strange journey, where Dynamo has implemented AWS, or Amazon pre-AWS. Cassandra was effectively a clone of Facebook. Then because Cassandra started to take off as an independent project, when AWS started to rise up, they re-implemented Cassandra, called it DynamoDB mostly to capture some of that market in particular, because Cassandra was very difficult to operate.

But the thing that Cassandra really had going for was this homogenous scale out, and it was one of the first systems at the time that even had a prayer of offering multi-data center support where you could take commodity machines, keep adding them into a running cluster and keep scaling it up. Everything else was based on the primary replica kind of asynchronous architectures. All writes had to go through this single machine. You divertically cale it, which

became increasingly untenable, especially in the cloud, especially given the hardware capacity at that time.

I think the industry was in a very unusual spot where you had a bunch of companies on the same social media market whose growth was aggressively outstripping Moore's law. Now you might see someone scale farther up vertically on PostgreS or what have you than you did at that time. It just wasn't an option given the rate of growth of these systems, and that really created this renaissance in database technology where people said, "We're not satisfied with the RDBMS. There's a lot of good things here, but from an operational and cost and scale out perspective, it no longer is fit for purpose."

**[00:12:31] JM**: Why didn't that renaissance happen within the Cassandra ecosystem?

**[00:12:36] EW**: I mean, Cassandra was part of the renaissance, but I think you're basically asking why didn't Cassandra deliver on that long-term platform promise? I think that's a nuanced question. A lot of it is situational. These systems are just really hard to build. It's hard to build a durable, reliable, highly available database on a single node. The conventional wisdom is to be that it takes 7 million bucks to build a mature database. It turns out, that's not even really true. It's more realistic to say, "After the move to distributed systems, it takes 20, 25 million bucks to build a distributed operational database, which is actually reliable."

So you get a lot of these project, including Cassandra kicking off, and Mongo is another one. Getting some early market traction and then trying to play catch up with these basic, what we consider legacy database capabilities, like transactionality, and constraints, and durability and that kind of thing. A lot of that work was pushed on to developers to manage, and we got eventually consistent systems. We got tuneable consistency, all that kind of stuff, which basically says, "The database can actually deliver the guarantees that you're accustomed to, but we give you some primitives of what you can compose in your application to make it more less work.

I think Cassandra – Because these systems got picked up in open source early, then they became highly path-constrained by their early adopters, by the use cases that did initially work, which were all the non-mission critical ones, because they weren't durable. They weren't

transactional. That kind of distracted these teams for a really long time from investing in the fundamental architectures of the system.

At the same time, I think the thing that really changed the market's perception of what is possible in a distributed database was Google Spanner, because it's kind of conventional wisdom now. But if you remember before, Spanner was announced. I guess Cloud Spanner came out like two years ago. The Google Spanner paper came out 5 or 6 years ago. Before that, it was widely believed and also continually marketed by the NoSQL vendors that distributed ACID transactions were literally impossible. So it was kind of like this cool fusion class effort where people said, "In theory, maybe there's a way. But in practice, no one's ever going to get there," and that kind of mentality persisted, and that worse is better mindset was continually promoted until the Google Spanner came out. They said, "No, we actually did it." That sort of planted the seed for a bunch of projects internal and external to start pursuing higher levels of durability and consistency in a distributed scale out data platform.

**[00:15:19] JM**: We'll get to talk about Spanner a little bit more, but I want to get back to your experience at Twitter so that we can make it concrete for people what the consequences of these failures at the database layer were. What was going wrong in the data layer and how did those failures cascade to user level problems?

**[00:15:43] EW**: So Twitter's data layer evolved through a classic series of steps. Initially, it was a single MySQL box. I think it was hosted in Joyent. Literally a single box, a hot spare replica that wasn't otherwise used. Then you started reading form the spares and you start vertically partitioning the tables to get more capacity per table and separate clusters. Then you start doing some temporal partition. You end up with a system sharded by hand six ways from Sunday.

The next step we had to take that system to is the true sharded architecture kind of similar to Vitess. So we had a system called Gizzard internally, which was basically equivalent to Vitess, which managed sharding across a bunch of commodity boxes, but node replacement was still effectively manual.

If you noticed, at each step of this trajectory, the system becomes more special purpose, more focused on just servicing the queries that Twitter needed at that time to scale, and it did scale,

and it was superfast, especially the timeline system was incredibly fast and incredibly efficient, which these systems are still used today, like the social graph and the timeline system for that reason. But the thing that they really didn't deliver was not the scale, it was the flexibility. Because if you don't have a general purpose platform, it becomes very difficult to develop new products in a cost effective way. Experiments become very costly, because you have to deliver them all at scale with special purpose distribution code just for that new feature or that new product line.

I think that's the biggest thing that was lost along that journey. We had no choice at the time. We had to keep this side up, but we didn't have the investment horizon from the business to really build that general purpose reusable platform that would let the developer work completely abstractly from the underlying scalability and consistency primitives of their dataset.

**[00:17:40] JM**: You started FaunaDB in 2012. Describe the initial spec for the database you wanted to build.

**[00:17:48] EW**: We started the company as a consulting company. We had built all these systems at Twitter, but we didn't want to just replicate the social graph system in some kind of commercial context. We weren't convinced that was a brought enough use case. So we wanted to explore the data market generally, and we did consulting with a lot of startups in particular and some larger companies trying to understand what are non-Twitter class data problems. We know what Twitter needed. We know what Facebook needed, but what is everyone need to get both scale and product flexibility long-term?

We discovered that the market was suffering from – Effectively, part of this worse is better mindset, in particular, polyglot persistence, where nothing is really good in general, but we'll try to glue together five different systems so that at least, in aggregate, all the checkboxes are checked, but they don't compose. So there'd always be like a MySQL or a PostgreS for transactional workloads. Then there'd be some table that didn't scale that was moved into a key value store, like Cassandra, or Dynamo. Then there'd be something that needed to be cached. So it'd be copied into Redis or memcache, and then there'd be a bus, like Kafka to try to glue it together, and be Elasticsearch to enable different indexing patterns.

You exit this process, which is a completely rational process, but your dataset is duplicated 15 times across five different clusters. You're spending a million bucks on Amazon a year and your engineering productivity has gone under water because you spend all your time on maintenance and integration. We said, "This isn't necessary. Information science doesn't demand that all these systems are separate. The problem here is that there's no underlying platform, which has solved the CS problem of delivering them together in industry."

A lot of the academic literature too was still at kind of this point solution mindset. We took this one system, we iterated on it, modified it a little bit. We made it better along this one dimension and then we just left it. So the general purpose nature of the original transactional database was totally lost.

After we saw that pattern repeated several times, we decided if we didn't build it, it was never going to get done. We moved into a product prototyping and product development and we raised venture capital in 2016. We raised our seed round from CRV. Then in the end of 2017, we raised our Series A from Google Ventures and Point72.

[SPONSOR MESSAGE]

**[00:20:38] JM**: Today's episode is sponsored by Datadog, a cloud scale monitoring service that provides comprehensive visibility into cloud, hybrid and multi-cloud environments with over 250 integrations. Datadog unifies your metrics, your logs and your distributed request traces in one platform so that you can investigate and troubleshoot issues across every layer of your stack.

Use Datadog to rich customizable dashboards and algorithmic alerts to ensure redundancy across multi-cloud deployments and monitor cloud migrations in real-time. Start a free trial today and Datadog will send you a t-shirt. You can visit softwareengineeringdaily.com/datadog for more details. That's softwareengineeringdaily.com/datadog and you will get a free t-shirt for trying out Datadog.

Thanks to Datadog for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

**[00:21:41] JM**: There is a term you're using here, data platform, and this is a term that's come up more and more in the last couple of years in the interviews that we've had on the show. Is there a notable difference between the terms database and data platform?

**[00:22:00] EW**: Yeah, there is, and that's probably the Genesis of Fauna and it's probably the reason for the name. At Twitter, we had a lot of product features and product lines that were backed by individual distributed or sharded databases. But if you look at what's fundamental to a platform and what were really made, SQL and Oracle system our back in the days, so revolutionary is that you got to leverage from composing multiple applications and multiple features in the same system. You could integrate through the dataset. You could rely on the database to enforce fundamental dimensions of your business model. That made it much easier for everyone else upstream to do your job.

I think the difference between a database and a data platform is whether the system has leverage, and we've put a lot of effort Fauna to make sure that that leverages there based on our experience at Twitter in particular. For example, the multi-tenancy system, the quality of service system. We offer Fauna as a serverless cloud product as well as an on-prem, or managed cloud product.

But our serverless cloud is not deploying a VM for each person who signs up. It's one global Fauna cluster, and we dynamically provision a tenancy within it. Then that gives us tremendous leverage operationally, because we can scale an aggregate instead of individually. If it was composed with individual databases the way RDS or something, and Amazon currently is, you don't get any sharing of resources. You don't get sharing of operational burden, and it's also worst for your users, because now it's expensive to create a database that's a fixed cost. That fix cost destroys your leverage.

**[00:23:43] JM**: To get into what you did to build FaunaDB, we should talk about some of the important papers that came out in 2012 that relate to FaunaDB. There's the Spanner paper from Google, which you mentioned earlier, and the Calvin paper, which I had heard less about before doing this show. Explain the relevance of these two papers, Calvin and Spanner.

**[00:24:09] EW**: So pre-Spanner, it was widely believed that distributed consistent transactions were impossible. The way Spanner saw this problem was by synchronizing multiple data centers on atomic clocks. There are effectively three generations now of distributed transactional architecture. The first, which also came from Google, is Percolator. Percolator basically doesn't scale beyond a single data center. There's one machine, which issues timestamp to all readers and writers. If you're not in the local data center to that machine, you have to go to that data center to get the timestamp. So effectively equivalent to your primary replica, traditional mode of replicating a SQL database, but scaled up to the data center scale.

What Spanner did was say, "We can get these timestamps via physical atomic clock hardware, and if we know the boundaries in which these clocks can drift, we can guarantee – In particular, we can guarantee strict serializablity or what they call external consistency, which is the highest level of transactional isolation.

This was mind-blowing for the industry, because it now let you do fast reads and writes with ACID isolation in a truly global, truly replicated, truly no single point of failure way. But at the same time, the Spanner paper were being published, the Calvin paper was being published by Daniel Abadi's team at Yale at the time. Calvin inverted the model and said rather than synchronizing the transactions on the time, what if we bundle transactions up ahead of time and we effectively synchronize the time on the transactions, which have been bundled together.

The big benefit of this attitude towards the replication of the transactions themselves is that it's no longer relying on this physical hardware. You don't have to operate anything special. You don't have to understand your clock skew. You don't have to manage anything. It's a pure software solution.

But the Calvin paper was basically ignored at the time. I think there are a couple of reasons for that. One reason is that the way the replication properties were articulated was in the context of being global, and people interpreted that to mean that it was similar to Percolator. Somewhere there's a single machine, which everything had to synchronize on rather than correctly interpreting it as being logically global saying, "We're creating a global order, but in implementation, there's nothing global about it at all. It's partitioned and scalable and highly available just like you would want to everything to be."

The second thing about it was that Calvin requires transactions to be essentially pre-declared before they're submitted to the database for execution. So it's a little difficult to adapt a traditional SQL model to it, although it is possible, because SQL is built around a session transaction model, where you open a transaction, read some stuff, your application messes with the data, write some stuff, read some more stuff, application interferes again. Eventually, it calls commit.

Calvin requires the transaction to be pre-defined. So that means you can't open and close a transaction at separate periods of time, and I can talk about how we work around this in Fauna. But basically, this meant people took the paper and said, "Oh, that's a cool idea, but it's not really applicable to the real world. We don't really understand how it's different than Percolator." It wasn't until we picked up in industry looking for what's genuinely best fit for purpose for a cloud native multi-data center scale out in a SQL solution that is starting to get attention again.

**[00:27:54] JM**: Just so we don't burry the high-level motivation for what we're talking about here, explain how these innovations that were coming out of these papers, how did they offer solutions to the problems of social network database scalability that you're thinking about when you were encountering these problems at Twitter?

**[00:28:21] EW**: So you cannot solve your productivity problems in our opinion unless you can rely on your database to be correct. Twitter has an internal project called Manhattan, which there's a similar goals to Fauna on the outside for the same reason.

If you think about writing a traditional application, the burden the developer has to take on if they can't rely on their database to essentially work is tremendous. You end up dealing with quorum consistency guarantees or lack thereof. CRDTs and other complex data types, which have nothing to do with the actual product goals of the products you're building.

But that's the burden infrastructure teams at these companies had to take on, and the outcome was special purpose systems, which took these eventually consistent primitives and presented them in a way that made sense for those specific product lines, but were not broadly reusable, did not offer composability, did not offer isolation in all the different senses of isolation, including

isolation of transactions and isolation of workloads and isolation of data centers, what have you, to their development teams.

This actually worse with the microservices and the container revolution, because now you don't have just one application, which is taking on some of the database level consistency concerns. You have 10, you have 50, you have 200, which are all trying to agree on how to handle failures in the underlying ability of the data platform to actually present a consistent view of the data. So the scale problem got solved, but this productivity problem did not get solved. Fundamental to this productivity problem is being able to rely on your database and your data platform to present a consistent view of reality to all the applications and service, which are integrating to that data.

**[00:30:23] JM**: There are so many databases that get classified under the vague category of NewSQL, and we've done shows on these. CockroachDB, Spanner, VaultDB, TiDB, now Fauna DB. For the average software engineer, this is really daunting to try to understand the subtleties between these different databases. Can you give us a rubric for examining tradeoffs in these different NewSQL databases?

**[00:30:59] EW**: That's a really good question. I agree. The market, at least superficially, is crowded and also confusing, and NewSQL is kind of a funny name, because people include us – I think, correctly include us in this category, even though we're not actually SQL. Fauna is a relational NoSQL database. We don't offer SQL syntax, but we do offer [inaudible 00:31:23] transactions in Fauna's native query language, which let you accomplish anything you would traditionally accomplish with the ORM backed by a SQL database.

But, effectively, your valuation criteria are twofold. There's a tradeoff between traditionally scale and high-availability, which you're trying to minimize. You want maximum scale and maximum high-availability, and then there's another dimension of transactional correctness that you're trying to maximize. These systems all kind of plot out different points on this multidimensional envelope of tradeoffs.

But what we're trying to do is find what is the absolute peak of maximizing availability, maximizing scale and never giving up transactional isolation and correctness, which is

effectively useful for any mission-critical data application, any data that's irreplaceable that relies on transactionality and consistency.

So you end up with the three models of transaction, transaction, consensus, dimension, where Percolator is effectively correct, although many of the implementations of it are not or don't need the potential guarantees of a Percolator model. But it gives a multi-data center scale out, and that would include TiDB, FoundationDB in that category.

You also get the Spanner-base systems, which increase scale, increase availability because they don't have a single point of failure around the clock oracle, but it comes at the cost of operational overhead and fundamental data integrity, because if you don't manage the clock skews properly, if you're not in an environment where the end-to-end software and infrastructure stack is tightly controlled, like it is in Google Spanner, you never really know if you're delivering the level of guarantee that you hope to guarantee. Because if the clocks drift, by definition, they are the synchronization point. If they drift, you don't know, because you don't have anything else to synchronize on. Eventually, you can detect that the drift has exceeded your tolerance. At that point, you don't know how long you've been in this pathological state where transactional correctness was not actually enforced.

So there're systems like Cockroach, although Cockroach is actually deviated quite a bit from Spanner specifically to work around some of these issues, but not all, where translating model from Google Spanner in Google Cloud to the public cloud, which is much less reliable. It has a lot of downsides. We like to believe we're the third generation of architecture here with Calvin, which has found a model where, at least as far as these three dimensions are concerned, you don't have to give anything up. You get your strict serializability all the time even if clock skew, it never affects data correctness. You get your scale out homogenous SQL style architecture, where you can always add new data centers, you can always add new nodes. It doesn't impact availability. It always improves throughput.

You also get one of the things which only Fauna can do outside of Google Spanner and Google Spanner's controlled environment, which is you get fast snapshot reads from any data center. So you effectively get a physician on the availability curve, which is otherwise not available, I guess, where you can read form the system as if it's a NoSQL database that doesn't coordinate

at all. You get superfast, single millisecond reads from every data center, distribute your data all around the world, but you never give up transactional correctness the way you do with something like Cassandra.

On the one hand, you get your traditional experience on the correctness side. It's never going to be violated even if your operations go haywire, but you get a scale out and high-availability experience, which is effectively indistinguishable from a NoSQL database. Fauna can lose a minority of the data centers and still maintain liveness for write. So if you want to tolerate losing three data centers, deploy seven and you're good to go.

**[00:35:34] JM**: The penalty of using atomic clocks to maintain the consistency and the serializability of your transactions that you referred to in the Spanner paper has, does atomic clock skew, does that happen regularly?

**[00:35:51] EW**: Presumably not. I mean, it takes time to read information, but barring quantum entangled PCI express cards that you could slot in, there's no better way to synchronize on the physical world than managing the drift of atomic clocks.

To be clear, atomic clocks do drift. Someone has to set the time of the clock, but the drift is so small that you can manage those windows of ambiguity down to – Initially, when the Spanner paper came out, it was nine milliseconds or something like that. I'm told they've pushed it lower down the two to three milliseconds range now.

But the problem is that's just the beginning. Just having a clock, which is well-synchronized doesn't mean that your application can access it in a predictable amount of time. You have the entire software stack above that as well as the hardware stack that has to be completely controlled. That's probably why, in my opinion, Google Spanner is relatively expensive, because you can saturate these machines, because if you have some kind of VM migration, if you have a stall in the page cache behavior, if you have anything else interrupt, these systems are not hard real-time systems. They're Linux with many layers of service, service stacks built on top of them.

If anything interrupts, the flow of data in a way that delays accessing what the current timestamp is, you'll lose correctness. That's the thing which is literally impossible to control in the public cloud, especially if you want to move a multi-cloud or hybrid cloud world. You can't use atomic clocks even if they are sensibly available if they're from different vendors, and you have no way to know if you're actually meeting your tolerance guarantees. You can know in a negative sense, you can know you've gruesomely failed them, but you can never affirmatively approve that you're actually meeting these guarantees.

[SPONSOR MESSAGE]

**[00:37:59] JM**: Logi Analytics is an embedded business intelligence tool. It allows you to make dashboards and reports embedded in your applications. Create, deploy and constantly improve your analytic applications that engage users and drive revenue.

You focus on building at the best applications for your users while Logi gets you there faster and keeps you competitive. Logi Analytics is use by over 1,800 teams, including Verizon, Cisco, GoDaddy and J.P. Morgan Chase. Check it out by going to L-O-G-Ianalytics.com/datascience. That's logianalytics.com/datascience.

Logi can be used to maintain your brand while keeping a consistent familiar and branded user interface so that your users don't feel like they're out of place. It's an embedded analytics tool. You can extend your application with advanced APIS, you can create custom experiences for all your users and you can deliver a platform that's tailored to meet specific customer needs, and you could do all that with Logi Analytics.

Logianalytics.com/datascience to find out more, and thank you to Logi Analytics.

[INTERVIEW CONTINUED]

**[00:39:28] JM**: For the people who are listening to this with the Twitter use case in mind, they're probably thinking like, "Does this really matter that much if a tweet gets out of order because of some atomic clock skew edge case?" They're thinking, "Twitter is not going to care about the

atomic clock skew edge case. They're going to flip a coin and they're going to pick Spanner or FaunaDB and it's not really going to matter for them. That may or may not be true.

But my sense is that there are applications for which this kind of edge case behavior is extraordinarily important. You think about life and death type of software applications where you would want a distributed, high available, consistent database. Can you talk about what is the application category for which these properties are so important? How big is that application category and what exactly are we talking about here?

**[00:40:26] EW**: Yeah. I think their perspective on Twitter is superficially correct.

**[00:40:32] JM**: Right, exactly.

**[00:40:33] EW**: If tweets are late, fine. If you reply to someone and someone else sees the reply before they see the original tweet, will they live? Sure, they'll live. But the impact within Twitter not having this level of data integrity in the core platform was detrimental twofold. The support burden was much higher, because you would have people who just didn't get the product experience they were expecting. The canonical example of this is user privacy. Someone sets their account to be private and then post something they don't want someone else to see. If those two actions aren't strictly serializable, someone else could see the thing before they get the privacy change.

Then you violated your contract, your contract of trust with the user and they lose confidence in the platform. You've caused them whatever kind of personal problem they're trying avoid by making this change. But at the same time there are aspects on the system which really rely on transactionality, and for that reason remained in legacy systems much longer, because these primitives weren't available. An obvious one is username registration. If you have people trying to acquire the same unique stream name, you have to pick a winner. You can't clean it up after the fact.

To your broader point, most of the work we do at Fauna is in domain, so it's very obvious that this level of correctness is critical. So a lot of work with financial services, maintaining ledgers, maintaining balance movement, whether that's for credit, or for cash, or for equity holding, what

have you. It has to be right, or a user generated content, where it's irreplaceable if you lose it. So that includes identity, accounts, that kind of traditional content, especially in some of the work we've been doing with NVIDIA and their GForce platform space.

Also, ecommerce is a very obvious vertical, where you have purchasing and inventory decisions being made, and that can be you bought something online that has to be shipped to you or could be your competing to reserve, a concert ticket or a hotel room, for which there's only one with other people who are trying to do the same thing at the same time for the same price.

I think, broadly though, it's kind of like asking, "Do you want a strong or weak memory model? Do you want replicated disks or not? Can you work around the issue in any specific domain?" Sure. Probably. That's kind of why NoSQL got started the way it did giving up everything but scale in the interest of keeping these products running. But you just don't want to, because it's always better to have a more trustworthy platform to build your product on.

**[00:43:28] JM**: Let's say we want to deploy FaunaDB and we want to host a brand new company called SE Daily Twitter. So it's just like Twitter, except it's for Software Engineering Daily listeners only, and we want to be architected to scale even on day one. Describe the architecture of this FaunaDB cluster that we're going to deploy for SE Daily Twitter.

**[00:43:53] EW**: So you have two options. You can use Fauna serverless cloud, which is pay as you go utility pricing, or pre-pay for a discount, the usual stuff. That gives you immediate access to a global cluster of Fauna nodes, which expands AWS and GCP infrastructure. Give you a low latency experience for your global user base, the global listenership of SE Daily. Then you can rely on us to scale it up behind the scenes as your usage grows and grows and grows. Or say you know you're launching out the gate to the hundreds of millions of listeners of the podcast. You want to operate your own infrastructure for compliance reasons, security reasons, that kind of thing. You want to specialize the hardware profile to make sure you get the maximum bang for your buck.

You can run Fauna in your own private cloud or on your own physical hardware. So you can do the same thing we did in our cloud. You can get different regions from different cloud providers.

You can spin up the Fauna JAR, because it's just a Java JAR. There's no service dependencies or other dependencies beyond the JVM, whatsoever.

On each machine you provision in each data center. Tell them where the other ones are. Let the cluster cohere itself and then you'll get the exact same access to a data platform that you now control that you do for using Fauna cloud. You can provision multiple tenancy context, you can have your production and your staging and your test database is all running on the same hardware. You can scale it up and down transparently from the application, all that kind of thing.

I think the big difference here is you don't just get a key value store that's global or relational store that's locked to a single data center. You get a system that does both. So your entire end-to-end application can be architected for a global user base with low latency local access to the dataset from day one.

**[00:45:52] JM**: Now let's say I send out a tweet and that tweet gets written to the FaunaDB instance that I'm communicating with. How does that write get propagated to the other instances? I know we can't go super deeply into the detailed communication protocol between just beyond the scope of the podcast, but give me an overview for the write path.

**[00:46:19] EW**: This is one of the critical innovations of Calvin, because in your traditional relational database or your Spanner system, in your Percolator systems, the transaction execution and transaction replication are intermingled. So Spanner, for example, elects a leader for each partition and every key within that partition has to commit to that leader. Those leaders are in different data centers, tough. You have to go coordinate with them, [inaudible 00:46:48], come back, result the transaction, go back, cleanup those [inaudible 00:46:52].

What Fauna does is separate the transaction log from the transaction replicas. So if you have a global cluster, you can configure your log to be as in many data centers in that cluster as you please. All transactions enter into the system and get committed to this log in a durable, replicated, highly available way.

Your latency profile for writes is to the majority set of that log that's nearest to you. So if you have a North American cluster, you're typically looking at in the 50 to 80 millisecond range per

transaction commit. All the replicas are reading off this log and applying those transactions in a completely uncoordinated way locally. So as soon as the transaction hits the local replica, it's made visible in that local replica and local clients can see it.

So you have essentially a semi-synchronous pipeline that only involves one global round trip on writes and no round trips whatsoever outside the local data center on reads. That's what really gives you the unique scale out profile of Fauna and Calvin.

**[00:48:04] JM**: Now, I would love to go deeper on this SE Daily Twitter example, but I know we're running low on time. So I'd like to zoom out and talk a little bit about your experience building this business. I think it's been 7 years if you include the time spent on your sort of consulting expedition. Can you just tell me about what it's like to build a database from scratch and then build a business around that? Why is that hard?

**[00:48:35] EW**: Yeah. I mean, both of those things are tremendously hard, and that's why there's very few entrance in this market to begin with. We basically have Spanner and a handful of startups fewer than one hand who've actually delivered a product to market at all. Amazon doesn't have a system which can guarantee global transactional isolation. Azure doesn't have a system. Essentially comes to the fact that this is a deep tech project, takes real CS innovation. You're not just applying existing patterns or putting forms online, although there are many Twitter, for one, examples of putting forms online that are tremendously important and successful. But at the same time, the audience for operational database is very risk averse, especially after people got badly burned by some of the vendors in the initial NoSQL days, like Cassandra, for example, and Mongo and that kind of thing.

So we're at a point where the industry knows how to evaluate whether these systems are actually correct and will no longer take it on faith from the vendor. But the burden of building them to be correct is tremendously high, because it's cutting edge, it's cutting edge research both on the academic side and on the industry side.

**[00:49:54] JM**: Tell me more about how you differentiate in your go-to-market strategy, because I've talked to a lot of people who are building various services that they're selling to developers or they're selling to, I guess, the CIO at a large bank, for example. The go-to-market –

Obviously, this is not like a “business podcast,” but the go-to-market experience for selling developer tools is really interesting and it’s kind of an art and a science that hasn’t really been talked about very much. So what have you learned that you can share with the audience?

**[00:50:29] EW**: We pursued kind of a barbell model. We offer Fauna and serverless cloud for the individual developer, for the hobbyist, for the new startup with only a handful of people. We also offer the on-prem or the enterprise edition, for the big business that has existing product lines that’s running enterprise style decision process, and our sales model internally reflect both of those things.

But I think one thing that differentiates our market from your traditional – Now, traditional bottoms up kind of grassroots developer adaption path for something like an orchestration or that kind of thing, is that for us, our customers are all making the same type of decision, because no matter how small you are, you still want your database to be correct and you want it to scale, because you aspire to grow your product big.

In some ways that makes our job easier, because can speak the language to the entire range of the market. But I think in some sense, the usual conventional wisdom applies, where you want to make your system easy to get started with, free to try, easy to create initial experiences of success in the developer community. Then that can translate into the opportunity to compete for enterprise deals or to get pulled along with a small startup that’s growing fast.

I think making a product that works, that meets a defined need, that’s easy to use, easy to adapt, has flexible pricing, those things never really change. One of the things that has changed, and I think effects, in particular, the open source community a lot, is we see a strong movement in the small and mid-market category of customers to manage cloud services.

So I think if you’re starting a new business now, it behooves you to make sure that you launch a cloud service out of the gate, because if it’s open source or what have you, you may get mind share, you may get adaption, but you’re just not going to capture the long-term business relationships with the mid-market and startups who are looking to outsource the operational burden and now even transform kind of the TCO experience by moving to serverless and utility price models, which are directly aligned with their business value.

One example of this on the analytics side is obviously Snowflake, where they're taking advantage of cloud economics similar to the way Fauna cloud is and delivering a fundamentally better experience across the board. We see their pattern continuing to penetrate further and further into the enterprise, and it will be a long time, if ever, for all these legacy systems and legacy buy-in patterns are gone. But everyone's building for a cloud world now too, and if you're launching a developer tool and you don't do that, you're not going to get very far.

**[00:53:21] JM**: Evan Weaver, thanks for coming on Software Engineering Daily. It's been really run talking.

**[00:53:24] JM**: You too. Thanks for the opportunity.

[END OF INTERVIEW]

**[00:53:30] JM**: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[END]