EPISODE 1303

[INTRODUCTION]

[00:00:00] JM: In 2003, Google developed a robust cluster management system called Borg. This enabled them to manage clusters with tens of thousands of machines, moving them away from virtual machines and firmly into container management. Then in 2014, they open sourced a version of Borg called Kubernetes. Now in 2021, CockroachDB is a distributed database designed with the Kubernetes architecture in mind. CockroachDB uses regular SQL and scales by automatically distributing data and workload demands. Their databases survive machine, data center and region failures and provide guaranteed ACID compliant transactions.

In this episode, we talk with Spencer Kimball, CEO at Cockroach Labs, about distributed databases and containerization. It was a great conversation about distributed systems, and particularly how to build a modern distributed systems product. I really enjoyed talking to Spencer, and I think he's going to be back on the show in the near future. So enjoy today's episode, and look forward to one in the future.

[INTERVIEW]

[00:01:02] JM: Spencer, welcome to the show.

[00:01:03] SK: Thank you, Jeffrey. It's a pleasure to be here.

[00:01:05] JM: It's a great pleasure to have you. I've done a bunch of shows on CockroachDB over the years, and it's an important set of technologies. Basically, the technologies under the ages of arguably open source Spanner. That's maybe reductive at this point. But more broadly, I think of Cockroach as falling into the category of modern distributed databases. And in this category, there's a lot of confusion. You have MongoDB obviously. You have Spanner. You have Cockroach. You have FaunaDB. You have VoltDB, DynamoDB. If I'm trying to evaluate this market, what are the features that I'm looking for? Am I looking for pricing? Am I looking for resilience, fault tolerance? Take me through the swath analysis.

[00:01:58] SK: Yeah, absolutely. Well, there're many dimensions as you point out, which is part of why there's so many solutions. The other reason, of course, is that operational databases are the largest market in software. When we really think about product segmentation, probably the largest dimension is kind of what's the modality of the operational database. And typically, people would break that down into NoSQL and SQL. And that's where, in the list of databases you described, I say there's the biggest dichotomy. And relational databases in the operational database market are by far the majority. I think it's depends on what analysts you ask. But the market is something about $65 billion a year and it's growing. The compound annual growth rates in the double digits, low double digits. Relational is probably 55 million of that 65 million. And so the other stuff is NoSQL, and it's graph databases, and document databases, like MongoDB. And there're a number of others. So that's the biggest one.

We're relational. And we've been relational not from the beginning, interestingly, and nor was Spanner. So I think it is fair to say that we are very inspired by Spanner. And I think, certainly, when we got started, an open source Spanner would be how we described ourselves. Spanner also started more as a transactional key value store. And then they realized that in order to really fulfill one of their primary purposes, which was to replace a thousand plus shard instantiation of MySQL that was used for AdWords, they needed to have some level of SQL compatibility. Otherwise, that use case just wouldn't move.

So we also started off as this is the easier problem to solve, is how do you make a distributed transactional key value store. But we also quickly realized SQL. Despite maybe the last decade, losing some of its primacy, remains the number one choice for operational database. And part of that is just, well, it's evolved for a long time. There's tons of institutional muscle memory around it. But I think more importantly, the evolution that's really happened in the category is about managing complexity.

So it's easy to start any project on any of the databases you mentioned. But it becomes increasingly difficult for any project and any database as a project gains lots of additional functionality and complexity. Relational databases have really found, I think, sane ways to manage that complexity, explicit data models, ways to alter your table structure, your indexes and so forth while the database is running. Those are things that are common to the enterprise-grade relational databases that actually are sort of by design not part of, for example, many

NoSQL systems where they say, “You don't need a data model. You can just get started and you can just add things willy-nilly.” And that's great when you get started. It becomes a big cumulative technical burden as you move along and you get to years two and three, and you don't have the same engineers on the project anymore, and people don't understand what the original data model was, because there's always a data model, by the way. So that's the biggest sort of dimension that separates the field, the very wide field of competitive distributed databases.

Now, let's say that there's this sort of core capabilities beyond that. Like what is the distributed database buy you? And there's a lot of differentiation in the marketplace. Even though sometimes when you look at all the different web pages, they say the same things. There is there is a significant differentiation. It can be hard to parse at times. Where cockroach is, I think, fundamentally different from anyone in the market right now, besides maybe Google Spanner, and it's not Cloud Spanner, it's what Google uses internally, is that we really are focused on a multi-region set of use cases. So the idea here, and again, is very much informed by the decade I spent at Google, it's pretty easy to get customers anywhere in the world, right? And for a big company, that's your fundamental reality, but even for a startup. You get users in Brazil, or you get users in Japan, or Australia. You want those users to have a first-class experience, not just something that is only good for users on the East Coast of the United States. That's hard to do, unless you're actually embracing a data architecture that puts the data close to the customer. And that's something that Cockroach has been focused on now for the greater part of our existence as a company. And that would be really one of the big reason you'd select Cockroach, if you really want to say, “I'd like to build the way big tech builds. I want to make sure that no matter where the users are, they have an absolutely first-grade experience. There're real time latencies everywhere.” In order to do that, you do need to embrace a database that has that differentiating capability. So that's one where we really stand out.

Probably the most common aspect of all the distributed databases you mentioned is high-scale in terms of data. Remember Mongo in 2010. That was a really big draw. It's like we're web scale. We give you that capability. We also make it easy, because you don’t have to learn SQL. And those are really the selling points of Mongo. And they're great selling points. But scale remains a really big category definer for distributed. And that's really the big reason that anyone embraces distributed in the first place. Like how big is my data going to get? How big can I let my data get without running into problems? And it's different depending on what kind of database you're talking about.

I'd argue, it's harder in the relational model. There's more disruption there. The disruption around scale is fundamental to the non-relational, the NoSQL databases. But we’re really – I think we've innovated in the last six years, as of course have Google and others, in really bringing high scale to the relational side of the market.

And then there's one other thing I'd mention. I know this is a really long answer, but it is a complex and competitive space. I think it is important to have a little detail to back it up, but is the consumption model. And so Cockroach very much started as open source. And we still are. We have a BSL license now. So it's not strict open source. But that was the dominant consumption model for databases in the last 20 years. And it really edged out the idea of closed source. And one of the reasons for that is open source made it extremely small amount of time required to get to some value. If you think about the sort of minimum time it took to get a closed source database that you could get your hands on as developer and start to kick the tires, even get something into production, I would take like 100 days. You’d go through legal, and procurement. You'd have sales people. And you have to make a decision to evaluate the competitors before you could ever commit to anything. You got to run a process, right? And so that could take some serious time.

With open source, you had this ability to, literally within hours, get something stood out, maybe minutes. And that is a huge advantage for consumption. We're entering a new world now. So the time to value is greatly enhanced beyond open source if you can consume something as a service, because you no longer have to learn how to run it and operate it. And so all of the databases you mentioned fall into different consumption categories. I think you didn't mention any of the closed source more traditional ones like Oracle, but even those are starting to embrace multiple consumption models. So really, when you look at Cockroach, not only is it something that you run yourself, which many of our customers do, but it's also something that you can have as a fully managed service in the cloud. And that's, I think, pretty common across. But what you're seeing now is even new consumption models beyond that, which we're also embracing. So, for example, serverless in the cloud, and serverless applied to a database just means that you as a user don't have to make all kinds of decisions that are tied to the idea of how many nodes do I have? Where do I have the nodes? How big are the nodes? What's my capacity planning?

But serverless in a database, you get to say as a developer, “This is just a SQL API in the cloud.” And I can start small and I can elastically grow to however big. I can elastically shrink to exactly what my needs are. I get charged only for what I'm using. What Cockroach is actually really moving towards is you don't even have to say what region or availability zone your data should be in or your database, because it actually spans all the regions. So as a developer, this is really enticing, because you don't have to make decisions up front that might be challenged by the reality of where you actually end up with customers, or how big your database needs to get and how quickly. Instead, it’s a just SQL API that's a seed when you get it, but it can grow into any size tree in any location where you actually end up with users. And that's, I think, the future. But you have all of these consumption models. And those also a very important dimension that does separate the different databases out there. So really, it's what kind of database, what kind of operational database, and that is very much going to informed by the application you're writing. And it's really about how you want to consume it. How that's going to work in your environment. And in all the things you mentioned, DynamoDB, Google Cloud Spanner, and so forth, those only run in the cloud as managed services. And in fact, they only, in those different cases, run within a single cloud, whether it's AWS or GCP.

Often, you have to situate yourself based on what kind of company, what kind of use case, and sort of keep all of those dimensions in mind in order to choose a database that's going to be the most appropriate. So you can't say that Cockroach is the best for all use cases. It's absolutely not. So it's a complex competitive landscape out there. And the choice isn't always easy. But if you know enough about what you want to accomplish, what your end state is going to look like, it does allow you to make the right choice, I believe, but it takes some education and some time spent really evaluating not just the surface marketing claims, but a little bit deeper understanding how the technology works. What happens in very real edge cases in production? And, yeah, what your ambitions are as a company and as a project?

[00:12:19] JM: Let me see if I'm understanding correctly your perspective on this domain. When I started this podcast, the CAP theorem was about as much as I knew about distributed systems. CAP theorem being you can have two or three, consistency, availability and partition tolerance at any given time. And there's a lot more detail. It's not that simple in reality. And in fact, even if you're just talking about something like consistency, are we talking about consistency between in-memory and disk? Are we talking about consistency between two different discs? Are we talking about consistency across different data centers? And then you could add in cost. How are we trading off cost? And the modern CAP style framing is what do you want? Like what are you willing to pay for? And can you get what you want out of a given platform without sacrificing too much implementation cost? And I think what I'm hearing from you is your goal with Cockroach is to provide configurability around those different goals for what your distributed database should be.

[00:13:33] SK: Yeah. I think that's definitely part of what I'm saying. I'd add a little bit to it and just say that even within Cockroach, there's lots of nuance in terms of, yeah, we're a CP system. But depending on what you're looking for beyond that, let's say you want this idea of availability, there's lots of nuance in terms of what that means. And I think, with all of the databases out there, there's always going to be tradeoffs. But it's surprising how effective you can find a point in the solution space that actually optimizes for everything you really care about. And the costs are things that you don't care about. So there's tons of flexibility.

Cockroach is what I know best. So I can give you a little bit of additional color there to make this more concrete. The idea of the CAP theorem with availability is actually interesting. And it's often misunderstood. It's not the same thing as high availability. Really, what it means is that any non-failing node in the distributed system is able to answer a query definitively no matter what else is up. So if you just got one node out there, you can ask it and it will give you an answer. It doesn't have to coordinate with somebody else. That's what availability means.

And what that means in practice is that you can have a split brain, because things can die that might have the right answer and the things that remain might not have the most recent data, but the things that remain can give you the answer. And you’re going to get the wrong answer, but it will always give you an answer if something is non-failing. That’s what available means.

High availability, which is I think what most people think of when they think of that word, availability in the CAP theorem, high availability is what's your SLA? How many nines do you have? How likely is this to stay up when there're various kinds of disasters? And there, Cockroach can be arbitrarily highly available. If you want five nines, well, it's like what's the risk of losing a data center? How many data centers do I have to add in order to make sure that I meet that probability? And that high availability is really something that you can dial as appropriate for your use case, and Cockroach can do it. So we are a highly available CP database. Not an available and CP database, which the CAP theorem correctly says you can't have everything, right?

This is a really great example. Like people worried that, "Okay, well, does that mean that something that is an AP system is more available than Cockroach?" Yes, in the CAP theorem. No in terms of high availability. You can make Cockroach arbitrarily highly available. So it's confusing. CAP theorem I think adds more confusion than it actually helped solve. And that's just my opinion. But whatever tradeoffs you think are necessary, there's often really interesting ways to not actually accept the tradeoff and actually get exactly what you want and to sort of hide the cost.

As another example, Cockroach in the most recent version has added something called non-blocking transactions. I mean, it's a really fascinating idea, and something that at first glance just didn't really seem like it would be possible. But in fact, it isn't. So what this capability allows you to do, think about a use case like Quora, where you want a global audience to be able to read all the questions and answers. Potentially –

[00:16:51] JM: How'd you know I love Quora?

[00:16:53] SK: Well, everyone does. It's great a system. I use it all the time.

[00:16:58] JM: Did you know – By the way, the first podcast I started was the Quora cast. It was an unofficial podcast about Quora people.

[00:17:04] SK: I did not know that.

[00:17:05] JM: That's how much I love Quora.

[00:17:08] SK: So I lucked out on this example. It's very resonant. Yeah, that's a very common use case, by the way. It's like there's a certain amount of data that you want local to the customer about their account, things you might be tracking, and so forth. But a lot of the data, you want to make sure that it goes out globally. So everyone can read it very quickly from wherever they're coming from in the world. They're browsing all the questions and answers and so forth. When you write, which actually in the scheme of the reads and writes in a system like Quora, writing is actually infrequent compared to reading. But it might be like 5% versus 95%. Huge imbalance.

So what you realize there is that there's an opportunity to pay a higher cost on writes if your reads are always super low cost, because you're doing 95% reads, 5% writes. With non-blocking transactions, what you're able to do with Cockroach is pay a higher latency on your writes. But when you read, you get not just the global replication like you get in an eventually consistent system. But we actually make it so that everywhere you read, everywhere around the world, is going to get the exact same consistent answer even as you're updating things in real time. Everyone reads the same thing. It's not like, "Okay, Tokyo is reading this. Australia – Or Sydney is reading another thing at exactly the same time," because Sydney just hasn't gotten the data, the update yet that's eventually percolating there. So that's a big problem if you're trying to build a relational system. So that wasn't an option for us. But we introduced a whole new transaction model. And what it does is pretty neat. It creates a transaction that's going to take effect in the future, usually by several 100 milliseconds. And it does the writing and so forth on a global basis, and it's going to take whatever that delay is, that several 100 milliseconds, which is a lot for write, but not if you're trying to build something like Quora. It doesn't matter if an answer takes several 100 milliseconds to propagate around the world. That's almost like an expectation.

But in the meantime, as that several 100 milliseconds is being used in order to coordinate a write globally, all of the reads can continue without having any kind of blocking or locking or anything like that. So all the reads continue to read the old thing. And when the switchover happens, it happens globally. And it happens at the exact same timestamp. There might be some absolute time differences, but everyone's going to read the same data after it becomes committed no matter where you're on the planet. And that's like a really fascinating capability. And it solves a problem that I think many people didn't think was solvable, and still don't, which is like how do you build a system like Quora and actually have consistency globally and fast reads? And the answer is you can't. That's what people think. In fact, the answer is you can. You just have to pay a higher cost on the writes, but you need a clever system to do it. And that was mind blowing to me, because I thought the answer was no too. But we've got engineers, thankfully, that are smarter than I am, that figured out how to do this. And it's almost magical. But there are things like that everywhere in computer science. And I think the opportunity is really to get clever there and sort of redefine how things can be built so you can build things better.

[00:20:04] JM: To my knowledge, Quora, and I realized this is taking your point further than you intended. But to my knowledge, Quora started on MySQL. And a lot of companies will start on some fairly common well-understood solution, MySQL, MongoDB comes to mind, DynamoDB, maybe Postgres are probably like the most common places to start. Are you seeing people start with CockroachDB or migrating to it? What's the story there?

[00:20:43] SK: Absolutely. And that's why we are Postgres compatible. So we wanted to make sure that starting on Cockroach wasn't learning something new. Because it’s just, I think, need this friction and how many different SQL dialects or database API's do you need out there. I think we don't need new ones. SQL is a fairly well-understood standard. We chose Postgres instead of MySQL. We could have done MySQL. They’re both great. We could have done something to look like Oracle that we probably have gotten sued by them. I think that part is really important to getting developers to start on Cockroach. Of course, we do have developers that start on Cockroach, and we have since the open source project was in beta. So that happens. The question is how do you get more developers to start on Cockroach?

And there’s actually a pretty interesting story. When developers look at something like Cockroach, traditionally, I think there's a realization if they've heard of it. And Cockroach is neat. It does cool things. I might not need that for this use case. It's like Postgres. It's compatible with Postgres. So maybe I start on Postgres. And if it succeeds, I'll move to Cockroach in the future. It's very hard to move databases though. So we want a developer that thinks that way and that's interested in Cockroach’s capabilities, because they think it's going to be a good fit for their long term ambitions. We want them to start on Cockroach. So I think that's one of the central problems to solve for any company that's a new database in this competitive, crowded space.

What we're really trying to do to solve that is to realize what makes a developer excited about a new database beyond the nice differentiating capabilities that Cockroach has? And I think the answer to that is can you make a developer's life easier? Because if you can do that, then you can get a developer's attention, really get their intention. It's going to be easier to use Cockroach than it is to use Postgres. That is a good reason to start on it. Because if that's true, and you get these outsize differentiators that truly explain the cloud and the distributed architecture, then it's a slam dunk, right? You get you get both things at once. So we've been trying to puzzle that out. How do we make a developer's life easier? And I think, there, really, the realization is that relational databases are pretty high-friction. So if you're going to get an RDS instance, which is probably the cheapest thing you can get. A production ready RDS and AWS cost about $100 a month. So you realize there's actually significant friction to acquiring a relational database as a service. And if we can reduce that friction, then we can make Cockroach actually an easier place to start, even though it's a kind of newer, more complex, more powerful kind of database. It looks like Postgres. So that's a good start. But can we make it so that the friction is low? And we're saying, “You know what? The right way to do that is to make it so that, for developers, relational databases don't cost money anymore. They're literally free. And free forever. And they should always be free.” Kind of like the way Gmail feels, right? Obviously, Google has been happily monetizing Gmail. Meanwhile, it's still free for everyone. How did they do that? Well, you realize that you can charge in the sort of corporate context for every Gmail user. You can start to charge a user when their mail spool gets too big. But I think it's mostly the sort of corporate side that they monetize.

For us, we want to make it so that you can log in with GitHub. You don't have to create an account or anything like that. And you can get a Cockroach cluster. You can get 10 of them. And you will have a perpetually free, very generous tier. So in terms of like how many requests a day you can use it? We want to make it so that you can run, essentially, any use case on Cockroach for free as a developer. Never get charged. Never put a credit card down. Until you actually reach like a product market fit level. So think about if you're a developer that's starting a company. And I don't know if you've done it yourself. But a new project, you typically have pre-production stuff. You're doing Dev tests. You might have some big regression tests that are running. You might use it for CI/CD. You're sort of spinning up databases quite a bit. You might have multiple production databases. It's quite a few things. We want to make all of that free.

Most of the early starts have to do some serious iteration before they really hit resonance with their target audience and get product market fit. We want to get all the way up to the point where you start to release scale exponentially. And then you would exhaust the free tier and start to pay with a credit card. And eventually, if you become, say, the next Airbnb, we would expect you to move from that sort of consumption-based model to really having a dedicated Cockroach cluster that is within your security footprint with VPC peering and that sort of thing. So there's sort of this journey we want a developer to take if they're a new start, which is a free entry point, which is very generous. Eventually, you put your credit card down and pay for what you use. And then when you really graduate to the big leagues, you're going to have a dedicated Cockroach cluster.

And the way we make all this work is pretty fascinating. It's really introducing multi-tenancy to Cockroach so that we can accommodate really fine-grained usage and make it so that we're not dedicating VMs to something where someone's really just kicking the tires. So we can efficiently share resources and use them where they're actually in-demand at any given point, say, in a current day. And that really lowers the cost for us. And so we can offer these databases for free. And then, fundamentally, all of that's paid for by our big customers that land in the dedicated space, and they want that real enterprise experience an additional level of scale and so forth. So that's sort of the journey we're on. But that's where, fundamentally, I think you're solving the problem of how do you get a developer to be interested enough in Cockroach that that's where they're going to start, as opposed to where they might think that they're going to go eventually if they're otherwise going to use Postgres?

[00:26:33] JM: Very interesting vision, the whole ease of use platform onboarding thing. So the database as a service experience is clear to me. It's understandable why that is desirable to me. Can you take me inside building for that kind of product vision? Because to me, that's a vision with some real economic implications. So are you operating your own data centers?

[00:27:06] SK: We're not. So right now we use AWS and GCP for our fully-managed service, and also our network, our serverless, which is currently in beta and will soon move to GA. Right now, the economics are much better to use the public cloud providers. But there is a level of scale where that ceases to be true. But it's a level of scale that is somewhere foreign to our future, because the clouds are very competitive, and will give you extraordinarily steep discounts as you scale up within them. Further, the database is just part of any application stack. So if we had our own data centers, then those would have to have very fast interconnects with where our customers actually want to run their application layers. Where there’s just going to be data egress costs and things like that, the sort of inter-cloud network bandwidth actually is much more expensive. And it's going to create more latency as well.

So the realization is even if we made some economic sense for us to run our own data centers for Cockroach, it would still be somewhat at odds with how customers want to use Cockroach. So they already have a cloud footprint. They want to make sure that whatever cloud they're in, they can use Cockroach as a service in the same cloud even in the same availability zones so that they sort of minimize the latency and the cost.

[00:28:26] JM: I've done a few shows recently on cross-datacenter replication and cross-datacenter fault tolerance. Can you tell me, what does Cockroach do to enable that? And what is the networking look like between data centers to ensure that level of fault tolerance?

[00:28:49] SK: Yeah. Yeah, geo replication is the concept, what we call our business continuity resilience capability. And yeah, you very much – The expectation is that with Cockroach, and suddenly, if you run it, if you use Cockroach cloud, the fully managed services is always true. But if you run it yourself, this is typically what people do. You're going to use probably availability zones within a region to do your replication. And if you think about what that means, it means that you can lose an entire data center. And you're going to have other copies of the data in the other data centers you're using.

We use what's called consensus-based replication, which is, I think, in advance in many ways from the asynchronous sort of primary, secondary replication that you'd use, for example, on Oracle with Golden Gate and what’s typical on Postgres and MySQL. MongoDB, Cassandra, Cockroach, Spanner, Aurora, they're all using this consensus-based replication. And what it gives you, which is really novel over the primary, secondary asynchronous replication, is that when you lose a data center or a replications site, with consensus-based replication, you're still going to be able to get the right answer. You won't have potentially lost data. It's what's called a recovery point objective. With Cockroach or other systems that use consensus-based replication, you're actually able to set that to zero. Like you won't lose data.

With a synchronous replication, it's very easy to see how you can lose data. Your primary gets the write, the commit, and it doesn't make it to the secondary. You lose the primary replication site, you failover to the secondary. You don't have all the data. So you wrote something. It got committed. And all of a sudden your application doesn't see it anymore when the failover happens. That causes postmortems and a lot of headaches for developers, and teams and so forth.

With cockroach, what you need in addition to the primary and the secondary is you need a third, at least. You can have five, you can have seven, and that's kind of where you can dial how much high availability you have. But typically, it's three. And so what consensus means is that whenever you do a commit, you're not just writing to one out of the three. You're writing two in majority. So two out of three, or three out of three. When that's true, you can lose a minority. And one of the things that remains, one of the replication sites is guaranteed to have the right answer. And they kind of coordinate and make sure the two that remained would coordinate and make sure that they're giving you the right answer between them. That's geo replication, consensus-based geo replication.

The really interesting stuff that kind of goes beyond that, when you really start thinking, “Hey, we have a database that's distributed. We can do things like consensus-based geo replication. We're also thinking, “What else is the cloud gives you?” It's not just availability zones within a region. It's actually multiple regions.” And so you say, “Okay, if we can lose an availability zone and still have uptime, could we lose the whole region and still have uptime?” And the answer is yes. So you can have your replication sites in three different regions. Let's say you used US Central, US West and US East. Now, you could lose the entire East Coast and still have Central and West that would be replication sites that can always give you the right answer.

The cost of that, and there's always a cost, is that you've introduced more latency between your replication sites. And so that means that when you want to get a commit, you're going to have to wait longer. Availability zones within a region, they’re pretty close together. Like tens or hundreds of miles across the country. You might have to have, say, 30, 35 millisecond latency to get a commit. So you pay that price, but you pay it if you want to say I need a different level of survivability. Not just losing an availability zone because a backhoe went through a network fiber optic cable. But I want to actually allow a hurricane to knock out a big chunk of the power on the East Coast and still have complete uptime in my business use case. So it's pretty fascinating.

And then I'll add one more step, which is quite exciting. And that's when you say, "Okay, this database is distributed." The point of distributed databases is to exploit the cloud. I think it's something that isn't always obvious at first blush. But exploiting the cloud, it means, "Hey, in the public cloud, you can get all these data centers close by. You're going to have fast consensus replication, lose a data center and still have continuity." You can even use regions across a continent and have even higher level of survivability, and potentially lower latency to West Coast and East Coast and central users can kind of pin their data, the main copy of their data close to them.

Then you realize the public cloud gives you access to every populated place on the planet. And you're going to have data centers potentially next to every customer you care about. How do you actually embrace that and exploit it? And there the answer is it's not geo replication, because you don't necessarily – Lots of use cases, Quora is an example of one where you often replicate a lot of the data you care about everywhere. But many other use cases think about financial ledgers, and gaming, and retail accounts. And yeah, I say it's actually more common than not not to want to replicate the data globally. You actually specifically want to keep the data near the customer. And you may actually be required to only keep it near the customer, because there might be data sovereignty laws and so forth. But when you start talking about users in Australia, if you want to give them a great experience, you have to domicile their data close to them. That gives them a nice low latency. Also, the data is in their legal jurisdiction. Everyone's happy. So that's the next stage. And we call that geo partitioning. When you realize that, "Okay, the planet is a big place, and there's significant latency between the furthest points on either side of the globe, for example." So you have to fundamentally embrace the idea of a distributed database when you start talking about speed of light latencies that you simply can't improve unless you have reimagined how your data architecture looks.

One interesting way to think about it's just there's different levels of scale that you care about in your business. What happens at sort of the local level in terms of not losing data? How survivable I want to be potentially even across regions? And then do I really have costs cameras everywhere are potentially anywhere? And I want to make sure they have a great experience. And so all of that is well-served with a distributed architecture. In fact, it's difficult to understand how sort of those broader levels of concern could ever be well-solved without a distributed data architecture.

[00:35:21] JM: Your mention of distributed ledgers made me really want to ask you about crypto related stuff, because I'm sure you have some thoughts, but I'm not going to go there for this – We'll save that for another episode. As far as modern consensus-based implementations, have there been development – So you were at Google, right? Okay. So what kind of developments have there been in implementing that kind of consensus?

[00:35:48] SK: It's quite a bit of research. It's very active. It's. So when I was at Google, the idea of consensus replication was very new. I think it is really only introduced in the late 90s, Paxos. Yeah. And so Google, to my knowledge is, well, certainly one of the first companies that productionized it. And that was in a system called Megastore. They might have used it somewhere else before that. And then of course, became a big part of Spanner.

Cockroach, when we were getting started, Raft was the sort of new hotness, and it was just to be a more comprehensible version of Paxos. I'd say that –I don't know. There're pros and cons to all of these systems. But since then, there's been incredibly detailed and nuanced work around Paxos and all kinds of things that I think if they'd been available to us earlier, we might have been interested in using. So there's very active research. Lots of PhDs are looking into the problem. I think maybe that one really important takeaway from this is that, while so much interesting work has been done, and we might have used something different from Raft, if all of that had been known to us when we started in 2014. We haven't changed it.

So you realize that consensus can have all kinds of sort of nuanced improvements to it. But ultimately, consensus is at root consensus. And if it works well enough, one learning for us is that – Let me tell you, one of my cofounders, Ben Darnell, he likes to say this, he spent a week implementing Raft. And then we've spent now six and a half years making it work. You can implement them as a school project, and many people do, the devils in the details. And it's incredible how hard it is to actually make these things work in production. So, yeah, you start with something. If it's consensus, I think it's the right way to do things. It doesn't matter which of the different flavors. I'd rather be starting a database in 2021 to choose exactly what would be the perfect thing. But maybe one more takeaway from this whole journey is just that the things that you imagine a database needs to do in 2015 evolve. And some of that evolution informs the interest in potentially having used a different consensus algorithm in the first place. But we didn't know those things when we chose the algorithm. We didn't understand how important geographic scale was, for example, to modern business use cases.

[00:38:25] JM: What do we need out of a distributed database in 2031?

[00:38:29] SK: That's a great question. I usually try to set the sort of threshold –

[00:38:33] JM: 2026?

[00:38:34] SK: 2025, '26. Yeah, that's kind of where my head is most of the time. I think the answer is really quite apparent. All you have to do is look at what big tech is doing. By big tech, I mean, let's call it the Fang architectures. Why has Google built distributed data architectures for everything they release? And they have platforms, really, internally where you don't launch something that isn't embracing a global footprint. Facebook has probably spent engineering millennia. Literally thousand plus years of engineering time building their hugely complex and extraordinarily functionally amazing distributed database architecture on top of hundreds of thousands of MySQL nodes. So why do these companies spend what has to be hundreds of millions of dollars on this R&D? And it's not just a single time cost, of course. It's cumulative. And it goes on and on forever, because you have to maintain these things. And the answer is because it gives you a competitive advantage. And so what does big tech fundamentally doing there? Well, they're solving for two kinds of scale. Well, there's resilience. So it's mitigating risk. You always want to be up. You can't screw up the customer experience by not even being available. So that's a huge part of it. But then it's like, "Okay, big tech is solving incredibly data-intensive use cases." But those had become very common. Even startups have them. Certainly, the gaming companies that do well, they go from wanting a three node Cockroach cluster to like potentially multiple hundreds of Cockroach cluster. It happen almost overnight. So data intensity is another big thing.

But the global reach, or at least the multi region, wherever you want to do business, is a huge part of what big tech is solving for, because their customers are everywhere around the world. And, again, that has become extremely common. So of course, the multinationals out there that are a bit older and need to catch up to big tech, or at least remain competitive. There's also every startup.

So when I really think about what is a distributed database need to do in 2025, ‘26? I think it's enable every developer, every new start to build the way big tech does. And those are fundamentally the capabilities Cockroach has. But the challenge for us started off as being just building that capability. Can we make a relational database that can do these things that Spanner can do? The reason Google built all that stuff and put that money in there? But then the new challenge is how do we bring that to every developer? Actually, it's a really incredible challenge, because if you want to have in the sort of traditional way of running Cockroach, let's say three regions around the world where you have customers, say, US, the EU, and APAC, that would imply that you probably want three availability zones in each region. So we're talking about nine different availability zones. You're going to probably want some number of nodes, at least two in each of those, maybe three. So potentially 27 VMs for Cockroach nodes. You're going to need nine different Kubernetes masters and that have costs within the clouds. You're going to need the right kinds of load balancers in each one, the level, the global load balancer. It's a tremendous amount of effort and fundamentally cost to give it to you in sort of the dedicated package.

So what we've really been working on and investing in is that serverless consumption model I was talking about, which is how do we take an incredible cost structure to build something like big tech and make it so that you can have fractional ownership, like a virtual cluster that sits on this truly globally distributed physical cluster? You get a fractional slice of that, which to you feels like a completely isolated cluster both from security, from noisy neighbors. And that's a huge challenge that you have to then grapple with. And then how do we make it so that the unit economics work for us, right? So that we can give that away for free and really accelerate all those new starts and make it so that developers look at databases differently and fundamentally, don't expect to pay for them, so that they can do more, they can do it faster? And, hopefully, choose Cockroach. So that's really what our vision is. It’s kind of how do you allow developers feet to really never touch the ground? By the ground, I mean are you dealing with an operating system? Are you dealing with Kubernetes? Are you dealing with instance types, and sizing, and regions?

I see a world where you basically are able, as a developer, to create something on your laptop, the backend, the mobile app, the web app, and launch that into the cloud, and have it scaled to any level of usage, any geographies, any clouds. And just to have that all happen where you pay for only what you use. So you start really small. You don't pay anything. You can scale to run a major financial institution's retail banking. So you want to have that whole journey happen for a developer without them having to deal with the operational complexity that's very common in 2021.

[00:43:37] JM: I know that the name Cockroach is supposed to convey that a CockroachDB instance can potentially survive like a cockroach can, can survive a nuclear catastrophe, for example, or the disappearance of a data center. The fact that you've built a company around this concept concerns me that other pieces of infrastructure on the Internet are not so reliable. Do you have any worst case scenario, Black Swan thoughts about what happens if, name your cloud providers, East Coast Instance, or East Coast geo like disappears all of a sudden? Or maybe even worse, a couple of those regions disappear. Are we just toast basically as a civilization?

[00:44:29] SK: We just got through a global pandemic. So my confidence in the global system has actually gone up, which is good, because it could have gone either way before this thing started, in my opinion. If the whole East Coast goes down, no, we're definitely not toast, right? If all of AWS goes down, like all of it, some sort of systemic flaw or some – I don't know. An actual deliberate attack of some sort. It's still not going to take us down, right? We have plenty of competitive offerings across the clouds. And, typically, most businesses choose one of the clouds. But that's something that is on most CIOs kind of roadmap of how do we even limit our systemic cloud risk? So I think it's not as much of a monoculture as might sort of seem to be at first glance. You start to look at all the different ways that things are built. But there are some monocultures in there, I think. There're things that are used across the clouds, like very critical pieces of software. There might be zero day exploits in there. So deliberate acts that really takes down the whole Internet is a possibility. And I suspect it would come from a state actor, if it did. That would probably take the world down. And it's hard to say what would happen there, but it would not be good.

We're not trying to protect it against that, right? Cockroach isn't going to stay up if that happens. Cockroach can stay up, though, if much more common failure scenarios. And that's what most people are planning for realistically. Lose a datacenter, lose a region even. Potentially lose a cloud. And we have customers that aren't in production, but are interested in having their replication sites in three different clouds, private cloud, AWS, GCP, connect them with fiber. And you actually could have AWS run into a pretty big systemic issue and still have complete business continuity.

Again, like it's always the details catch you, right? Any complex application isn't just a nice distributed application layer and then a nice Cockroach cluster underneath it, let's say. As soon as you get to any reasonable level of complexity, you're using all kinds of other services. Some of them are AWS specific. Some might be using Confluent for data connectivity. You could be using Elastic, using Mongo. And this is not uncommon. Most use cases do use all kinds of different tools and services out there. So the question is what really happens if you lose a cloud, or you lose a region, or you lose an availability zone because you've created this complexity with a lot of different things connecting? And typically what happens in those things, when one thing stops working well, or at all, things start to pile up behind it, and it starts to take down other systems, have cascading impacts.

The only way to solve for that, fundamentally, is you've got to run the full scale outage test. And you have to do it with some degree of periodic frequency. Otherwise, whatever you build, the complexity quickly goes beyond your last test. And you've probably introduced another sort of failure point. So I hope I answered the question there. But I think there's a world that made, maybe that 2026 timeframe, where there's really good stacks that help people manage that complexity, and still get the kind of survivability across the whole complex architecture that they're looking for.

If you use these vendors, you connect them in these ways, even things like low-code platforms that are hosting many different applications that do lots of different things, but all on the same architecture, right? That could be like really a monoculture that all fails together. But also, it's more likely that you can use the economies of scale of having that one platform host many use cases to do the right kind of testing so that all of the use cases survive when you have one of these sort of major systemic failures.

So listen, it’s I think where a lot of the interesting work remains to be done in terms of proving out the potential for a lot of these technologies, something as simple as Raft, right? But it's underlying so much now in the larger ecosystem. And then I think now the thing you have to turn your attention to is the various layers above Raft that become increasingly complex. How do these complex systems interact in practice? And what happens if you have these major failures?

[00:48:51] JM: All right, we're up against time. Just to close off, you are definitely running one of the most successful infrastructure companies that have been started in the last six or seven years. Give me a few counterintuitive lessons you've learned about building a successful infrastructure company in this era.

[00:49:14] SK: I don't know if this is counterintuitive, but by far, the most difficult scaling problem is engineers, engineering talent. And it feels like it's only becoming harder. And so I'd say that that's probably a good signal that if you're thinking about a career choice, becoming a software engineer is not a bad way to go. I suspect most of the people listening to this blog have already made that choice, or this podcast. So that's good. It's good for the future of our industry, I think.

Counterintuitive, let me think about that. Well, here's one for you. When I started this company, I never thought of myself as being interested in building technology that then had to be sold to big, I guess, stereotypically slow moving enterprise companies. Choose your global 2000, fortune 500 company. I felt like, “Oh my god! The amount of –” I Came from Google, where I was building for Google engineers internally. There's very fast moving. You could have very sharp edges. And then the idea that, “Okay, we're going to help modernize, help the digital transformations of these big businesses.” That felt like I can't believe I'm doing this, because that's going to not be something I enjoy. And it turns out that is incredibly enjoyable. So it's the sort of, I think, other half of what the challenge is and the excitement is actually getting these big customers and making them successful. And so that, just from a pure technologist perspective, has been a surprise for me. And I'd say that's been one of the most rewarding parts of the journey that we've been on as a company and me personally as an engineer, and also as a CEO.

[00:50:58] JM: Spencer, thanks for coming on the show. Great talking.

[00:51:00] SK: It's been my pleasure. Thank you, Jeffrey.

[END]