EPISODE 979

[INTRODUCTION]

[00:00:00] JM: NoSQL databases provide an interface for storing and accessing data that allows the user to work with data in an unstructured fashion. SQL databases require the data in the database to be normalized, meaning that each object in the entire database has an entry or a null value for each field.

One advantage of NoSQL is that the different objects are de-normalized, meaning that the different objects in the database can have unique fields. There's a widely held belief that NoSQL databases do not scale or that there is some significant penalty the developer will pay for using a NoSQL database as soon as their app becomes popular.

The truth is much more subtle than that. NoSQL databases can perform as well as or better than SQL databases if the developers know the query patterns that their applications make. SQL databases will be a better choice in the condition where the database has a very wide spectrum of access patterns. But in many cases, an application actually has a narrow range of different requests for the database, and a NoSQL database can perform very well if the database is structured and optimized for these requests.

Rick Houlihan is an executive with Amazon Web Services who works with database teams and engineers to optimize their products and their database infrastructure. Rick joins the show to discuss the tenants of NoSQL and describe the fundamental contrast between NoSQL and SQL database limitations.

We are hiring a software engineer who can work across both mobile and web applications. This role will include work on softwaredaily.com, our iOS app, and our Android application. We're looking for someone who learns very quickly and can produce high-quality code at a fast pace. We're looking to move beyond the world of just being a software podcast into more of a platform of information about software. If you're interested in working with us, send an email to jeff@softwareengineeringdaily.com.

We're looking for somebody who is hungry and wants to learn quickly and wants to build lots of software. If you are that person and you're hungry, it doesn't matter what your experience level is as long as you have built and shipped meaningful applications. Send me an email, jeff@softwareengineeringdaily.com.

[SPONSOR MESSAGE]

[00:02:30] JM: Today's show is brought to you by Heroku, which has been my most frequently used cloud provider since I started as a software engineer. Heroku allows me to build and deploy my apps quickly without friction. Heroku's focus has always been on the developer experience, and working with data on the platform brings that same great experience. Heroku knows that you need fast access to data and insights so you can bring the most compelling and relevant apps to market.

Heroku's fully managed Postgres, Redis and Kafka data services help you get started faster and be more productive. Whether you're working with Postgres, or Apache Kafka, or Redis, and that means you can focus on building data-driven apps, not data infrastructure.

Visit softwareengineeringdaily.com/herokudata to learn about Heroku's managed data services. We build our own site, softwaredaily.com on Heroku, and as we scale, we will eventually need access to data services. I'm looking forward to taking advantage of Heroku's managed data services because I'm confident that they will be as easy to use as Heroku's core deployment and application management systems.

Visit softwareengineeringdaily.com/herokudata to find out more, and thanks to Heroku for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:04:04] JM: Rick Houlihan, welcome to Software Engineering Daily.

[00:04:06] RH: Hey, great. Thanks for having me. I really, really appreciate it.

[00:04:09] JM: You've said that the history of database technology is a history of a data pressure. Explain what data pressure is and how it relates to the databases that we use today.

[00:04:19] RH: Yeah, sure. That's a great question to start with. I always like to startup with that kind of analogy of we've always had these series, like I said, peaks and values of data pressure. What that really means is it's the ability of systems that we're asking to process our data to process that data in a reasonable cost or in a reasonable time. When one of those dimensions is broken, that's what I'd like to refer to as a technology trigger. Over the years we've invented many things as we've encountered these technology triggers and data pressure.

[00:04:49] JM: There are several explanations for why NoSQL rose in popularity after SQL had been the predominant database type. We've explored some of those different theories on this show. Give me your historical perspective for why NoSQL became popular.

[00:05:07] RH: Well, sure. I mean, really what it comes down to, again, as people started to process volumes of data, the relational database that we've used for so many years turned out to not scale so well. That really points back to the reason why it was invented in the first place, and the relational database came into being because, again, we're experiencing data pressure, it was the cost of processing data that was preventing us from scaling, and the relational database decreased the pressure on the storage systems, because the normalized data model de-duplicated that data and allowed us to free up storage, so to speak, which was really the most expensive resource in the data center 3 or 4 years ago.

But now, today, fast forward, we pay pennies per gigabyte and we're paying dollars per our CPU minutes, and really the CPU is no longer just this fixed asset that's kind of spinning in idle loop when it's not doing anything else. It's an asset that we can use to do other things. So joining data and running complex queries is really not something that we'd like to spend our money on, so to speak.

NoSQL databases allow us to use a de-normalized data model that's finally tuned to the access pattern. It allows us to use simpler queries that don't burn up the CPU so much, but it does this at the cost of the storage, which is actually today quite fine, because storage is so cheap, and really the CPU is where we want to drive the efficiency.

[00:06:29] JM: With that said, is it a free lunch given infinite storage space? The decision to choose between a SQL and a NoSQL database, or are there penalties in terms of latency or some other penalties at a high level while we're talking to high level?

[00:06:46] RH: Yeah. No. Absolutely. NoSQL database do really well when we have a very predictable access pattern. When something is a repeatable process and we're thinking kind of a transactional application, online transaction processing or OLTP apps. These are great applications for NoSQL databases, and that's because, again, if we have a well-understood access pattern, then I can structure the data that's very specific and tuned to that particular pattern.

Relational databases, they're agnostic to the access patterns, right? But that really means they're optimized for none of them, but that's okay because they have a really good purpose in life, and their purpose is to satisfy the need for an ad hoc query. That kind of application is what we would typically call an OLAP style application, right? Online analytics processing.

For those types of applications, you don't really know what kind of questions you're going to be asked. What the access patterns are going to be. They might be different today than they are tomorrow. For those types of applications that normalize data model with an ad hoc query engine makes the most sense. There's a small set of applications, those OLAP applications, that still makes sense for the relational database. But for everything else, we should really be thinking NoSQL first, because it drives a lot of efficiency into the system and the cost of scale is so much lower that it's well worth the consideration.

[00:08:05] JM: Let's talk through the anatomy of a NoSQL table. First, explain how does a really big table get divided up into partitions using a partition key.

[00:08:17] RH: Yeah, sure. This is fundamental premise for all NoSQL databases. They all scale the same way. Typically what you're going to do is you're going to define some sort of item that we're going to insert into a collection. This item is going to be uniquely identified on some attributed DynamoDB, that's called a partition key. In MongoDB, that's the under bar ID.

At some point you're going to choose to use a shard key or assign a shard key with MongoDB that allows us to distribute this data across and arbitrary what we call key space. Now in DynamoDB, the partition key serves that purpose. When I define the partition key in a DynamoDB table or in any wide column database, I use this partition key value as the shard key.

What we're doing is we're taking this value. We're going to create an unordered hash index across this logical key space and then we're going to start laying these items out across that key space. If I need to scale the system in either throughput or storage capacity, what I'll do is I'll split the key space across multiple physical nodes. This gives me the ability when I query this system. As long as I provide that shard key or that partition key condition within the query, I know exactly where to go on the array of nodes that are servicing the key space. There could be thousands of servers participating in the servicing of this key space.

But as long as I get that partition key quality condition, I know exactly which server to go to. This is what gives me fast and consistent performance in any scale for NoSQL databases, again, because it's automatic. As soon as I query the system, the request writer knows exactly where to go to get the data. I don't have to search across the key space to find that information.

[00:09:54] JM: What is a sort key?

[00:09:56] RH: Okay. A sort key is when a wide column database, what you're doing is you define a primary key which consists of both the partition and the sort key. In a partition key, it uniquely identifies an item in a partition key only table.

When I add the sort key, an interesting thing happens. The partition key now identifies a folder or some sort of container, and the sort key uniquely identifies the item within that container, and when I query the system, I can do two things. I query with a partition key quality condition that says, "Go here," and then a sort key condition which can include range queries, like greater than, less than, or between. Things that will give me the ability to kind of filter the items that live in this partition, and we're going to play a lot of games when we do the data modeling, because

what we're going to use that sort key attribute and specific conditions on the sort key attribute to limit the items that are being returned with I query a given partition.

You can think about partition on a given table in DynamoDB or any wide column database as a collection of items, then there's going to be sort key conditions that I'm going to use to retrieve certain items from within that collection.

[00:11:05] JM: We'll keep working through a few of these terms just because I want to set it up for a more higher level or a more rich conversation. But just to continue moving through some of this technical vocabulary, could you define the term partition overloading?

[00:11:20] RH: Sure. Partition overloading is when what we're going to do is we're going to create multiple types of partitions on the table. Typically, in a NoSQL database, we're going to insert all of the items that we need that we're interested in. They don't all have to be the same structure. That's one of the big differences between a DynamoDB table and maybe your traditional relational database.

When I define a table and a relational database, all of the items or all the rows on that table look the same. In a NoSQL database, when I define a table or a collection, really what that is, it's an object collection. I'm going to throw a lot of objects in there. They can be all different types.

When we talk about partition overloading on a wide column database, what we're really saying is that certain partitions are going to describe different types of entities within your application. I might have user partitions. I might have warehouse partitions. I might have product partitions. I might have sales people on my table. Each one of these partitions will be identified using a unique partition key, and when I query the system, I'm going to use the sort key conditions and the partition key conditions that match the types of access patterns I'm running, right?

I might be interested in getting all the orders for a given user. I'm going to query the user partition with the day range condition that's going to say, "Give me all the orders for a user within the last 30 days." But then I might want to know for a given warehouse which parts are on back order. So I might query the warehouse with a different condition that's saying, "Hey, give me all the parts and filter on a back ordered state for the parts that this warehouse has." This gives me

all the parts for any given warehouse that might be back ordered. There're totally different query conditions querying different types of partitions for different types of data, and that's what we mean by partition overloading.

[00:13:08] JM: Does the database accessor, like the client who's accessing the database, do they need to know anything about these sort keys or partition keys, or is this stuff that's taken care of somehow under the covers by the database itself?

[00:13:25] RH: You kind of need to understand what data you're storing on the system, right? When I define items, I'm going to define items that have a partition key and a sort key and I'm going to typically use generic names for those attributes, things like PK and SK. Then every item I insert on the table is going to have an attribute called PK. It's going to have an attribute called SK. The values that I insert there are going to dictate what type of item that I'm really storing in the system.

When I query the system, I'm going to say the PK equals X. Whatever X is defines the partition that I'm querying. Then the sort conditions I'm using will be specific to the access pattern. If I'm querying a user partition, then I'm going to use sort conditions that are going to retrieve items from that user partition. If I'm querying a different type of partition, I might use totally different sort conditions, and that is definitely up to the client, the accessor, the user. These are going to define the access patterns. They're going to define the data that's being stored and they're going to define the conditions that we're going to use when we query the data.

This is actually a process we typically go through when we design a table. We're going to define your entity model. That entity model is going to really define what types of partitions leave on your table. Then we're going to look at and identify all the various access patterns that we're going to run against that entity model. Then we can go and we can start to actually model the data, define the partitions, define the item types. When we're done, we're going to produce a list of access patterns, a list of indexes or tables that need to be queried for each individual pattern and a list of sort key conditions and filter conditions that will be applied to produce the results that we're looking for. That's like a general process.

[00:15:06] JM: I want to make sure I understand the partition term and access pattern correctly. Let's say I have a database and a ton of orders, and these orders have lots and lots of fields. They have the name of the customer, the state that it's being shipped to, the city that it's being shipped to, the color of the box that's being shipped in. I don't know.

[00:15:31] RH: True. All kinds of different descriptions.

[00:15:34] JM: All kinds of different fields. Let's say I pick the partition key of state, like the state, is it Texas? Is it California? Is it Michigan. If I pick that partition key of state, that means that the database is going to be partitioned. It's going to be split up physically in terms of those different states so that – I might do that because I might have lookups where I'm frequently perhaps getting all of the orders from a specific state. So then my database is pre-optimized. It's going to be laid out to be able to serve that kind of query.

[00:16:11] RH: Sure. Absolutely. That's a valid – It depends on the nature of the aggregation or the grouping of the data that we're trying to produce. If we're trying to group data by state or by zip code or by city, absolutely those are valid partition keys. We get into the mechanics of actually creating those aggregations. We need to understand that NoSQL databases, typically, when I create a partition or a shard, what I'm doing is I'm saying, "Okay. Here's a group of data that lives in one place." That one place is typically one server. That one server has limited throughput.

If the workload that we're trying to drive into that aggregation is larger than what one storage node can handle, then we need to logically split the key space so to speak. You create like state zero and state one and then we can split that workload across multiple physical nodes and so on and so forth. Maybe we have to create for a whole state, aggregating orders by state. That might require maybe 8 or 9 storage nodes to participate because of the rate of orders that would be received by the system. These are the game we'll play to try and get that workload to spread out horizontally, and that's really the key to NoSQL, right? You don't want to pin too much work to any given storage node. We need to get that work spread out across multiple storage nodes in order to scale.

[00:17:37] JM: Just a little more terminology and then we'll get to the fun stuff. Another term is secondary indexing. What is secondary indexing? Why is it useful?

[00:17:45] RH: Sure. When we create partitions on the table, we're going to define the types of entities in a system. We're going to load objects or items into those partitions that are related to those entities, and those are going to serve some sort of primary access patterns so to speak. When I query this partition with these conditions, I'm going to get certain set of items.

Now I might have the need to create totally different aggregations of those items, and the example here might be if I have customers and customers need to access their order data, then my primary access patterns might be querying the table by customer ID using day range conditions to produce orders in the last 30 days or whatever.

If I have those same orders, I might have an accounting workflow that executes on a monthly basis and says, "I need to get all the sales reps orders for the month." Every one of the orders a customer places has a sales rep ID associated to it. Now if I create a secondary index, I'll index on the sales rep ID and that will be a partition key on the index. If I use the sales rep ID as the partition key for the GSI, now when I query the secondary index, the global secondary index or GSI using the sales rep ID and a day range condition, I'll use data as the sort key again. I can say, "Give me all the orders for this sales rep for the last 30 days."

What I've done now is I have one grouping of orders on the table that's grouped by customer, and that's the way we write the data into the system. If I create a secondary index on the sales rep's ID, then I can query that index and get the orders by a sales rep for my accounting workflow at the end of the month. You can think of indexes as a way to create additional partitions, to regroup the data, to support additional access patterns, and this is an interesting thing about NoSQL, is people say, "We don't join data in NoSQL."

But if you think what I just described, that's really what I'm doing. I use the secondary index to join the data across partitions to produce a different aggregation or a different result set. So it's more of a modeled join than it is an ad hoc join, and that's really the difference, right? Relational databases have ad hoc queries. I could join data arbitrarily. NoSQL databases that I can join data, but I have to do it on the index. I have to do it in a way that's kind of I have to think about it

beforehand and structure these storage containers in order to produce the aggregations that I'm looking for, but it's generally the same process.

[00:20:08] JM: Do these secondary indexes, do they copy the data that already exists or are we just making lookups that are going to be pointing to a data that we already have?

[00:20:21] RH: Yes. That's up to the user. We can choose when we define the index to project all of the data from the original items, to project just some of the data, or to just project the attribute keys that we want to index, right? It's up to you when you define the access pattern. You could say, "It's a very infrequent pattern. Maybe I just want to know the items that match. I don't expect those result sets to be really large." So we'll create an index that's keys only that's going to be very lightweight from a storage perspective. You're not going to have to pay a lot for that.

Then when you get the items that match, you'd go back to the table and do maybe a batch get item and get the data that you need from those items. You might say, "Well, no. My access pattern is a little different. I actually need all that data when I access off the index, and I do this with high frequency and high velocity, and I want low latency. So let's project the items completely on to the index that's going to double my write capacity cost, that's going to double my storage cost, but it's going to optimize my read."

This is the question. When you create the index, do I optimize for the read? Do I optimize for the write? That decision is informed by the nature of the access pattern, right? What's the velocity of the data? What's the shape of the data? How much does it cost me to optimize for that read? Is that a cost I want to bear or would I prefer to optimize for the write and pay less? That's a decision the user makes where they analyze the access pattern.

[00:21:45] JM: In order to scale, databases need to be sharded often times. In DynamoDB, does the database designer have to figure out how to do this sharing, or is sharing taken care off by the database implementation?

[00:22:02] RH: In all NoSQL databases, there's a condition that we call hot keys, and hot keys really are about how much throughput am I asking to give in logical key to provide? In

DynamoDB, we use relatively small storage nodes, and the reason we do this is so that we can scale the system quickly. Legacy technologies like MongoDB and Cassandra, they'll use very large storage nodes because they're essentially an entire server.

The downside using large storage nodes is that the scale becomes expensive because the more shards I have, the longer it takes to add a new shard. The reason why is because I have to actually replicate some data on to the new shard before it becomes available. When my system becomes large, that replication process can literally take weeks and even months. I'm working with some of the largest legacy technology deployments from MongoDB and Cassandra in the world today, and the largest cluster I'm aware of is 45 shards for MongoDB. They started replicating shard 46 in July. They expect it to be done in February sometime. This is not an uncommon experience.

With MongoDB, or with DynamoDB, we use very small storage partitions because we want to be able to scale quickly from zero to a million request per second, but this comes in a cost. It comes with a throughput. There's an issue with throughput with DynamoDB partitions, right? You have to be aware of this. Write sharding in DynamoDB is something that the user does need to become aware for scaled out workloads.

But generally speaking if we select the right partition key, that's not something that the user has to be aware of, because that partitioning occurs under the covers for NoSQL databases. As long as I'm not trying to pin too much workload to a single logical key, then we can go ahead and get the system take care of that. I hope that's a good answer.

[SPONSOR MESSAGE]

[00:24:07] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[00:25:55] JM: Now, I think we've given people the proper building blocks mentally to understand why databases get tuned in different ways, because a database is, at a high-level, just a ton of data, and we simply don't have the tools today, the futuristic tools to be able to just say, "Look, we've got a big lump of data and we've got a magical API that's going to figure out the best way to always query that data." This magical API is going to turn your data into the correct database and it's going to give you all the correct ways to query it and there's going to be minimal cost and everything. This is just such a huge optimization problem that we as an industry are going to figure out overtime with new databases, new things that are built on top of those databases.

The state of database design and choice today, a lot of it depends on the read and the write paths of your application. If you have an application that's writing time series data at a very, very high-volume, but that data is very rarely accessed, then you have a certain kind of application that you may have a certain kind of database need for.

On the contrary, if you need a database that suits a ridesharing application, maybe you need a different kind of application, and the point is that different clients are reading and writing to that database in certain ways, and the patterns of the reads and the writes are often – There's also the shape of the data. That's a third dimension that we can explore here.

Are there some certain read and write patterns that you can illustrate to help us understand how we should be making these choices that we've outlined earlier, things like sort key and partition key?

[00:27:53] RH: Yes and no. I mean, there are general guidelines. I don't know if there's any kind of general guidelines that we can deliver, but there are certainly some decisions that the developer needs to make when they're designing their schema, because there are tradeoffs in cost and performance, right?

You mentioned the velocity of the data, the shape of the data, really, the frequency of the queries. I can use an example to talk to this. I had a customer who is maintaining a secondary index so that they can find exception conditions on their table and the exception condition that they were looking for they need to check for once a week. They had about 1-1/2 terabytes of data on the table. They created a nice, efficient global secondary index. They could query the global secondary index for the exception condition and retrieve the two or three items every week that we're actually in that exception condition.

The problem with this implementation was that they were projecting all of the data on to the index which duplicated their storage cost and it duplicated their write capacity cost. They have barely sizable items. The items were around 12 kilobytes. In DynamoDB, that's 12 WCUs per item write. I think their write capacity on the system was about 150,000 steady state, 150,000 WCUs. It's a pretty expensive table, and they were essentially doubling their cost by writing those items to the index.

The first optimization they went through was to say, "Well, since there's only two or three items that we need," they just went to a keys-only index and they stopped projecting all the extended data, and that was good. That dropped their write capacity by about 70% or 80% requirement, but they were still running at about 15,000 to 20,000 WCUs fairly steady state.

The cost I think was somewhere around $2,000 or $3,000 a week. I basically said, “Look, why don’t we just once a week we table scan this table. We’ll put the WCUs on your table up to or RCUs up to a million RCUs? We’ll spin up a couple of EC2 instances. We’ll table scan this thing. It will take us about 20 to 30 minutes. We’re done. You’ll pay – I don’t know, $100 total cost during this 20 or 30-minute window while we’re table scanning, and it will drop your weekly cost by thousands.”

I mean, long story short, it’s not so much always that we want to most efficient way to read the data. That end solution was the grossly inefficient read, right? Table scan the entire table to find two or three items, but we’re only doing it once a week and the cost of maintaining the index was in the thousands a week. The net tradeoff to the customer was huge savings.

This is what we’re really going to do when we start to look at how do we choose to structure the data. We’re going to make these types of decisions based on the frequency, the velocity, the shape of the data that we’re trying to store and whether or not we want to optimize the read or the write. It’s really going to be so access patterns specific that it’s hard to say what’s the general guideline. Long answer, but that’s kind of where it has to go.

[00:31:12] JM: That’s a great answer. We have a lot of episodes about exploring the query patterns of transactional processing versus analytic processing. Transactional processing being like a write or a read to a single record or some specific small collection of records perhaps, or just I guess more accurately just a write to the production database in many cases, although even that’s kind of a crude way of putting it.

Then OLAP being these analytic processing systems where you have really high volumes of data. You’re off doing aggregations, like maybe you’re summing the profits that you’ve made from all the orders across the entirety of a gigantic ecommerce website.

Are NoSQL databases specifically better suited to OLTP applications or is there a way to use NoSQL databases as OLAP applications?

[00:32:09] RH: Okay. Yeah. I mean, that's a great question. Generally speaking, NoSQL strong suit is the OLTP app. It's a well-defined access pattern. It's a repeatable predictable way to access the data. That's what you're going to expect from an OLTP type application. That's what I'd say. That's how I would describe OLTP, right? It's a repeated process. It happens the same way every time. It's like the shopping cart at amazon.com, right?

When I process my order, it's the same process that runs when you process yours. It's no different than anyone around the world processes their shopping cart every time they hit the button. The same thing happens. That's OLTP. That is a sweet spot for NoSQL.

Now the other side is the OLAP application, and I would really qualify that into two buckets. You've got kind of operational analytics, which is almost a repeatable process. It's something where I'm looking for summary, KPIs, key metrics that are driving the information back into my system. Things like current request rates, current order rates, total sums, counts, downloads, all these types of things that we try to generate as a kind of report metrics.

For those times and workloads, NoSQL can be a very interesting choice. As long as you have a mechanism to do change data capture processing, right? As I insert items on the table, as I update items on the table, I want those writes to appear on some sort of a change data capture pipeline where they can be processed and I can actually write back summary information to top level metrics into items that I reinsert back on to the table. In this way, we can offload the pressure of the compute of having to drive those things like counts, sums, averages, top in, last in. Most of max-min. Most of the type kind of KPIs that people are trying to calculate can be done in this CDC pipeline.

This is actually something we found at Amazon that we started doing very early on before we even chose to migrate off of our Oracle systems to NoSQL. We are creating rollup tables in our Oracle databases to hold these KPIs because the cost of doing those queries in real-time, the compute cost was just too high right.

We would maintain rollup tables for sums, averages, like downloads per song or whatnot for your Amazon music. Don't calculate that in real-time. It was too hard and too expensive. So once we did that, we're kind of starting to realize these are kind of eventually consisting KPIs,

which is fine and actually works really well with time series data, because if you think about it, time series data loads into a time bound partition and once it's loaded it doesn't change. If I calculate the total sums, averages and whatnot, I'm looking to write that back as a summary item. It's a lot cheaper to select that item than it is to compute that value every time somebody needs it, and the reality is the data hasn't changed. So it works really well.

Now, the other type – Again, long answer. But the other type of analytics is that ad hoc analytics, right? Kind of think like a market analyst is coming in and who knows what happened today, and he summoned data on different dimensions. He's grouping on different aggregations every day. Those types of applications has definitely the realm of the relational database. Again, predictable access patterns, predictable analytics.

I would say operational analytics, things that allow us to kind of compute using a CDC pipeline. Those are great use cases for NoSQL. The third category of kind of the ad hoc analytics, I guess you'd say ad hoc OLAP, that's the relational database.

[00:35:50] JM: Since you are such a diehard NoSQL advocate and you have just made an advocacy for SQL, we have to go a little bit deeper there. When you're talking about an application where SQL actually is relevant, are you talking about a situation where you would need to use a data warehousing system or are you saying that there is a conceivable scenario where you would advocate somebody using a relational database like MySQL or Aurora?

[00:36:20] RH: Absolutely. I'd go MySQL or Aurora. For example, if you were to walk-in, if I was considering application to migrate to NoSQL, first thing I would do is look at the server logs. What are those queries look like? Do I see a log that says, "Hey, I run these 15 queries 20 million times a day." That's probably a really good candidate for NoSQL. That workload might be considered an OLTP workload. It might be considered an operational analytics workload. But that's what the query log is going to look like. It's going to be a same set of queries running all the time.

If I walk in there and I look at the query log and over the last week who the heck knows what's happening. Every day is different, right? Then that's probably something I'd want to consider a relational database for.

Now, you mentioned data warehousing. That really starts to speak to what is the nature of the workload. What is the scale of the data? When you start to kin of pick a database for a particular workload, you really have three dimensions. You have what's the need for pattern flexibility? That's really ad hoc queries. Do I understand my patterns, or are they all over the map? What's the need for scale? Do I need to have infinite scale of this system for all practical purposes and throughput and storage?

The last definition, the last dimension is really efficiency. How fast do I need these results to come back? That's kind of like an triangle of data. I call it the pie theorem, right? You can pick two. If I need pattern flexibility and infinite scale, then I'm really looking into your data warehouse, your decision support system. This could be like a Red Shift. This might be Amazon's Athena. But this is where I don't expect queries to come back quick and I need to run these ad hoc queries on large datasets.

If I'm looking for infinite scale and efficiency, now I'm talking about NoSQL. I'm talking about DynamoBM. I'm talking about predictable access patterns. If I need that pattern flexibility, that ad hoc query engine and I need efficiency, then I'm really not looking to scale, then that's your relational database. That's what we really got to understand here, is that relational database is going to operate very rather efficiently when I need ad hoc queries, but only at a limited scale. So this is where our tradeoffs get into – If we start to scale beyond with the relational database can handle, then we're looking at – We better be looking at a different type workload, which is that data warehouse to that decision support system. That's going to run better on a Red Shift or an Athena.

[00:38:51] JM: It sound like to boil down what you're saying, if you know the query pattern of your application, NoSQL is a good candidate. If you have such a diverse range of different queries that people are running, you're not going to be able to pick partition keys and sort keys and secondary indexes that are actually going to make sense. You're just going to – If you try to do that, you would have so much bloated database infrastructure that would just not make any sense.

So, you'd be spending way too much money. You'd be spending way too much time maintaining this thing. It'd better off with just the simple – Maybe you could go through some normalization process and get your data in a SQL database.

[00:39:32] RH: Correct. Absolutely. That's really what it comes down to. If I can predict the access pattern, let's go NoSQL. If I can't, then let's try and get this thing in a relational footprint somehow.

[00:39:41] JM: Yeah. I was watching a talk you gave and something that I guess I didn't realize was that if you do know these access patterns, NoSQL can actually be more efficient to query than a SQL database. Is that accurate? The query efficiency is actually going to be better?

[00:40:02] RH: Oh, yeah. No. Absolutely. I mean, if you think of a NoSQL database, what is a NoSQL databases? It's a distributed hash table, right? This is what we're doing. We're taking that partition key. We're creating [inaudible 00:40:11] hash index. When I need that thing, what am I doing? I'm querying the hash index for that particular key value. That's a time constant function, right? I mean, querying a hash table.

As a matter of fact, if you look at the relational database, if the relational database has no indexes and –So let's imagine a worst-case scenario, right? We've got a query, we're joining two tables in a relational database. There is no indexes on the sort dimensions or on the join dimension. I'm going to have to go ahead and have a large results to that set from the outer table. I've got a large number of rows on the inner table. Worst-case scenario, what does database do? It creates a hash table. It basically does a sequential scan of the inner table and creates a large hash table on the join dimension. Then it iterates through the results sent from the outer table and it queries the hash table for each result to get the join. This is what it does.

If the relational database determines that the most efficient data structure to query when all hope is lost and I have no indexes, then that most efficient data structure is a hash table, then it seems like storing your data in a distributed hash table would be pretty ideal, right? The reason why it does that is because it's a time constant function to query a hash table.

If you look at indexing, let's say it's an index query on a nested loop join in a relational database, that's an optimal scenario. Small results sent from the outer table. I'm indexed on the join dimension. I'm going to go ahead and basically do – What is it? It's an n log n time complexity. That's the best I'm going to do in a relational join. I'll take time constant over analog at any time as far as big O.

[00:41:54] JM: I see you as an evangelist for NoSQL, and I don't know much about you other than what I've seen from your talks. I can imagine you have gotten this way from too many conversations with people who are saying, "Wait. You're a full-time NoSQL person? NoSQL that like non-web scale technology for like hackers and stuff? Why would you spend your career doing that?"

I guess you can tell me whether or not that personality assessment is correct, but I'm also just curious. Is this like a new discovery that we can actually make these NoSQL databases perform well or have people just been confused the whole time?

[00:42:36] RH: No. People have been confused for a long time. I mean, including myself. As matter of fact, I apologize to customers that I talked to ten years ago about SQL, because I told people over the years to do some pretty crazy things.

What happened really in my evolution was about four years ago or so, five years ago, I was tasked with the migration of Amazon's relational service infrastructure over to NoSQL. So we had – What was it? I think we had it somewhere around 350 what we call tier 1 services. These are revenue makers group for Amazon. If any one of those goes down, we're losing money. All of those had to go over to NoSQL. That was a mandate from the business. We had 12,000 tier 2 services, which basically run various aspects of everything.

When these go down, we're not necessarily losing money immediately, but some of these things are pretty critical. Most of those had to go as well, and my team was – They formed this team called the Black Belt Team, and my team. I headed up that team and it was my job to make all that stuff work. We went through the first year of this process. The team thought they were doing a pretty good job. Then we started to move some of the bigger workloads across and we realized that it's actually really easy to do things incorrectly at small scale and not know. You

could really mess things up badly. As long as you're not scaling it out, you won't know how bad it is.

Actually, I believe today strongly that the vast majority of NoSQL deployments out there are exactly that. They think they're doing it right. They think they're having good results, but they just haven't scaled yet. Once they do scale, they're going to find out how badly their modeled. But there is a way to make it work, and we had to do that. We had to go through the pain. We had to figure all these things out, and we actually had to figure out that all data is relational. There's no such thing as non-relational databases. Anyone who says that he really doesn't understand the way the database works – There's not any data I'm aware of that's non-relational. Everything has relationships, and we need to be able to model relational data in a NoSQL database. We just need to do it in a de-normalized manner, and you have to do it in a way that makes sense for the application in order to make it work.

This is why I talk a lot about design patterns, best practices, when you're actually working with your data in order to be able to optimize the model and drive the efficiency that we want from the system, because if you're just putting everything into a big giant blob and creating large documents, the chances are you're actually spending more with NoSQL that you would be on the relational database. I think that's where a lot of this opinion comes from, from people who've done it incorrectly. Burn their fingers and they go back to the relational database instead of actually figuring out a use the technology correctly.

[SPONSOR MESSAGE]

[00:45:29] JM: Being on-call is hard, but having the right tools for the job can make it easier. When you wake up in the middle of the night to troubleshoot the database, you should be able to have the database monitoring information right in front of you. When you're out to dinner and your phone buzzes because your entire application is down, you should be able to easily find out who pushed code most recently so that you can contact them and find out how to troubleshoot the issue.

VictorOps is a collaborative incident response tool. VictorOps brings your monitoring data and your collaboration tools into one place so that you can fix issues more quickly and reduce the

pain of on-call. Go to victorops.com/sedaily and get a free t-shirt when you try out VictorOps. It's not just any t-shirt. It's an on-call shirt. When you're on-call, your tool should make the experience as good as possible, and these tools include a comfortable t-shirt. If you visit victorops.com/sedaily and try out VictorOps, you can get that comfortable t-shirt.

VictorOps integrates with all of your services; Slack, Splunk, CloudWatch, DataDog, New Relic, and overtime, VictorOps improves and delivers more value to you through machine learning. If you want to hear about VictorOps works, you can listen to our episode with Chris Riley. VictorOps is a collaborative incident response tool, and you could learn more about it as well as get a free t-shirt when you check it out at victorops.com/sedaily.

Thanks for listening and thanks to VictorOps for being a sponsor.

[INTERVIEW CONTINUED]

[00:47:19] JM: I mean, there's a few reasons why there is so much confusion around this database, like this questioning of the viability of NoSQL at a fundamental level. There're a lot of reasons for that. I mean, databases are just hard and they're complicated and most people don't ever spend the time to go deep on them, because why would you? You're just using it for your application. Who cares?

But I do wonder, are any of the things – You've talked about this, kind of this conversion of yourself from a doubter of NoSQL or questionnaire of NoSQL to somebody who really sees the value in it. Does this mirror how the DynamoDB team has evolved or was DynamoDB architected in a way from day one that took all this intelligence into account? Was this just a small quiet team that knew exactly how to make NoSQL databases work?

[00:48:13] RH: That would be great. No. No. As a matter of fact, I think that – Again, the first use case that DynamDB was really built for and it did a really good job was a key value access pattern, right? Give me this partition key. Give me all the data that's associated with this given partition key, and that is a valid use case for NoSQL. That is the one where most people are succeeding with their NoSQL deployments.

Think like a session data, user session data for a web application. NoSQL database makes a really good and easy data store for that, because you have some session ID. You've got a blob of data associated with that session ID. Perfect key value access pattern. I always need all the session data whenever the user logs in. It's no problem.

But when you get into the more subtle aspects of NoSQL and the way that we work with the data, what you start to realize is that these entities might consist of data that lives across in a relational database many tables. But when I joined the data together across these tables, I'm not necessarily pulling the data from every table that has something that's related to a given user or a given entity. That would be extremely expensive.

So if I push all that data into a big giant document that describes this giant entity, then I have the same inefficiency with my NoSQL system. Essentially, every time I access that blob, I'm getting all of the data that's reading everything, yet most of the time when I access data that's associated to something or related to something, what I really want is that one row. The one row from that one table that maybe has just a couple of fields on it and I don't want to have to read a 12 megabyte document so that I can go get that data and then write that 12 megabyte document back to the disk when I needed to update a single image or within that document. That's what happens with most people who are using NoSQL today, which is what drives their inefficiency. This is where you need to start to realize how do I model this relational data that lives across multiple objects and so on and so forth. It annoys the old database.

[00:50:14] JM: Today, if I understand your work correctly, you help people restructure their databases or move from their current data systems to a NoSQL or improve their NoSQL schema stuff. You're kind of helping companies that have an existing installation work more efficiently. Is that right?

[00:50:35] RH: Yeah, generally. I'll help them migrate from their relational databases. A lot of companies are looking at the same problems that Amazon had, cost of relational databases, extreme at scale, painful to manage. They're looking to get into NoSQL databases, especially cloud native NoSQL where they don't have to manage the infrastructure. I do a lot of work with our more strategic customers to help get them off of the relational databases and move them into NoSQL or build new applications on NoSQL or do exactly what you just said, is help

customers that are having pain with existing NoSQL and get them running well. Not only for DynamoDB, but I work across stacks, MongoDB, Cassandra, you name it.

[00:51:15] JM: Presumably, someone with your knowledge could work on database design. You could go work on the Aurora team. You could go work on the Dynamo team. What is it that makes you like to do what you're doing as supposed to database design?

[00:51:30] RH: I do that actually. I report directly to the general manager of Dynamo DB. I'm considered a global strategic asset. So they moved me on to strategic opportunities. I tend to move the needle pretty quick in my engagements. A lot of that is I'm helped by the fact that the – This this isn't black magic, right? It's not like it takes a rocket – You don't have to [inaudible 00:51:49] rocket science to understand how to do this modeling. It's really cool. I love it, because when I talk to the developers, you'll see this – I call it the light bulb moment, right? It's like when all of a sudden, "Oh, that's how it works."

Then it's just like a normalized data model. You weren't born understanding a parent-child relationship or many-to-many relationship or how to create that join table. But once it was explained to you, it was kind of like, "Oh! Yeah, of course. That works. I see how that works." It's the same thing with NoSQL. Once you kind of get it, it's like, "Oh! You don't really need me anymore."

A lot of my engagements with these accounts are typically I'll fly in for a day. I'll do a two or three hour design pattern session. I will do a maybe a day or two, a half day or a full day of design reviews with their teams. I get them going on their data models. I'll see a couple of guys hit that light bulb moment. Once I see that, I know I'm never going to have to go back. They'll typically ask me this, "How can we –Once we get our model, can we call you back and see if it's good?"

I'd say about half the time they give me a call and I might help them optimize somewhat. But generally speaking, these guys, they take it and run with it once they get it. It's just like developers are developers, right? Once they understand the concepts, they don't need to be handheld.

[00:53:07] JM: It sounds pretty rewarding.

[00:53:08] RH: Yeah, it's great. Of course, your question is why don't I do database design? I get to help the team prioritize features, define features. The NoSQL workbench for DynamoDB was a tool that I helped bring to market, and there is several features within DynamoDB that I work on as well from a product management perspective. I get the best of both worlds. I have my foot on both sides of the fence.

[00:53:33] JM: That's awesome. To address another technical subject, technical pattern, you alluded to this a little bit earlier I think with the discussion of change data capture. Every transaction at a DynamoDB instance is written to a DynamoDB stream. What are some patterns for taking advantage of that event stream?

[00:53:53] RH: Okay. Yeah, sure. Yeah, I use that a lot to do these summary metrics, summary rollups, the kind of things that you call I guess operational analytics. Maybe I need to know a top in, a last in or max-minimum, or account sums, averages, all kinds of different calculated metrics so to speak. So as data changes on the table, what I'm going to do is I'm going to define a lambda function, and one of the nice things about DynamoDB is that there're a lot of guaranteed contracts here. There is a guaranteed contract between the table and the stream that every write that happens on the table will appear on the DynamoDB stream. It's going to get cashed for 24 hours.

Then if you choose, you can use Lambda to process the stream. When you do that, you get a guaranteed contract between Lambda and the stream and that it will fire on each and every item at least once.

Generally speaking, it will fire only once, but there is a rare condition in which maybe the container which the Lambda was executing went offline or crashed and might it have to re-fire. In that case, we might actually process an item twice. When you code those Lambda functions, code them to be item potent. But as long as you do that, then you're guaranteed the processing from the table to the stream, from the stream to the Lambda, to wherever it needs to go.

Now as far as what are the patterns people use, they'll write that data back into those top-level metrics and create these summary items that people can then select using a time constant query as supposed to calculate all that data by reading it and trying to calculate it. The other thing they'll do is replicate that data into other systems. So a couple of things DynamoDB doesn't support. We don't support geospatial indexes and we don't support full text indexes. I have a lot of customers that will sync that data into Elasticsearch when they need to run those types of queries and then they'll query the Elasticsearch index for geospatial, for example, and then query those items off the DynamoDB table using batch get items. That's a valid use case for the stream.

Other often things people do with it, they'll role that data up into S3, create so parquet files. They can run Athena queries on that if they have ad hoc analytics that they need for daily or scheduled reports. Think of it just kind of, again, change data capture. Anytime you need to change the date on the system, maybe a trigger event, right?

There could – If anything on the table ever changes state from X to Y, I want to run this process of notify as somebody or something like that. It's really a neat kind of – I guess you'd say stored procedure engine so to speak for DynamoDB that scales completely independently of the table, which is another nice aspect of the system, because it's fully distributed. We don't have to worry about stored procedures that are running in your Lambda functions impacting the availability of the data on your table. These two things are completely disconnected and they scale independently of each other, which is a really nice system to give you that kind of processing pipeline.

[00:56:47] JM: We focused most the conversation on NoSQL a little bit on SQL and a little bit on time series, a little bit on data warehousing. We haven't really touched on the increased popularity of these distributed queuing systems.

So the Kafka style, Kinesis style, append only, distributed log systems. This became quite a powerhouse of a trend in data management, and I guess – I'm wondering, well, kind of two questions that I'll just kind of bundle into one because I know we need to wrap up. But do you have any perspective for how that trend is going to change data infrastructure overall? As we are writing all of these data that kind of looks like a change data capture, but we're persisting it

as a big distributed database and any other reflections on the broader data platform vision and kind of predictions for the future as a whole.

[00:57:50] RH: Sure. It's interesting you talked about it. So you're really talking about kind of stream data processing, and I do see that becoming much more of a, I guess, front and center component of the modern application stack so to speak. It's what drives machine learning, right?

I mean, we called this stuff back in the day complex event processing and pattern recognition, and this is kind of out on steroids, being able to take the data, shove it into Kinesis, fire hose, roll it up into parquet files. Drop into S3. Run Athena queries over the top of it. Execute stream processing, stream aggregations, time bound queries of the event stream so to speak. Find all the events in this time window that match these conditions and create these reports and aggregations off of it. These are interesting types of workflows. They kind of touch a little bit on NoSQL. I guess a backend store some of the data, but I will see that this is like the leading edge of the input into what really will eventually be the AIML engines, right?

I kind of look at AIML today and look at what they're doing. I say it's not so much different than what we were doing 10 or 15 years ago. We just call it different things, right? Honestly, there's not a lot of difference here. But as far as future of database technology, I really see the – Amazon is pushing a purpose built solution, portfolio solutions.

I think that we'll start to – I totally agree with that philosophy, but I believe that those solutions are going to be – It's not so much that we want to use different types of databases for a single service or a single app. I want to build services that are backed generally speaking by a single database. But different services have different requirements, and different databases are going to have different features that are going to satisfy those requirements.

I think NoSQL databases are generally speaking going to evolve into the cloud. I think legacy technology providers like MongoDB and Cassandra, these technologies were built to deploy on-prem. They're going to need to evolve those solutions. MongoDB's Atlas is really a solution that's built to host MongoDB instances, and that model just does not translate very well into cloud deployments, right?

I kind of put that into the same boxes of what we call lift and shift cloud deployments, where you take a legacy enterprise data center, you re-create the exact same thing in a virtual space in the cloud. You deploy a bunch of instances into it. The guys login. It looks just like it did when they were deployed at the Equinox Data Center in Virginia, right? But that's not the best way to use the cloud, right? You're not going to gain the efficiency of cloud deployments until you embrace cloud native technologies that give you the elasticity and the cost efficiency of massively distributed shared backplane services like DynamoDB. We cannot scale elastically if I'm using dedicated server instances.

So I really look for solutions coming from MongoDB, Cassandra as NoSQL providers. If they want to live in the next generation of NoSQL, they need to develop cloud native solutions that can compete with the DynamoDBs of the world. We're kind of doing that to a certain extent, right? We launched managed Cassandra service with Amazon just recently at Reinvent. That is a Cassandra native scale out. If you think about how we built like Aurora Postgres or Aurora MySQL, we've done the same thing now for NoSQL. We have the DynamoDB storage engine and we have basically a full new stack that's sharing components of the technology and delivering cloud native scale for Cassandra using the native Cassandra distribution as a frontend, and it's really powerful.

So this is what we're going to start to see if these solution providers want to survive. Now, we're helping Cassandra with this, because it's fully open source. We can't help MongoDB, because they have a close source license and they hold on to their code. So if they want to start working with us on this, maybe we can do it, but they need to embrace the cloud native architecture if they want to succeed.

[01:02:12] JM: I would love to see such an armistice. I have advocated for one, but that's a subject for another show. Rick, thank you so much for coming on the show. It's been really fun talking to you.

[01:02:22] RH: No problem. Thank you very much for having me. I really appreciate it. Thanks so much.

[END OF INTERVIEW]

[01:02:35] JM: Embedded analytics is the way to add dashboards to your application. Are the dashboards that are in your application engaging your end-users or are they falling flat? According to analyst firm, Gartner, the UX of embedded analytics has a direct impact on how end-users perceive your application. Fortunately, you don't have to be a UIUX designer to build impressive dashboards and reports. Logi Analytics has come up with six steps that will transform the user experience of your embedded analytics.

Logi Analytics is the leading development platform for embedded dashboards and reports, and unlike other solutions, Logi gives you complete control to create your own unique analytics experience. Visit logianalytics.com/sedaily to access six basic principles that will transform your dashboards. That's logianalytics.com/sedaily. That L-O-G-Ianalytics.com/sedaily.

[END]