EPISODE 607

[INTRODUCTION]

[0:00:00.3] JM: Relational databases have been popular since the 1970s, but in the last 20 years the amount of data that applications need to collect and store has skyrocketed. The raw cost to store that data has decreased. There's a common phrase in software companies; it costs you less to save the data than to throw it away. Saving the data is cheap, but accessing that data in a useful way can be expensive. Developers still need rapid row-wise and column-wise access to the data. Accessing an individual row of a database can be useful if the user is logging in and you want to load all of that user's data or if you want to update a banking system with a new financial transaction.

Accessing an entire column of the database can be useful if you want to aggregate the summaries of all the entries in the system, like the sum of all financial transactions in a bank. These different kinds of transactions are nothing new, but with the growing scale of data, companies are changing their mentality from thinking in terms of individual databases to thinking about distributed data platforms.

In a data platform, the data across a company might be put into a variety of storage systems; distributed file systems, databases, in-memory caches, search indexes, but the API for the developer is kept simple, and the simplest most commonly understood language is SQL.

Marco Slot is an engineer with Citus Data, a company that makes PostgreS scalable. PostgreS is one of the most common relational databases, and in this episode Marco describes how PostgreS can be used to service almost all of the needs of a data platform.

This isn't easy to do as it requires sharding your growing relational database into clusters and orchestrating distributed queries between those shards. In this show, Marco and I discuss Citus's approach to the distributed systems problems of a sharded relational database. This episode is a nice complement to previous episodes we've done with Ozgun and Craig from Citus in which they gave a history of relational databases and explained how PostgreS compares to the wide variety of relational databases out there.

Full disclosure, Citus data is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:02:23.7] JM: Life in the cloud is great. Application integrations are more abundant, data is easier to access and software can be more effective for cross-functional teams, but due to the highly interconnected nature of cloud infrastructure and continuous delivery, you're bound to have anomalies and errors within applications.

VictorOps empowers teams to minimize downtime. Use VictorOps to manage on-call schedules, contextually alert teams when something goes wrong, and collaborate around an incident. VictorOps also integrates with hundreds of service partners to provide better visibility and communication across all aspects of your monitoring, alerting and incident response.

Head over to victorops.com/sedaily to learn how VictorOps help teams continuously deliver and maintain uptime. Again, that's victorops.com/sedaily, V-I-C-T-O-R-O-P-S.com/sedaily. See for yourself how VictorOps incident management makes incident resolution.

[INTERVIEW]

[0:03:38.4] JM: Marco Slot is a database engineer with Citus Data. Marco, welcome to Software Engineering Daily.

[0:03:44.5] MS: Thank you very much for having me.

[0:03:46.7] JM: Today we're talking about SQL databases and how those are advancing. SQL databases have been around for a long time. Why haven't the scalability problems of relational database has gotten solved?

[0:04:01.5] MS: I think it takes a long time to develop a database. SQL databases have been under development for decades. I mean, they've built up a lot of functionality and some of that functionality they were in terms of being able to do very complex joins and foreign keys and

other features. They were able to do because they were always working on a single machine, and that was for a long time good enough. But I think at some point, maybe 10 years ago, maybe a bit earlier, you get kind of the rise of the internet and social network and apps later on, and those kind of really created, started generating a lot of data and we have this need for very large-scale databases that stored humongous amounts of data and sometimes needed to keep it in memory to be accessed quickly.

So taking all the functionality of a SQL database and kind of starting dividing that over multiple servers turned out to be quite challenging, because if you actually pull apart the data and you then try to enforce a relation between rows that are on different machines or you try to join rows that are on different machines, you constantly have to go over the network, and the network is much slower than, let's say, memory or CPU.

So that was not something that could easily be done. So I think the first reaction to the need for scale was to create much more primitive databases, which could only maybe do keyvalue stores. You could store by particular key and lookup by a particular key, and then based on the key you could route it to a particular machine and you didn't really have any of these relational operators. There're no relations between keys or you cannot group particular sets of rows together, and that's scaled well. But, yeah, I think since then, last few years especially, I think the need for SQL databases is still there because we kind of want to build interesting applications that do more than just keyvalue storage. So I think more recently, maybe the past 5 to 10 years, there've been a lot more development around kind of scaling, actually scaling SQL operations.

[0:06:22.8] JM: We have well-defined principles around scaling application servers. So we could keep the logic stateless. We can be conscious of monolithic and micro services approaches and gauge trade-offs between those two. We can also make sure we have some sense of auto scaling in the world of application servers. When we're talking about databases, what are the well-defined principles that we have established around scaling a relational database?

[0:06:55.9] MS: I think there we're, to some extent, still in a bit of an experimental phase where different companies, different groups are trying out different things. But I think there are patterns

emerging of, let's say, good ideas, things that work quite well in terms of being able to provide to rich SQL functionality and also doing that at scale. I think one important principle kind of on the data modeling side is that of co-location.

So when you distribute a table in a SQL database, you typically pick a column, which is the distribution column or partition column and then the data is going to get divided into shards and each shard contains a particular range of distribution column values or sometimes a range of hashes of distribution column values.

What kind of turns out to be a good idea is that you actually have the same column to all of your tables and you distribute them all by that column and then discharge that contain the same range are co-located on the same machine. Because then that enables you to do drawings on the distribution column, and there's many applications that can take advantage of that. For example, a multitenant application like, SaaS applications that have a lot of independent users. They might decide to distribute all their tables by tenant ID, and then when their application makes a query for a particular tenant, well, all data is always in the same place. So that query can go to one particular machine, and it still do joins between different tables, but then that one particular machine can answer that join, or you could do a query across all your tenants as long as it – And as long as it joins on the tenant ID, actually, the drawing can entirely be local without kind of actually having to exchange data between all the servers. So that is then actually very efficient.

So the sort of principle of co-location enables a lot of this type of SQL functionality, not just joins, also foreign keys, also the ability to in search into one table and select from another table in kind of a distributor way across all the shards is very powerful.

Another concept, which I think occurs in several distributed SQL databases is that of reference tables, and sometimes the naming differs a bit, sometimes it's like dimension tables or replicated table, but basically the idea is that these are smaller tables. So they're not in need of scaling across many servers, but they can actually be replicated to all the servers. They’re small enough to do that.

When we do that, then we can actually join these reference tables with the distributor table by any column. An example could be you have – I don't know, a big purchases table, orders table and big customers tables and they're distributed by customer ID, co-located, and so you could join by customer ID, but then you also have a product category table, and there's no way you could distribute that by customer ID.

But then It's probably small, probably doesn't change much. So you could then replicate that. Then that, again, enables a whole new kind of set of SQL functions in terms of being able to join those tables on any column without going over the network, which makes it quite fast. So that's kind of on the data modeling side.

Outside of the database, in terms of more deployment, I think a cloud is quite important in terms of being able to deploy a distributed database and gets the most out of it, especially network-attached storage, Elastic Block Store on Amazon and other similar services in Google and Azure. Those are kind of almost crucial in terms of providing kind of persistence in a way where we can lose a note and then bring in another one back and reattach it to the persistent volumes, and that kind of now – There's an abstract version of that in Kubernetes where you have stateful sets and persistent volumes where every node has as a number or every pod, and if you lose the pod it gets replaced with a pod with the same number and attaches to the same storage.

So that's a very useful bit of infrastructure, which is available in the cloud, but usually not on-prem. The other thing is S3 or storage systems that can kind of do this archive. So not many systems have the scale to actually do a backup of a large distributed database with terabytes of data, but S3 and the Google storage and equivalent systems, they actually do have that scale. So in order to do backups, kind of running these databases in the cloud is quite important. It's very hard to do it on-prem.

[0:11:47.1] JM: Many of the things that you just outlined were about the storage strategies. How do you shard a database? Where are you storing your data to in order to maintain the various ACID qualities of your database, or maybe at least the durability element of it?

A lot of the interesting work that you have done in the world of PostgreS, or SQL more generally, is around the access patterns, or at least this is what I've heard you talk about. The access

patterns and the querying, how you query a distributed database. Once you have it sharded and it's partitioned across different logical nodes, you want to be able to query it. Okay, we can figure out how to do a distributed query, but it's a harder problem to do a distributed query efficiently or fast, and one way that you approach the world of distributed queries is through the lens of relational algebra. And relational algebra for some people listening might sound like a dusty old field. It was popularized in the 1970s by somebody working at IBM. Why is an old field like relational algebra – It almost makes me feel old to say something like relational algebra. Maybe that's my association with my computer science professors that might have taught me relational algebra, but why is this important to study when designing modern distributed databases?

[0:13:24.8] MS: What's funny is that often we read papers and they're actually usually very hard to read because they had to be scanned and the scan is not perfect. I mean, originally, they were done on typewriters probably, so they're actually poor quality. But I just think algebra never – Like, math never changes, right? It's always the truth. So algebra was as good a representation of SQL 30 years ago as it is today, or 40 years ago.

I think it's probably more important now than – The relational algebra is more important now than ever because it provides you with a way to reason about the SQL query especially and applies certain transformations on your distributed execution plan for that SQL query where, because they're just kind of algebraic transformations, you kind of preserve the correctness, but you're able to find an algebra tree that is actually much more efficient when done on a cluster.

So, for example, you might have to distributed tables and your drawing them on a column where they're not co-located. So a distributed database will typically have to do is reshuffle the data over the network. Now, if you start looking at the algebra tree for your query, there are operators in there such as filter to table, projection, join, and if you build that algebra tree for your SQL query, some of the operations are commutative, meaning you can kind of switch them around, like 1 +2 is the same as 2+1. So it doesn't matter which order you do them in.

But if these are like a filter and a join, the order matters a lot. It matters a lot whether you end up reshuffling your entire table over the network, then doing a join, and then filtering, or you first filter out maybe like 10 rows, you send them over the network and then you join them. So I

think, yeah, relational algebra, it's a convenient way for us to reason about, “Okay. How can we come to a more efficient query plan, where we first try to reduce the data set to as little as possible because the network is slow, because we don't want to send much data over the network?”

It also comes in in operations like aggregates. If you some sum, you don't want to pull the whole table over the network and then calculate the sum. You want to kind of take the sum of each individual shard and then take the sum of the sums. Kind of reasoning about or finding the best way to do this, I think for that relational algebra is actually still essential.

[0:16:04.2] JM: If I'm performing a transaction on a distributed relational database, how does that differ from transactions on a single machine?

[0:16:14.8] MS: There's quite a few things that go into doing a distributed transaction. I mean, one of the things is – So you have the independent servers and you’re at some points maybe starting local transactions on this server, which are part of a bigger distributed transaction that should commit as one thing and become visible as one thing. But one of the problems you run into quite quickly if you start making modifications across multiple nodes in the same transaction is that you have locks in a database. Basically, lock protects, for example, prevents two transactions from updating the same row concurrently. One will get the lock and will finish and the other one will block until the lock is released.

Now, what happens, because SQL databases provide a very flexible way of doing begin and update the lead, insert in one transaction, is that you can get into death locks, where one transaction obtains a lock, like lock A, and then one is lock B, another transaction has locked B, but it wants lock A. Now they’re actually stuck waiting for each other.

Now, in a single database, there's just kind of a background process that then starts and sees, “Hey, actually, I have two transactions which are kind of forming a cycle and I should kill one of them,” because there is actually no way to finish both of them at that point. But when you move into a distributed database, that logic no longer works.

So to actually prevent your database from getting stuck all the time without ever recovering, one of the first things you need is a way to deal with distributed death locks, and probably what you need is to gather all the lock crafts in one place kind of periodically and then pick out, “Hey, actually, I have this transaction on one node and another transaction on another node and they belong to the same distributed transaction.” This distributed transaction, one is waiting for two on the first note, and two is waiting for one on the second note. So, actually, distributed transactions one and two are stuck waiting for each other. So we should kill one of them. So that's one aspect.

Another aspect is, “Okay, at some points, from the client perspective, the client should just be sending these statements, begin update, delete, commit.” At some point at commit, we want to make sure that we never get into a situation where the transaction succeeds on one node, but fails on another. That would be bad, then our database gets into this very terrible state.

To deal with that, one of the techniques that I think is quite commonly applied is two-phase commit. So this is a way of telling the database or telling the storage nodes, “Hey, all the stuff I just did kind of remember it, but don't commit it just yet. Don't show it to anyone else yet.”

So a two-phase commit first kind of does this. It prepares the transaction on all the nodes, and at this point if anything goes wrong, it can still go back and abort everything. But at some point, if it succeeds with preparing all the transactions, it decides, “Okay. Actually, now we’re good. I can commit.” Then the node that’s coordinating all of these, which is probably the one that the client connects to, durably store a record of saying, “Okay. This transaction, commit it.” Because after that, anything could go wrong, but we should still be able to recover from that situation.

So at a later time, the transaction should be kind of committed. But if a node fails, we should still be able to – Once that node comes back, we should still be able to commit it. So it's important that kind of transactions are done in these two phases, where first, the state of the transaction is durably stored, and then once the coordinator takes the commit decision, then we actually proceed. If something breaks, then at least we have remembered to commit decision and can recover from it.

So these are kind of just some of the things that kind of happen that's don't really come up with in a kind of single node database. But once you go to multiple nodes where you can do transactions that span across multiple nodes, you need to start solving these problems.

[SPONSOR MESSAGE]

[0:20:28.4] JM: Today's episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for cloud scale infrastructure and applications. Datadog provides dashboarding, alerting, application performance monitoring and log management in one tightly integrated platform so you can get end-to-end visibility quickly, and it integrates seamlessly with AWS so you can start monitoring EC2, RDS, ECS and all of your other AWS services in minutes.

Visualize key metrics, set alerts to identify anomalies and collaborate with your team to troubleshoot and fix issues fast. Try it yourself by starting a free 14-day trial today. Listeners of this podcast will also receive a free Datadog t-shirt. Go to softwareengineeringdaily.com/datadog to get that t-shirt. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[0:21:33.6] JM: Let's maybe give an example that we can toy around with. So what if I had a relational database that had every single podcast episode from every single podcast in history? I think there are 500,000 podcast shows that have been published and I'm sure there are number of episodes in each of those podcasts. So if I had a big distributed relational database of all podcast episodes, if that database instead was just all of the podcasts about databases. I know there are some podcasts specific to databases. Those podcasts would probably fit on a single database node. Even if I had a complex transformation that I wanted to perform on that database, it wouldn't be too difficult. We could easily reason about how to run some kind of transformation on that single node database. But if we tried to scale that transaction to the relational database of all 500,000 podcasts and every single episode indexed and the categories and perhaps the show notes and the links to the MP3 of that episode, what kinds of challenges would we encounter when we're trying to scale a complex transformation to a bigger database?

[0:22:58.5] MS: One of the things that happens is that – I mean, you could have – So SQL databases have such a kind of rich language where you can do – You can update a row, but you can also update a set of rows based on the result of a query. One thing that happens that you need to think about is, "Well, will my query actually return the same results everywhere? Because otherwise I'm going to update the database in different ways, in different places." This is comes up especially if you do kind of replication.

One way to do replication in a distributed databases you take that update statement with the sub-query and you send it to multiple places at the same time. So you apply the same transformation. Actually, because you also need to make sure that the sub-query actually returns the same thing everywhere, you need to take – Often, you take very aggressive locks to make this work correctly or to keep the replicas consistent. So one thing that means is that for a database to actually support such commands and still be able to do other things at the same time, you need a kind of different replication model where you do more kind of streaming replication.

I mean, I'm very much on the PostgreS side. So the PostgreS replication mechanism would be, "Okay. This update happens and it starts writing you rows and it writes those to a log and the log actually gets replicated as like one single stream into another nod, and those rows are applied on the other side." So that's a way to actually keep that complex operation kind of running efficiently without actually having to lock the entire database.

[0:24:44.2] JM: Before a distributed database engine is going to execute a query, it needs to evaluate the costs of various approaches. This is the role of the query planners. So a query planner is going to plan how to execute a complex query. What are the requirements for a query planner?

[0:25:07.7] MS: There is actually within a query planner, like there's different architectures that people have used to build query planners. So one thing we've worked on or I've worked on a lot lately is kind of having more of a layered query planner where you take a kind of brief look at the query and see, "Can I do something really simple with it?"

In a distributed database, the really simple thing to do would be to check, “Can this query be answered by one node? If yes, send it to that node and let that node worry about how to answer the query,” and then on that node you would use typical techniques from relational database, or maybe that node is a relational database. But then if that query cannot be sent to one node, well, the next step you might look at, “Can I actually do this query in kind of a single route where I’d make all of my nodes do one thing and they send me their results and I merge them together.”

Actually, at that point, it’s still relatively simple. You’re just essentially looking at the filters in the query and the key there is joining by the distribution column. Are all the tables in the query joined by the distribution column? If you can't do that, the next step might be, “Well, are there certain kind of offending parts in the sub-query? Maybe there's a sub-query with the limits, and then that usually means I need to evaluate that limit separately because I might need to evaluate it globally.” So I’d take out that sub-query and I execute it independently and push the results back into the cluster and then do the rest of my query.

So a lot of SQL queries can kind of go through that process and then be answered. But there is also a class of distributed SQL queries where that is not the case and particularly the ones that join multiple tables that are distributed, but they don't join them on the distribution column. This happens especially in kind of a data warehousing reporting use cases where you just have lots of different tables from different sources together and you want to answer some complex business question, then you're probably okay with the query taking a while longer, but not months.

So in that case, the query planner has to consider, “Okay. If I have a join between a few tables, I mean, what's the lowest-cost way of doing that join?” In the distributed level, basically, you kind of have a few different – Well, you mainly have repartitioning operations and you have to consider, “Okay. If I join three tables, A, B and C, if I want to join A with B, I need to repartition both of them because they're not joined on the distribution column. Then I want to join with C, but that’s, again, joined by a different column so I need to repartition both sides. But maybe actually A and C are joined on the distribution column, as just B, which is weird.

So what a database typically does is it generates all possible orders, all possible ways of executing a join and executing a query, and then kind of figures out the cost of that, where in the distributed case, the repartition operations is really expensive, but the co-located join operation is really cheap. For that, it can use the sizes of the individual shards. It can look at maybe some statistics that it periodically fetches from the data nodes. Once it has the optimal join plan, it can send the individual queries to the storage node, and the storage nodes are in a better position to make decisions for themselves, like they can have local statistics and decide, “Okay. What locally is the best way to do the join or to look up the rows in this table?”

Though there're also some databases that have kind of completely centralized query planner and it would be kind of interesting to kind of do a more rigorous study of like how these two approaches compare with doing all the planning in advance or deferring some of the planning to later on. I am actually not sure, but I'd be curious.

[0:29:20.7] JM: I certainly feel like I could do an entire show on query planning because there are so many trade-offs to how you write a query planner or how the query planner itself is going to approach a query itself. I think one proof point of this is that people write their own custom query planners. What are some use cases? What are some examples of places where a developer would want to make their own custom query plan? They've got a SQL database. It's on PostgreS. They want to have their own query planner. Why would they want that?

[0:29:57.3] MS: Well, I think writing your own query planner really from scratch is probably a bit like doing your own encryption, not something you should do. But I think the fact that like PostgreS allows you to override the query planner – Well, I think it's useful in two ways. One is it gives the opportunity for kind of extensions to kind of define really new functionality, like adding a GPU computing to PostgreS or adding distribution to PostgreS. But apart from that, if you're more kind of a database user, then the fact that you have these hooks actually allow you to kind of just nudge the query planner in the right direction as well.

So if you have a particularly complex query which is important to your business and the way the database ends up doing it is not quite the most efficient way and you know there's a more efficient way, it’s very useful if you can just kind of add some – Give the query planner some better execution plan for that particular query or for a particular type of table. This is not going to

be for everyone, but there are certainly businesses out there that whose kind of livelihoods depend on their database being fast.

So in those cases, you might at least augment the plan that PostgreS generates. Then for kind of the wider community, there's extension builders building whole new query planners that just kind of enhance the functionality of the database or expand it in some way, which the database itself does not provide.

[0:31:37.6] JM: This gets us into one of the appealing aspects of the PostgreS ecosystem. So as example, you could have an extension to the query planning facility which would allow you to change how your query planner works because of your domain specific needs. More broadly, the extension ability is true for any step of the SQL query pipeline. So there's a bunch of different steps in this query pipeline. There's the planning step, the execution step, the access methods. Many other steps to the execution of a SQL query, well, I guess the planning and execution and successful outcome of a SQL query, and people who are listening may not be familiar with the sequence of steps that goes on under the hood when you run a relational database query. Why are there so many different phases in that process of a single SQL query?

[0:32:40.9] MS: I think the main reason there are so many phases is because SQL is – Or implementing SQL is pretty complex. Even the tiniest detail of a SQL query is probably more complicated than you'd expect, like parsing a table name. It seems straightforward, "Can you read the characters?" and you look up the table. But actually, I mean, that table could be dropped concurrently, or renamed concurrently, or the permissions could change, or maybe you didn't specify the namespace of the table and so now a new table showed up in a different schema, which actually now overrides the previous name that that table, or a table that resolved to.

So even the tiniest details in a database are actually fairly complicated. I think what happens is partly because PostgreS has always been developed as a kind of community project. There's not one company or one person behind it. They make sure that the code was very structured and modular and easy to read, and that is also just one of the reasons why the whole process is very well broken down into steps of a parser, a planner, an optimizer and then execution is broken down into multiple steps.

At some point, like the code was so modular that someone thought, “Well, then we could just as well allow people to override these steps.” So they started adding function pointers where you could just set – Replace one of those steps with your own function and you could still call back to the original functions. So you could just augment it with some new feature.

This kind of actually then created this explosion of people trying different things on top of PostgreS, and it actually works at many different layers. You can add a function to PostgreS that does something custom. You can add a new type. You can add a new operator, but you could also change part of the query plan or you could read from another or database inside PostgreS and then use the results from that inside your query.

So I think just the way PostgreS developed led to modularity, which then led to extensibility, which then led to how crazy array of new projects where PostgreS is almost kind of becoming the platform, the operating system for a whole set of extensions.

[0:35:04.5] JM: So why is this so useful for all these different phases of the PostgreS execution pipeline to be modular? What does that enable for the PostgreS ecosystem?

[0:35:16.8] MS: Well, it enables kind of anyone to add new functionality to PostgreS for themselves or as a new product or to solve their problem. Sometimes it's simple things. Because you can plug into different stages of execution, you can maybe add some – I don’t know, auditing at that part of the execution. Like, “Hey, a query was executed. We need to remember this,” or some monitoring tools, or you could actively kind of explore new kind of trends in databases like GPUs or distributed databases.

Yeah, when you do that, then that kind of expands the functionality and the capabilities of the database quite a lot. I think that's quite unique to PostgreS, that if you look at the core, it's probably – Well, I think it’s probably the most mature open source database, but it's probably similar to other relational databases in many respect. But if you look at the whole ecosystem and the functionality that you can add, such as like post-GIS, which is for a kind of dealing with spatial data, and you can do columnar storage to an extension. You can do distribution to an extension. So an ecosystem, PostgreS is like by far maybe the most powerful database.

[0:36:38.7] JM: What's another example of a phase in that pipeline? Like I said, planning step, execution step, access methods, all of these different things that can be broken up. You gave the example of a high-level example where you might want geospatial data to have some specific facets of its entire database, I guess. What are some other examples where you would want extension functionality?

[0:37:06.5] MS: I think one of the most popular APIs that PostgreS provides for extensions is the forum data wrapper. So this is a way to kind of plug a different data storage system, be it a different database system, or a file, or something completely different into the database as if it were a table. So if you're writing the code for this, you have to implement a few functions, like an insert function and a select a row function. Nowadays, they're expanding it to have some like aggregates to be able to send aggregates into the underlying storage system.

So when you do this, you can actually use that other system as a table in PostgreS and you can join it with the tables that are already in PostgreS or with other forum tables that talk to some other underlying system. It becomes especially useful because it also provides the facility to take filters from the query and send those to the other storage system. So if you have another database hooked up there, you don't have to retrieve the whole database. You just retrieved the rows that you need to answer the overarching query.

I think this is used a lot and it's used both for data migration. It's used for kind of PostgreS acting as a central SQL engine that talks to a lot of different systems. It's used to kind of create new storage formats where the data is, for example, compressed on disk. So this is extremely powerful.

[SPONSOR MESSAGE]

[0:38:49.5] JM: At Software Engineering Daily, we have user data coming in from so many sources; mobile apps, podcast players, our website, and it's all to provide you, our listener, with the best possible experience. To do that we need to answer key questions, like what content our listeners enjoy? What causes listeners to log out, or unsubscribe, or to share a podcast episode with their friends if they liked it? To answer these questions, we want to be able to use a variety

of analytics tools, such as mixed panel, Google Analytics and Optimizely. If you have ever built a software product that has gone for any length of time, eventually you have to start answering questions around analytics and you start to realize there are a lot of analytics tools.

Segment allows us to gather customer data from anywhere and send that data to any analytics tool. It's the ultimate in analytics middleware. Segment is the customer data infrastructure that has saved us from writing duplicate code across all of the different platforms that we want to analyze.

Software Engineering Daily listeners can try Segment free for 90 days by entering SEDAILY into the how did you hear about us box at signup. If you don't have much customer data to analyze, Segment also has a free developer edition, but if you're looking to fully track and utilize all the customer data across your properties to make important customer-first decisions, definitely take advantage of this 90-day free trial exclusively for Software Engineering Daily listeners. If you're using cloud apps such as MailChimp, Marketo, Intercom, AppNexus, Zendesk, you can integrate with all of these different tools and centralize your customer data in one place with Segment.

To get that free 90-day trial, sign up for Segment at segment.com and enter SEDAILY in the how did you hear about us box during signup. Thanks again to Segment for sponsoring Software Engineering Daily and for producing a product that we needed.

[INTERVIEW CONTINUED]

[0:41:20.2] JM: So if I can mix-and-match these different query extensions based on a query, PostgreS could be quite adaptive to any query I could throw at it. So you could imagine, if I just have a complex query, maybe PostgreS could switch out the query plan. Maybe there's a different query planner that would be more effective for a given query. Is that important? Should PostgreS itself be able to just understand what my query is and what to optimize and swap out and switch between different extensions based on what the query actually is?

[0:42:01.0] MS: I think that would be like a nice vision for the PostgreS future. I mean, it can kind of do it to some extent where it provides – For example, you can create an extension that

just provides alternative execution plans using some different technology, like be it distributing the query or GPU, and then PostgreS can essentially just use this cost-based optimizer and say, “Well, actually, in this case, GPU is faster. In this case it's lower,” and then pick the plan based on that.

It depends a bit – In other cases, if the data is, for example, in a different location, then you don't actually have to have many options. You have to kind distribute the query or send it to that other location. I think there is a future where a lot of these things come together. For example, PostgreS would be able to decide, “Hey, I'm actually – I'm getting big. I'm getting pretty busy. I should kind of spawn a new node and scale out to that new node in a kind of much more dynamic way.” Currently, a lot of these decisions are still like manual, you pick the distribution column, you do decide when to scale out. I think there is a kind of longer-term future where that becomes a much more automated process where you don't have to think about it.

Also, for example, index creation in PostgreS is still a very manual process. You have to think about it and create the indexes, but I think that's something that, let’s say, PostgreS five years from now might be able to do itself.

[0:43:35.1] JM: Right. Related to that, there was that results out of Google about machine-learned database indexing. That was the idea that machine-learned database indexes could be used to improve database efficiency. Do you see any other applications of machine learning to database architecture?

[0:43:56.9] MS: Yeah. I mean, I really love that concept of – I had never thought of that being an option, but now that it's out there, yeah, you do wonder what else could you do. I think one thing I see users struggle with in, well, PostgreS world – So PostgreS, because it has such advance index types, one of the index types that kind of some of the bigger users really like is partial indexes. So this is the idea that you have an index, but it only – For looking up rows officially, but it only applies to a subset of the database in the filter. So only rows that match the filter are index and only queries that have to filter will use the index.

So this allows you to do this kind of micro optimization, where if you have one particular customer that makes one particular type of query, you can make an index for it. But the problem

with that is – I mean, it's great to have the possibility, but it's very hard to decide when you actually create these in Nexus and which in Nexus to create?

So if there was like kind of maybe a machine learning mechanism that kind of analyzes your queries over time and the access patterns, maybe does some profiling, and kind of from that finds, "Hey, actually, everything gets better if you just add this index." I mean, the thing that's always hard to avoid are having to make trade-offs. So there's a question of, "Okay, well, if you create an index, my writes get slower. So to what extent am I willing to let the database slow down my writes?" We probably need to think about, "Okay. What's a good way to define my performance requirements for my database that then my database itself could figure out how to meet by creating the right indexes or kind of may be optimizing queries in a certain way?" Yeah, that's kind of an exciting new development.

[0:45:50.9] JM: The vision that Citus Data has, which is where you work, if I understand correctly, is around the building PostgreS into a data platform. When I think of data platform today, I think of these different systems that are wired together. So you have Elasticsearch, you have HDFS, you've got Hive queries that run on HDFS. Maybe you've got five different databases that are running against that HDFS file system, or they use the HDFS file system as the source of truth and then they have materialized views that are different pictures of the data that's across HDFS. It's a lot of sprawl. It's a lot of complication.

So if I understand correctly, Citus believes that you could just do all of this with PostgreS. Is that an accurate description of the data platform idea that you have?

[0:46:53.4] MS: Yeah. I think that's – I mean, you probably can't do everything on PostgreS, and I mean it's definitely not a platform for storing raw audio files or something like that. But if you look at kind of a lot of sort of modern data architectures or so many different components, and to some extent that's because they're making different trade-offs. But what's interesting about this extensibility of PostgreS is that you can actually just make these trade-offs within one particular database platform. So if you need a different type of storage to optimize for a particular use case, you can actually just use a different storage module in PostgreS and still have kind of the same general interface where you can use SQL and you can use the other extensions.

If you look at the functionality that like a lot of these systems provides, it's usually a subset of what a SQL database can provide, be it full text search for just kind of queuing and storing an event queue or like even key value storage. I mean, all of those can be expressed as kind of SQL commands. So if you have like a scalable SQL database, which has the ability to kind of adapt to different workloads, because you can kind of plug different – Or you can make different trade-offs by plugging in different kind of access method and storage methods then, yeah, you're almost kind of thinking why would I actually need to become an expert at 10 different data storage systems if like one can actually do everything? So I think, yeah, that's definitely kind of like longer-term vision.

[0:48:38.6] JM: There are companies with PostgreS systems today. There are also companies that might be getting started and they want to stand up a relational database for a greenfield application, and you want to be able to cater to those different sets of customers. So how does the product development process for Citus differ from the customers who are standing up a greenfield application versus the ones who are migrating their old application to the Citus distributed platform?

[0:49:13.7] MS: You often see kind of some differences between different use cases or classes of use cases. So if you think of, for example, SaaS apps, which there are a lot, and they usually start out with a SQL database. They have an application that makes pretty complex SQL queries. Now, for them, probably in the first couple years of the company they don't really have the problem of having to scale out their database across multiple servers.

I mean, they could make that decision early on to kind of shard, to go for a sharded solution and just start small and then grow. But very often they are actually already coming with a complex app. For those users with Citus, we try to kind of cater to the requirement that they need to be able to run – Like the SQL queries they use today, they want to continue using those because they can't just – They don't want to change their whole app just to scale out their database.

So the multitenant model actually lends itself quite well for that, because if you shard your data by a tenant, and if you're coming from an existing app, but it usually means maybe you have to add a tenant ID column to many tables because maybe previously you implicitly had that

column through relationships via other tables, but you didn't actually explicitly stored in the table. So you need to do some, in that case, schema changes to kind of cater to distribution. Maybe your queries needs extra filters to make sure that Citus can then like route it to the right node that stores the data of that particular tenant. There's usually some – This takes a few weeks usually to kind of do this migration, but then once you've done that, it's actually a pretty straightforward to migrate, because everything is still PostgreS. You can just keep using your existing tooling.

So we find this as actually an easy process, especially if you're at the point where your single database servers is falling over, then you kind of really need to do this. That's one kind of use case we try to cater to and that's also by providing migration tools where you can do migrations without downtime. Then there's another use case which – Well, few other use cases, but one other use case where people usually start from scratch on Citus, is a real-time analytics use case, which is where you have this large stream of events of like time series data and you want to provide some analytics applications on it such as a dashboard, which kind of continuously gets updated with the new data.

This is something you can't naturally do in PostgreS because it usually requires some highly parallel operations, which PostgreS doesn't – Well, that has some parallelism, but not in all the operations that it needs to have. So in that case, those users would build a pipeline where they copy into Citus, maybe build a rollup table and then their dashboard selects from the rollup table.

Because of the distribution and because Citus paralyzes the operations, like the insert select operations and the copy operations across all the cores in the cluster, you can achieve really high volumes and you can actually – Because you're in PostgreS, do really kind of interesting things in terms of queries and we also cater to that with some additional extensions, like an extension for keeping track of maybe the top 10 IPs that hit a particular page, or the top 10 queries on a particular search engine. We provide like additional PostgreS extensions that help you build those kind of – Or track that.

So there are some other use cases. There's kind of more key value storage kind of traditional NoSQL use cases that use Citus. Yeah, we see quite different requirements into like the entire

picture of those different use cases where one usually first starts on PostgreS and then migrates and one usually starts from scratch and then grows.

[0:53:11.9] JM: What's the hardest part of building a manage database service?

[0:53:17.6] MS: Well, there's many hard parts, but I think we've made it easier by the fact that like in the Citus cloud team has a lot of experience building, running a managed service with a few of them coming over from Heroku. They kind of had some principles with them that really helps the stability of the service. Such as the fact that all the database nodes are continuously archived into S3, and then when we kind of add a new replica into the system, we actually recover it from S3, for example. All those techniques that were kind of already learned over a couple of years, we've applied those to Citus cloud and that has been quite successful.

I think where we kind of, let's say – Or the hardest part is just things that relate to distribution and a distributed database, where sometimes debugging an issue in a distributed database can be much harder than debugging an issue on a single database, because you've got connections going crisscross. I think like that's one of the challenges, just the understanding what's going on is harder when you build a distributed database service. Yeah, I think we're getting better and better at that.

[0:54:40.2] JM: Speaking of an application for the future, you've spent some time working on self-driving car systems. In the interviews I've done with people who are working on self-driving cars, I sense that there are a ton of problems that are quite hard to solve. There's model deployment. There're simulations, which have tons and tons of data. There's all these data labeling issues, because you're driving around, you're gathering photos of your surroundings and trying to model the universe based off of the photos that you're gathering. What are the data platform requirements of self-driving cars?

[0:55:24.7] MS: They're pretty heavy. Sometimes the research has tried to sell their work and saying,, "Well, self-driving cars are much more efficient so they save on CO_2," and you look at the trunk and there's like a big server farm.

[0:55:37.1] JM: Right.

[0:55:37.5] MS: Yeah. So, I mean, the most typical kind of set up in terms of sensors is that the self- driving car has a LIDAR on top where it generates a point cloud, which kind of – At 60 frames per second, and typically there is actually multiple servers kind of that each get the same point cloud, but then track different objects or in that using different techniques. Nowadays, there's bit more attention to machine learning techniques, but there's other techniques to define those objects, and it's a difficult problem.

Actually, one of the things that my PhD research was about is can we reduce those requirements a bit at the point where we have a couple of self-driving cars and some of the more binary decisions we can just share. I mean, you have to kind of consider – Well, we might be able to talk to each other over the wireless network, but sometimes it might be broken. Sometimes someone might be lying. But once you're in the state where you have a couple of self-driving cars, actually the data requirements go down a bit because now you kind of know what the car in front of you is going to do or what the car on the side of you is going to do. So that was like one of the goals of my research to kind of through cooperation kind of make it more efficient.

[0:56:51.1] JM: If I am a self-driving car company, I want to minimize my concerns about infrastructure. Let's say you're trying to convince a self-driving car company to use Citus Data. You're saying, "This is the data platform you should be using." What are the arguments you're going to make to them?

[0:57:10.0] MS: I'm not sure if SQL is up to the task of treaty point cloud analysis, but I mean I think the most important thing you eventually need is just to prove like rubber meets the road, you actually drive millions and millions of kilometers without kind of getting into any accidents. I mean, that's the best way to convince anyone.

Also not just millions and millions of kilometers, but actually doing with things like fog. I think those are – The basics of driving on a highway and driving through a simple city with not too many pedestrians on the road, those are kind of solved, but then the problem of, "Okay. What if it starts raining and LIDAR stars deflecting? What if it's foggy and we can't actually see the next object at all?" Dealing with those is, I think, a very much an unsolved problem and it's more

down to coming up with smarter algorithms than actually kind of increasing the processing power or something.

So, yeah, I think once those problems can reasonably be tackled, and it's kind of a mixture of coming up with a car that has a particular set of sensors and a particular software to deal with those situations and perhaps the ability to communicate without a car such that it can kind of get more information about its surroundings. I think you have to look at it at like as a whole thing, not just as a piece of software or as a computer, but as a car and what capabilities that car has and how it uses those capabilities.

[0:58:44.9] JM: Okay. Well, Marco Slot, thank you for coming on Software Engineering Daily. It's been really great talking to you.

[0:58:49.9] MS: Thank you very much for having me. It was like very interesting coming here talking about distributed databases and self-driving cars, my two favorite things.

[0:58:58.0] JM: All right. Thank you.

[END OF INTERVIEW]

[0:59:03.1] JM: Nobody becomes a developer to solve bugs. We like to develop software because we like to be creative. We like to build new things, but debugging is an unavoidable part of most developers' lives. So you might as well do it as best as you can. You might as well debug as efficiently as you can. Now you can drastically cut the time that it takes you to debug.

Rookout rapid production debugging allows developers to track down issues in production without any additional coding. Any redeployment, you don't have to restart your app. Classic debuggers can be difficult to set up, and with the debugger, you often aren't testing the code in a production environment. You're testing it on your own machine or in a staging server.

Rookout lets you debug issues as they are occurring in production. Rookout is modern debugging. You can insert Rookout non-breaking breakpoints to immediately collect any piece of data from your live code and pipeline it anywhere. Even if you never thought about it before

or you didn't create instrumentation to collect it, you can insert these nonbreaking breakpoints on-the-fly.

Go to rook out.com/sedaily to start a free trial and see how Rookout works. See how much debugging time you can save with this futuristic debugging tool. Rookout integrates with modern tools like Slack, Datadog, Sentry and New Relic.

Try the debugger of the future, try Rookout at @rookout.com/sedaily. That's R-O-O-K-O-U-T.com/sedaily. Thanks to Rookout for being a new sponsor of Software Engineering Daily.

[END]