EPISODE 1057

[INTRODUCTION]

[00:00:00] JM: A relational database often holds critical operational data for a company including user names and financial information. Since this data is so important, a relational database must be architected to avoid data loss. Relational databases need to be a distributed system in order to provide the fault tolerance necessary for production use cases. If a database node goes down, the database must be able to recover smoothly without data loss, and this requires having all of the data in the database replicated beyond a single node. If you write to a distributed transactional database, that write must be propagated to each of the other nodes in the database. If you read from a distributed database, that read must return the same data that any other database reader would see, and these constraints can be satisfied differently depending on the design of the database system. As a result, there is a vast market of distributed databases out there, from cloud providers to software vendors.

CockroachDB is an open source globally consistent relational database. CockroachDB is heavily informed by Google Spanner, which is the relational database that Google uses for much of its transactional workloads. Peter Mattis is a cofounder of CockroachDB and he joins the show to discuss the architecture of CockroachDB. The process of building a business around a database and his memories working on distributed systems at Google.

Full disclosure; CockroachDB is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:41] JM: When a large percentage of the population goes into quarantine, the dynamics of Internet traffic change. Some companies need to scale down quickly in order to save money, while other companies like streamers and ecommerce retailers are scrambling to keep up with unprecedented demand.

CockroachDB is a distributed SQL database that makes it simple to build resilient, scalable applications quickly. CockroachDB is Postgres compatible, giving the same familiar SQL

interface database developers have used for years. But unlike older databases, scaling with CockroachDB is handled within the database itself so you don't need to manage shards from your own client application. Because the data is distributed, you won't lose data if a machine or a data center goes down. CockroachDB is resilient and adaptable to any environment. You can host it on-prem, you can run it in a hybrid cloud, and you can even deploy it across multiple clouds. Some of the world's largest banks and massive online retailers and popular gaming platforms and developers from companies of all sizes trust CockroachDB with their critical data.

Sign up for a free 30-day trial and get a free T-shirt at cockroachlabs.com/sedaily. That's cockroachlabs.com/sedaily.

[INTERVIEW]

[00:03:12] JM: Peter, welcome to the show.

[00:03:14] PM: Glad to be here. Thanks for having me.

[00:03:15] JM: You were at Google for almost a decade and some of that time was spent working on the file system; Colossus. Today, you work on CockroachDB. How does building a file system compare to building a database?

[00:03:30] PM: It's a lot easier for two reasons. I mean, the surface area of a file system is much smaller than the surface area of a database. The thing that was a huge challenge with Colossus is everything inside Google is about enormous scale. You think externally like, "Oh! We're going to have a cluster machine, maybe a hundred machines, or a thousand," and Google is like, "Tens of thousands of machines. That's what we need to scale out to." Now it's like hundreds of thousands of machine. Your systems are running in enormous scale, and scale was just like the thing to be concerned about all the time.

But on the surface area, what I'm referring to as the surface area is just the number of API calls and API entry points and the number of operations that a file system performs is much more limited than a database and especially much more limited than a SQL database. A SQL database like CockroachDB has a gigantic surface area. You're thinking of file system. You open

files, you read some bytes, you write some bytes, you have to worry about – For Colossus, we had to worry about replication and availability and redundancy. We still have all that replication availability redundancy prompts inside CockroachDB, but then we have to deal with all the semantics of SQL. We have to deal with the SQL execution, the SQL optimizer. We have more stringent backup restore requirements importing from other systems. Just the complexity of CockroachDB is much, much higher than it is for Colossus.

[00:04:45] JM: When you were at Google, you worked with Spanner and Bigtable. Tell me about the use case for those respective data stores.

[00:04:55] PM: Yeah. The original use case for Bigtable was used in the indexing pipeline at Google, and for the indexing pipeline, it wasn't really a transactional system per se. They're kind of doing these boatloads and operations and then perform the implementations as part of the indexing pipeline process. Yet, very quickly, other developers inside Google saw this as being like, "Hey, can we use this for other purposes?" and they started kind of extending and asking for additional functionality that made them more appropriate to use for end-user applications. The indexing pipeline is really an end-user application. It's a batch process that takes place inside Google.

We started seeing use cases where people are using Bigtable. Trying to saying like, "Oh, can I put Gmail on top of Bigtable?" Just you know, that I worked on Gmail early on. I was one of the early developers working on Gmail. So I was very familiar with that backend, and it was not built on top of Bigtable. Bigtable came about after the initial Gmail architecture was built.

But as soon as Bigtable came about, people are like, "Ah! I want to start using this. It seems like a great way to do things. It scales horizontally. It gives me kind of the API stuff I need to build an application on top of," and yet the very quickly started running to the friction points too. The frictions points doesn't have transactions. It has these kind of semantics that were not quite exactly what applications wanted, and that's where the initial Bigtable authors went back to the drawing board and said like, "Ah! We really, really need to do something that's more like a traditional database," and Spanner emerged from there.

Spanner, they really were like, “Ah! Okay, let's sit down and think about how we really want to build this thing for the types of applications that people are struggling to use Bigtable for.” There’s actually intermediary system here called Megastore. Megastore was kind of tacked on top of Bigtable and it added some of that transactionality and indexing functionality that an application might want on top of Bigtable, and yet it was kind of glommed on. It wasn’t kind of in the grids. You can almost see Spanner as reimagining of what corporating Megastore into Bittable from the get go would look like. It's quite a bit more than that, but that is like one way you can do it, but it’s a complete rewrite of those systems.

Very quickly, the ADS database at the time was running into scale problems and the ADS people are like, “Hey, we would either replace it with something else. Can we actually use Spanner for this purpose?” Originally, Spanner didn't have a SQL interface to it and the ADS folks needed this SQL interface. There's no way they're going to convert their existing system to something that didn't have a SQL interface. They went ahead and implement this system called F1 that sat on top of Spanner. Eventually, this got enough usage that Spanner and F1 people who were working very closely together just merged the two together. I'm not actually sure now. I’ve been gone from Google for a number of years. I’m not sure if F1 actually exists separately from Spanner now. I know Spanner has SQL functionality just built into it.

[00:07:40] JM: You mentioned that would Bigtable, there was an inability to have transactions. What does that mean? I think of a transaction as fundamental to a data storage system.

[00:07:52] PM: Yeah, file systems or data storage systems. They kind of have transactions in a funny way. They're very limited, but it’s not what you traditionally call a transaction. I think when I use the term transaction, that you can have an arbitrary set of operations. There’s some amount of isolation between concurrent transactions. So that either transactions will get aborted if there's conflicts or there will be some kind of locking mechanism present inside of it. The transactions you get from a file system don't really have that. You have some amount of atomicity you get from certain operations like rename. But you don't get like real kind of full-fledged transactions. Bigtable did not have kind of full-fledged transactions. There was no notion of flocking. The application itself would have to be structured in such a way that it won't create conflicting operations, or if it did have conflicting operations, that you would be able to recover from those.

One of the ways you do that is there's often natural sharding that takes place inside an application. To give you a concrete example of that is an application like Gmail. The natural sharding of Gmail is on a user basis. If you were to implement Gmail on top of Bigtable, one thing you could do is just kind of say like, "Oh! Here is my frontend server that is talking to Bigtable, and I'm going to kind of have some external locking mechanism that ensures that only one of my frontend servers can actually be manipulating a user's data at a time." That kind of locks out all that data even though it's external from Bigtable itself.

[00:09:18] JM: This was back in the early 2000s, and you're talking about data infrastructure inside of Google. Now, outside of Google, there were plenty of people who had large database deployments but did not have the Google infrastructure. I've done some interviews with companies in that vintage, and a common problem that I hear in the scalability issues was that they would have to handle the scalability with client sharding. The client would have to talk to individual database shards. So the internal database infrastructure wouldn't take care of the scalability. You actually had to defer some of the shard management or the shard communications to the people who are just like accessing the database as like internal service developers. Is that consistent with your understanding of what was going on outside of Google? The problems that people were encountering managing database infrastructure?

[00:10:25] PM: Certainly, and it was actually taking place inside Google as well. One of the systems inside Google that used databases heavily or traditional SQL databases was the ad serving system, and this was built on top of MySQL. They actually had sharded MySQL and actually changed the number of shards over time that's resharding operation, and that resharding operation inside Google, outside Google, wherever it happens, is usually incredibly painful.

This was one of the motivations for that ad system to actually eventually migrate on to Spanner, is they were just getting tired of handling – I think they went from tens of shards of MySQL, to hundreds, to eventually thousands, and they changed things. Went back to hundreds, and it was about up to thousands again. But just kind of a nightmare of complexity of dealing with that kind of sharding at the application layer.

I mean, really, this is one of the kind of core tenants of CockroachDB, is we feel that if you're having to do this sharding work external to the database, you're essentially pulling some of the database functionality into your application. Taking that burden on of implementing a database inside your application and often times you're giving up things. Very frequently, you don't have cross-shard transactions, so your transactions have to be limited to a single shard. You're not dealing with the easy scalability the same way that a database itself might deal with it internally. This is just too much of a burden for application developers.

[00:11:42] JM: You cofounded Cockroach Labs in 2015, and this is long after you were working at Google. This is like as late as 2015, people are still having issues with this application level sharding with these, just relational database management problems. As late as 2015, why wasn't this problem solved?

[00:12:09] PM: I mean, that's a good question. Why wasn't this problem solved? I'm not sure why it wasn't solved. Certainly, one of the reasons we actually founded Cockroach Labs is after Google, my cofounders and I, we went off to another startup. Then we eventually wound up at Square, and at Square we saw some of this exact same pain points in a monolithic MySQL database. They were trying to figure out how to pull bits of it to keep it scaling further. They're having to buy increasingly large hardware to run this database on, and they're eventually looking how to shard it and we're like, "Ah! Why is this still happy?" That was 2014. Why is this still happening in 2014?

We took a look around, and there is essentially people who are looking at NoSQL systems and being like, "Ah! I got this horizontal scalability, but it doesn't have the functionality applications most applications want. They don't have transactions. They don't have indexing. All the niceties you would get from traditional SQL database." The other end of the spectrum, we had the SQL databases, and the SQL databases are like, "Oh! We'll give you all these functionality, but sorry, you're going to have to do sharding eventually when you get to a certain scale." Why wasn't there some marriage in the middle? I'm not quite sure.

It feels like Spanner was leading the way in what we should do, and we came around at the right time identifying the fact that this is now possible. Spanner is like proof point that this is

possible where previously people said, “No, if you want to have the scalability, you have to give up all these functionality,” and the answer is, “Really, no you don't.”

I think some things didn't actually change technology-wise. Networks are superfast now. I mean, historically, when SQL databases were being developed, computer networks were very slow. Now, communication between two machines within a data center is faster than accessing your local disk. Computer networks are very fast. They continue to get faster.

Another advancement is advances in replication protocols. Paxos has been around for whatever it is, 20, 25 years now. Notoriously hard to implement, but we've increased [inaudible 00:14:02] dramatically over that time. Raft is a similar consensus protocol, but even since 2015, the advancement in the academic understanding of consensus protocols, it feels like there's just been this Cambrian explosion of understanding of these protocols and really understanding that Paxos, Raft, all these other ones that academics have come up are just different points on this spectrum of replication protocols.

[00:14:27] JM: With all those incremental advancements, improvements in networking, improvements in consensus. I mean, CockroachDB is commonly referred to as an open source version of Spanner, but there is so much in the way of different design decisions you could make if we talk about, again, these incremental improvements to infrastructure. How much has the design of CockroachDB diverged from what you originally built when you were looking at the Spanner paper as something of a reference implementation?

[00:15:05] PM: Yeah. Well, the most obvious way diverged is when we first got started five years ago. SQL was not on our roadmap. We were looking at trying to adapt and understand what the Spanner API was, which they don't really go into at in depth in the Spanner paper. But looking at some kind of keyvalue API, we’re taking a look at Cassandra and some of the other systems out there, the NoSQL systems, and seeing if can adapt it and then introduce transactions and more full-fledged indexes. The big divergence is at some point along the time we’re like, “Actually, what we should do is embrace SQL,” and I think I was a fantastic decision in hindsight. Didn't realize how monumental when we made that decision.

But above and beyond that, like the description in the Spanner paper is high-level. When you actually get down to the nitty-gritty of like the bits and pieces, I think there's like still a lot of similarities with Spanner. But saying that we're an open source Spanner, I would say maybe more were inspired by the Spanner design, and they certainly blazed the path that we walked in for a while, but now we've gone off on our own path.

[00:16:08] JM: Why is it – That original Spanner paper, they had their own API. Why did that database have its own API? Why didn't they implement it with SQL from the outset? Is there something about having a fully SQL compliant interface that's difficult to build?

[00:16:26] PM: Yeah. It's a huge monumental task. Now you can look and see if there're components you can pull off-the-shelf. Can you amply repurpose aspects of like the Postgres SQL implementation or perhaps the MySQL implementation? But all those things are kind of tightly wed together. There's not like you can't just cookie-cutter take the SQL execution engine from Postgres and slap it on to something else and really hit the scalability point that we're after with CockroachDB or the Spanner designers were after with Spanner.

I think another aspect of that within Spanner is the people designing Spanner were the people who designed and implemented Bigtable, and Bigtable had its own API, and that API was kind of informing some of their decisions for spanner as well.

[SPONSOR MESSAGE]

[00:17:14] JM: This episode of Software Engineering daily is sponsored by Datadog. Datadog is a cloud monitoring platform built by engineers for engineers, enabling full stack observability for modern applications. Datadog integrates seamlessly to gather metrics and events from more than 400 technologies including cloud providers, databases and web servers.

Easily identify slow running queries, error rates, bottlenecks and more fast with built-in dashboards, algorithmic alerts and end-to-end request tracing and log management from Datadog. Datadog helps engineering teams troubleshoot and collaborate together in one place to enhance performance and prevent downtime. You can start a free trial of Datadog today and

Datadog will send you a free T-shirt. Visit softwareengineering daily.com/datadog to get started. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:18:19] JM: You cofounded Cockroach Labs in 2015 and you launched the database company with an open source repository. How long did it take you and the team to get the database ready for a release into the open?

[00:18:36] PM: Oh! My memory of this is vague. I mean, I think it was – Was it two years it took to get to her 1.0 data release? I'd have to go look that up. Time flies so to speak. It's hard for me to even recall this. It's been five years and how far we've come in that time period. It took a little bit longer than we imagined. We already had some pieces in place.

I think part of starting any company is you have to forget how much work is going to be involved and almost consciously be ignorant of the amount of work that's going to be evolved. We were a little bit optimistic in how quickly you're going to be able to get out that first version, and it took longer than we expected. Part of this was because our aspirations grew. Our first version had SQL, and when we started the company, we didn't actually know SQL is going to be in there. I think we made that decision 6 to 9 months after the company founded. Then I think the 1.0 release was maybe two years after we came out. I'd have to look that up.

[00:19:27] JM: Do you recall what was the hardest part of that first implementation? I mean, you're saying you underestimated the difficulty of the undertaking. Was there some particular implementation detail that you recall as being really hard to finish?

[00:19:43] PM: There was no one particular implementation detail, but I'll tell you a little story about our kind of stability woes at some point in time. As you're developing any new piece of software, especially a complex piece of software, there is a period of time where it's not working at all. Then it kind of flips over and starts to work. It's not like a bright line where it goes from nonfunctional to functional. It kind of transitions, and that transition, it's like, "Oh! Hey, we can run it for a period of time. Oh! It crashed. Oh! Okay, we'll fix some bug. Let's run some additional queries against it. Oh! It crashed."

We're doing normal development. We're doing unit tests and integration test, but it was just trying to get clusters of Cockroach stable for long periods of time. That took quite a bit longer than I think any of us expected. We went to this period of we called the stability code yellow, and code yellow is a term that borrowed from our Google days. Where a code yellow is some kind of problem that is a company impacting problem. If we don't get past this, the company is going to die. It was all hands on deck for the stability code yellow that lasted for a couple months and we kind of dropped all the feature work we were working on and just focused on addressing the stability woes.

At the time, there is like a dark period. We're like, "Are we ever going get beyond this?" It went from the database running for minutes at a time, to hours at a time, to days at a time, to weeks at a time. Now we're at the point like years later, that understanding the database running for years at a time is no question. But just in the moment there, it feels pretty dark, like, "Ah! How are we ever going to get this thing just stable and get through all these bugs and whatnot?"

You do it. I've seen this happen several times. When you're in the midst of it, it never feels great. Then you get to the other side and you're like, "Ah! Okay, I see what had to be done." In hindsight, there might've been some additional work we could have been doing upfront on the integration stability testing, maybe. Not quite sure. I just feel that some, like any big effort like this has to go through a period of the dark times. I remember similar things happening on Gmail shortly before we launched the original version of Gmail, some dark days there as well.

[00:21:54] JM: You are working on the database with your cofounders. What was the division of labor? Did you have a stark division of labor or did you all just pile in and get an understanding of the entire infrastructure of the database?

[00:22:10] PM: We had a division of labor. It wasn't stark. It was more we kind of divvied up big areas between us. For example, very early on, my cofounder Ben Darnall, he worked on Raft and all the pieces of Raft. I kind of jumped in into a variety of places. I ended up getting like the initial work on the SQL execution and original SQL heuristic planner going. Spencer, he'd actually kind of put all the groundwork. He laid the base foundation originally and then started

doing work on the distributed transaction protocol, but I wouldn't say the divisions were stark. We all jumped in into the various areas where necessary.

There was also division of labor in terms of kind of the responsibilities of the founder. Spencer Kimball, he's our CEO. He's one who paints division, goes out and works with the investors. He's a much more outward-focused. I act as the CTO. Much more heavily involved in technology, and Ben acts as the chief architect. He's like kind of the final arbiter of design decisions, especially contentious ones.

[00:23:12] JM: Let's talk about the engineering and the actual architecture of CockroachBP. The underlying data is stored in sorted keyvalue pairs. How does this data layout of being arranged in keyvalue pairs? How does that compare to how data is laid out on disk in a relational database like MySQL?

[00:23:35] PM: It's relatively similar. Underneath the hood in a traditional OLTP-based relational database, you're going to have some kind of something similar to keyvalue storage. It might not be called keyvalue storage, but if you squint at NodB, which is the backend for MySQL. At least is the default backend for MySQL. It's a B-tree storage, and B-tree is storing keys and values.

They might be not storing actual string keys and string values the way we are in CockroachDB. There might be delimiters inside those, and I actually don't know the details what NodB does. But some level it's kind of still keys and values. You have keys that are part of the lookup. Those might be delimited, as I said. Then you have values. The values might also be delimited.

Inside CockroachDB, at somewhere, there are keys and values where kind of the contents in the keys and values are opaque, but the higher level, it's SQL. SQL has arranged to format those keys and values so that you have that kind of calm structure inside the keys and the values. That's very similar.

There is also OLAP databases. When you talk about the relational database landscape and analytics database and OLAP database, those are frequently column stores. The difference between kind of a transactional processing database in one of these analytic databases and have they store data on disk is the column stores arrange to store all the values for particular

column kind of adjacent on disk and they can load it up, all that column data in one easy read and process it in a more efficient manner. The benefit of this is it makes you read queries much, much faster, but it slows down your write and your update worries.

[00:25:11] JM: What is the translation process for a query? There are these five layers in CockroachDB that defines how a SQL query is eventually translated into a keyvalue query. You have the SQL layer, the transactional layer, the distribution layer, the replication, and finally the storage layer. Why do you need these five layers for managing a query?

[00:25:37] PM: Well, this is part about like how much effort it is to go through and the expertise you need at each level. Down at the lowest level, how we're storing things on disk. We have this MVCC, multi-version concurrency control layer, built on top of RocksDB, and there's just deep expertise. You have to know about interacting with the disk, storage on disk and the management there. The layers are kind of like oriented in terms of knowledge and expertise areas.

Above the MVCC, that's where you have replication. What are all the complexities in dealing with Raft and making the high-performance replication protocol? Of that, you have distributed transactions. Distributed transactions are not simple and it's layered on top of the replication just so that we can actually have a group of engineers who have that expertise and competence there.

Then above the distributed transactions, you have SQL. SQL itself is broken into a SQL optimizer, which takes care of the planning of the query; and SQL execution, which takes the plan that optimizers spit out and then executes that plan to completion. It's just, again, different areas of expertise. The people working on optimizer, they have to be very knowledgeable about relational algebra and the validity of various transports and how to apply table statistics into a cost model to decide which query plan to execute.

The execution folks, they have to be experts at having and optimizing the execution of a particular query plan. Each query plan is composed of these various operators, and the internal details of those operators are complex. I think like the kind of layering you sees is some aspect

of managing the complexity. Keeping the complexity from pervading everywhere and having it scoped down to a size that a single team can manage.

Something to be aware of though here is sometimes these layers, they can get in the way. We're seeing academic research come out right now that are saying, "Hey, having distributed transactions built on top of a replication layer, there's kind of a duplication of effort there." Look at what we can do if we actually blend these two layers together and have them be aware of each other. This is something that we're paying attention to. It's something we've thought about. Sometimes the layers are just getting in the way and you might want to get rid of them.

[00:27:44] JM: What are the places where the developer might want to configure something and change something in one of these layers? Would that ever happen? The actual database user, they want to tweak something that would affect specifically the transactional layer, or are these layers really only in the purview of the people who are actually building CockroachDB? They're not database operator areas of modification.

[00:28:11] PM: They're really for the most part in the purview of the Cockroach Labs developers, CockroachDB developers. There are knobs in some of these areas, and to give you an example of that down at the low level, there's the cache in RocksDB, a block cache, and the size of that block cache affects the performance of read queries. That's a knob that's available for external tuning.

There're also knobs for controlling certain things in the optimizer. Those are almost never tuned, but they exist. There're a few other knobs in the system. But for the most part, kind of having a preponderance of knobs I think gets in the way of the usability of a system. We've tried to have those things just do the right thing and automatically tune themselves. I can't say we've achieved that goal perfectly, but that's certainly the goal.

[00:28:58] JM: You've mentioned RocksDB a couple times, and RocksDB is an underlying storage engine that powers various database systems. Explain what the purpose of a storage engine is.

[00:29:11] PM: The purpose of RocksDB. At some level, a database has to write stuff down on to disk, and kind of naïvely you might think, “Oh! I’ll just open a file and write some data in the file, but then I have to read that data back.” That means structuring the data that I wrote into the file in a certain way that makes it easy to find it.

Now, there are systems, there's been research in this area for as long as there have been relational databases for even longer. One of the ways to structure things on disk is a B-tree. This is a balanced tree structure. RocksDB is slightly different. It uses something called a log structured merge tree, but it’s conceptually like if you step back and squint to a log structure merge tree and a B-tree, both kind of provide the same primitive, which is that can write keys and values into the structure and then I can get them back in order, in sorted order.

Taking care like the details of how that happens, it is complex. For a B-tree, you have these rebalancing mechanisms. For RocksDB, you have a series of SS tables, some levels and compaction heuristics. But the fundamental need there is I need to have some way that I can write these keys and values and have them be persistently stored on disk and then read them back later in a desired order.

[00:30:24] JM: All of the underlying data is divided into ranges, and these ranges are each replicated. For each of the replica sets, there's one replica that's called a leaseholder, and that's the replica that handles the read and write requests. If you have all of the data that is replicated at least three times. You've got at least three servers in the default case that contain one of the instances of the data, why not have each of those replicas serve requests? Why do you need to designate one specific replica as the replica that's going to be actually performing or handling the read and write requests?

[00:31:12] PM: Yeah. It's an interesting question. It would be desirable to be able to read for many replica. It’s desirable for several reasons. You get kind of better aggregate resource usage in your cluster. Maybe one of those replicas is nearby you, so you avoid geographic latency when doing the read.

The problem with being able to read from every replica is how to do that consistently. If I was allowing to read from any replica and write came in somewhere else, there's this whole

coordination that has to take place there. The various ways to do that coordination, there's not that many, and one of them is to actually kind of funnel all the request through this central chokepoint, which is this leaseholder replica for the range. But it's always fundamentally about providing consistency, consistent answer. You wouldn't want to have the case that two reads to the same metadata hit two different replicas and they get a different answer just because one replica is slightly behind. That would be a big no-no for our whole consistency story. It'd be very surprising to the application developer.

[00:32:17] JM: I'd like to understand the interface between CockroachDB and RocksDB, the storage engine. In a little more detail, can you tell me where does the abstraction boundary between CockroachDB and RocksDB lie? What's the interface between those two?

[00:32:36] PM: Yeah. The interface is essentially RocksDB exposes relatively small set of primitives. You have operations for setting the keys, deleting keys, deleting a range of keys. You can actually ingest a whole set of keys as well. There's this ingestion operation.

What is exposed up to CockroachDB? There's a this MVCC layer that I mentioned before, multi-version concurrency control that sits on top of RocksDB, and that essentially kind of tweaks those primitives that RockDB provided and then kind of exposes them up to the next layer up, which is this kind of per node storage system.

What it looks like is you are able to – Within the Cockroach node, just as one component knows about talking to RocksDB and it has operations such as put a row, retrieve a row, scan a whole bunch of rows, and those get translated fairly directly into API calls on RocksDB.

Kind of the surface are there, it's relatively locked into kind of one fraction of the code. Maybe 10% of the CockroachDB code base knows about that. But there have been some RocksDB-isms that have escaped beyond those boundaries. Kind of the primary one is RocksDB has this file format on disk. It's called SS tables, sorted string tables. This is – It's not like a formally defined format, but it's not a format that's going to be changed. We actually end up leveraging that format as our backup format that is used for storing backups up in the cloud storage like S3 or Google storage.

Another instance where RocksDB-isms have escaped passed this API boundary is something that's a little bit kind of down in the weeds. RockDB, in addition to having operations for setting the value, you can actually have this thing that's called a batch, and a batch provides a container for a group of operations that RocksDB guarantees will all be applied atomically. When you write a batch, you're guaranteed like all the operations of the batch, they either occur atomically or they didn't occur at all. Inside the batch, there's a representation of the format of these operations, and that kind of internal batch representation has leaked out of some of our abstractions.

I don't feel too bad about either of these leakages just because these formats are so you well-specified and understood that we could eventually get rid of RocksDB and replace it with something else and there won't be a problem. I should mention that that's actually something that's coming down the path. We are looking to replace RocksDB with another system that's going to be compatible with these data storage formats.

[00:35:08] JM: Interesting. Did you say you're planning to replace RocksDB entirely?

[00:35:12] PM: Yeah, that's the intention.

[00:35:15] PM: What's the motivation for doing that?

[00:35:16] PM: The motivation for doing that is maybe traditional SQL databases. They actually have their own storage engine inside. MySQL tightly wed to nodB, and while there's a RocksDB backend for MySQL, it comes with a lot of caveats, and not all of the MySQL functionality available if you use that RocksDB backend. Cassandra comes with its own LSM tree internally. They're looking at this other RocksDB backend, but again it comes with caveats.

Now the thing about using RocksDB is absolutely the right decision when CockroachDB got started. We wouldn't want to have to reinvent the entire world. But over time, there are customizations we want to make and we have to own that storage layer. Now there's kind of two obvious pass forward that we could kind fork RocksDB and use it. RocksDB has a ton of functionality that we do not need, and we couldn't fork it, strip out all that functionality to own it. But there's this other thing that gets in the way here, which is RocksDB is entirely written in C++.

All the rest of CockroachDB is written in Go, and we really see this barrier between the two languages. Probably, you might've heard of CGo. If you're familiar with Go, you've heard this interface called CGo, where Go can really easily call to C, and yet despite how easy it is, it isn't as easy as just accessing some native Go library directly, and we can't reuse functionality that's been part of the rest of CockroachDB inside or closer to the code that interfaces with RocksDB.

We also see tons of opportunities to evolve the storage layer in service of CockroachDB. To give you one example of that, some of our restore and import mechanisms, they are bottlenecked right now on the performance of RocksDB. Again, this is something we could look at to fix inside RocksDB, and yet the overhead of doing that versus doing it in a storage engine that we own, doing it in a storage engine we own looks to be significantly less.

[SPONSOR MESSAGE]

[00:37:19] JM: When you spend your spare time learning, you can accelerate your career. O'Reilly lets you learn through high-quality books, videos, courses and interactive experiences. O'Reilly content has been built over decades. They're a trusted source of effective technology education. If you're an individual leveling up on your own, you can use O'Reilly to chart a course for your career goals. If you manage a team or a company, you can get access to O'Reilly's career development resources for your whole organization.

Go to softwareengineeringdaily.com/oreilly to explore O'Reilly's e-learning experiences. You can build the skills you need to future-proof your career. Check out softwareengineeringdaily.com/oreilly, and thank you to O'Reilly for being a partner with Software Engineering Daily for many years now.

[INTERVIEW CONTINUED]

[00:38:20] JM: This problem of making a globally consistent database, this has obviously been noticed by other people in the industry, cloud providers, other vendors. Tell me about the design decisions that CockroachDB has made that contrast with the other attempts, the other companies that are building globally consistent SQL infrastructure.

[00:38:50] PM: Yeah. There're some other startups in this space. Some of them born out of academic research. The one that comes to mind there is FaunaDB. It's born out of a research project called Calvin. One of the decisions that the Calvin research was showing, "Here's how you do distributed global distributed transactions." The caveat they have is, "Oh! You need to specify all the operations in your transaction upfront," and I've heard claims from the FaunaDB folks that they know how to do interactive SQL transactions.

Let me make sure I'm clear what I mean by interactive transactions. Interactive transaction or conversational transaction is one where the application starts their transaction, does some operations, gets the results, applies application logic that's present inside the application itself. Does some other operations, can do this multiple times and then commits the transaction. This is a very common pattern inside applications itself.

FaunaDB, or at least Calvin, was saying like, "Hey, if you can actually specify everything about the transaction upfront, we can do some really neat techniques inside the database in order to optimize it," and yet it becomes with this big caveat and a significant change in how applications have to be architected. We feel that that change is a big burden and could be a big headwind against their adaption.

There's a company out of China, PingCAP, which develops a database called TiDB. Sure, it's a lot of similarities with the CockroachDB architecture, but they have some differences. Some of their stuff is more centralized or not as focused on geographic replication as we are.

Then we have like some of the big players. I mean, clearly, Spanner, which we share a lot of the architectural decisions with. With Spanner, Google doesn't seem to be pushing as hard as I would've expected. I think they could be pushing it harder. Amazon and Aurora, Aurora is offering kind of some geographic distribution now, but the current architecture involves a single master. I don't know. I know they've been talking about having multi-master Aurora for a while. I don't know if that's been – I can't recall if that's been released yet. I mean, there is quite a big difference between having a single master node that has to coordinate stuff and becomes a bottlenecked versus allowing any node and having the cluster scale out to enormous numbers of nodes.

[00:41:04] JM: In the competitive landscape, operating a business as well as building a complex distributed database, one thing I wonder is how do you convey the subtle engineering differences to the end user? Is the end user – Is the developer market discriminating enough to be able to see the differences between those different database architectures or does it come down to kind of marketing?

[00:41:39] PM: I think there's a lot of marketing involved. Certainly, there is a subset of the developer community that is discriminating and it can understand these differences. Are these the early adapters or is it larger than that? Not quite sure. There's certainly a lot of marketing. I mean, SQL databases when they first come out, came out into existence 30 or 40 years ago, must've been incredibly confusing and complex, yet over time we figured out the right way to explain these differences and explain them in a way that makes sense.

I think we're still working on this. It's a work in progress and it's something that needs to be focused on. You figure out how to translate the kind of expert terminology into analogies that makes sense for everybody. We're still working on that. We'll probably be working on that for the lifetime of the company frankly.

[00:42:24] JM: Have you changed the implementation of CockroachDB significantly since Kubernetes came out?

[00:42:31] PM: The answer is yes, but not because of Kubernetes. The internal implementation of CockroachDB is constantly seeing enhancements. To give you one concrete example of that, our distributed transaction protocol has evolved significantly since the first implementation version of the distributed transaction protocol in version 1.0 versus what's currently in the product today. They are significantly different. The current version is significantly better, significantly faster, lower latency, gets better parallelism.

But all that came about not due to Kubernetes. Kubernetes is clearly this tidal wave that's taken over the industry, and it just happened to coincide with the way CockroachDB nicely just meshes with the Kubernetes architecture. We've done some minor adjustments to fit in more cleanly. We're continuing do some adjustments. We're looking right now, we're building out a specific CockroachDB operator to make the Kubernetes Cockroach experience even better. But

there haven't been like huge architectural changes necessary, and I think that was more luck than anything else.

[00:43:32] JM: In addition to architecting CockroachDB, you have to stand it up on the cloud providers or you have to figure out a repeatable architecture pattern when a developer stands up a CockroachDB instance on AWS, for example. What happens when they deploy it? I'd like to know more about what happens in the design and deployment of a cloud product. You have to make this compliant for both AWS and GCP. I don't know if you build it with entirely kind of like open source pieces or like does deploying on AWS use the AWS load-balancing infrastructure and deploying it on GCP, it uses GCP load-balancing infrastructure. What's the process of getting this thing stood up on a cloud provider?

[00:44:25] PM: Yeah. I mean, there's a choice here, and the first division is, is this a customer standing up on AWS or GCP? They have all this flexibility in what they can decide to do. They can use open source tools for load-balancing. They can use the AWS load balancer. We generally see people using Kubernetes now, and Kubernetes also has some load balancers inside of it as well. That's kind of the first decision.

Then there's also do you want to even be running it yourself? This is where we had this product Cockroach Cloud, which is we want to run the database for you. Running anything in the cloud, running any system is challenging, and we want take that burden off developers and take it upon ourselves. Frankly, we're the experts in CockroachDB. We should be able to run it better than you. We should be able to run it more efficiently. We should be able to anticipate problems, make sure everything is set up correctly.

When we're running CockroachDB for you and on Cockroach Cloud, we support currently AWS and GCP, and we're running these CockroachDB clusters on top of Kubernetes. What we've decided to do though is to use the native bits that are part of the cloud infrastructure. If you're on GCP and you want virtual private cloud peering work using the – You have to go through the Google APIs to use their VPC peering implementation. Similarly for AWS, we're using the AWS load balancer, the Google load balancers and whatnot. This just gives the best experience on those platforms, though it does add more burden, but that's a burden we feel we should be taking on.

To see how this plays out though, is we haven't had a support for Azure. It's on a roadmap, but we're going to have to do kind of custom work in order to support Azure rather than just dumping in something that worked exactly that same bits of code on every cloud. It really would be nice at some point if everybody, the entire tech ecosystem could agree on cloud APIs and just make them symmetric across all the cloud providers, but we're certainly not in that world.

[00:46:18] PM: What do you think is the market size for distributed consistent SQL databases?

[00:46:24] PM: I really hope it's huge. I mean, I'm not sure what the market size say right now is for kind of traditional SQL databases, but it's in the 30, 40, 50 billion range. I expect that something like CockroachDB, we don't see it as taking up a niche spot on there. We see it taking up a significant spot. Anything you've traditionally used a normal kind of single node SQL database for, you can use CockroachDB in that same mode. CockroachDB runs perfectly fine on a single node. The advantage is, "Oh! I want high-availability." Add a couple more nodes to the cluster and you get it kind of just easy. There's no headache there.

Our expectation is we want to capture up and down that into the – That entire spectrum of the market. Not just occupying purely high-end clusters that have hundreds of nodes. We see this. We have many, many customers, 5, 10, 20 node clusters perfectly happy there with the expectation that as their systems grow, they'll just continue to expand their clusters.

[00:47:27] JM: The idea is really to be a pure successor to MySQL or Postgres as SQL infrastructure that scales more reliably or more readily.

[00:47:39] PM: Yeah, absolutely. We're also looking at scaling down the other way. There is a certain minimum node size you have over MySQL instance. Let's get lower than that. Let's get smaller than that.

[00:47:49] JM: Can you tell me more about that? How can you shrink the typical size of a SQL deployment?

[00:47:57] PM: Well, it's not that so much you shrink the size of it, but just consider right now you're using – You undoubtedly use Gmail for email. If you don't, you're from another planet as far as I'm concerned, but you're using some kind of hosted email service. You don't have a dedicated machine in a data center running your email. Your email is being shared inside a much larger system. There's probably like – Somewhere inside Google, there is a cluster with thousands of machines and you're occupying a fraction of some subset of those machines.

The experience is you have your email. It's like an email database just for you. I mean, we can do the same thing with a Cockroach cluster of subdivide even a Cockroach cluster. Give a thousand users a perception that they're getting their own kind of custom cluster even though it's being represented and implemented on the backend across on top of a single much larger cluster. It's kind of giving that perception versus reality.

Another example of this is just the virtualization of hardware. You go and you start a VM up in Amazon. You're not getting a dedicated machine. You're getting a fractional of machine and it's all kind of – Smoke and mirrors isn't quite the right phrase, but it's all technology in the backend to take a physical machine that might have 96 cores and give you one CPU of that or a fraction of a CPU even.

[00:49:13] JM: Give me a lesson in engineering management that you've learned since starting Cockroach five years ago.

[00:49:19] PM: That's a great question. The thing that frequently comes to mind is communication. You just have to communicate and over communicate and say the same thing again and not be surprised when there is some population, some subset of your engineering population that didn't hear something or is confused by something, which is bound to happen.

I think prior to CockroachDB, I had worked in smaller teams and led smaller teams and felt like easy enough to get everybody on the same page just as you scale and you grow. You just have to just get that Zen attitude that, "Oh! I answered that question last week. I'm going to answer it again this week," and just give them the same pitch and say it again and again. That communication is just critically important.

[00:50:01] JM: How do you think the current economic downturn will affect infrastructure companies like Cockroach?

[00:50:08] PM: Yeah. I go back and forth on this. I mean, clearly, this downturn is once in 100 years kind of thing. It's not like previous crises, financial crises. We can't really be looking for economic stimulus. We're looking for disaster recovery. I heard something, an analogy, that's like everywhere in the world is getting hit by a hurricane for three months straight. There are companies out there still spending. There are companies out there still making – Spending money and buying infrastructure, and there's others that have just completely turned off the tab. We need to be focusing on those companies that are still spending. Clearly, Amazon is still delivering. Various other banks are still operating. The government is still operating. There are people buying stuff still. I think we're going to come out of this, the economy overall is going to come out of this. The infrastructure spending is still going to continue. The near-term impact is just – Ask me on three different days, I might give you three different answers.

[00:51:09] JM: That's very consistent.

[00:51:10] PM: Yeah, it's not consistent, but they're consistent day-to-day. So I don't have to do consistent internally.

[00:51:15] JM: Right, exactly. Okay, just one last topic I wanted to talk about. We do a lot of shows on databases. We also do a lot of shows on data infrastructure. The funny thing about like data engineering infrastructure, like if you're talking about really large data workloads like Spark workloads or nightly batch Hadoop jobs. The thing with these – Or machine learning, training machine learning models. The thing about these is the date infrastructure there is less consistent if we're talking about things like the "Lambda architecture" or the fact that you have data being read from these large data lakes or from some data warehouse where a large portion of data has been pulled into a data warehouse from a data lake. This data may be slightly out of date.

The way that we build infrastructure today, the fact that the data is slightly out of date is not really a problem, because usually we're finding aggregations or we're finding averages, we're just kind of averaging and mashing the state altogether. I wonder if is there a desire or is there a

time in the future when we are going to need highly consistent OLAP data infrastructure as well?

[00:52:28] PM: I'm not sure if they'll need to be highly consistent, but I think having faster data pipelines is something interesting. We have a friend company, sibling company maybe, which is called Materialized.io, and they're working on just really fast kind of materialized views for data, data pipelines.

I think there are definite use cases, strong use cases for that, and yet there is also other use cases .I'm sorry to be down on you guys, Materialized, but there's other use cases where the latency concerns are not a problem, and we see this like the end of day reconciliation reports and whatnot. If you missed your transactions at midnight and then you can start it up and start your reconciliation process and it's like, "Yeah," like things will get reconciled the next day too and somehow that perfectly up-to-date that those superfast latencies are less important. Yeah, I see a world for faster, more consistent analytics. I think that's going to be slightly more niche.

[00:53:34] JM: Agreed. Yeah. We had the Materialized team on several weeks ago. I guess they're in New York also.

[00:53:40] PM: Yeah, that's correct. Yup. Hard to tell exactly where they are nowadays. We're all just isolated due in our rooms, right?

[00:53:48] JM: Right. I completely agree. I think we are slowly sliding into virtual reality. It doesn't really feel as we anticipated. We're not looking at rendered avatars of each other, but it does feel like we're in a kind of virtual reality.

[00:54:02] PM: I kind of had a similar thought the other day of what'd be like if there's instantaneous transportation or transportation was no longer part of your life and essentially we don't have the instantaneous transportation, but transportation is certainly no longer part of those people's lives, and that's kind like this thing that it feels a little bit like the future right now.

[00:54:18] JM: Yes. Yes. It's a little unnerving. I think that is part of what makes this whole virus experience so unnerving. It's just the changes in our daily habits or lack thereof. Anyway, Peter, thank you for coming on the show. It's been really fun talking to you.

[00:54:33] PM: Yeah. Pleasure of being here.

[END OF INTERVIEW]

[00:54:43] JM: Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we're being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it's definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[END]