EPISODE 1094

[INTRODUCTION]

[00:00:00] JM: DynamoDB is a managed NoSQL database service from AWS. It's widely used as a transactional database to fulfill keyvalue and wide column data models. In a previous show with Rick Houlihan, we explored how to build a data model and optimize the query patterns for a NoSQL database. Today's show is about DynamoDB specifically; partitioning, indexing, query semantics, normalization, table design, and other subjects. We talked through how to be cost-conscious and how to integrate with event-based AWS lambda triggers.

Alex DeBrie is the author of the *DynamoDB Book*, a book whose title speaks for itself. Alex has a comprehensive experience with DynamoDB and he joins the show to share that experience through a detailed discussion of use cases and strategies related to DynamoDB.

If you are interested in sponsoring Software Engineering Daily, send me an email, jeff@softwareengineeringdaily.com. The show reaches 30,000 engineers each day, and if you are interested in reaching those engineers as well, we'd love to hear from you. You can also become a paid subscriber to the show by going to software daily.com and clicking on subscribe. That would help us support the show and it would mean a lot to us. Thank you.

[SPONSOR MESSAGE]

[00:01:24] JM: If you haven't tried Airtable, one way to think about it is like AWS with a visual component. And I say that because it's a very new way of creating software. Just like when AWS came out, it was quite a new way of creating software. Airtable is a low code platform with the ability to become code-heavy if you want it to.

Airtable has gone through years of development as a modern approach to a spreadsheet. And beyond being a spreadsheet, it's a modern approach to a database backend. With that effort gone through, Airtable introduced blocks, which is a system for building rich application components that sit on top of your Airtable backend providing a user interface or a map or whatever you want out of your Airtable block.

Airtable is now releasing custom blocks so you can build your own Airtable components. And if this doesn’t sound exciting, that’s probably Because you haven’t tried Airtable. You can make Airtable custom blocks out of JavaScript and React components and they work through the Airtable SDK to enable developers to make their own functionality on top of Airtable. You can make a modern authenticated real-time CRUD app for users, their admins or for yourself. And if you’re looking for the right time to get started, that time is now. You can enter the Airtable custom blocks hackathon at airtable.devpost.com and get started. It’s a great hackathon. It’s $100,000 in cash prizes, and you can start by building a database, then move up the stack to build a fully-fledged custom block using the Airtable SDK.

You can really build anything with Airtable, and you can compete to win $100,000 in cash prizes. So it’s quite a good time. So just go to airtable.devpost.com, try out a new way to build software with Airtable. Thank you to Airtable for being a sponsor of the show.

[INTERVIEW]

[00:03:32] JM: Alex DeBrie, welcome to show.

[00:03:33] AD: Thanks, Jeff. Thanks for having me.

[00:03:34] JM: You have written a book on DynamoDB, and it’s really comprehensive. How did you get started working on DynamoDB?

[00:03:43] AD: Yeah, good question. I would say my first experience with Dynamo was probably four years ago and I just wanted to use it for a very simple internal application at the company I was using for a time. It was like Slack/command for internal company usage and I just need some quick data storage. I use Dynamo there pretty incorrectly, but use it there. But then I moved on to Serverless Inc. that that's the creators of the serverless framework, and DynamoDB is really big in the serverless world for a number of reasons. I'd say that's why it really ramped up my Dynamo usage.

I used it pretty incorrectly for quite a while, and it wasn't until – Let’s see. It would have been Christmas, 2-1/2 years ago. I think like 2017, the end of 2017. I listened to Rick Houlihan’s Reinvent talk from that Reinvent and it just sort of blew my mind on how Dynamo should be used and compared to how I was using it. That really sort of lit my fire, and then that Christmas break, I ended up watching that talk a bunch of times and then creating a resource called dynamodbguide.com, which is a free resource out there.

Basically, it was a walk-through to some of the DynamoDB principles. How to use the API? How to think about modeling data? Things like that.

[00:04:48] JM: Rick Houlihan, of course, the famous NoSQL commentator, DynamoDB commentator and presenter. A quote from him from your forward, “There is one underlying principle when it comes to NoSQL. At the core of all no SQL databases, there is a collection of disparate objects tied together by index common attributes and queried with conditional select statements to produce result sets. NoSQL does not join data across tables. It effectively achieves the same result by storing everything in one table and using indexes to group items.”

Explain how DynamoDB works at database despite not having multiple tables.

[00:05:28] AD: Yeah. I think it’s very different for people, especially if you're coming from that relational world. But I think the two biggest things to know about DynamoDB and really most NoSQL is, number one, you’re going to be limited on sort of how you can query it. With DynamoDB, there's this notion of a primary key, which you have to include on every single item. And most of your data access is going to be done on that primary key.

If you're clearing your data, you need to use that primary key. You can’t just sort of select off any column or index to any column like you could do in a relational database. That's the first one. You’re sort of limited in how you can access it. You need to be intentional that way. The second one that Rick sort of mentioned is there's no joins. So you can't sort of combine this data like you do in a relational database where you normalize your data and then at query time you pull it all back together and assemble it as you need it.

Because of that, you need to model your data differently, and what you do is you're basically intentionally designing your data for your access patterns. You're thinking about your access patterns, where instead with a relational databases, you sort of put all your data into its tables. You normalize as you should, and then you think, "Hey, what are my access patterns to actually write the query? Join those or the other? Maybe add some indexes?"

With Dynamo, you think access patterns first before you design your data at all, and you're basically laying it out in a way that matches how you need to access it. If you need to – If you have a one-to-many relationship and you have a parent item and its related items, you need to retrieve all those together like in a join operation. You're actually just going to pre-join those or write those into the same collection so that then you can query those very efficiently and get them so they're altogether at read time rather than having to pull them together from different tables.

[00:06:58] JM: DynamoDB, it's been around for a while. How has the database evolved over the years? You say it's really become a database platform more than just a database.

[00:07:09] AD: Yeah. It's quite different from when it was first launched, and some of the features by the time I got involved, they are already there. But I think even things like secondary indexes weren't there at the beginning. Secondary indexes are ways you can basically set these up on your table to give you additional primary keys as when you write that item to your base table. Amazon will handle sort of replicating it out into this secondary index that allow these additional access patterns for you. I think that made it a lot easier, because you don't have to worry about sort of maintaining all these different copies of your data to handle different access patterns, so secondary indexes is a big one.

But even in more recent years, you have things like transactions where you can operate on multiple items in a single request, and if one of those write operation fails, then the whole operation will be rolled back. That gives you a lot of control, a lot more flexibility than you had before where you had to do sagas or different things like that. So that helps a lot.

Another pretty interesting thing is DynamoDB streams where you can enable this on your DynamoDB table and then you can – Basically, anytime a write operation happens on your table, they're just going to drop a record into that stream. If you're familiar with things like Kafka or Kinesis from Amazon, just drop a record in there and then you can consume that stream as you want and use it to fan it out to different microservices in your application or to put it into an analytics system, things like that. It gives you the sort of reactive string base architecture that you like. Those are the big ones.

They've also really innovated on the billing model, which I think is pretty cool. I mean, I think the original billing model is pretty awesome where you're paying for read and write units directly. So rather than saying, “Hey, I want this much CPU and this much RAM,” and you're trying to guess how that turns into queries and what sort of performance you’re going to get. They just say, “Hey, this is how many read capacity units you want to have and how many write capacity units,” and that's how many read and write operations you get per second. So you don’t have to do that translation. Once you’ve done your sort of capacity planning, you don’t have to translate it into CPU and RAM. You just get the reads and writes that you need.

But then in the last couple years, they’ve also added an on-demand mode where you don't have to provision that upfront. You can just do paper requests just like you do with lambda are API gateway or SNS or any of these other sort of serverless services where you can just say, “Hey, I’ll pay for every request that comes in. I'm basically never going to get throttled,” and you'll pay a little premium for that, but I think it's worth it to avoid that capacity planning and just basically never have to worry about getting throttled.

[00:09:25] JM: Let’s go a little bit higher level. For somebody who's just unsure of how DynamoDB is used, what applications it’s a good use case for. I believe it’s typically OLTP transactional applications. What are the core concepts of DynamoDB? What do I need to know as to whether this is a good fit for my application?

[00:09:50] AD: Yeah. I like to break it into sort of two types of applications where I think it fits really well for. Number one is sort of the very high scale application where you maybe out-scaled a relational database. And this is why DynamoDB was built in the first place. It matches this sort of internal implementation at amazon.com called Dynamo. There's a paper on that. You can go read. Werner Vogels is one of the authors of that. But basically, you have these really high scale applications at Amazon, the shopping cart, the inventory system, things like that. As they sort of looked at how those applications were used, they saw that they're not using some of the more advanced features of a relational database like joins, like complex filtering, things like that. They're basically doing single record lookups, maybe a collection of records, but not sort of advanced stuff. I think the high scale applications where you see it a lot, and it’s used a ton internally at both Amazon and AWS, but also at places like a Lyft and Redfin and different companies like that.

The second place I think DynamoDB has really taken off is in the serverless world, and that’s just because the billing model that I mentioned earlier, that fits pretty well with that serverless mindset. But really, it's more about how the connection model works and how the provisioning model works, and it fits really well in the serverless ecosystem where things like relational databases don't fit as well without lambda compute model where you have this hyper ephemeral compute spinning up all over the place. It just doesn't really work well for something like Postgres and MySQL. So I think a lot of people started picking up DynamoDB for those use cases as well.

I always strongly recommend it for those two use cases, high scale applications and then serverless applications. I think it works really well for basically all OLTP applications. You can use it for any of those even if you're not high scale, even if you're not doing serverless. But I think at that point, it's more a point of personal preference. If you're more comfortable with something like MySQL or Postgres, you can use that, but I think it will fit well there as well.

[00:11:45] JM: The data modeling process for a NoSQL database, it's going to be fundamentally different from a SQL database. If I am designing my data model for a shopping cart application, for example, what are some of the patterns I’m going to want to keep in mind or some of the principles I want to keep in mind when designing my data model?

00:12:12] AD: Yup, sure thing. I think the biggest thing you want to think about first is, number one, really think out your access patterns first before you actually start designing your data model, and that's pretty different than relational, where relational, you think of your objects. You put them into tables and then you write your queries. With DynamoDB, you’re going to think of your access patterns first. So with that shopping cart, you might have add item to cart, fetch cart, checkout, create order, different things like that. Really, list all those out and think about the different elements of data you're going to have there. Think about the objects you're going to have there and where you're going to need them in different patterns and really just actually write all those out on a piece of paper what you're going to have. Then you start modeling your data to handle those access patterns specifically. Basically what you're doing is you're shaping your data to handle this access pattern to have very efficient writes, very efficient reads so you can handle these access patterns quickly without doing joins. I think that's a little bit funky and that's a big change for people, but that's going to be the biggest –The biggest thing is it's going to give you advantages. If you actually design for your access patterns rather than design in an abstract way and then trying to map your access patterns on top of it.

[00:13:17] JM: NoSQL databases are sometimes called schemaless, but this is actually a misconception. Can you explain why that is a misconception? Why does a NoSQL database actually have something of a schema?

[00:13:30] AD: Yeah, sure. I think a lot of people are like, "Hey, it's schemaless. I don't have to think about my schema. It gives me a lot of flexibility," and I think that way will lead to a lot of pain. Basically, how you should think about schemaless databases is the schema is not going to be enforced by the database itself. If you're using a relational database like Postgres, like MySQL, you'll define the different columns on your tables. You'll define the types. What the default values are? What can be known? What can't? Then if you try and write something in there that doesn't conform to that, it's going to get rejected back to you.

With NoSQL or schemaless databases like DynamoDB, that's mostly not going to happen other than the primary key that we talked about earlier. Each item needs to have that primary key. But other than that, you can basically put any attributes on your items or records that you want, and that's not going to get enforced at the database layer.

In that sense, it's schemaless because there's not a schema at the database layer, but you still need to have a schema somewhere. Now, basically what happened is that schema enforcement has moved up into your application layer. So you have to realize that you're taking on this additional burden of sort of managing your schema in that way and making sure that you know what data you're writing in so that you know that when that data comes back out and you want to show it to someone, it maps what you're expecting. So you need to do that work there and understand that, but the schemeless part is just saying the database itself will not enforce that schema.

[00:14:48] JM: When the schema's are not enforced on the database level, but in the application code, you could – Isn't it a problem that could have a legal writes that could get done? Why is that not a problem?

[00:15:03] AD: I mean, that's totally right, and I guess you just have to think where is this data coming in and where could invalid data come from. If you are accepting web requests, maybe you validate that before it even gets into your application to say, "This is what the payload body is looking like and what I'm expecting," or maybe if it's a combination of different data in your application at the very boundary where you're actually interacting with Dynamo and you're writing to Dynamo or reading from Dynamo, you're doing the validation there before you write it in to saying, "Hey, I want to conform to this schema. If it doesn't, throw that exception before you actually write that garbage into my database and I have to clean it up later on."

You just need to think, "Hey, where can this bad data come in and how do I guard myself against it?" That's part of the burden you're taking on with a NoSQL database. But I would say, at a lot of places that have basically turned a relational database into a NoSQL database where they're using it in a very keyvalue way and sort of ad hoc columns or things like that. They've turned off a lot of that schema validation logic anyway where the database isn't going to be validating that data for you, at least in prod just because it does slow down the speed on your database.

[00:16:06] JM: A developer often learns how to do relational data modeling before they get into NoSQL kinds of modeling. SQL has been dominant for a really long time. If I'm a developer that is very familiar with SQL and I want to make a move towards NoSQL, I'm going to have to make some adjustments. What are some of the relational behaviors that a developer might have to change when moving to NoSQL database modeling?

[00:16:38] AD: Yup. I think the biggest one is going be around normalization, because when you learn relational data modeling, you learn all these principles of normalization, first, second, third, normal form, all that. Basically, if I can boil it into a sense, it's basically don't repeat yourself for data is how I describe normalization. So you try not to repeat yourself. If you do have a bit of data that's referenced by multiple records, you often split that into another table and have item sort of point to that record.

The great thing about that is you get some nice data integrity benefits where if you need to update that record, you only need to update it in one place rather than in 4 or 10 or a hundred different places. But then the downside is you have that query time downside of joining together that data to pull together these normalized records. I think denormalization is probably the most difficult part for people to learn.

The other part that I think is kind of weird is there is a split between what I call application attributes and indexing attributes on your items. Application attributes, that’s what I call them, but they're basically just the attributes and properties that make sense in your application. If you have a user, this would be like the username, the first name, last name, birthdate, address, all that stuff that's actually useful in your application. But then you’ll have what I call indexing attributes, and these are like your primary keys for your base table or the primary keys for your secondary index. They’re going to look kind of funky. They probably contain elements of those application attributes in there, but they’ll also contain weird like PREFIXES and hashtags or just things like that that help to sort of organize and structure your data, especially structure different types of entities in the same item collection near each other where you can retrieve them all in one query.

The items definitely look very weird, and I think that's a hard part. Finally, the last hard part is with a relational database, you can go look at a table, and it looks like a spreadsheet and it sort of makes sense. You can make sense of it like a table, whereas in a NoSQL database, in DynamoDB, you're going to look at it and you’re going to have all these different item types and they don't have the same attributes and it's just harder to sort of make sense of just looking at it as a table. I think one thing you have to take on there is write some tooling around that to where if you have these particular access patterns that you want to debug, now you don't go to your MySQL workbench or something like that. You'll you probably have some scripts that say, “Hey, give me the user and all the users items,” or something like that, right? Some script where you can pass in a user ID and it's going to print out those objects for you rather than sort of looking at it in an admin tool or something like that.

[SPONSOR MESSAGE]

[00:19:10] JM: This episode of Software Engineering Daily is sponsored by Datadog. Datadog is a cloud monitoring platform built by engineers for engineers enabling full stack observability for modern applications. Datadog integrates seamlessly to gather metrics and events from more than 400 technologies, including cloud providers, databases and webservers. Easily identify slow running queries, error rates, bottlenecks and more fast with built-in dashboards, algorithmic alerts and end-to-end request tracing and log management from Datadog.

Datadog helps engineering teams troubleshoot and collaborate together in one place to enhance performance and prevent downtime. You can start a free trial of Datadog today and Datadog will send you a free t-shirt. Visit softwareengineeringdaily.com/datadog to get started. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:20:14] JM: The data model in DynamoDB, it can be either keyvalue or wide column data model. Can you explain what each of these is and how they differ in modeling approaches?

[00:20:29] AD: Yup. Sure. A keyvalue is going to be – if you're familiar with programming language, it's going to be like a dictionary or a map or a hash, anything like that, where you have a key and you can go get that in your data store and you get the value for it. It's just like a one-to-one mapping between things. I would say the keyvalue is used for pretty simple applications where you're only retrieving a single item at a time. But often, you need to retrieve multiple records at a time, multiple related records or a parent record and all of its children or different things like that. That's where you'd reach for that wide column store. And it's sort of like a two-dimensional hash map.

What it really is, is it's like a hash where the value of these each hash is a B-tree. So now you're getting into a little more complex computer science terminology there. But what I compare it to is imagine you had a bookshelf full of dictionaries and maybe the dictionaries are all in different languages. That first value in that wide column, you figure out which dictionary you want to go get to. Maybe you want to go pick up the English dictionary, and then the value – Or all the items within that particular item collection that have that same partition key, which is the first value in that wide column, those are to be sorted in order just like a dictionary would be. So then you can say things like, "Hey, give me all the items between dog an elephant or something like that." You can get a nice range of values there. It's sort of like that two-dimensional thing where you're saying, "Okay. First off, find me the right book and then give me a range of values within that book."

[00:21:56] JM: If I am going to model my database, eventually I'm going to implement it. There's a difference between describing the model of my database and actually implementing that. Tell me about how a developer goes from modeling their database to implementing it.

[00:22:14] AD: Yeah. I think that's pretty easy with DynamoDB. I think most of the work is in designing the data model with DynamoDB because, again, you're listing out your access patterns and you're basically saying, "Okay. How am I designing the primary keys or design secondary indexes to handle these access patterns? Or if it's this write operation, I may be using this write transaction. If it's this read operation, I'm using a query," or maybe just a simple get item to retrieve a single item. You're basically listing all those out.

At the end, you've got it all written out. You haven't done any coding. You sort have designed your entire data model. Then it's just a matter of going and implementing those, and you make a – I usually make like a class for each entity than I'm building there that knows how to sort of turn an object in my application into a DynamoDB item. Then I have some simple data access functions that I just pass those objects into. They write them to Dynamo and go – I find that the implementation part with DynamoDB is the easiest part. You need to learn a little bit of syntax around how to write conditions and expressions or key condition expressions or update expressions. A few things like that, a little bit of syntax. But really, it's the modeling aspect that's going to be a hard part of it. And then once you've done all that, you basically have a spec to go implement.

[00:23:28] JM: We've kind of glossed over some of the core concepts of DynamoDB. I'd like to go into some of the core concepts and then talk about things like indexing and how to define the write keys. But if we start with just the core concepts of DynamoDB, let's say we're defining a database to manage all my podcast episodes. It's a simple example that we're both familiar with, and each of these episodes has topics. It has a guest title. It has the host. It has the date. Tell me about how this data could be modeled and describe the core concepts that we would need to know for creating this database.

[00:24:14] AD: Yup. Sure. You think about the different entities you might have in your application. In this one, we might have podcasts. You might have hosts and guests, as you mention, and topics. Those would be the first sort of things I'd think about there. Then you'd sort of think about your access patterns again. You get podcasts by ID. Maybe get most recent podcasts. Get podcasts by guests. Get podcasts by topic, things like that. And once you've sort of listed those out, now you'll start thinking about modeling.

In terms of the core concepts you have there, the foundational one is going to be an item. That's going to be like a row in a relational database, but it just refers to a record where you have some combination of data together there. Your data can – Each item can have what are called attributes on them, which are just the actual data themselves, and those can be things like strings and numbers or they can also be complex values, like lists, and maps, or dictionaries, whatever you want to call those.

I guess a collection of these different items is going to be in a table. So you might compare that to a table in a relational database, but the big difference between a DynamoDB table and a relational table is a DynamoDB table will probably have different types of entities in it. It might have – In this example, it might have people, which would be the host and the guess on your show. We would have podcasts themselves. We might have maybe an entity representing a topic that has some information about that topic itself. You'd have all these types of entities in the same table rather than being split out across multiple tables like you would in a relational database.

Now, when you create that table, you're going to create something that's called a primary key. The primary key is going to uniquely identify each item that goes in that table. Each item that you write to that table needs to have that primary key and it needs to uniquely identify that item. So you can't have two items with the same primary key. They would just overwrite each other if so.

There are two types of primary keys. One is going to be just the simple primary key that has just a single value. This is the partition key. This is when you're using Dynamo and that keyvalue method, as you mentioned before. The second type, more common, is what's called a composite primary key that has two elements. That's going to have a partition key and then was called a sort key as well. When you're using that composite sort key, it's the combination of those two [inaudible 00:26:21], the partition key and the sort key that uniquely identify an item.

You can have multiple items in that table with the same partition key. It will have a different sort keys or with the same sort key and different partition keys. Likewise, I think the last important concept you really want to know with DynamoDB is one called secondary indexes, which we've touched on a little bit. But basically, you can create a secondary index on your table and you specify some attribute names that are going to serve as the primary key in your secondary index. Now, when you write items into that base table with your main primary key, if those items also have the attributes that you've specified for your secondary index, DynamoDB is going to copy them into the secondary index without reshape data. So then you can query them based on those, the primary key values for that secondary index. It's going to be the exact same data, but they're just handling, replicating that over for you in this different shape that's going to allow you to query in different ways.

For example, with this podcasts example we've said, you might have a podcast item where the primary key is based on the episode number or something like that. So you could easily look up an episode by its number, but then you also want to group items together by topic. Maybe you make a secondary index where the partition key is going to be, the topic is going to be encoded in there, and then the sort key for that would maybe be episode number. Then when you query against that secondary index, you could give it the topic name. It's going to give you all episodes with that topic in order, by episode number there.

[00:27:45] JM: In a more complex data model, how would I choose indexes to define? Give me some general principles for how to choose indexes to define in DynamoDB.

[00:27:58] AD: Yeah. The big thing is any time you want to sort of group items together in a particular way or maybe filter down in a particular way or sort in a particular way, you'll probably use a secondary index. You can handle some of those main access patterns by the other primary key in your table, and then additional ones you can handle elsewhere. Just going back to this podcast example, maybe the primary key in our database allows us for that keyvalue access of a particular episode by its episode number. But then we want to get the most recent episode. So maybe we would group them together by month or week or something like that that would allow us to sort of go to the most recent month and find the most recent episodes in order there.

Also, you could group by topic as we were mentioning. That would be another secondary index you would have where you would be using the topic somewhere in that partition key to group items together and then ordering them by episode name. You could also do by guest if you have a lot of repeat guests. I would say you probably don't have a ton of those. But yeah, if you want to allow people to look up by guest and see people that have come multiple times, you could maybe do a secondary index that way that would allow that. Yeah, I think those would be the basic ones. But the big thing is whenever you want to group or sort or filter in a particular way, you'll probably use that primary key, because those primary keys, those are doing most of the data access work for you, because that's how you're accessing your data. So you want to you want to use those in a way that do some of that filtering for you.

[00:29:24] JM: There's this tradeoff between defining indexes and the latency that can occur in having updates. If I make a bunch of indexes over my data, then if I write a new piece of information to that database, that update to the database may need to be indexed. That's going to take a while for that data to be available. Is there any rules you have around how frequently one should define an index? Because obviously there's a tradeoff in that definition.

[00:30:04] AD: Yeah. One thing I would say there is you usually won't see much of a write impact on having multiple indexes. This gets a little into the weeds because there are two types of secondary indexes. There's a local secondary index and a global secondary index. Use global secondary indexes in almost all situations, but local secondary indexes will probably add a little to latency because they do have consistent reads if you want. So they probably need to wait a little longer on how that write works. But for the most part, let's just simplify it down and say you're using a global secondary index. That's going to be replicated out to that global secondary index asynchronously. I don't think you'll see much impact on the write path there, at least on that sort of initial write. What you will see is there is some latency if you're reading from that secondary index. You could see slightly out of date data as that data is replicated into your secondary index. That's something you need to account for in your application. What sort of consistency requirements do you need? Are you a bank? Or you need this to be exactly consistent or can you handle a little inconsistency there?

That's one thing I would say. But the going back to the tradeoffs around secondary indexes, I think most of the tradeoffs around secondary indexes are actually around cost, because if you have – With DynamoDB, you're paying for reads and writes. And if you have secondary indexes, you have to pay for reads and writes on those secondary indexes as well. If you have 8 secondary indexes, and every time you write that podcast record in there is going to get replicated out to 8 different indexes, now you're paying nine times for that write rather than just once if you just have the base table itself. I mean, one thing you can do is try and reuse the same index for multiple access patterns and there are some funky ways to do that. But yeah, I think read and writes are really going to be the biggest thing rather than sort of write latency itself.

[00:31:46] JM: What about joins? Join is a relational operation. We would join between two different database tables. How do we overcome the need for joins in a NoSQL scenario if we don't have tables?

[00:32:00] AD: Yup. Good question. When we talk about using DynamoDB as a wide column store, which we talked about before, which has two values of partition key and a sort key. If you're using DynamoDB in that way, you can use what's called a query operation, which allow you to act on items that have the same partition key, so have that same value in the partition key and you can read multiple items in the same request.

One thing you can do is sort of pre-join your data by putting the parent item with its related items. There's a lot of examples in the book on this, but let's say you have an e-commerce application and a user makes an order, and an order is made up of different order items. So you have different types of records there and sometimes you want to sort of fetch the whole summary of the order where you want to get the order itself and all the related order items that were ordered in that order. You can put those into what's called the same item collection, which means those items have the same partition key. And now you can use the query to operate on that item collection with the same partition key and get all those values. What you've done is you've basically pre-joined your data by writing them into the same item collection with that same partition key so that you can get them all in one request.

I think that's probably one of the most common ways to handle joins. Another thing you can do is just de-normalized your data. Like I was saying before, Dynamo allows you to use things like maps and arrays on your items themselves. So if you have something that has some related items but not a ton of them, not an unbounded number of them, you can actually just store those related items on the parent item itself.

One example here, again, going back that ecommerce application. Imagine when a customer creates an account. You allow them to save multiple mailing addresses, or credit cards, or things like that. Since it's unlikely that they're going to have 10,000 credit cards, you could just store that as an attribute on that parent user item, and then rather than having to join a crossed multiple records like you would in a relational database, you just have it stored directly on that user. And whenever you fetch that user, all those credit card details are going to come back with it.

[00:34:00] JM: If we're talking about partitioning, how does DynamoDB choose which partition to put a record in? Tell me about the partitioning strategy for designing a database.

[00:34:14] AD: Yeah. I think the partitioning thing one of the coolest parts of it. Like I said, every single record is going to have a partition key, and this is going to be true whether you have that simple keyvalue model that just has a single part of the primary key. That going to be called the partition key, or if you have that wide column model that has two elements, the partition key and the sort key, it's still going to have that partition key on there. So every item you put in DynamoDB is going to have that partition key.

When you're writing an item to DynamoDB, the first thing that – The sort of frontend of DynamoDB, which is called the request router, what it's going to do is it's going to take that partition key. It's going to hash it and then it's going to figure out where the value for that hash function which partition it lies on? Basically, you're DynamoDB table is going to be split across all these different partitions, which are basically different storage instances across all different nodes in the AWS infrastructure. Those partitions are maybe going to be 10 gigs max, but they're going to be split all over the place and then they're going to spread your database base on that partition key. Once they hash that value, then they just have a map that says, "Okay. Here are the 10 partitions that are storing the data for this table and here's where partition one starts and where it ends it. Here's where partition two starts and ends." And then based on what that hashed partition value is, they can look on which node it goes to and then actually go make the request to that node.

The really elegant thing about that is just how well it scales at, because if I have a table with five gigs of data, it's probably just going to be one partition. But if I have a table that has 10 terabytes of data and it's on however many partitions that would be, it's still going to be up pretty fast lookup. It's going to be 01 look up, because they're going to hash that. They're going to look it up in a map that has those config values and then they're going to go down to that node.

If you have that 10 terabyte table, based on that 01 lookup, now they've already narrowed it down to 10 gigs of data that they can go search within. And your search space is much smaller, and that's what basically gives them essentially infinite scaling. And what I think is the best feature of DynamoDB is that you're going to get the exact same performance when you have five gigs of data as when you have 10 terabytes of data. You just like don't have to think about refactoring your application to add indexes or change these joins or different things like that as your application slows down like you might in a relational database.

[00:36:24] JM: Let's talk about query optimization. When I'm writing queries against my DynamoDB table, tell me about how to optimize those queries.

[00:36:34] AD: It's interesting, because I would say every database has a philosophy, and I think DynamoDB's guiding philosophy is they will not let you do something that won't scale, and this goes back to what I was just talking about with this partitioning thing and it's going to work the same at 5 gigs as it does as 10 terabytes. But basically, the way the operations are set up in DynamoDB, there's not much you have to optimize. You basically have individual item lookups. So you can do get items. You can do insert item, update item, delete item. Those are all very simple keyvalue ones. It's going to be basically an 01 operation. Then you have this query operation that you can do. When you do a query, you have to operate within a single partition. So you have to specify the partition key that you want to operate within. , you're doing that 01 lookup upfront to narrow it down to a specific partition. Then you can do basically arrange search within the values for that partition. And because the values for that partition are sorted, they're going to be implemented as a B-tree behind the scenes. You basically say what the start value and what the end value is that you want to go read. Again, it's just like going to a dictionary and finding a particular range of values.

But because of that, because you basically have these two pretty well bounded operations, you have that 01 lookup and then you have whatever the B-tree range search is, like a log in or something like that. I can't remember. But there's not a whole lot of a query optimization you have to do as long as you've done your data modeling correctly, which DynamoDB is going to force you to do upfront when you have a low amount of data, when you don't have any users and you just have a little bit of test data. It's going to make you write your queries correctly there so that's going to scale up well and you won't have to sort of re-optimize your queries later on.

This is – I guess as long as you're not using the scan operation. There is a scan operation as well, which just operates on your whole table. Some people use that. They scan their whole table and then filter down client-side. That's not going to work as you scale, because you can't be scanning your database every time.

[00:38:24] JM: What about when my a data model changes really dramatically or when the amount of data in my database scales up? Are there ways that certain queries can degrade in their quality?

[00:38:40] AD: When you talk about the amount of data really scaling up, I would say that's almost never going to be a problem provided you haven't done a few things. And number one is you haven't used or heavily relied on the scan operation. If you're using the scan operation on a hot path, that's not going to scale very well. The second thing is if you have operations that page through multiple results. Basically, with a query and a scan operation, when you do those, DynamoDB is going to limit the results you can get back to one meg of data. They're only going to read a max of one meg from disk there. If you have more than that, then you're going to have to paginate through that.

If in your application patterns you’re relying on paginating whether through an item collection in that query operation or across your whole tail with a scan operation, that's not going to scale well, and that's one of the ways you can sort of have something blowup from you over time. But honestly, you can go to a codebase and do a search in there and just make sure they're not scanning. If they're doing any pagination, just look at that and think about how big item collection could get over time. And as long as you’ve sort of checked off both of those boxes, you should almost definitely be able to scale.

I guess the last one I would say too is there's a notion of a hot partition or a hotkey where any given partition value can only – It has a maximum number of reads and writes it can do per second. And this numbers is pretty high. It's 3,000 read units or 1,000 write units per second, and that's on a single partition. That's going to be like on a particular user or a particular order and all of its order items, things like that. It's not across your whole table. As long as you're not exceeding that 3000 read units per second, 1000 write units per second and you’re not handling those pagination issues, you should be fine there.

Now, second question you had there is about migrations, and I think that's a more difficult story and it depends on what sort of migration you're doing. The first one is if your data model is mostly the same but you're just doing some little migrations, maybe that's adding some attributes to some items or maybe you’re adding new entity types into your table. I think you can do that pretty straightforward. You might have to do basically in ETL process where you scan your whole table, decorate your existing items with some new attributes and then go on and then you’re good to go. But even that is a pretty I think straightforward operation where once you've done it two or three times, you’re going to be in good shape and you understand the pattern there.

The harder one is if you're – If your data model actually truly changes quite a bit where you need to pull all the items out and reshape them, write them all in a different form. That's going to be difficult whether you’re using DynamoDB or relational database or anything like that, and that's going to be a bigger ETL migration. That's going to be ad hoc. I can't give you sort of solid rules for that if you data model changes that drastically. But in terms of just like adding additional access patterns, I actually don't think it's as scary as people think, and there are a few well-worn patterns that you can follow for those.

[SPONSOR MESSAGE]

[00:41:36] JM: Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we're being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it's definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[INTERVIEW CONTINUED]

[00:43:13] JM: What about the consistency model? Is there anything to keep in mind about eventual consistency or the strong consistency? Just give me an overview of the DynamoDB consistency model.

[00:43:25] AD: Yup, sure thing. Just as quick briefer for listeners, I guess you can think about strong consistency and eventual consistency, and it refers to when you're doing a read, have you seen sort of the most recent write or update operation for those records you're getting? In a strong consistency model, you can be sure you're getting all the most recent writes and updates. In an eventually consistent model, you're maybe saying, “Hey, I might not have gotten the most recent reads or updates.

With DynamoDB, when you're operating against your base tables, so not a secondary index, but your base table in this instance, the default consistency is eventual consistency, which means you might not get the most recent up-to-date thing, which we’re talking about. Maybe if there's a write within the last 200 milliseconds, you might not have gotten that update yet. But it’s generally going to be correct.

On that primary table, you may opt in to a strongly and consistent read, which will ensure that you’ll get the most recent updates for that operation. The basic difference there is going to be how many read units you end up consuming. If you go with that eventually consistent read, you actually consume half the read units that you would otherwise consume from that operation.

Now, if you're using a secondary index and specifically a global secondary index, you don't have the opportunity to opt into that strongly consistent read on that secondary index. With a global secondary index, you have to get an eventually consistent read, which means you’re potentially getting into some lag in data values, which is usually 200, 300 milliseconds from your base table, but it's possible that you’ll get some slightly stale data. So something you need to consider there, and you can't opt into a strongly consistent read on that global secondary index.

[00:45:04] JM: The integration points for DynamoDB plug in closely to serverless technologies, Lambda in particular, AWS Lambda. Why is DynamoDB so compatible with Lambda? What are the patterns around using DynamoDB with Lambda?

[00:45:24] AD: Yeah. I mean, they're a bunch of them. I'd say the biggest one is using DynamoDB feels like using any other AWS service you're using. In these serverless architectures, you’re often taking advantage of these fully-managed services, like S3 for blob storage, like SQS for queuing, like SNS, for pub/sub broadcasting type things. For each of those, you're pulling in the AWS SDK, whether that’s Boto3 for Python or whatever it is. You're creating a client and then you're just calling a method on that client to sort of interact with that service.

With DynamoDB, it's the exact same thing. You're interacting over HTTP. It's using AWS IAM for authentication, which if you’re using Lambda, you've got some credentials injected in that container so you don't need to think about username and password credential management, whereas if you're using a relational database, now, in your Lambda function, you need to spin up this persistent TCP connection and you need to think about credential management. Also, you probably have your database network partition, right? It's not accessible to the public Internet, because you don’t want your Postgres available to the public Interests. Now it's in a VPC, which means you need to put your Lambda in a VPC. And for a long time, that resulted in some pretty severe cold start penalties, 10 or 15 seconds, which is just unacceptable for user-facing web applications.

That was a pretty severe problem for people, and I think – Oh! The final thing with relational databases is MySQL, Postgres, they often have a connection limit, which limits the number of open connections you can have to that database, where if you’ve got 30 different functions and there are 5 to 10 instances of those function containers running, you might exceed that connection limit and now you can't connect to your database, where DynamoDB doesn't have that. You can connect as many times as you want. It's all HTTP. It's all IAM.

I think that connection model and just the interface model, it feels very natural and it works a lot easier. A couple other things I would say, DynamoDB works really well with infrastructure as code, where I think relational databases don't quite as well. Usually, you can provision the database itself via infrastructure as code. But then you sort of have this post provisioning step you need to do prepare it by creating your tables and adding indexes and things like that and where does that fit in Terraform or cloud formation? You don't really have that with Dynamo. You can create everything you need in cloud formation and in Terraform with DynamoDB.

Then, finally, that billing model where you can do that on-demand billing model with DynamoDB. You pay per request. And I can have a totally pay per request infrastructure that has HTTP. It has compute. It has data storage. It has queuing. It has blob storage. It has pub/sub and all that. And every single element of that is pay per request, which is pretty nuts. Where with a relational database, you're saying, “Hey, I want this big a server and I’m going to pay this much for it this month, whether it's overnight. And I'm not using it,” or anything like that.

[00:48:12] JM: What about Dynamo streams? There are there's a stream API associated with DynamoDB. What is it Dynamo stream and what applications would I use that for?

[00:48:23] AD: Yeah. Sure. A DynamoDB stream is just you can configure that on your table to basically say, “When a write operation happens on my table, whether that's an insert, an update, a delete, anything like that, if data changes in my table, drop a record in that stream to tell me what happens,” and it's going to say, “Hey, it was an insert operation,” and you can skip that item as it looked before. You can get it as it looked after, anything like that, same with update, same with the deletes. Now you can consume that and do different things with it.

If you have a microservices architecture, maybe you have this user service, but then different services in your application need information about those users. Maybe they need to update the address whenever a user updates an address or different things like that. Maybe they're storing a replica of that user in their service. What you’re going to have in that user service, you have that stream going on. You have something consuming from that stream and sort of reconstructing it into some sort of events. You can say, “Hey, user created. Here's the new user,” or, “User is updated. Here are the values that changed,” or, “This user was deleted and here's the user ID.” And you can spread that through SNS or Event Bridge, whatever you want to do and on different microservices. Pull on to that and say, “Okay. Now this user has changed. Let’s eject it from our copy of the data,” or “Let's update this address,” or whatever that is. So you can do that stuff.

Another thing you can do that is very popular is just hook it into analytics systems. So maybe you dump that record into Firehose to S3 and then you do query with Athena into Redshift, something like that. Because DynamoDB is not going to be great for running big analytic queries. It's good for those OLTP applications. Maybe you want to get that data out and put it somewhere where you can say, “How many users sign up this month, or how did our order volume go up this month compared to the last month?” Things like that. You can put these records into an analytic systems by hooking into those streams and doing it that way.

[00:50:06] JM: Are there any models for interacting between Dynamo streams and Lambda, like typical patterns for triggering lambdas off of something that happens in a stream?

[00:50:19] AD: I think Lambda makes it pretty easy to consume streams in AWS and whether talking about Kinesis streams are DynamoDB streams, it's pretty easy. You write your Lambda function and you say, “Hey, this event comes in. This is how I’m going to operate on it.” You hook it up to the stream, whether it’s that Dynamo stream or the Kinesis stream. And they’re going to handle all that triggering stuff for you, all the sort of check pointing of where your function is in terms of processing that stream, and it really takes off the heavy lifting for you.

I see people complain about stream processing in AWS and especially around Kinesis, and if you're not using Lambda, I think streams are a much bigger headache, because now you have to use like the Kinesis client library or different things like that and maintain your checkpoint. And there’s just like a lot of overhead work, whereas if you're using Lambda, it’s so easy. You have a function and it says when there are records in that stream, give it to me and I'll go do something with them. But other than that, I don't have to worry about anything, and it's pretty great.

[00:51:13] JM: Near the end of your book, you modeled the entirety of GitHub and you made a really detailed example for how that entire DynamoDB instance would be laid out. Can you tell me about some of the lessons that you are trying to convey in architecting this very, very detailed data model? I mean, GitHub has as all kinds of things. There’re users, and pull requests, and comments, and reactions, and you really modeled all of it.

[00:51:42] AD: Yup. Sure thing. I had a lot of fun doing it. I think the biggest thing for me is I've seen people online saying, “Hey, you can’t do complex things with DynamoDB. It's great for keyvalue storage or maybe a simple data model that has two or three types of entities in it, but you can't do complex relational stuff.” To me, that was just – I knew that wasn't true, especially because of how Amazon and AWS used DynamoDB pretty heavily, and they’ve got complex data models. It clearly can be done, and I just wanted to sort of show that to people and show, “Hey, this is how you handle multiple different relationships and really sort of traverse those relationships and rearrange and handle these different complex access patterns.

I pick GitHub because I think it's pretty easy for people to hook on to. You don't have to explain the data model as much. They already sort of grok it, but then you can have something pretty advance. I think it has 10 or 12 entities in there, and I can't remember how many access patterns, but 30 or 40. Really, just understanding that process of looking through the different screens, thinking about the different ways I'm going to access this data. The different reads, the different writes, the different constraints I'm going to have and then just walking through that. I would say I learned a ton from doing that, but I think that probably the best way to help people understand that like DynamoDB could do way more than just keyvalue store.

[00:52:56] JM: It's clear that you put ton of effort into this book. There are some excellent diagrams, just tons of information. How long did it take you to write the DynamoDB book?

[00:53:06] AD: That depends on what you're talking about. In terms of the book itself, I would say last year in 2019, I sort of worked on it nights and weekends from maybe July through September. I was like, "Hey, I'm going to release this by October," and then I realized I wasn't, because I just got tied up with other things. So then it was sort of on hiatus for a few months. Then December is like, "Okay, I really need to finish this book. I want to focus on it," and that's when I decided to quit my job. I quit. My last day was in the middle of January. So then from January 15th to – I released it April 7th, I believe, April 7th or 8th, somewhere in there. First week of April. Worked on it pretty much full-time there. That was basically my full-time job with a little bit of consulting there. Another two and a half, three months of that, a lot of stuff there. But I'd say that's the writing part of it. But learning DynamoDB took me a lot of time, because two and a half years ago, wrote dynamodbguide.com, which I wasn't very good at Dynamo when I wrote that, but it's sort of at least it helped like funnel a lot of interesting stuff to me where then like people would ask me questions even though I was an expert, but at least I was like out there. I was one of the few people like trying to learn this stuff. I think I learned a lot over those two years and before I even started writing the book just because I had this sort of focal point and people would reach out to me with questions. That helped a lot too. Yeah, probably if I would have sat down four or five months of actually writing the book, but a lot of ad hoc time here and there as well.

[00:54:33] JM: Let's finish with just some comparisons to other databases. DynamoDB is not the only NoSQL scalable database. There's Cassandra. You got FuanaDB. You've got – Well, actually I guess FaunaDB is not exactly in the same category. At least there's Cassandra. Are there's any other NoSQL –I guess there's like React.

[00:54:58] AD: There's Mongo.

[00:55:00] JM: Oh, yeah. Mongo, sure. How does DynamoDB compare to the other non-relational scalable databases?

[00:55:08] AD: Yup. Sure. I think, Cassandra it's the most sort of directly similar to in terms of data model, also a wide column store. The big differences there are not really around the data model as much, but just other aspects. Cassandra is open source. You can run it yourself. You can run it on any cloud. You can run it on-prem. Whereas Dynamo is proprietary to AWS. It's also fully managed, which means you don't have to worry about any that. Pick your poison there. Do you want open source and movable or do you want fully managed? I think those are the big differences there.

Mongo verse Dynamo I think is interesting, and I said earlier on that, like every database has a philosophy, and I think the philosophies of DynamoDB and Mongo are very different. I said DynamoDB, their philosophy is we don't want to let you do anything that won't scale. And that's basically true. As long as you're not using the scan or the pagination, like anything you do at one gig is going to work at 10 terabytes, which means it's restrictive upfront, which means you need to plan your access patterns way more up front. And it feels like a burden if you know you're only going to have 10 gigs of data. That's frustrating.

But then Mongo, I think their philosophy is totally different, where they're focused on ease-of-use and maybe developer happiness. They allow you to put these documents in there. They allow very flexible indexing where you can index on all sorts of different things. You can index on nested fields. You can index your – If you have arrays on documents, you can index those. You can just to sort of all sorts of flexible things and optimizing for that. They have this aggregation framework where you can do aggregations on the SnowSQL, which is something you probably shouldn't do at scale. But if you have small databases, you can do that and it works well.

I would say if your Mongo can work at scale, but a lot of the things that are nice when you're just starting out with Mongo, like these flexible indexes, like the aggregation framework, are totally not going to work if you actually get to 10 terabytes scale. That's where you can – It's sort of like a relational database where you can do these things that work at 10 gigs. And then once you get up to a terabyte, you find out that they are just going to kill you and now you have to rework that whole application and do this giant migration and all that stuff. It's pick your poison there on whether you want to do more work upfront or more work later on. That's the big thing. Then Mongo, again, being open source, you can host it yourself. Some of the same decisions about Cassandra versus DynamoDB there as well.

[00:57:28] JM: All right, Alex. Well, it's been really awesome talking to you about DynamoDB. Any final thoughts on the database or the experience of writing the book?

[00:57:38] AD: Yeah. I don't think so. Thanks for having me on. I've been a huge fan of the podcast for a while. I'm very bullish on DynamoDB and its future. I think I still think learning and education is like the biggest pain point for everyone, where everyone knows the relational model and people are still getting used to the NoSQL model. I'm hoping we're pushing that forward there. I think where the community is starting to grow around. Not just Dynamo, but around NoSQL in general. I'm excited to see sort of what the future holds there.

[00:58:02] JM: Thanks, Alex.

[00:58:03] AD: Thanks, Jeff.

[END OF INTERVIEW]

[00:58:13] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[END]