EPISODE 1413

[INTRODUCTION]

[00:00:00] JM: Database product companies typically have a few phases. First, the company will develop a technology with some kind of innovation such as speed, scalability or durability, then the company will offer support contracts around that technology for a period of time, before eventually building a managed hosted offering.

PlanetScale is a database company built around the Kubernetes based Vitess Technology. Sugu Sougoumarane is the CTO of PlanetScale, and formerly worked at YouTube, where he built large scale SQL databases.

In today's show, he talks about building out the core PlanetScale technology, tuning PlanetScale's consensus model, and the development of the hosted cloud service offering of the company. If you are interested in sponsoring Software Engineering Daily, we reach over 25,000 engineers per episode. And 250,000 engineers per month. If you are promoting a product or looking to hire engineers, Software Engineering Daily can be a great place to get your message out to a large populace of experienced engineers.

To find more information about sponsoring Software Engineering Daily, go to softwareengineeringdaily.com/sponsor. We'd love to hear from you.

[INTERVIEW]

[00:01:15] JM: Sugu, welcome to the show.

[00:01:15] SS: Hi, thank you.

[00:01:16] JM: You've been writing articles about scalable consensus, and you've been building PlanetScale for several years at this point. I want to ask, obviously, it's harder to achieve consensus at a large scale, then a small scale. So, when you think about a gradient from what you would consider a small deployment, maybe something like three nodes, to a much larger

deployment where I don't know, you have hundreds of nodes or thousands of nodes, or whatever you consider large, where are the places where consensus starts to break down and demand that you make tradeoffs?

[00:01:52] SS: So actually, the dangerous thing about consensus is if you go and ask somebody, what is the level of safety you want for your data, they will always go to an extreme. Say, I never want to lose data. The danger about consensus is that you may configure a system that way. It may actually work for a period of time, and sometimes, it may work well for even like a year or so, right? The problem happens when how such a system tolerates failures, right? If there is a failure, can the system continue to make forward progress? Until you see such – some of these failures are really, really rare. Some of these failures can happen like once every two years, once every year.

So, at that time, you don't like suddenly, if the system completely freezes up, and there is no way to recover, and then you go into this panic situation. So, the way I see consensus is you have to trade that off with availability, and then make good decisions where if, at some point of time, the system encounters failures or drops in performance, this system is well balanced enough that it can continue to make forward progress. I don't know if that answers your question.

Typically, if you have a very small system, like three nodes and stuff, such failures, you may never see in your entire life. But if you're running like tens of thousands of nodes, these failures become more common. In many cases, some of these failures are so common that a human cannot come in and intervene and investigate. Because the failure is complex, it takes time to analyze, and while you analyze, your system is not serving any data. So, it is better to think through all those failure scenarios and make a decision upfront about what is the decision you would make if you encountered that, and then code that into the software that makes those decisions. I think that answers your question the way you asked. You can ask me clarifying things on this.

[00:03:58] JM: Yeah, I guess what I'm curious about is in practice, where do you start to see situations that arise where consensus breaks down and you have nodes disagreeing in a programmatic fashion that demands a leader election?

[00:04:16] SS: Yeah. Typically, consensus doesn't really break down. It is a question of like, for example, let me see if I can think of a use case scenario. Let's say there are seven nodes. And then you say that for your data to be saved usually in a seven, nobody runs a seven-node system. So, I'm taking an extreme example. In a seven-node system, if you use a traditional algorithm like Paxos or Raft, it would require four nodes to have the data, and only then consensus would be considered as reached.

So, in that scenario, if there is a node failure and a network partition, then your system is stuck and is not able to make forward progress, because there is no situation where you are able to reach for nodes. But then you want seven nodes for other reasons, availability. You may want to serve traffic from another data center, and et cetera, right? So, in this case, so such a system, if you go like that, such systems don't scale well. But the thing that I was trying to talk about is, if you have seven-node system, and you still want good durability, you still want to be able to say, "If one other node, two other nodes have the data, it is sufficient." It is not necessary for four nodes to receive the data.

If you are able to specify that, then you get the best of both worlds. You get the durability that you want, because you're unlikely to lose three nodes at the same time, or two nodes, depending on your tolerance to failure. And at the same time, you can add more nodes to the system to satisfy other things that you need from the system, like availability, scalability, et cetera. So, what I was trying to do is decouple the durability requirements from the number of nodes that you would want to run in a system where you want to increase the number of nodes for other reasons. So, that's kind of what I was driving at in those blog posts.

[00:06:25] JM: What was the impetus for writing those blog posts?

[00:06:29] SS: So, it is mainly because today's world is a lot more complicated than the simplistic approach that traditional consensus systems use. For example, you go to a cloud, you have zones, you have regions. Sometimes you have cross continent nodes. And if you add all these things, the simplistic approach is very hard to configure and maintain. So, you want to be able to decouple your durability requirements from the fact that you have nodes all over the place, and you have architectures that are all over the place.

For example, if you're a stock trading application or a bank, you may want your data to be across geographical regions. But if you are as a retail shop or something, being across nodes is good enough. But then you may still want to have other safeties, even if there is no losers zone, sometimes you may say, “I still want the safety of data being somewhere else. But when I operate at high speed, being within a region is good enough, as long as they are in different zones.”

So, there are more and more complex rules that people come up with, that you cannot specify using a traditional consensus system. If you were able to separate this out, then it's a lot easier to say what you want and achieve.

[00:07:52] JM: Can you tell me more about how consensus challenges have evolved over the, let's say, last 10 years as cloud architectures have become more and more complicated?

[00:08:06] SS: Yeah. The main reason, consensus, I think is the original Paxos paper is tens of years old, I believe. When it came out, it was not really as important, because systems are not distributed as they are today. And the main reason is because of the law of scalability of compute, basically, CPUs have stopped growing in speed. So, the only way you achieve more and more is by distributing things. And once you start distributing things, software that was written to work on a single computer is not relevant anymore.

Now, you want software to work with other pieces. You want to split your software into smaller pieces, put them on different computers and have them talk to each other. And when this started happening, suddenly, consensus became the theoretical foundation to solve some very difficult problems about how the software pieces, keep their data consistent with each other. So, it's only going to get more and more important and there's going to be more and more demands on more flexible systems, I believe.

[00:09:18] JM: When you say flexibility, the test uses a system, the test which PlanetScale build with, use a system called Flex Paxos. What is meant by that flexibility? What are you talking about there?

[00:09:31] SS: So, Flex Paxos actually, there's a paper by Heidi Howard and Dahlia Malkhi. They published it a few years ago, that actually is the foundation for what I've talked about, is that paper, essentially, when it came out, it wasn't as clear but it is a lot clearer to me now, separates this durability from the number of nodes that you want to operate in a system. You can basically say, I want 11 nodes in my system, but my durability requirements are only two nodes or three nodes. You can specify that and the safety rules that exist with Paxos apply on such a system also.

Whereas if you use Paxos on like 11-node system, you would require to reach like six nodes before any transaction is considered complete. Whereas if you use Flex Paxos, you can continue to operate with the same safety using just like being able to eat three nodes when you save your data.

[00:10:34] JM: How is that possible? Isn't that going to be inherently less safe if only half of your nodes or some lower percentage of your nodes have received an update?

[00:10:43] SS: Yes, that's an excellent question. So, the one thing that – this is something that confused me. It took me literally years to understand is Paxos is actually two algorithms working together. So, there is one algorithm which basically ensures that your data is saved in more than one place. That's one side of Paxos. The other algorithm is the one that safely elects a new leader, where if one leader is down, you elect a new leader, and then that leader is responsible for saving data in more than one place.

So, those are actually two algorithms, but they work hand in hand. And if you look at live systems that run at higher QPS, you do the first part, at really, really high QPS. High scale system, do tens of thousands of transactions a second. We test doing millions of transactions per second, right? So, that is part one of the algorithm. Part two of the algorithm comes into play only when there is a failure, or there is a need for maintenance. If a node goes down, then the second algorithm comes in and if safely elects a new leader, or if you're like rolling out software, you want to bring that node down yourself, you elect a leader, and then upgrade that node and then bring that node back online.

If you look at this thing, on one side, you're doing millions of QPS. On the other side, you are doing fail overs or leader elections, which may be once a week, once a day. So, you don't want to equally balance both these processes. You can say that the high QPS system does not require that many nodes so that it has lesser roadblocks for it to complete its transactions. But the one that happens less frequently can have more nodes involved, more nodes required to make forward progress, because it happens only once in a while.

So, that is essentially what Flex Paxos allows you to do, is balance this out where the high QPS system really needs only two or three nodes. But the failover requires a lot more like in a three-node system, the failover for 11 – sorry, in 11-node system, if you say three nodes are required for consensus, then you need 11 minus 3, which is 8 plus 1, 9 nodes to perform a successful failover. Does that make sense?

[00:13:21] JM: Yes, it does. So, to be clear, the issue with a more rigid Paxos is that you're just going to have higher latency when you have leader a leader election?

[00:13:37] SS: So, given that network is very flaky, right? So, network is don't have – network performance is not predictable. There are like automatic back offs. There are all kinds of things built into TCP IP. So, you cannot always expect the network to have consistent performance. And the more communication you add between nodes, the higher you would see in terms of effective latency on the client side. If one node has to talk to only two nodes, and if at that time, like let's say the network blips once every second, you are less likely to see that one second, that every one second blip. If you have to reach only two nodes, versus if you have to reach six nodes, you will see that blip a lot more often, which effectively translates into performance degradation on the client side.

[00:14:36] JM: When I was reading about Flex Paxos, I was surprised I hadn't heard about this before. I just heard about Paxos and Raft. Is there a reason why Flex Paxos is not more widely used among databases?

[00:14:51] SS: I find that really really surprising. I have no idea why – I thought this this will catch on like wildfire. I have been excited about it. I've actually refined like the blog posts that I wrote are basically refinements from that Flex Paxos. We are doing this in Vitess, and the Vitess

community is pretty excited about this. But I don't know why others have not taken this on. I feel like systems would be a lot more stable, you can achieve better availability if you used Flex Paxos, but I don't fully understand why it has not caught on yet.

[00:15:31] JM: So, I just want to go a little bit deeper. Again, the thing that is being made more flexible in Flex Paxos is the node count that you need to reach before you are determining your – the node agreement count that you need to reach before you're determining yourself as having reached a –

[00:15:57] SS: Like having satisfied the durability requirement.

[00:16:00] JM: Durability. The durability requirement. Durability, I guess that the definition here is that you're not going to lose data, is that correct?

[00:16:09] SS: Correct. Correct. Of the more accurate way to state it is a system can tolerate up to these many node failures before losing data. So, if you say three nodes, and then more like even within Google, Google doesn't expect three nodes to be lost at the same time. They say you can lose up to two nodes. So, they have this N plus two rule. Typically, you set some limit that is reasonable, and for all practical purposes, those limits almost never happen in real life.

[00:16:43] JM: How do you test a consensus protocol? Do you have some dedicated automated tests to simulating failures?

[00:16:52] SS: So, there is actually, I forget the name, there is a theoretical thing it's called Tal or something. There is actually like a formal way to test and verify that consensus is accurate. But I don't know, I haven't used that system. What we plan to do is we have four Vitess. We are pretty confident about all the corner cases that we believe need to be covered. This system is actually under development as we speak, and we are writing tests to specifically recreate each and every one of those failures to make sure that the system handles those failures correctly.

[00:17:35] JM: I think of the work on consensus algorithms as fundamental to what PlanetScale was initially – what I understood, that PlanetScale initially was built around, which is just raw database scalability. I feel like the company has progressed to a point where now you're moving

towards database usability, beyond just database scalability. Do you have like separate teams within the company, some of which that are focused on core engineering of the database, fixing up edge cases and potential problems working on Vitess? And then other teams that are working on some of the higher-level functionality?

[00:18:24] SS: Yeah, absolutely. It is necessary, because each of this task is so deep, that it is not possible for a single engineer to understand and grasp and solve everything. So, we actually have multiple specializations within PlanetScale. Even within Vitess, there are specializations. There is a team that does query serving, there's a team that works on materialization, and there's another team that works on cluster management. And this is part of cluster management consensus. And then beyond Vitess, we have a team that works on orchestration, like being able to roll out all these Vitess nodes and managing them, which is based on Kubernetes. And then we have probably two other teams, one focused on writing the front-end application and another team on design and UX, and those areas. So yeah, it's multiple teams and multiple areas of specialization.

[00:19:25] JM: What's the day to day right now for the core database engineering side of things? What are the teams, I think about like something like Flex Paxos. I imagine that's pretty fully baked in the product at this point, but I'd love to know more about what the core database engineering teams are working on.

[00:19:47] SS: Yeah, so the Flex Paxos product, we call it VT Arc. It's a wordplay on VT Arc, because it's Vitess. There is a team now putting together – so we actually have the Alpha Out, which doesn't cover all the corner cases, but we know what they are. So now we are in the process of iterating forward and closing those gaps. So, that's the consensus team. The cluster management team. But then the problem that planet scale solves is a gap that is existed in software forever, right? So, there are this whole set of new engineers that have come that don't – that have their own big stacks to deal with. They have Rails to work with, they have, what is the? Tailwind. There are so many stacks to learn. Learning all those stacks, and at the same time, knowing how to operate a database is kind of unfair as an expectation. What they are looking for is these things to be packaged. That is the other side of things.

So, for that thing, the part of Vitess that helps is the materialization section of Vitess, which actually allows you to copy data from one form and transform it into another form. So, that core product of Vitess is used for like one of PlanetScale's signature feature, which is schema deployments. And then that same thing is also now used for doing safe automatic import from an external database. So, if I'm already running on MySQL, myself, or on RDS, and want to import the data into PlanetScale, just click a button, and all the data is safely imported for you. And then you can also revert it, because we can actually reverse that replication back to the source.

So, we have all those cool features that we have built, which is the built on the core materialization feature, which we call the replication in with Vitess. That's the second team. And the third team, we just make sure that more and more queries work and that they work better, because Vitess is a Sharded system. And some queries that would trivially work on a single instant, my instance, MySQL, are a lot harder to make work in a system that is Sharded. Sorry, that was a long answer.

[00:22:00] JM: No, it makes sense. Can you go a little bit deeper into that? What makes it hard to build around a Sharded system?

[00:22:09] SS: Yeah. There are two big challenges in a Sharded system. As soon as you make a decision about sharding, you've essentially made a decision to put your data in different places. But then as soon as you put your data in different places, when you run a query, that query may want you to combine data. And that way, the way that query wants you to combine data may make it awkward, like for example, you may say, select all rows ordered by a column. But if the system is not Sharded by that column, you are essentially asking this proxy to fetch all the data from all the rows and then do the sorting, right? Which means that you have to fetch the whole database. But the whole reason why I Sharded the system is because it doesn't fit on one machine.

So, at that time, we have to come up with clever ways of achieving that, like in this very simple example, I used a very simple example, there are a lot more complex ones. What we do is we actually ask each Shard to return the rows in that order. And then we we merge the results as they come and give you an audit version on the front end. So, as these queries get more and

more complex, it becomes harder and harder to achieve these things. That is where the challenge lies in terms of Sharded, being able to serve complex queries in a Sharded system.

[00:23:37] JM: Can you tell me a little bit, we've discussed this in previous episodes, but you've had a little bit more time since we last spoke. I'd love to get a macro reflection on how the product has evolved since the company was started, and some of the major milestones, some of the major challenges that you've overcome along the way as you've been taking the Vitess to a full-fledged product.

[00:24:05] SS: So, I believe the challenge with Vitess has been, I would say usability and that's what we've been focusing on. So, for example, making all queries work is a push towards usability, right? When somebody is running MySQL, and they come to Vitess and they Shard it, and then some of your queries start failing, that's a usability problem. So, if we made that query work in a Sharded system, more and more systems can migrate more easily. And then like the way I would put it is, Vitess was built for SREs to run a system. We are slowly moving that towards Vitess can be you know used by anybody. If somebody comes in, they shouldn't have to do like 200 things before they get Vitess up and running, is what we've been mostly focused on. And VT Arc product is the final piece in this where if you now bring Vitess up with VT Arc, it pretty much manages itself. You do not have to do – you have to do very little tinkering with Vitess itself once you brought it up this way.

[00:25:09] JM: So, we talked a lot about the database engineering, getting into some of the higher-level features, you built in database branching. Can you talk a little bit more about what database branching is? And then we can talk about some of the engineering challenges of that?

[00:25:26] SS: Yeah. The first version we released, we call it database branching, but it was just schema branching, which actually is a huge leap forward, because nothing like this was ever done before. Being able to view schema deployment as you would do software. You branch your code in software, make changes and then merge it back. Viewing schema deployment along the same way, that was basically all Sam. You should ask him about it, when you talk to him. He will be able to tell you better. It’s something not only makes it easier, but it's also intuitively understandable for a typical developer, because they have been doing that with code all the time.

So, when developers saw this, they just caught onto this pretty, pretty fast and quite easily. That is the schema branching. What we have now extended it to is actually data branching, because the one challenge with schema branching is yes, you've branched your schema, you've applied your changes, now you want to test it. But then when you test it, there is no data, and most of the software written is not testable, unless you prime it with some data. The next step that we are doing is when you do the schema branching, we will also copy the data on top of it. So that, after that, once you apply the schema changes, you can instantly test your software against that data.

The next step, we are likely going to do – so this works if your production is really tiny. If it is just a gigabyte, or 10 gigabytes, 100 gigabytes, you can branch it. But what if your data size is hundreds of terabytes? The thing we are going to be looking at is when you do this branching, you actually copy a subset of the data to the target, and then that will then allow you to test on a smaller subset. There are lots of questions around it, but we are talking.

[00:27:31] JM: How has the development of database branching compared to the development of the core database product? Obviously, it's much different. But the how does the testing and the engineering roadmap compare?

[00:27:47] SS: So, you're talking about Vitess, the core database product?

[00:27:52] JM: I'm talking about the database branching. What's been the process of –

[00:27:58] SS: Yeah, so these features are all core, built on top of core Vitess features. The data branching relies on Vitess' ability to backup and restore data. So, the main database is continuously backed up. When you branch, we basically restore from an existing backup.

[00:28:17] JM: Got it. So, if I want to make a branch to, let's say, I just want to test something, I want to test a new database schema, what's going on under the hood? Can you walk me through the path of issuing a schema change?

[00:28:37] SS: Yeah. So, what you do is – the UX is very simple, right? You go to the thing and say, click a button and saying, create a branch, and it says, what's the name of the branch, and poof, your branch is created. Under the covers, what we do is actually create a brand-new cluster for you. In that cluster, we apply the schema that is in the source database. So, your branch is created. If you requested a data branch, then we also find the latest backup from that cluster and then restore it from there. So, then your branch is created.

After that, what you do is you apply your schema change. When you apply your schema change, we have this new cool schema with that we call online DDL. There are many talks by Shlomi about it. He's one of the developers behind it. And what that uses is Vitess' replication feature to apply your schema change without downtime. So, even if there is a lot of data, it doesn't do any locking, everything is done asynchronously and the switchover is atomic and almost instant. So, the frontend never notices that the schema has changed. When you apply your schema change, that schema change takes place and then you test your software. Then after you tested your software, you say, "I want to merge this back into the main database." And then that time, a new workflow that is initiated, which actually compares your schema against the source schema, finds out what is the diff, and then says, "Oh, this is the diff that we are going to apply to the source", and then goes through an approval process. And after the approval process goes through, you click a button, and the same online DDL process now runs on your source database and performs the schema update.

[00:30:32] JM: How does this compare to a schema migration within a database system that may not have this kind of database branching feature? Why is database branching make for more safe migrations?

[00:30:52] SS: So, the one additional safety that we are adding now, which really, really makes it safe is, we are now going to publish a feature. Actually, it may already be out, which is the ability to revert your schema change. So, if you perform a schema change, and you realize, "Oh, God, something broke, somebody forgot to change their code", and stuff like that, you can instantly revert that change. So, I can talk about that later.

But the reason why schema has been – so if you go and ask any developer, why many developers go, avoid using a relational database, and if you go and ask them why they do it is

because, I don't want to face another schema migration, is almost a universal thing that every developer would say about a relational database. The reason is because the process is actually completely haphazard and not organized. Typically, what a developer will do is, “Oh, I want to add a column. Okay, where do I do it? I don't even have a testing database.” Many organizations have this staging database, it’s a common database that everybody uses for testing. So, I just go and make that change in that database. And as soon as I make that, I bring somebody else because they are also sharing that same database with you, right? That is my number one.

Let's say you’ve gone through all that. And then you then go want to deploy the schema into your production, and then you will now have to go, essentially, every developer would say, “Now, I have to go deal with the DBAs, because they are going to throw a tantrum about deploying schema changes.” Because they have five other schema changes lined up that they have to deploy, because DBAs typically deploy schema changes into your production, and the DBAs are under the gun for not breaking production. So, they have to review the schema change, they don't know why the schema changes are coming. The way they do it is they slow it down for you, because they don't want to make too many changes into production at the same time. So, you are now behind a queue of a DBA that has to review your code, your schema change to review that it is okay.

And then, when the schema is deployed in production, that can have issues like table can get locked for an extended period of time during which there is software downtime. So, that is another problem with schema changes. All these changes, kind of add up into a very big hurdle that developers have to cross to make schema changes, to the extent that pretty much developer says, “If I can just do this feature without a schema change, I would try to do that.” So, they start taking shortcuts, trying to say, maybe I can cram my data into this blob, because changing the schema, I have to go through this huge process. There are things that developers do to avoid schema changes, because of these hurdles. Essentially, what we have done is removed these hurdles, and which allows you to be more creative about how you want your data to be laid out.

[00:33:59] JM: That's great. So, at this point, we've talked about features of the database itself, and limited talk now about getting PlanetScale to a place where you can have a managed cloud

product. I've interviewed a lot of database companies and this is frequently something that's quite difficult, it's a mandatory phase of your product development as a database company, where you have to build a managed cloud product where somebody can just spin up a remote instance and it's managed by the company, what was the process for building a managed cloud product like how difficult was it?

[00:34:40] SS: So, actually the challenging part is that the minimum requirement for a viable database product is very, very high. There is a huge amount of plumbing, observability, automation, and also, on the front-end side you have to add authentication, security. All these, what I would call static requirements for any product to be viable had to be built. That was the biggest challenge for us, getting all that built up. And to the extent that people can come in and trust the product to be usable. Now, that we built all that, on top of this, now we can build really beautiful and attractive features.

[00:35:25] JM: I was looking at the architecture for a deployed PlanetScale instance, can you tell me, what were some of the, I guess, the learnings or the insights you had, as you were designing a deployment of a particular PlanetScale instance? How did you decide how to do backups and how to do observability on a single instance? And then we could talk about how to manage those instances at scale?

[00:35:55] SS: Yeah, so a lot of this experience came from the automation that we built at YouTube, for Vitess. The biggest pieces are being able to write code that typically humans do otherwise in a regular system. For example, Slack runs Vitess. But in Slack, you don't create a database every day. In Slack, there's like – databases, and people live with those databases. But then when we built PlanetScale, we had to create and drop databases all the time. Because now users come in and say, “Oh, spin up an instance. You have to create a database for it.” So, that is kind of the challenge that we had to face in PlanetScale. Fortunately, all the work that we did in YouTube, that we pushed into Vitess came in really, really handy for this type of automation.

I'm trying to see if there is something more specific that I can talk about. The testing of this was the hard part, mainly because this was not something that has been done before, even with

Vitess, right? Bringing up and bringing down thousands of databases per day is not something that people do normally. So, doing those things was pretty interesting.

[00:37:22] JM: Did you build cloud specific deployment setups like? Do you have a Google Cloud version that uses Google Kubernetes engine and Amazon version that uses AKS? Did you have more generic?

[00:37:38] SS: So, the core product is generic. The core product actually can run on Google Cloud or Amazon. But the currently the planet scale features are all on AWS. The branching and those features are currently only on AWS.

[00:37:54] JM: Gotcha.

[00:37:57] SS: But we are capable of bringing up clusters, just PlantScale clusters in Google Cloud, if needed.

[00:38:05] JM: Do your customers typically want to use the manage cloud product? Or do they like to manage it on their own cloud resources so they have finer grained control?

[00:38:15] SS: They definitely want us to manage. Some of them are willing to do shared credentials, where we use their credentials to manage the software, but they definitely at least, the predominantly – most of the customers definitely want us to manage the product.

[00:38:32] JM: Gotcha. What about like, the larger, like the massive tier of scale, like Square? Say specifically, but there are some gigantic customers you have or Slack?

[00:38:44] SS: Yeah, so those want to run on their own, for one reason or another. I can't name names, but some of them are actually considering when we found PlanetScale, we didn't have the PlanetScale managed cloud database. So, the way we started PlanetScale, we started supporting companies like Slack and Square. But then now, some of them are thinking, “Oh, the PlanetScale manage looks attractive, maybe we should migrate that into the PlanetScale managed.” So, that is definitely something that customers are considering. Not all of them, because many of them have varying restrictions, requirements that they have to go through. But

then, I believe that it's only a matter of time before people – because the way I see it is, a product company also invests in technology, but eventually they want to focus on the product, and then technology kind of becomes a burden for them. And they would rather outsource that burden to a company that is specialized in managing that technology.

[00:39:49] JM: Right, of course. You mentioned observability earlier. I'd love to know what is required to have scalable observability across a large number of cloud database instances.

[00:40:05] SS: Let me see if I can – this is difficult to answer without a use case. I'm trying to see if there is a use case. For example, like, we have the ability to take backups. And once in a while, a backup breaks. If a backup breaks for too long, and the way we architected is the backup depends on a previous backup, plus the bin logs is the way I would like to state it.

So, if a backup breaks for too long, then there are not enough bin logs available to actually build a new backup. In this case, typically if a backup breaks, it's fine, somebody can go in and take care of it. But then, if now you have thousands of instances, backups break all the time. But when a backup breaks, I want to be known, I want to be notified before the time expires, because I need to go fix it. So, in this case, we had to build observability for backups. Same in the case for failover. Same is the case for stuck branches, right? So, all these things at scale, you have to build observability. Otherwise, it just falls after our visibility, and then it becomes a bad user experience. So, we have observability for query performance, we have observability for failover times, we have like observability for backups. What else do we have observability for, time to take for cluster to come up. So, all these are metrics that we closely follow in PlanetScale.

[00:41:42] JM: Do you use off the shelf monitoring software? Do you do use vendors? Or do you –

[00:41:48] SS: We do use vendors. Am I allowed – yeah, I've used Chronosphere.

[00:41:53] JM: Nice. Okay.

[00:41:55] SS: I think we have publicly stated somewhere that we do use Chronosphere.

[00:42:00] JM: Can you say anything about your monitoring schema and how you like aggregate potential errors and stuff? Because I mean, this amount of data that you must have across all those instances is pretty tremendous.

[00:42:11] SS: Unfortunately, I wasn't involved in that. But I can talk about some of the high-level problems is collecting too much data is a problem. We had to actually tone down in many areas to reduce how much data we were collecting. It just multiplies in some cases. But there is no science behind that. It's mostly trial and error. You say, “Oh, this data would be useful”, and you start collecting and then it just completely blows up the system, so we go back and say, “Okay, maybe we should not collect data for every table. Maybe you should not collect data for every query.” So, we tone it down that way, just based on how overloaded the system gets and find out why it gets overloaded. There's no science behind this. It's pure iteration.

[00:43:00] JM: Now, you've seen a lot of customers come through the door, and a lot of them are maybe not exactly on the scale of YouTube, but quite large. Have you seen any distinctive differences in query patterns or use cases that have challenged some of your preconceived notions around Vitess and forced you to make some changes around the database?

[00:43:25] SS: It is likely that I think the biggest challenge that I can think of is ORMs. There are many users that come in with ORMs that do weird settings in the database that Vitess did not like like. Or precooked up applications like WordPress, right? Somebody says, “Oh, I'm going to deploy WordPress on Vitess.” And then WordPress sets all these variables, session variables in MySQL, and Vitess says, “What are you doing? Like I cannot do connection pooling, you should do this.”

So, I would say, that's a very distinctive problem that we had to solve in Vitess, and we are still refining that part of it to make it even more efficient. And like WordPress, or a specific ORM likes to do settings, something about transaction found rows, it's a weird setting in MySQL, that some were old ORM settings, even though MySQL recommends you to don't set it. But then if the application sets it, then we have to honor that setting, right? That's definitely one of the biggest challenges that we have to solve in terms of weird behavior that the application performs.

[00:44:36] JM: So now that the company is getting mature, what are the big projects that you have planned for the future? Or is it more incremental and process of sales and integrations with cloud providers? Is there anything monumental you have planned? Is it just a like a battle of inches at this point, and kind of more of a sales and marketing challenge?

[00:44:59] SS: I feel like we are building something monumental. I feel like we are filling a gap, a very painful gap that has existed in the software world that has been ignored for a very, very long time. And it's been there for so long that people have forgotten. I've kind of learned to live with that pain. I feel like we are now finally solving that problem, which is the high-level way I would describe it, is the ease with which you refactor code. You cannot refactor your data with that same ease. That I believe, is the core of what we are solving. Refactor your data, like you refactor code.

So, if you look at software lifecycle, you build software, and then you design your data. And then after a few months, you say, “Oh, a new feature has come in, software evolves.” You just make code changes. But you cannot change the data as easily as you change code. And what we want to enable is the ability to refactor the data along with the code so that your data structure remains in sync with your chord structure.

This impedance mismatch, as I would call, it, makes your software grow faster and bigger. And eventually, the fact that your data has remained the way it is, starts to hold back the forward progress of your software. But if you're able to refactor your data, just as you would refactor your code, eventually, then your software can continue to grow much faster, and will not eventually get held back by the fact that your data is organized a certain way.

So, the way I see it is like the schema migration is our first step. If you look at schema, it is actually a schema change. It's actually a specific instance of a bigger problem, right? Adding a column to a table is one instance of a bigger problem. What if you want to split a table into two? What if you want to re-Shard your database? That's actually a transformation. What if you want, let's say, re-Shard a table a different way, right? So, all these are valid changes that you want to perform in order to keep your data up to date with your software. I believe, this is what PlanetScale is solving at the core, and it is life changing in my opinion.

[00:47:22] JM: Awesome. Well, Sugu, thank you so much for coming back on the show. It's always a pleasure to talk to you.

[00:47:24] SS: Thank you very much.

[END]