EPISODE 737

[INTRODUCTION]

[00:00:00] JM: When he was at Facebook, Venkat Venkataramani saw how large volumes of data were changing software infrastructure. Applications such as logging servers and advertising technology were creating fast-moving semi-structured data. The user-base of Facebook was growing. The traffic was growing and the volume of data was growing, and the popular methods for managing this data were insufficient for the applications that developers wanted to build on top.

In previous episodes about data platforms, we have covered similar difficulties as experienced by Uber and DoorDash. Incoming data is often in JSON, which is hard to query for large data science jobs. So the JSON data is transformed to a file format like Parquet, and this requires an ETL job. Once it is in a Parquet file on disk in a data lake, the access time is slow.

To query the data efficiently, the data must be loaded into a data warehouse, which can load the data into memory, often in a columnar format that is easy to aggregate. Imagine being a developer at Facebook, or Uber, or DoorDash and trying to build a simple dashboard, or a machine learning application on top of this data platform. Where do you find the right data? How do you know it's up-to-date, and what if you don't know the shape of your queries ahead of time and you haven't defined indexes over your data?

The access speed will be too slow to do exploratory analysis. There are many steps in this process and each of these steps creates friction for applications developers that want to build on top of big data. It also creates opportunities to lose data or have time discrepancies between those pieces of data. Since even Facebook was having trouble managing this problem of the data platform, Venkat figured that there was an opportunity to build a company around solving the data platform for other software companies.

Venkat is the CEO of Rockset, a data system that is built to make it easy for developers to build data-driven apps. In Rockset, a piece of data can be ingested from data streams, data lakes and databases. Rockset creates multiple indexes and schemas across the data. Because there are multiple models for querying, Rockset can analyze an incoming query and create an intelligent query plan for serving that query.

Venkat joins the show to discuss his time working on data at Facebook, the untapped opportunities of using that data, the architecture of his company, Rockset, and also the changing economics of cloud resources. How it makes it possible to build an application like Rockset where it may not have been possible before cloud services made storage and compute so much cheaper.

We are conducting a listener survey. We would love to know what we're doing wrong and what we're doing right. Go to softwareengineeringdaily.com/survey and check it out. It would be super helpful and it would also enter you into a chance to win a piece of schwag if you enter your email address, otherwise it's anonymous, and you can win t-shirts, a hoodie, mugs, things that are available at the Software Engineering Daily Store. You can also sign up for our newsletter at softwareengineeringdaily.com/newsletter.

With that, let's get on with this episode.

[SPONSOR MESSAGE]

[00:03:58] JM: Your audience is most likely global. Your customers are everywhere. They're in different countries speaking different languages. For your product or service to reach these new markets, you'll need a reliable solution to localize your digital content quickly. Transifex is a SaaS based localization and translation platform that easily integrates with your Agile development process.

Your software, your websites, your games, apps, video subtitles and more can all be translated with Transifex. You can use Transifex with in-house translation teams, language service providers. You can even crowd source your translations. If you're a developer who is ready to reach a global audience, check out Transifex. You can visit transifex.com/sedaily and sign up for a free 15-day trial.

With Transifex, source content and translations are automatically synced to a global content repository that's accessible at any time. Translators work on live content within the development cycle, eliminating the need for freezes or batched translations. Whether you are translating a website, a game, a mobile app or even video subtitles, Transifex gives developers the powerful tools needed to manage the software localization process.

Sign up for a free 15 day trial and support Software Engineering Daily by going to transifex.com/sedaily. That's transifex.com/sedaily.

[INTERVIEW]

[00:05:47] JM: Venkat Venkataramani, you are the CEO and cofounder of Rockset. Welcome to Software Engineering Daily.

[00:05:52] VV: Thank you. Thank you for having us.

[00:05:54] JM: You were at Facebook for 8 years. You started in 2007 and I'd like to spend a bit of time talking about scaling Facebook, because you are first on the team that built the backend for Facebook chat and then eventually you got into caching infrastructure, data infrastructure, a lot of different things. Let's start with that first chat product that you worked on.

Facebook chat was a unique product in relation to something that people would build today. Like if you build a chat product today, you wouldn't have scale from day one, but Facebook was already a popular product. So when you were building Facebook chat, you knew that this had to be scalable from day one. How did that affect the architecture way back in 2007?

[00:06:39] VV: It brings back a lot of old memories. Yeah, thanks for the nostalgic trip. It was a very challenging project mostly because it was a completely web-based chat. Back in the day, AJAX was just a thing and we were doing these long polls and that kind of pattern was not well-understood. There were lots and lots of issues both from sort of the client side JavaScript and on the backend scaling. I would say, yeah, scaling was a – We spent a lot of time making sure that the product would work up on launch. So we came up with this idea to sort of do a dark launch, which is really how we did scale this thing, because there's no amount of workload that we could create in our own sandboxes that could mimic, at that point, 40, 50 million monthly active users all actively using the chat product.

So the way we actually did scale this thing back then was to launch the product on all the client side without actually any UI and sending kind of like randomly pick a friend and send our fake kind of chat message or pretend that somebody is actually tabbing and sending a chat message and then tracking how many of them are successful and whatnot. Scaling that kind of like, we call that dark launch, and doing a dark launch was extremely important, because I think we thought we were done. We thought it would work, and then we did the dark launch and we realize nothing really worked. We spent about two months from the day we thought we were ready to the day we were actually ready.

[00:08:02] JM: So basically had like the – You deployed the client, but it just wasn't rendering anything on the UI?

[00:08:08] VV: Yes.

[00:08:09] JM: Then randomly you would have a message that would be sent technically from my account to maybe your account, or it would send a bunch of messages and we would have a conversation, but you're just mimicking a conversation sending like "crack", or, "Hello, I'm Jeff," or "Hello, I'm Venkat."

[00:08:27] VV: Yes.

[00:08:28] JM: And you just did that at enough scale to find out if it would work without actually revealing to users that that was happening. That's pretty interesting. What kinds of issues did that reveal?

[00:08:39] VV: Oh my God! I mean, every part of the stack, nothing really worked. We thought everything would work and nothing really worked when we first did the dark launch. I mean, at every part of the stack, there were some scalability issues. The number of concurrent connections that we thought we were ready for, the memory all had poor connection. Suddenly crashing a bunch of backend servers. We were mostly on Apache Zend PHP stack back then.

The Facebook, kind of like the PHP application servers we're running, and the per message overhead, processing overhead, was way too much. That we realized if enough people started chatting, maybe even like a small percentage of the people started actively chatting, the number of endpoint hits just from the chat product to the web backend was higher than all of Facebook put together. We had to basically go and optimize that so that – Because every page load is explicit step from the user, except that a chat message cannot be as expensive as a frontend page load. So we ended up just figuring out tons and tons of things like that, and then we'll find one bottleneck and then we fix it and then find the next bottleneck. It's kind of like telling to say we were in that mode for about maybe two and a half months or something like, and then we cranked up all these thresholds. Like we had these fake thresholds on how many users should we pretend chatting right now and how often should they be sending messages and things like that.

We cranked it up by the end of the second month high enough and things were working very well to the point where the day we actually launched to all the users, we were so confident it's going to work, because we tested it at even much harder higher scale than what actually happened on the launch day.

[00:10:18] JM: After building chat, you worked on scaling memcached infrastructure and other aspects of caching. Explain the role that caching played in Facebook's early days. Why is caching infrastructure so important to Facebook?

[00:10:35] VV: Memcached was just amazing back in the day. I feel like I often tell this story where there're a lot of services that Facebook built. There was only one thing that actually truly scaled to Facebook scale, and that was Memcached. Everything else was using Memcached as scaling kind of like layer. Say you build a microservice for newsfeed or you build a microservice for ad serving, and you'll just be suddenly having to scale to millions of operation per second, but a lot of the time you're repeating the same operations over and over and the actual backend that is doing their operation can't really do that at that scale. So you use Memcached as a look aside cache.

So it was very, very important that we had a very high-performing, highly available, highly reliable Memcached not just for one part of the product, but for all of Facebook. Back in the day when Memcached used to go down for even – I don't know. Like let's say tens of minutes, most of Facebook for a lot of people would be a blank page. We can't render anything, because the other backends that Memcached was kind of like fronting can't really take the kind of throughput and the load that Memcached would have no problem dealing with, because it's not really trying to do too much work, too much computation to process a request, but everything that it was fronting in all the backend services that Memcached was fronting per request has to do a lot of work, and we wouldn't have been able to scale any of those backends during the hypergrowth years of Facebook in 2007, 2008, 2009 and even much later than that without the power of Memcached.

[00:12:09] JM: When I think back to that time, Facebook I think was the earliest highly-interactive multimedia product that was so widely used. I'm trying to think of other products in that category, but I guess there's kind of Gmail, maybe Google Search, maybe some online games. But you probably didn't really have patterns to follow. There weren't really any companies that had done stuff like this, except maybe Google.

[00:12:38] VV: That's a very good point. I used to say this in all of my TED Talks back when I was managing online data infrastructure at Facebook, where the social networking kind of backend workload was just very, very different mostly because of the huge fan-ins and the fan-outs. What I mean by that is you have to build, let's say, a search backend and there's a lot of – It's a read-only index. There are a lot of hard challenges in order to build a web scale search engine, like what Google did both in terms of ranking problems and in terms of just scaling the online serving index itself to be so fast.

But at the end of the day, you still have to figure out how to index the web and keep it replicated in many different places so that on query, the search [inaudible 00:13:20] is slow. But there were other search engines. There were other things from where they could have been lessons learned that is useful.

Then the emails, like the Hotmail and the Gmail and those were the kind of like big products that were really popular, and the biggest difference there is that they're all very partitioned. You log in and you look at your email, a good email backend would just simply start prefetching your inbox when you're starting to type your password. Then it can give you a very, very good snappy experience, because it's very predictable what the person is going to come and look at, which is all the hundred email in their inbox.

But in a social network, I think the problem is all the datasets are extremely interconnected. When enough people login, they're practically looking at everything that was created in the last few days or few weeks or what have you. So the problem really came to how do you basically cache and serve a very popular data-hungry product that is highly interconnected where a lot of the traditional approaches won't work? The kind of problems where the huge fan-ins that would happen on the server side, where let's say I still remember the Steven Colbert in 2008 starting a Facebook group to say, "I'm running for president."

That database that we use to basically host that particular content would just key lower, because the entire – There's like a big percentage of login population once they interact with that one piece of content. A single server just would not be able to deal with that all even if that is the only piece of content that that server is serving. Those are the kind of scalability challenges that kind of extreme hotpots that happen on the server side and the huge fan-ins that would happen on the client side, because you're not just going and logging into one server.

You're asking for some small pieces of content from lots and lots of different machines and very quickly when all of them come back, you're trying to render the kind of Facebook newsfeed like data rich experience, and all of that required tremendous scaling and tremendous kind of systems infrastructure. There weren't too many places we could look up to and say, "Hey, we should build it like that. I think a lot of the ideas had to be – Like infrastructure had to be kind of like built by Facebook for Facebook during those years.

[00:15:35] JM: Yeah. This was pre-cloud computing, or just around the start of cloud computing. You didn't have access to really good CDN infrastructure I don't think, or CDN infrastructure that would serve Facebook's purposes.

Anyway, you were an engineering director at Facebook eventually after nearing – Around the end of your 8 years at Facebook. You were building data systems for profiles and friends and messages and photos, and you took those lessons from the early days of building caching infrastructure and this chat system. You turned this into 8 years of working on data infrastructure.

When you zoom out and you think about the perspective of the 8 years you had at Facebook of building all that data infrastructure and solving all these problems and seeing, in some sense, the future, because a lot of the problems that Facebook had in its first 8 years or in the 8 years that you were there are things that people are having to deal with on a regular basis these days.

What are the big lessons? What are the most important takeaways from the engineering time you spent at Facebook?

[00:16:47] VV: What are the takeaways? If I have to zoom out and look at the perspective, I think I was blessed to work with some amazing people in those teams and in infrastructure and product and engineering that I would say Facebook was not the first social network, but it was definitely the first one that scaled to a billion active users. There was a lot of team, did a lot of hard work for that to be true. The online data infrastructure team, one thing that I'm very proud of that is a big takeaway is that reliability matters and performance matters that I think I'm very proud that Facebook was largely up through the entire – Those 8 year time period and all the efforts of my team were necessary for it. Definitely not sufficient by any means, but definitely necessary.

The other thing that I would say, the philosophy of what we were doing at my previous team at Facebook was to make it very easy to build products and services at Facebook. There are lots of data coming in and we really wanted product engineering to be bottlenecked by their creativity. What is the most engaging application could we build and not really be curtailed by what can the online data infrastructure do for them? What can it handle and whatnot?

That was kind of an ideal goal. We were never really there to be really honest, but we got quite loss. There were entire products that would get launched and build on top of the online data infrastructure that we would build that we wouldn't even hear about until they launched. We would read about it in the news just as like everybody else that, "Hey, Facebook launched another product." But no surprise that it was entirely built on top of the server side and the online data abstractions that we had built and managed and scaled.

Those were the key takeaways I think, right abstractions matter, making it very easy to build data driven products can have sort of a foundational impact on how quickly an organization can move and how quickly they can innovate and how much they can put the data to use to build very, very interesting and engaging products. That was my key takeaway, and a lot of that was kind of like what really prompted us to start Rockset to do the same thing in the cloud now for anybody building applications in the cloud.

[SPONSOR MESSAGE]

[00:19:00] JM: Managed cloud services save developers time and effort. Why would you build your own logging platform, or CMS, or authentication service yourself when a managed tool or API can solve the problem for you? But how do you find the right services to integrate? How do you learn to stitch them together? How do you manage credentials within your teams or your products?

Manifold makes your life easier by providing a single workflow to organize your services, connect your integrations and share them with your team. You can discover the best services for your projects in the manifold marketplace or bring your own and manage them all in one dashboard. With services covering authentication, messaging, monitoring, CMS and more, Manifold will keep you on the cutting-edge so you can focus on building your project rather than focusing on problems that have already been solved. I'm a fan of Manifold because it pushes the developer to a higher level of abstraction, which I think can be really productive for allowing you to build and leverage your creativity faster.

Once you have the services that you need, you can delivery your configuration to any environment, you can deploy on any cloud, and Manifold is completely free to use. If you head over to manifold.co/sedaily, you will get a coupon code for $10, which you can use to try out any service on the Manifold marketplace.

Thanks to Manifold for being a sponsor of Software Engineering Daily, and check out manifold.co/sedaily. Get your $10 credit, shop around, look for cool services that you can use in your next product, or project. There is a lot of stuff there, and $10 can take you a long way to trying a lot of different services. Go to manifold.co/sedaily and shop around for tools to be creative.

Thanks again to Manifold.

[INTERVIEW CONTINUED]

[00:21:15] JM: There's something about what you just said, this idea of having a team mentality where you really wanted to unlock the creativity of the product engineering side of the house by removing these frictions around data infrastructure, and the vision for that being unlocked for other companies is kind of inspiring and I think that's what we're going to get into with Rockset.

I've done a number of shows recently where we've talked to companies like DoorDash, and Uber, and Airbnb about the data infrastructure at these companies, and it's so far from a solved problem. You have search problems. You have Spark query problems. You have batch processing problems. You have real-time streaming issues. You have all of these issues around what you could bucket as data infrastructure or data platform and nobody really has a handle on what are the best patterns to solving this. Explain what you're doing at Rockset. How are you approaching this problem of data infrastructure?

[00:22:30] VV: I think holly approaching this at a level is I think there are too many specialized systems that all were born in the pre-cloud era, that I think cloud economics are fundamentally kind of obsoleting them and requires a fresh start or a fresh perspective.

At a high-level, yes, a lot of people are struggling with a lot of data and there are lots of different kinds of problems and I don't believe that there's going to be one database to rule them all or anything like that, far from it. I think there's always going to be different application requirements for different kinds of datasets and you'll always have a multitude of data management systems to build a very complex application and logistics like Uber, or DoorDash, or Facebook, absolutely.

But I think the complexity is also too much in the lots and lots of specialized systems running on dedicated Linux servers, because I think everybody is still trying to solve the problems using software that was built to run on data centers on dedicated Linux hardware. I think when we look at it on how the cloud is changing it is a lot more about how the cloud economics is fundamentally different.

So what we want to do is we want to see what is the better abstraction in the cloud and how do we make things simpler while keeping the power of the system just as high? What are the right abstractions in the cloud and how do we simplify the story rather than asking people to duct tape together multiple disparate data management systems to solve any problem, build any application or do any automation that is data-driven? How do we eliminate a lot of the complexity and how do we simplify this whole story? That is what drives us to do what we are doing at Rockset.

[00:24:17] JM: The cost structures of the cloud have changed the economics of what you can do with a managed data platform. Explain why that is.

[00:24:28] VV: Very simple thing actually. It's kind of a very duh moment, which is like whether you rent one server for hundred hours, or hundred servers for one hour, it costs you exactly the same in the cloud. This was never true in on-prem, because it's not even an option to buy a hundred servers and use it for an hour and return it to your hardware provider. That's not a thing. So you have to capacity plan that. You really have a limited kind of computational quota to do anything. Then all the software optimizations makes sense, where you have these three machines, squeeze every last drop out of it because that's all you're going to get for the next – I don't know, three years. The more efficient you are, the more I can extract out of this particular investment that I've already made. Almost all of the software that was written in the pre-cloud era were built with that mindset.

In the cloud, what we're basically saying is if you can parallelize something, you should always parallelize something as long as your schedulers and your infrastructure can very quickly grow and shrink, and shrinking as just as important, because you can redial all the CPU and all the compute and all the storage when you're not using them, then you save a lot of money.

If you could parallelize it without increasing the cost, if your software is pretty good at doing that, then you should not only ask for the same kind of efficiency and the same kind of cost dynamics, but you should also expect things to be much, much faster in the cloud than what you're used to, because there's no reason to just do the slow route just because you happen to have installed your software in the cloud on only three machines. If your software can't quickly grow and shrink based on your demand and based on your desired SLAs, then that means it's time to change your software in the cloud and not be stuck with slow performing systems in the cloud. That's kind of like the fundamental realization that we had, which is why I think we have done a lot of software engineering based on these kind of cloud principles that we have observed and our software architecture is build ground-up in the cloud to take advantage of the fluidity of the hardware that is available.

[00:26:40] JM: To some degree, you can use these abundance of resources to speed things up to have a wider degree of how you're indexing things. For example, if you ingest a bunch of data in Rockset, as I understand, you're keeping a bunch of different indexes over that data. Anybody who's worked with databases, if you want to speed up a database, you're often times just adding an index that is a faster way of running a certain type of query. If you can maintain more of those indexes, then you can have more performant responses to more types of queries. Is that an accurate abbreviation of what you're doing to some degree at Rockset?

[00:27:26] VV: Correct. That's very, very good abbreviation of what we're doing. At the end of the day, we are a cloud indexing company. All we need is – There are other ways that we are also very interesting, which is like you don't need to describe the shape of the data or the shape of the queries and we automatically index the way – Schematize your data and index in more than one way so that a wide spectrum of your queries are fast. So that's exactly correct, what you said.

The cool thing is why are we able to do this now? Why wouldn't a multiple indexing strategies work in data centers? I think it comes down to, again, being able to leverage the storage hierarchy that is available in the cloud really. If you were to go and build a software that requires multiple combination of storage systems, one for fast in a hot data and one for cold data and somebody needs to configure, "Now, you're dealing with multiple vendors and multiple hierarchical storage systems," that all might not have the same kind of performance from one installation to another. So it's kind of very hard to build a working system on hierarchical storage on-prem. It's not that it's not possible. It's very, very difficult to get it working in one setup, in one data center, but building a software product that would just work with a multitude of them and still give you a really good performance is very, very hard. That is why I think it hasn't really been done before.

But when you look at it in the cloud, you have all different hierarchies and there's a very nice sort of cost to performance that is very well documented, very well understood on what is the read latency for your local RAM, locally at that storage, block storage, remote block storage, in SDDs, remote block storage in disks and all the way to things like S3, Amazon S3 and other blob stores and Glacier and what have you.

The hierarchy is not only deep, but also very well understood and very reliable to something that you can actually take a hard dependency on as building a system software. If your software can leverage it, now you can actually make, let's say, two copies of your data and keep the two copies in different indexing organizations so that you can actually now automatically speed up a wide spectrum of queries without asking too many configuration and setup questions from the user.

Again, I think it's really the cloud that they're readily available, cheap and hierarchical storage services that are available in the cloud that really enables this kind of a technique. Without that, I think it would be very hard to do something like this.

[00:29:50] JM: We had a show a while ago with somebody from MongoDB where we were discussing database query performance and how to speed things up or the tradeoffs you can make, and one of the things that the guest was exploring was the fact that when you keep more indexes over a certain piece of data or over sets of data, the more indexes you keep, the longer it can take to have consistent updates when you ingest a new piece of data, because when you ingest a new piece of data, then you have to go and update all of the indexes that are over that piece of data. If you keep too many indexes and you have high-consistency requirements over your queries, then you can get into this issue where whenever you're ingesting a piece of data, you're locking the database while you're spending time updating those indexes. Is that another tradeoff that you face when you're keeping this wide variety of indexes over the data?

[00:30:51] VV: Absolutely, but also I think what kind of indexes. What you said is absolutely correct. In a traditional, what is generally considered the MongoDB like sharded databases are called turn partition databases. The turn partition databases, yes, I think the indexes are built in a certain way where the more and more you have them, yes, the cost of a write operation goes higher and higher and that is just like basic math. There's nothing spectacular about it.

The way when we say we build different kind of indexes, we don't actually build hundreds and hundreds of types of indexes. We actually largely build two, maybe sometimes three indexes, but what type of indexes really matter? What we do is we build an inverted index, which is like a search index on all these data. What that does is whenever you have a query coming in that has arbitrary filters on your incoming data where you're looking for, "Show me everything where this property is this value and that property is true and this other property is within this value range and what have you." Search engines are extremely good at running arbitrary filters and very quickly getting – Finding the needles in the haystack.

So we build that index to be able to very quickly grow from large volumes of data to the small portion that your query is actually targeting, and we're very, very good at that, faster, because it does distributed processing across many different cores and many different machines. That is one type of index we do.

The second type of index is for analytical queries when you want to do large scans over huge volumes of data. For that, the state of the art are columnar indexes, and columnar databases and data warehouse use that extensively. That is the second type of organization that we also keep behind the scenes. We actually don't make many copies of the data. Every index is some ways a copy of your original data. That's why it's able to serve the queries, because the data also lives in the index. We largely make two copies of the data, but our optimizer, our SQL kind of query processing layer knows which one to use for what parts of the query.

If you run a query that says, "Find everybody whoever worked at this company. Now, go and retrieve all of those people's names or whatever, and then look at all the places where they live. Then now go find other people that live from that location," or something like that, the first query is the one that needs to now do a very quick filter. The first part of the query, I mean. Then we use kind of a search or an inverted index for that. Then for the subsequent processing, we need to do a lot of other distributed SQL processing and aggregation and there we have built a custom distributed query processing engine in C++ that can elastically grow and shrink the compute resources that are allocated for that particular query just so that we can continue to keep it fast.

Then if your query involves a large scan over large amounts of data, then the query optimizer that we have built will automatically pick the columnar organization of the data where you can actually do large scans over just portions of the data that you want to do a lot more efficient than what an inverted index or a search index would be able to do.

[00:34:03] JM: If I understand correctly, depending on what query is issued to the Rockset database, or data storage system, whatever you want to call it, depending on what query is issues, that query may use a different index.

[00:34:20] VV: Correct.

[00:34:21] JM: Okay. Now, I think I read in some of the documentation that you are not trying to build what is called an OLTP database. This is not a transactional database for storing a user account, for example. This is more for aggregations, or search indexes, or doing search querying, doing kind of ingestion of click stream data and exploratory analysis, things like that where you don't need strongly consistent up-to-the-microsecond data. Is that correct?

[00:34:56] VV: That is absolutely correct. We are not a transaction processing data backend. I think a single node well optimized MySQL or PostgreS server can take someone's transaction processing system a really, really long way. They're very, very good transaction processing systems. If you really look at – There's a data explosion. Everybody talks about data is the new oil and what have you. If you really look at where is this coming from, they're not coming from – Because people are buying more. Those kind of explosions are not based on, in my opinion, kind of transactional processing exploding the roof.

Even when a business grows, the data – You take any company that is growing, their data needs grow exponentially not because they're doing exponentially more transactions or orders. It's because they're collecting a lot more information on a lot of that information that dominates. I think your data kind of workflows and data streams and all the data that you accumulate are all what we call kind of even data.

What explodes is machine-generated data, data that are automatically kind of created when somebody is using a product just so that you can understand what aspects of the product are working and what aspects of the product are not and things like that. So when you look at the explosion of the data, the problems that dominate the complexity of the data architecture and all the things that you want to do it all come from these kinds of datasets where one application is generating it and another application is trying to consume it and flows in a real-time stream from the application that generates it to the various sets of other applications and microservices that you might want to build that consumes those datasets and those data streams. That is where we think things are way too complex, because people continue to think about this kind of like bimodal type of data management systems where you're either a transaction processing system or you're a warehouse.

We really think the data management kind of backends is missing a third leg. There's an increasing number of operation applications that you want to build that needs to work with real-time data. It doesn't have to be transactionally consistent, but the data latency has to be in milliseconds and cant' be in hours or days. You still have large volumes of that that cannot fit at a single machine, and now you want to build a fast, interactive application on that dataset and how do you do that? That's where I think a lot of the complexity lies today, and that is really what we're trying to eliminate at Rockset.

[SPONSOR MESSAGE]

[00:37:34] JM: We are running an experiment to find out if Software Engineering Daily listeners are above average engineers. At triplebyte.com/sedaily, you can take a quiz to help us gather data. I took the quiz and it covered a wide range of topics; general programming ability, a little security, a little system design. It was a nice short test to measure how my practical engineering skills have changed since I started this podcast. I will admit that though I've gotten better at talking about software engineering, I have definitely gotten worse at actually writing code and doing software engineering myself.

But if you want to check out that quiz yourself, you can help us gather data and take that quiz at triplebyte.com/sedaily. We have been running this experiment for a few weeks and I'm happy to report that Software Engineering Daily listeners are absolutely crushing it so far. Triplebyte has told me that everyone who has taken the test on average is three times more likely to be in their top bracket of quiz course.

If you're looking for a job, Triplebyte is a great place to start your search. It fast tracks you at hundreds of top tech companies. Triplebyte takes engineers seriously and does not waste their time, which is what I try to do with Software Engineering Daily myself, and I recommend checking out triplebyte.com/sedaily. That's T-R-I-P-L-E-B-Y-T-E.com/sedaily. Triplebyte, byte as in 8 bits.

Thanks to Triplebyte for being a sponsor of Software Engineering Daily. We appreciate it.

[INTERVIEW CONTINUED]

[00:39:33] JM: So let's take a typical use case in advertising. Let's say I'm building a social network and users are scrolling through that social network. Their mouse is moving around the screen. There are ads that are appearing in the newsfeed. There are stories that are appearing in the newsfeed. As a user is moving their mouse around the page and clicking on items in the feed, the social network needs to collect this clickstream data, and this is a ton of data. This problem is very similar to a self-driving car that's driving around collecting tons of data, or a log ingestion tool over a high-volume web application where you have high-volumes of some data that is structured that has a really nice well-defined schema to it and some data that is less structured, where you have kind of unique events or like a log message, for example. A lot message is kind of – It has unstructured elements to it, but it also has a structure to it.

We're kind of talking about the requirements for a data store that would be able to capture any of these kinds of data and be able to build indexes and be easy to query for lots of different applications that we might want to build against that application, because if you're working at Facebook or any other social network that has this highly interactive mouse around clicking on items, clicking on ads, you want high-fidelity into – You want to have this data platform that different internal product teams can go and use, can go and leverage and build whatever kinds of applications on top of it they want.

What are the requirements for this kind of data store and what is going to happen as that data is being ingested into Rockset?

[00:41:25] VV: Yes. I think that's exactly the type of applications where I think we can really shine, where you have some datasets that are probably real-time and semi-structured, but you really want to also be able to join that with other structures and other data sitting in other places. So what we say about Rockset is you can go from useful data to useful applications with a single click.

With Rockset, what we do is get an account, point us at your dataset. It could be structured as semi-structure, real-time or not, it doesn't matter. We will automatically ingest it, continuously ingest it. We'll automatically schematize it and we will instantly start pouring very fast SQl queries on top of the data.

So say you have a stream coming in, clickstream or what have you, and the data is semi-structured. You don't need to describe the shape of the data ahead of time to Rockset. Point your data at Rockset and it will look like a fully indexed, fully optimized SQL table on the other side. The whole read API is offered as a SQL or REST. So if you know how to work with REST APIs and you know SQL, you already know how to build an application on top of Rockset.

That's basically our abstraction, our view on how we are making this whole process a lot, lot simpler. We're looking at, in terms of use cases, think of the personalization layer that you are sort of referring to in these kinds of like kind of social and ecommerce kind of applications where based on the behavioral data that these products are collecting, which is like what are you searching for? What do you like? What do you not like? What kinds of content do you interact with more than others and those kinds of behavioral data, compared with the past transactional data on like, "Who are your friends? What are the other products that you previously bought from this website?" You really want to bring these two together in real-time to really build a highly personalized experience for your end user, and that is the kind of an application or an example where you really have to bring both kind of structured data about the user and about the past transactions and the semi-structured real-time streams together. The simpler you can make it, the more and more data power your application and your automation is going to be, and that is the kind of set of applications and use cases that we excel at.

[00:43:47] JM: Actually, I want to dive in a little bit deeper to this structured versus unstructured data question, because I think some people get a little confused about this. It took me really long time to have a basic understanding of structured versus unstructured data. Can you give me an example of a data stream and fields that would be semi-structured or how you would describe it structured versus unstructured, and then let's talk about how you ingest that data and how you derive schema from it.

[00:44:17] VV: Sure. Structured data is very well understood as what would actually live in a SQL table and a MySQL or PostgreS or one of these relational databases. Semi-structured, largely the first two, when we say semi-structured, we refer to things like JSON, things like Parquet, where there are a couple of issues with that. Number one is you can't ask for a list of all the fields that might appear ahead of time in a lot of these kind of datasets, because it's maybe coming from third-party or it's coming from your own frontend application and it's very easy for – Today it looks like there are like, let's say, 17 different fields and values and tomorrow there's going to be 18 or 19 and there's really no kind of schema registry or what have you where people don't do ALTER TABLE add column before logging in a new field.

Semi-structured, the shape of the data is fluid. It will be changing on you a lot, and there could be lots and lots of different fields and values unlike a fixed set of columns in a table. That's one big difference between semi-structured and structured.

Then the second thing is when it's semi-structured and coming in fluidly, the values could also be complex. In a typical traditional structured database, the values would either be numbers or strings or Boolean or what have you. But in a semi-structured dataset, the value itself could be a nested object, or a nested array of things. If you look at, for example, the Twitter file host API, which is a really good example for semi-structured data real-time data stream, every tweet is a semi-structured JSON object that comes when you subscribe to their API. A couple of those fields are what you'd expect. What is the text of the tweet? What is the time that tweet was created and what have you?

But then when you talk about, “Okay, who’s the author of the tweet?” That is a nested kind of object within the tweet that gives you, you know, it says “user”, and then the value is itself lots and lots of fields and values. It’s like this is the name of the user. This is the screen name. This is actually the place where they logged in from or something like that. There are lots of information about the user that is a nested object. This would have gotten normalized in a typical structure database where there’d be a user I.D. and a user table separate from the tweet table, but that is actually not nested within – One is nested within the other in these kind of semi-structured datasets.

That kind of gives you an idea of the differences. So the fluid shape of it, the complex structure of it makes it semi-structured that I think traditional SQL base systems often really struggle with dealing with these kinds of datasets.

[00:46:55] JM: Okay. The Twitter Firehose is a great example, because, as you said, there’s schema around things like the username, or the time at which the tweet was made. But even just the text of the tweet itself is – Well, I guess it’s structured in a sense that it’s just 140 character message, but it’s unstructured in the sense that anything could be in that message. But I guess you’re also talking about the unstructured nature, meaning there can be nested fields. So you could have – If you’re talking about who is this user, you can have nestings of where maybe this person didn’t put in a first name and last name. So there’s nothing in the first name, last name nested field. But other users might have a first name and a last name. So you have these different nesting structures.

Okay. So how does that impact the ingestion process? Because we’re talking about Rockset as this data platform that can ingest all these different types of data streams, whether they have schema or not, and you can impose schema on them, because in order to derive an index or a useful index from something, you have to impose some kind of schema on that, whether it’s an inverted index for a search engine or some other kind of index. Tell me what happens during the ingestion process of data for Rockset.

[00:48:14] VV: Exactly. Very good question. What you said is largely true for structured databases that in order to create an index, you need to know the structural data that you’re indexing. That is one of the fundamental things that we're challenging and questioning with our strong dynamic typing. Our type system is strongly typed, but also dynamically typed and that we internally use for our query processing, which is why when data like the Twitter Firehose comes in, just think about what it would take today before Rockset if you were to build an application on top of that? If someone comes to you and say, "Hey, you're a data engineer or an application engineer, I want to be able to build arbitrary – I want to ask arbitrary questions on top of this data. I want to find out all the tweets in the last two weeks that has certain words in it or were tweeted by a certain set of users, or that had a certain kind of like stock ticker symbols it them. I want to aggregate and find the most popular stock ticker symbols in the last 15 days across the tweets that I have and all sorts of these kinds of complex questions."

Now the first thing you're going to say is, "Well, I can't query JSON, so I need to ETL this into a structured database so I can do these kinds of complex filters and aggregations. You're going to be like, "Okay, how do I do that?" Now you need to go and take a random sample of the dataset and then you're going to look at like, "Okay, how do I flatten this out into a SQL table," and some of them would be – You'd be able to flatten this and you would sort of like build some kind of an ETL script either on the real-time or on a static dataset of tweets that have been dumped in some source. Then what happens if these API continued to evolve and there's more information in them than what you had, let's say, a month ago. Now your application now needs to exploit that, but your ETL scripts don't know about them. So now you go and redo the same thing again and again and again.

Instead, how does it work with Rockset is, literally, if there's a real-time stream, let's say it's Amazon Kinesis stream or it's coming in Apache Kafka and there's a topic that has a list of JSON tweets coming from the Twitter Firehose API, you literally have to get an account with Rockset and just give us read success to that topic or that Kinesis stream. You don't need to specify anything about what's in it. We would automatically deduct it's a JSON format and we will automatically schematize it and make the entire Twitter stream no matter how much how nested it is and you have nested arrays of nested arrays of nested arrays, it doesn't matter. We enable direct SQL-based access for all of those fields to un-nest them on the fly to filter out the set of data that is malformed that you don't want to include in your data processing to do type inspection to say, "Only show me when this field was this type, because other types are invalid for this type and I want to ignore them during my query processing and filter them, aggregate them, sort them, join them with another dataset." You can do all thoughts of complex analysis on that. On the automatically schematized and indexed data that Rockset does without having to discover the shape of the data.

How exactly it works is we look at every one of those JSON document coming in and we don't try to match it to some prebuild schema. If you have setup something that you want us to enforce, that we can do, but out of the box, we don't try to match it against some golden format of what we expect. We look at every document, every record that comes in in and by itself and we shard it to pieces. We retain all the pieces of content, which is here are the fields, here are the values, and this field happens to have this value in this type, in this instance of the document. We are basically organizing our data in a way that would allow us to do very fast and efficient SQL processing on top of that organized data without sacrificing performance at scale.

So every piece of content, you can think of it like sharded into small little pieces while preserving the type and the value all of that intact without having to comply with some predefined shape, which is where I think you suddenly have these alerts in traditional systems. We don't have any of that. We'll ingest them as the records come in in real-time and organize them in our backend. At query time is when we really try to match your query with like SQL query processing time. We try to match your query with what the data actually we have seen.

For example, say you have a zip code field that comes in mostly as integers and once in a while it is a string. We will store all the zip codes that came as integers as integers and all the zip code values that came as strings as strings in our backend when we are organizing them and indexing them up no ingest. But at query processing time you can say – You can write a query that says, "Show me all records where zip code is greater than zero," then we'll automatically only match all the integers and floating point numbers that we happen to have for that field at query processing time.

But then say you write another query that says, "Find me all zip codes that is equal to this particular string." Then at query processing time, we'll go and match all those records where that particular field happens to be of type string and the value happens to match what you provided.

So by preserving the type of the data and the values all in place when we're ingesting the data and indexing the data, allows us to do this kind of type binding to do the dynamic typing at query time. So at query time, you can interrogate it, you can write the query if the data suddenly gets malformed. You just need to tweak your query a little bit and very quickly unblock yourself and keep going in sort of break your [inaudible 00:53:53] pipeline and reloading your data and go backing and rescanning your data, revalidating your data again and again every time your data kind of evolves on you. Instead, whatever you use to do at ETL time at write time, you can now do that at query processing time, which would speed up a three-week project to be done in a matter of hours or days.

[00:54:12] JM: Now something that I was hoping we'd be able to get to, but I don't think we have enough time, is a discussion of RocksDB, because the name Rockset I think has some association with RocksDB, that the database is a database storage system that was built within Facebook. I think RocksDB is used as a storage engine in several other databases too, but I think maybe we have to save RocksDB for another episode.

I think we should close by talking a little bit about the data engineering problems that you're seeing in your early customers and how those compared to what you saw at Facebook and what you're seeing people build with Rockset since they have a newer solution. Yeah, what are the problems that you're seeing, the data engineering problems you're seeing in your customers and how do those compared to what you saw at Faceboo?

[00:55:09] VV: Very interesting. I think people try to largely build similar kinds of applications, but I think the biggest difference between what our customers see and what happen within Facebook is that the overall "the quality of the data", or the messiness of the data is very, very different. I think with our customers, a lot of the time they're trying to connect third-party datasets with some internal datasets that they have.

This is a problem that is even harder than the kind of data problems that I think honestly we had at Facebook, because largely the data that was created within Facebook was largely created by the Facebook application itself and it wasn't like third-party datasets that Facebook was trying to make sense out of.

Here, whether it is a marketing operations application that somebody is trying to build, or market intelligence. What's happening in the market? What's happening with my competitors? What products is moving? What products are not moving? To answer these kinds of questions, enterprises have to combine external third-party datasets and semi-structured data with that internal data, which is I think a really interesting kind of challenge, but very different than what I think we saw at Facebook.

So that is something that has been very interesting for us that these kinds of market intelligence applications and sales operations, customer support operations in the real-world needs to really – Is a lot more complex than even what I think some of the backend applications at Facebook was dealing with. So those are the kinds of use cases where I think we really shine, and so those are the kind of use cases that our customers are doing, whether it is data science or semi-structured data, building personalization engines on these semi-structured and structured datasets. All kinds of IoT sensor data, use cases. As I said, sales automation, like business operations automation kinds of use cases.

All of these things where I think almost all of them or sort of in certain ways very different and some ways much harder than honestly what we experience at Facebook.

[00:57:12] JM: Venkat, thank you for coming on Software Engineering Daily. It's been really fun talking to you about Rockset.

[00:57:16] VV: Thank you. Yeah, I think we have – Go to rockset.com. There is a "Get Started for Free" if you're interested in that, you can get started for free. There's a free tier and all you need to do is go to rockset.com and click that green button and we'll hook you up with an account, and I can't wait to see what people are going to build on top of Rockset.

[00:57:34] JM: Okay. Well, thanks for coming on the show. I'll talk to you soon.

[00:57:36] VV: Thank you. I appreciate you having us.

[END OF INTERVIEW]

[00:57:41] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on-the-fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team who have talked about building the product in previous episodes of Software Engineering Daily, and it's great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily.

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]