All Articles

The What, Why, and When of Single-Table Design with DynamoDB

I've become a big proponent of DynamoDB over the past few years. DynamoDB provides many benefits that other databases don't, such as a flexible pricing model, a stateless connection model that works seamlessly with serverless compute, and consistent response time even as your database scales to enormous size.

Yet data modeling with DynamoDB is tricky for those used to the relational databases that have dominated for the past few decades. There are a number of quirks around data modeling with DynamoDB, but the biggest one is the recommendation from AWS to use a single table for all of your records.

In this post, we'll do a deep dive on the concepts behind single-table design. You'll learn:

- What is single-table design
- Why single-table design is needed
- The downsides of single-table design
- The two times where the downsides of single-table design outweigh the benefits

I recently did a live debate on this subject on Twitch with Rick Houlihan and Edin Zulich. Check out the recording here. Rick is more staunchly in the single-table camp, so be sure to watch that as well.

Let's get started!

What is single-table design

Before we get too far, let's define single-table design. To do this, we'll take a quick journey through the history of databases. We'll look at some basic modeling in relational databases,

then see why you need to model differently in DynamoDB. With this, we'll see the key reason for using single-table design.

At the end of this section, we'll also do a quick look at some other, smaller benefits of single-table design.

Background on SQL modeling & joins

Let's start with our good friend, the relational database.

With relational databases, you generally normalize your data by creating a table for each type of entity in your application. For example, if you're making an e-commerce application, you'll have one table for Customers and one table for Orders.

Customers		
CustomerId	**CustomerName**	**CustomerBirthdate**
741	Alex DeBrie	05/26/1988
742	Albert Einstein	03/14/1879

Orders		
OrderId	**CustomerId**	**OrderDate**
11578	741	12/20/2019
11579	910	12/21/2019

Each Order belongs to a certain Customer, and you use foreign keys to refer from a record in one table to a record in another. These foreign keys act as pointers — if I need more information about a Customer that placed a particular Order, I can follow the foreign key reference to retrieve items about the Customer.

Customers		
CustomerId	**CustomerName**	**CustomerBirthdate**
741	Alex DeBrie	05/26/1988
742	Albert Einstein	03/14/1879

Orders		
OrderId	**CustomerId**	**OrderDate**
11578	741	12/20/2019
11579	910	12/21/2019

To follow these pointers, the SQL language for querying relational databases has a concept of *joins*. Joins allow you to combine records from two or more tables at read-time.

The problem of missing joins in DynamoDB

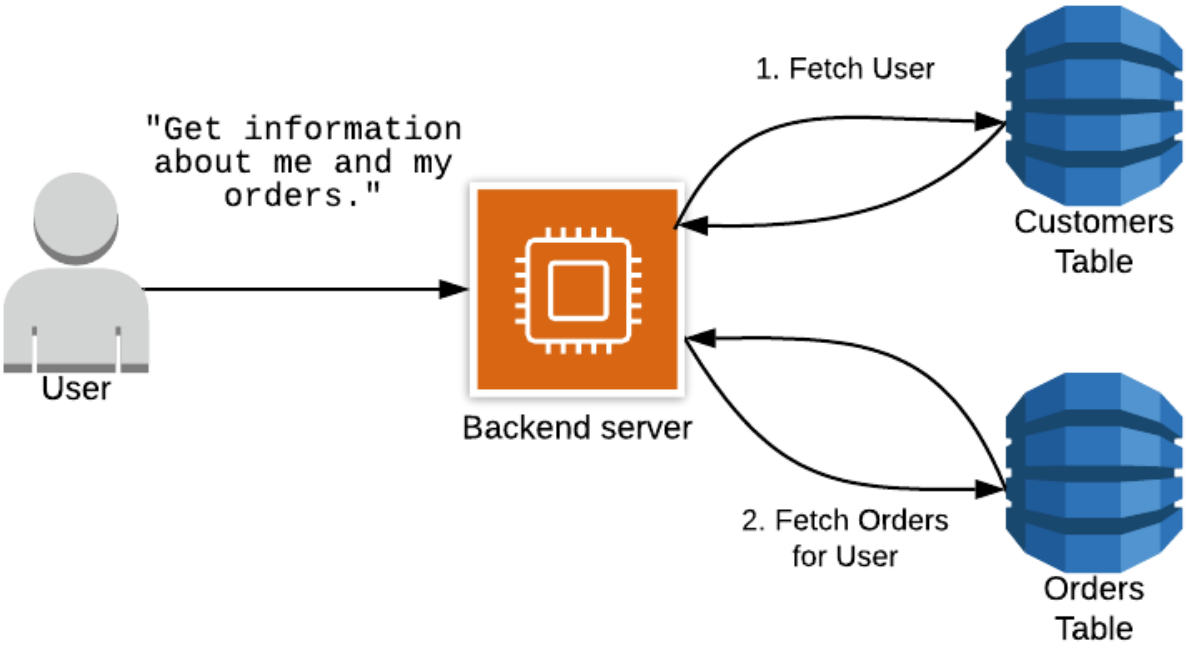
While convenient, SQL joins are also expensive. They require scanning large portions of multiple tables in your relational database, comparing different values, and returning a result set.

DynamoDB was built for enormous, high-velocity use cases, such as the Amazon.com shopping cart. These use cases can't tolerate the inconsistency and slowing performance of joins as a dataset scales.

DynamoDB closely guards against any operations that won't scale, and there's not a great way to make relational joins scale. Rather than working to make joins scale better, DynamoDB sidesteps the problem by removing the ability to use joins at all.

But as an application developer, you still need some of the benefits of relational joins. And one of the big benefits of joins is the ability to get multiple, heterogenous items from your database in a single request.

In our example above, we want to get both a Customer record and all Orders for the customer. Many developers apply relational design patterns with DynamoDB even though they don't have the relational tools like the join operation. This means they put their items into different tables according to their type. However, since there are no joins in DynamoDB, they'll need to make multiple, serial requests to fetch both the Orders and the Customer record.



This can become a big issue in your application. Network I/O is likely the slowest part of your application, and you're making multiple network requests in a waterfall fashion, where one request provides data that is used for subsequent requests. As your application scales, this pattern gets slower and slower.

The solution: pre-join your data into item collections

So how do you get fast, consistent performance from DynamoDB without making multiple requests to your database? By *pre-joining* your data using item collections.

An item collection in DynamoDB refers to all the items in a table or index that share a partition key. In the example below, we have a DynamoDB table that contains actors and the movies in which they have played. The primary key is a composite primary key where the partition key is the actor's name and the sort key is the movie name.

Primary Key		Attributes		
Actor (PARTITION)	**Movie (SORT)**			
Tom Hanks	Cast Away	**Role**	**Year**	**Genre**
		Chuck Noland	2000	Drama
	Toy Story	**Role**	**Year**	**Genre**
		Woody	1995	Children's
Tim Allen	Toy Story	**Role**	**Year**	**Genre**
		Buzz Lightyear	1995	Children's
Natalie Portman	Black Swan	**Role**	**Year**	**Genre**
		Nina Sayers	2010	Drama

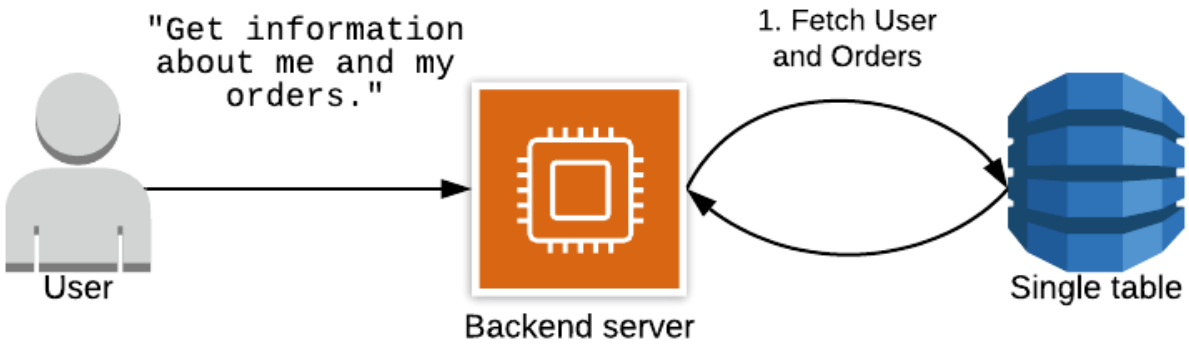
You can see there are two items for Tom Hanks — Cast Away and Toy Story. Because they have the same partition key of `Tom Hanks`, they are in the same item collection.

You can use DynamoDB's `Query` API operation to read multiple items with the same partition key. Thus, if you need to retrieve multiple heterogenous items in a single request, you organize those items so that they are in the same item collection.

Let's look at an example from my DynamoDB data modeling talk at AWS re:Invent 2019. This example uses an e-commerce application like we've been discussing, which involves Users and Orders. We have an access pattern where we want to fetch the User record and the Order records. To make this possible in a single request, we make sure all Order records live in the same item collection as the User record to which they belong.

Primary Key		Attributes				
PK	**SK**					
USER#alexdebrie	#PROFILE#alexdebrie	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		alexdebrie	Alex DeBrie	alexdebrie1@gmail.com	03/23/2018	{"Home":{"StreetAddress":"1111 1st St","State":"Nebr
	ORDER#5e7272b7	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	5e7272b7	PLACED	04/21/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","C
	ORDER#42ef295e	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	42ef295e	PLACED	04/25/2019	{"StreetAddress":"1111 1st St","State":"Nebraska","C
	ORDER#2e7abecc	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		alexdebrie	2e7abecc	SHIPPED	12/25/2018	{"StreetAddress":"1111 1st St","State":"Nebraska","C
USER#nedstark	#PROFILE#nedstark	**Username**	**FullName**	**Email**	**CreatedAt**	**Addresses**
		nedstark	Eddard Stark	lord@winterfell.com	02/27/2016	{"Home":{"StreetAddress":"1234 2nd Ave","City":"Wir
	ORDER#2eae1dee	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	2eae1dee	SHIPPED	01/15/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L
	ORDER#f4f80a91	**Username**	**OrderId**	**Status**	**CreatedAt**	**Address**
		nedstark	f4f80a91	PLACED	05/12/2019	{"StreetAddress":"Suite 200, Red Keep","City":"King's L

Now when we want to fetch the User and Orders, we can do it in a single request without needing a costly join operation:



This is what single-table design is all about — tuning your table so that your access patterns can be handled with as few requests to DynamoDB as possible, ideally one.

And because everything looks better in fancy quotes, let's say it one more time:

The main reason for using a single table in DynamoDB is to retrieve multiple, heterogenous item types using a single request.

Other benefits of single-table design

While reducing the number of requests for an access pattern is the main reason for using a single-table design with DynamoDB, there are some other benefits as well. I will discuss those briefly.

First, there is some operational overhead with each table you have in DynamoDB. Even though DynamoDB is fully-managed and pretty hands-off compared to a relational database, you still need to configure alarms, monitor metrics, etc. If you have one table with all items in it rather than eight separate tables, you reduce the number of alarms and metrics to watch.

Second, having a single table can save you money as compared to having multiple tables. With each table you have, you need to provision read and write capacity units. Often you will do some back-of-the-envelope math on the traffic you expect, bump it up by X%, and convert it to RCUs and WCUs. If you have one or two entity types in your single table that are accessed much more frequently than the others, you can hide some of the extra capacity for less-frequently accessed items in the buffer for the other items.

While these two benefits are real, they're pretty marginal. The operations burden on DynamoDB is quite low, and the pricing will only save you a bit of money on the margins. Further, if you are using DynamoDB On-Demand pricing, you won't save *any* money by going to a single-table design.

In general, when thinking about single-table design, you should consider the main benefit to be the performance improvement by making a single request to retrive all needed items.

Downsides of a single-table design

While the single-table pattern is powerful and ridiculously scalable, it doesn't come without costs. In this section, we'll review some of the downsides of a single-table design.

In my mind, there are three downsides of single-table design in DynamoDB:

- The steep learning curve to understand single-table design;
- The inflexibility of adding new access patterns;
- The difficulty of exporting your tables for analytics.

Let's review each of these in turn.

The steep learning curve of single-table design

The biggest complaint I get from members of the community is around the *difficulty* of learning single-table design in DynamoDB.

A single, over-loaded DynamoDB table looks really weird compared to the clean, normalized tables of your relational database. It's hard to unlearn all the lessons you've learned over years of relational data modeling.

To those who are avoiding single-table design because of the learning curve, my response is this:

Tough.

Software development is a continuous journey of learning, and you can't use the difficulty of learning new things as an excuse to use a new thing poorly.

Later on in this post, I will describe a few times when I think it's OK to decide not to use single-table design. However, you should absolutely understand the principles behind single-table design before making that decision. Ignorance is not a reason to avoid the general best practices.

The inflexibility of new access patterns

A second complaint about DynamoDB is the difficulty of accommodating new access patterns in a single-table design. This complaint has much more validity.

When modeling a single-table design in DynamoDB, you start with your access patterns first. Think hard (and write down!) how you will access your data, then carefully model your table to satisfy those access patterns. When doing this, you will organize your items into collections such that each access pattern can be handled with as few requests as possible — ideally a single request.

Once you have your table modeled out, then you put it into action and write the code to implement it. And, done properly, this will work great! Your application will be able to scale infinitely with no degradation in performance.

However, your table design is narrowly tailored for the exact purpose for which it has been designed. If your access patterns change because you're adding new objects or accessing multiple objects in different ways, you may need to do an ETL process to scan every item in your table and update with new attributes. This process isn't impossible, but it does add friction to your development process.

The difficulty of analytics

DynamoDB is designed for OLTP use cases — high speed, high velocity data access where you're operating on a few records at a time. But users also have a need for OLAP access patterns — big, analytical queries over the entire dataset to find popular items, or number of orders by day, or other insights.

DynamoDB is not good at OLAP queries. This is intentional. DynamoDB focuses on being ultra-performant at OLTP queries and wants you to use other, purpose-built databases for OLAP. To do this, you'll need to get your data from DynamoDB into another system.

If you have a single table design, getting it into the proper format for an analytics system can be tricky. You've denormalized your data and twisted it into a pretzel that's designed to handle your exact use cases. Now you need to unwind that table and re-normalize it so that it's useful for analytics.

My favorite quote on this comes from Forrest Brazeal's excellent walkthrough on single-table design:

[A] well-optimized single-table DynamoDB layout looks more like machine code than a simple spreadsheet

Spreadsheets are easy for analytics, whereas a single-table design takes some work to unwind. Your data infrastructure work will need to be pushed forward in your development process to make sure you can reconstitute your table in an analytics-friendly way.

When not to use single-table design

So far, we know the pros and cons of single-table design in DynamoDB. Now it's time to get to the more controversial part — when, if ever, should you *not* use single-table design in DynamoDB?

At a basic level, the answer is "whenever the benefits don't outweigh the costs". But that generic answer doesn't help us much. The more concrete answer is "whenever I need query flexibility and/or easier analytics more than I need blazing fast performance." And I think there are two occasions where this is most likely:

- in new applications where developer agility is more important than application performance;
- in applications using GraphQL.

We'll explore each of these below. But first I want to emphasize that these are exceptions, not general guidance. When modeling with DynamoDB, you should be following best practices. This includes denormalization, single-table design, and other proper NoSQL modeling principles. And even if you opt into a multi-table design, you should understand single-table design to know why it's not a good fit for your specific application.

New applications that prioritize flexibility

In the past few years, many startups and enterprises are choosing to build on serverless compute like AWS Lambda for their applications. There are a number of benefits to the serverless model, from the ease of deployments to the painless scaling to the pay-per-use pricing model.

Many of these applications use DynamoDB as their database because of the way it fits seamlessly with the serverless model. From provisioning to pricing to permissions to the

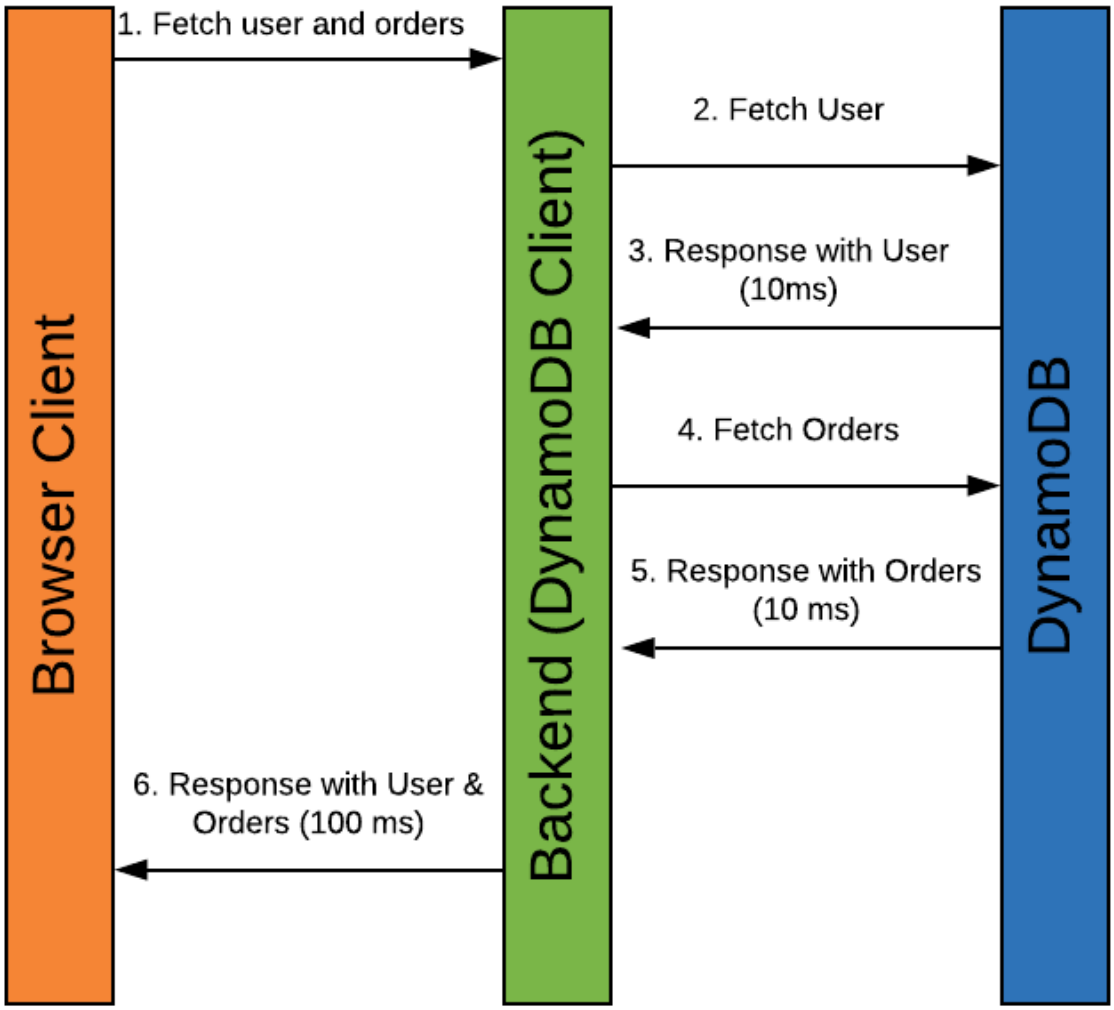
connection model, DynamoDB is a perfect fit with serverless applications, whereas traditional relational databases are more problematic.

However, it's important to remember that while DynamoDB works great *with* serverless, it was not built *for* serverless.

DynamoDB was built for large-scale, high-velocity applications that were outscaling the capabilities of relational databases. And relational databases can scale pretty darn far! If you're in the situation where you're out-scaling a relational database, you probably have a good sense of the access patterns you need. But if you're making a greenfield application at a startup, it's unlikely you absolutely require the scaling capabilities of DynamoDB to start, and you may not know how your application will evolve over time.

In this situation, you may decide that the performance characteristics of a single-table design are not worth the loss of flexibility and more difficult analytics. You may opt for a Faux-SQL approach where you use DynamoDB but in a relational way by normalizing your data across multiple tables.

This means you may need to make multiple, serial calls to DynamoDB to satisfy your access patterns. Your application may look as follows:



Notice how there are two separate requests to DynamoDB. First, there's a request to fetch the User, then there's a follow up request to fetch the Orders for the given User. Because multiple requests must be made and these requests must be made serially, there's going to be a slower response time for clients of your backend application.

For some use cases, this may be acceptable. Not all applications need to have sub-30ms response times. If your application is fine with 100ms response times, the increased flexibility and easier analytics for early-stage use cases might be worth the slower performance.

GraphQL & Single-table design

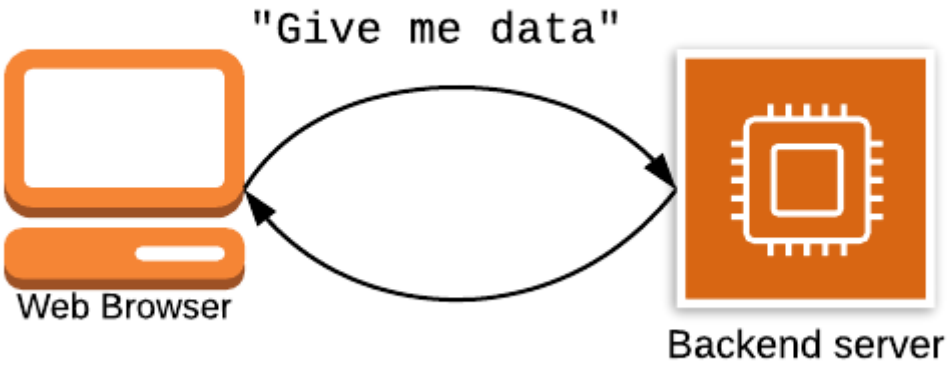
The second place where you may want to avoid single-table design with DynamoDB is in GraphQL applications.

Before I get 'Well, actually'-d to death on this one, I want to clarify that yes, I know GraphQL is an execution engine rather than a query language for a specific database. And yes, I know that GraphQL is database agnostic.

My point is not that you *cannot* use a single-table design with GraphQL. I'm saying that because of the way GraphQL's execution works, you're losing most of the benefits of a single-table design while still inheriting all of the costs.

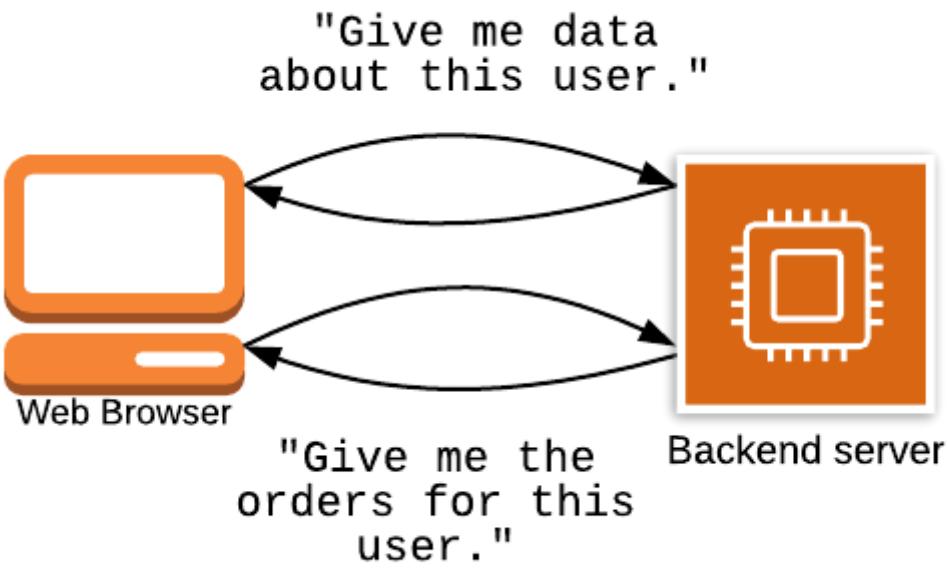
To understand why, let's take a look at how GraphQL works and one of the main problems it aims to solve.

For the past few years, many applications have opted for a REST-based API on the backend and a single-page application on the frontend. It might look as follows:



In a REST-based API, you have different *resources* which generally map to an entity in your application, such as Users or Orders. You can perform CRUD-like operations on these resources by using different HTTP verbs to indicate the operation you want to perform.

One common source of frustration for frontend developers when using REST-based APIs is that they may need to make multiple requests to different endpoints to fetch all the data for a given page:



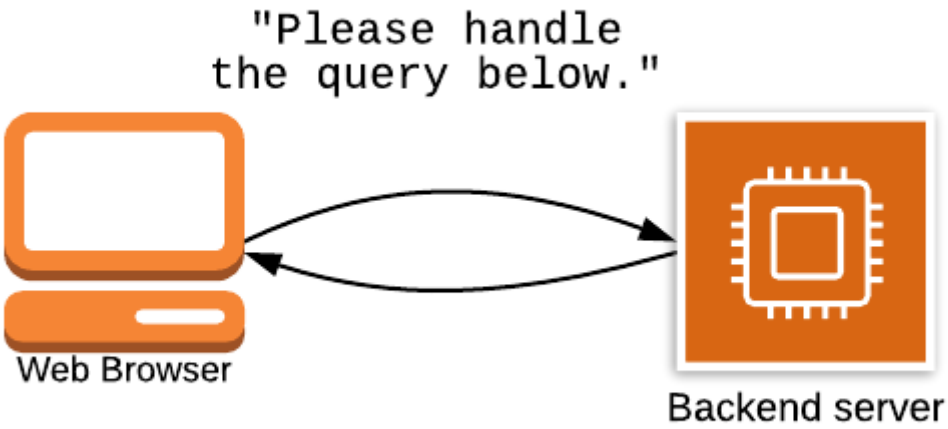
In the example above, the client has to make two requests — one to get the User, and one to get the most recent Orders for a user.

With GraphQL, you can fetch all the data you need for a page in a single request. For example, you might have a GraphQL query that looks as follows:

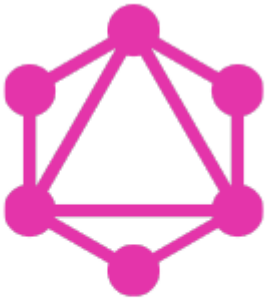
```
query { User( id:112233 ){
    firstName
    lastName
    addresses
    orders {
      orderDate
      amount
      status
    }
  }
}
```

In the block above, we're making a query to fetch the User with id 112233, then we're fetching certain attributes about the user (including firstName, lastName, and addresses), as well as all of the orders that are owned by that user.

Now our flow looks as follows:




```
query {
 User( id:112233 ){
    firstName
    lastName
    addresses
    orders {
      orderDate
      amount
      status
    }
  }
}
```



The web browser makes a single request to our backend server. The contents of that request will be our GraphQL query, as shown below the server. The GraphQL implementation will parse the query and handle it.

This looks like a win — our client is only making a single request to the backend! Hurrah!

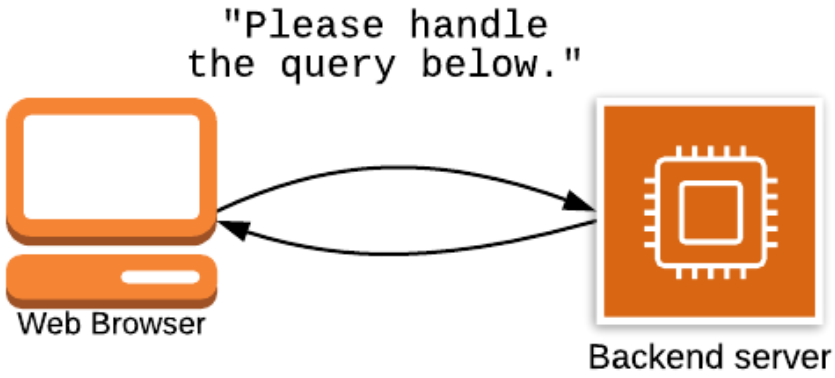
In a way, this mirrors our discussion earlier about why you want to use single-table design with DynamoDB. We only want to make a single request to DynamoDB to fetch heterogenous items, just like the frontend wants to make a single request to the backend to fetch heterogenous resources. This sounds like a match made in heaven!

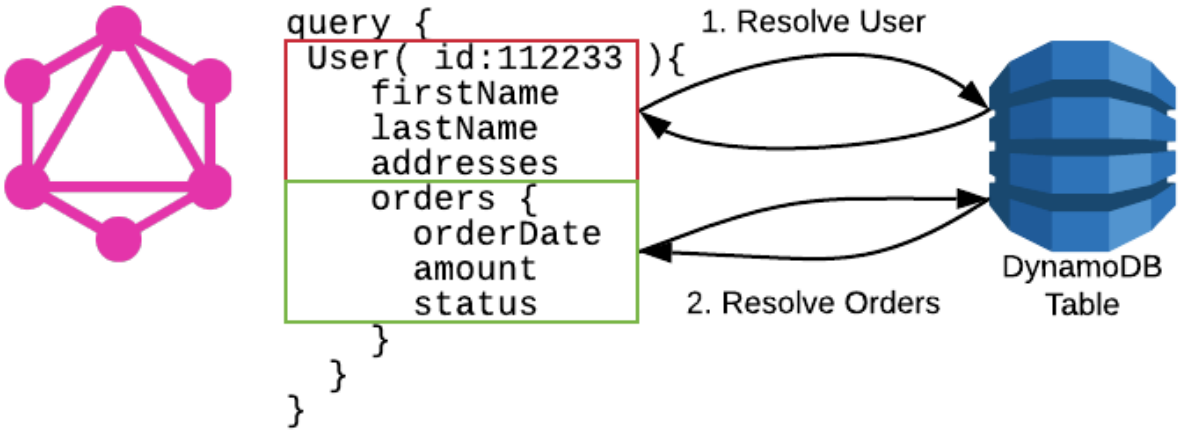
The issue is in *how* GraphQL handles those resources in the backend. Each field on each type in your GraphQL schema is handled by a resolver. This resolver understands how to fill in the data for the field.

For different types in your query, such as `User` and `Order` in our example, you would usually have a resolver that would make a database request to resolve the value. The resolver would be given some arguments to indicate which instances of that type should be fetched, and then the resolver will fetch and return the data.

The problem is that resolvers are essentially independent from each other. In the example above, the root resolver would execute first to find the User with ID 112233. This would involve a query to the database. Then, once that data is available, it would be passed to the Order resolver in order to fetch the relevant Orders for this User. This would make subsequent requests to the database to resolve those entities.

Now our flow looks like this:





In this flow, our backend is making multiple, serial requests to DynamoDB to fulfill our access pattern. This is exactly what we're trying to avoid with single-table design!

None of this goes to say that you can't use DynamoDB with GraphQL — you absolutely can. I just think it's a waste to spend time on a single-table design when using GraphQL with DynamoDB. Because GraphQL entities are resolved separately, I think it's fine to model each entity in a separate table. It will allow for more flexibility and make it easier for analytics purposes going forward.

Conclusion

In this post, we reviewed the concept of single-table design in DynamoDB. First, we went through some history on how NoSQL & DynamoDB evolved and why single-table design is necessary.

Second, we looked at some downsides to single-table design in DynamoDB. Specifically, we saw how single-table design can make it harder to evolve your access patterns and complicates your analytics work.

Finally, we looked at two situations where the benefits of single-table design in DynamoDB may not outweigh the costs. The first situation is in new, fast-evolving applications using serverless compute where developer agility is paramount. The second situation is when using GraphQL due to the way the GraphQL execution flow works.

I'm still a strong proponent of single-table design in DynamoDB in most use cases. And even if you don't think it's right for your situation, I still think you should learn and understand single-table design before opting out of it. Basic NoSQL design principles will help you even if you don't follow the best practices for large-scale design.

If you have questions or comments on this piece, feel free to leave a note below or email me directly.

Published 5 Feb 2020

AWS DynamoDB

AWS Data Hero providing training and consulting with expertise in DynamoDB, serverless applications, and cloud-native technology.
Alex DeBrie on Twitter

Anton Babenko • 8 months ago

You probably saved me couple hours of reading about GraphQL and DynamoDB patterns!

2 ^ | v • Reply • Share ›

Alex D Mod ➚ Anton Babenko • 8 months ago

Great to hear! :)

^ | v • Reply • Share ›

Aidan Lawn • 4 months ago

Thanks Alex, helpful article. Does Aurora serverless change recommendations at all? It feels like it provides best of both worlds, serverless type pricing and scalability with good old relational data modelling. Trying to choose for new project, be interested in your opinion.

1 ^ | v • Reply • Share ›

Alex D Mod ➚ Aidan Lawn • 4 months ago

Hey **@Aidan Lawn**, it doesn't change it too much for me. I think Aurora Serverless doesn't scale quite as well as I'd like for it to truly replace DynamoDB. It still has the horizontal scaling issues (and thus gradual performance reduction) of a relational database. Also, the 'serverless' scaling characteristics are a bit slow for most production use cases. AWS markets it more as a way to save on test environments or other infrequently used databases rather than as a rapidly scaling production database.

^ | v • Reply • Share ›

Antonio Presto • 8 months ago

Hi Alex, great job, thanks! Just a detail, there is no need to add a resolver to return an object within another object in graphql, the internal object can come directly from the root object.

1 ^ | v • Reply • Share ›

Mr J • 2 months ago • edited

I was just wondering what the best way to get around not being able to use CONTAINS when running queries in DynamoDB (I know you can use CONTAINS in filters). For example (using BEGINS_WITH along with the sort key) if I had a product like the Echo Dot, and I wanted the user to be able to find the that product using both the words

"Echo" and "Dot" would it be advisable to create a Gobal Secondary Index for the second word in products that have two word names? For example based on the code I've written the user would potentially search for "Dot". We would first search the table and upon finding no matches we would automatically be redirected to search the second word index. The problem I see with this model is that every time there was no match we would be running two searches regardless of whether the product's name had two words in it. Is this the best way to do this? My goal is to avoid using scans on tables or indexes that are potentially large.

Alternatively, I thought I could just list the product twice in the sort key once under "Echo" and once under "Dot" then with if both records had a matching attribute, like for example an identification number, I could filter to just the first or second match.

Reply • Share ›

Alex D Mod → Mr J • 2 months ago

Hey **@Mr J** , good question. Your use case sounds more like a search-based use case, and those usually don't work well with DynamoDB. You would probably want a different approach for that access pattern.

An e-commerce store is going to have a variety of use cases that may be served by a variety of services, depending on the size of the application. If you have multiple services, it may make sense to have certain parts, like inventory, shopping cart, and deliveries, managed by DynamoDB while having others, like product search, handled by a different database.

Let me know if that helps!

Reply • Share ›

Mr J → Alex D • 2 months ago • edited

Yeah you're probably right. I did humor the idea of search keywords as a potential sort key but the number of permutations makes this an unreasonable approach. For example say I have a product name like "Echo Dot One" composed of three words the permutations would be as follows:

Dot Echo One 0a46a90a-162d-4067-b5c3-82c268bf1ebd

Dot EchoOne 0a46a90a-162d-4067-b5c3-82c268bf1ebd

Dot One Echo 0a46a90a-162d-4067-b5c3-82c268bf1ebd

Dot OneEcho 0a46a90a-162d-4067-b5c3-82c268bf1ebd

DotEcho One 0a46a90a-162d-4067-b5c3-82c268bf1ebd

DotEchoOne 0a46a90a-162d-4067-b5c3-82c268bf1ebd

DotOne Echo 0a46a90a-162d-4067-b5c3-82c268bf1ebd

see more

Reply • Share ›

irwansyahwii • 2 months ago

I have had experiences in DynamoDB where when the user INSERT new records it will not seen from the DynamoDB console directly or from querying the database but it will shown after we wait for a while. This weird behaviour has contributed to many duplicated data in the database. I wonder if anyone knows how to overcome this issue?

Reply • Share ›

Tanaxia • 2 months ago

Great write up! I'm curious what your thoughts about how to actually parse the results from a single table database in a typed language (i.e. Golang)? Especially when fetching an entire "object" that consists of many rows of different data types.

The SortKey will have a different value for each data type of course, but how would one go about best solving how to map these to different properties in a main object from a list of many result rows?

Thanks!

Reply • Share ›

Alex D Mod → Tanaxia • 2 months ago

Good question, **@Tanaxia** ! I would say you generally have an idea of the shape of the response you'll get back -- e.g., the first item is an Organization item and the remaining items are User items. You can use that to help parse out the objects.

I generally store one object in my application in one item in Dynamo, absent special circumstances.

Does that answer your question?

Reply • Share ›

Tanaxia → Alex D • 2 months ago

Cheers!

Reply • Share ›

Zbigniew Ledwoń • 2 months ago • edited

What about the case when one Item belongs to many say Users - then you have to duplicate this Item for each User, right? This not only causes data redundancy but also Item update nightmare. But I guess this is a cost of super fast reading of such data.

Reply • Share ›

Alex D Mod → Zbigniew Ledwoń • 2 months ago

Good point, **@Zbigniew Ledwoń** ! A many-to-many relationship is different and a

little more tricky. It really depends on your needs. If the data changes frequently and would be duplicated widely, I would recommend against duplication due to the item update issue you mention. You'll likely need to use multiple requests to handle it. However, if the data is immutable or basically immutable, the duplication can work with many-to-many as well.

1 ^ | v • Reply • Share ›

Bikash • 3 months ago

Does single table design lead to a monolith architecture wherein we put other domain information in the same table? If we take example of review and rating domain, which in this case can be modeled by storing them in the same table. But is that the recommended way? In the service based (or micro-service) design, probably that would go in separate service, hence a separate table unless we want to share the same table between two services, which will be frowned upon. If one choses to store the rating and review info on separate table, should s/he also duplicate the profile data in that table again?

^ | v • Reply • Share ›

Alex D Mod → Bikash • 3 months ago

Hey **@Bikash** ! Good question.

You shouldn't think of DynamoDB as a single table for your entire application. Rather, it should be a table per service. Essentially, anywhere you would have a separate relational *database*, you should have a single DynamoDB *table*.

As for duplicating data between tables, it's the same set of considerations with a relational database. If you split up that data in a relational world, would you need to copy profile data from one database to another? If so, you'll need to do the same with DynamoDB. In general, I prefer to not duplicate data as that indicates the domains may not be as separate as you think. However, there are situations where it is required.

Hope that helps!

^ | v • Reply • Share ›

Bikash → Alex D • 3 months ago

Thanks, that's helpful.

^ | v • Reply • Share ›

Benjamin Lvovsky • 4 months ago • edited

Single table design doesn't work for a case of a need to point in time rollback for a subset of table data. For example right now we store large information fro multiple various companies, each company has huge data from financial daily data to many other. So when we need to use a great feature of point in time rollback only for one company (no way we want to mess other companies data) we just cannot do it with single table design pattern. The whole one-fits-all approach is not a good way for any software design. It all depends on the specific requirements.

Reply • Share ›

Alex D Mod ➚ Benjamin Lvovsky • 4 months ago

I completely agree with your point about one-size-fits-all. I think this post even shows it -- I state the benefits of using single-table design but also the downsides. Each person needs to weigh those for themselves.

As for your point about rollbacks, I think it's somewhat orthogonal. Your problem seems to be an issue of multi-tenancy: you can't put data from different companies in a single table. The single-table design is more about putting different types of entities (e.g. Organizations and Users and Orders) into a single table, regardless of multi-tenancy.

Reply • Share ›

Benjamin Lvovsky ➚ Alex D • 4 months ago

The way how "single-table design" is usually presented as one app - one table. Multi-tenancy is about data, and the above is for functionality per application. So it looks like the selling approach has to be changed to "one app-one table, as long as the nature of the data allows" :)
I really think no need for a push "one app-one table" as it really depends on multiple specifics. One of them this orthogonal multi-tenancy data related specifics. I am sure there will be plenty more orthogonal to the main idea or not.

Reply • Share ›

Alex D Mod ➚ Benjamin Lvovsky • 4 months ago

True, but I think you could do 'single table' (as usually conceived) in a multi-tenant app or even do 'single table' with "one-table-per-tenant".

Compare it to a relational database. If you wanted to handle multi-tenancy like you're describing in a relational database, you may have a database schema per tenant. But for each tenant/schema, there would be multiple tables (one for Organizations, one for Users, etc.). This is where DynamoDB single table is different: you're not using a table per entity like you would in relational.

And I agree we don't need to put it for everything -- I say as much in this very blog post. :) But I do think it's worth understanding the reasoning behind it and the concepts, regardless of whether you actually use that pattern.

Reply • Share ›

Cristi Scutaru • 4 months ago

Wonderful presentation. In fact, most of your articles explain complex things in a detailed but easy to understand manner. You certainly became one of my top sources of information on DynamoDB.

Reply • Share ›

Alex D Mod ➚ Cristi Scutaru • 4 months ago

Thanks, **@Cristi Scutaru** ! Glad you liked it.

Reply • Share ›

DQ • 4 months ago

Hi Alex,
Thanks for the great article, and your brilliant talk at @AWS reinvent + various other times i have seen you on AWS online talks.

Just wanted to say that twitch link is no longer valid, any chance this is available on a different link?

Thank you!

Reply • Share ›

rodrigo2 • 7 months ago

Nice job!

Reply • Share ›

Connor Leech • 9 months ago

Awesome high level overview of DynamoDB single table design. Thank you!

Reply • Share ›

Alex D Mod ➚ Connor Leech • 9 months ago

Thanks, **@Connor Leech** ! Glad it was helpful :)

Reply • Share ›

Rajnish Srivastava • 10 months ago

In the middle of discussion with Rick, he mentioned something around "selective lambda" - https://www.twitch.tv/video...

What did he really mean? Can you point me to some literature / context that can help me understand the concept? Thanks Alex.

Reply • Share ›

Alex D Mod ➚ Rajnish Srivastava • 10 months ago

Hey **@Rajnish Srivastava** , I think Rick is referring to the ability to have a Lambda process a stream that only processes certain records. For example, if you had both Users and Orders in a single table, you might be able to configure a selective Lambda that only operates on updates to User items, not Order items.

It's similar to the filtered streams proposal I mentioned in my DDB wish list: https://www.alexdebrie.com/...

Let me know if that helps!

Let me know if that helps!

Reply • Share ›

Rajnish Srivastava ➚ Alex D • 10 months ago

Thanks Alex! This makes sense.

Regards,
Rajnish

Reply • Share ›

Cesar Carlos Calderon Muro • 10 months ago

Pretty comprehensible explanation! Thanks!

Reply • Share ›