# GraphQL, DynamoDB, and Single-table Design

April 12, 2022 · 19 min read

**Alex DeBrie**
Founder, DeBrie Advisory

I've written and spoken a lot about data modeling with DynamoDB over the years. I love DynamoDB, and I feel like I understand its pros and cons pretty well.

Recently, the topic of the compatibility of GraphQL and single-table design in DynamoDB came up again, sparked by a tweet from Rick Houlihan that led to a follow-up tweet indicating that I should update my post about single-table design where I mentioned that GraphQL might be an area where you *don't* want to use single-table design with DynamoDB.

Twitter is a bad medium for nuance, and I think the question of whether to use single-table design with GraphQL is a nuanced one that depends a lot on why you are choosing to use GraphQL.

If you want the TL;DR Twitter version of this post, it is: You absolutely can use single-table design with GraphQL, and I know some very smart people who are doing so. However, I don't think it's *wrong* to chose not to, depending on your specific needs + preferences.

Further, I think Amplify made the right choice in opting for a table-per-model default.

***Post-publishing note:*** *if you want to see some good practical tips on how to use single-table design with AppSync + GraphQL, check out this video from Adam Elmore on his setup. He also has some good conceptual points on when he does (and doesn't) use single-table design principles in his application.*

If you want the full details, check out the post below. We will cover:



Let's goooooo.

## What are the benefits of GraphQL?

I want to start off by understanding the potential benefits of GraphQL in order to understand why people are choosing to use GraphQL.

Note that I refer to these as *potential* benefits. People may adopt GraphQL for all of these reasons, or there may be one or two reasons that are more important. Further, they may use GraphQL in a way that eschews one or more of the other potential benefits. I don't think these decisions are necessarily wrong -- it depends a lot on context and the particular needs of a team and product.

Finally, there are a few benefits of GraphQL that I don't mention here, such as the ability to specify the exact fields you want to save on network bandwidth. It doesn't mean these aren't important for some situations. Rather, they're not that relevant to the single-table vs. multi-table debate.

### API type safety

The first reason people like GraphQL is the type safety from the API.

Single-page applications make requests to a backend server to retrieve data to display in the application. On a shopping application, the requested data could be a list of products in a category or the contents of a customer's shopping cart. On GitHub, the requested data could be a GitHub repository or a list of issues for the repository.

In a standard JSON-based REST API, it's likely there is a schema for this data, at least theoretically. The problem comes in reality -- many frontend developers find that the returned data shape is unreliable, and developers need to implement lots of defensive coding to ensure they have the data properties they are expecting.

GraphQL helps with this problem by having a defined, typed schema that is available from the backend. Further, most GraphQL implementations will ensure that a response from the backend adheres to this schema before sending to the frontend. This makes it significantly easier as a frontend client of a GraphQL-based service as you have greater confidence over the returned data.

### Fewer network requests from the frontend

A second reason that people like GraphQL is to reduce the number of network requests from the frontend to the backend.

Let's return to our previous examples with a single-page application. If I'm implementing a shopping cart for my single-page application with a typical REST backend, I might need to make a number of calls to render the shopping cart page:

- One call to fetch the current N items in the shopping cart
- N calls to fetch up-to-date details about each item, including the current price and availability

This is the dreaded N + 1 problem. The N + 1 problem is commonly used as an argument against using ORMs as evidence they can make suboptimal queries that add additional load on your database. However, we're often making this same problem on the frontend! By using single-page applications with a RESTful backend implementation, our frontend needs to make a number of requests to render the page. Further, there may be a sequential nature to this, as I can't make the N calls to fetch the item details until my first request to retrieve the shopping cart contents. Now I have a waterfall of sequential requests that make my application seem sluggish.

GraphQL reduces this problem by allowing a frontend client to make a single request that retrieves a graph of data. In a single GraphQL query, I can retrieve both my shopping cart and all of the items in the cart. As soon as this request finishes, I can paint the entire page without waiting for subsequent requests.

Notice that this doesn't necessarily eliminate the N + 1 problem entirely. As we'll see below, it could move the N + 1 problem to the backend. The only thing GraphQL does here is eliminate the N + 1 problem *from the frontend application and developer*.

### Flexible querying patterns

The third and final benefit of GraphQL for the point of this post is that GraphQL provides more flexible querying options for clients.

We saw above that GraphQL can eliminate the N + 1 problem from the frontend. Another alternative to reduce calls from the frontend to the backend is the Backends for Frontends (BFF) architecture pattern. With this pattern, backend APIs are implemented specifically for the frontends that will be calling them. For example, we may have an endpoint specifically for rendering our shopping cart page that assembles both the cart contents and updated details about the specific items on the backend before returning a complete response to the frontend.

Once fully implemented, the BFF pattern looks a lot like GraphQL. I can get the full graph of data I need in a single request rather than by making multiple calls to my backend.

The main difference between BFFs and GraphQL is in flexibility.

BFFs are designed specifically for my frontend experiences, and thus changing my frontend experience requires changing my backend as well. This can mean pulling in additional teams, dealing with backward compatibility issues, and other factors. As a result, BFFs can be slower to evolve.

In contrast, GraphQL has provides significantly more flexibility. With GraphQL, you publish a schema that describes the shape of your data. A client can make any request against that schema they need, and the GraphQL backend implementation will do the work to assemble the data. Thus, changing requirements on the frontend (usually) do not require backend changes as well. It's as simple as rewriting the GraphQL query for a page to change the desired data.

Generally, GraphQL backends use isolated, focused resolvers that retrieve specific pieces of data. This can result in N + 1 queries on the backend. For example, imagine our shopping cart query below:

```
{
  cart(username: "alexdebrie") {
    id
    lastUpdated
    items {
      id
      quantity
      name
      sku
      price
```

```
    }
  }
}
```

The `cart` resolver would resolve details about the shopping cart overall, whereas details about the `items` would be handled by a separate resolver. On the backend, these would be handled sequentially -- after the cart resolver finishes, the items resolver would be called to fetch the items for the cart.

### Key takeaways

If you look closely at the benefits above, you'll notice that GraphQL makes life quite a bit easier for frontend developers. They don't have to be as defensive about the data they get back from an API. They don't have to make a waterfall of network requests and deal with partial errors. And they can iterate on the frontend without requiring backend assistance. Because the resolvers are implemented generically, independent of the originating query, it's easy to reshape a response simply by changing the GraphQL query input.

Because of this, I've generally seen more GraphQL adoption in engineering teams where frontend developers have more sway. **There is nothing inherently wrong with this.** Most engineering teams have some inherent bias toward a particular type of developer and optimize their systems accordingly. I think REST APIs make life a lot easier for backend developers -- by simplifyng the amount of data on any particular request -- by pushing a lot of that complexity to clients (often the frontend developers).

## DynamoDB and GraphQL

Now that we know a bit about some of the benefits of GraphQL, let's take a look at how DynamoDB and GraphQL interact. First, we'll cover some basics about DynamoDB. Then, we'll discuss whether you should use GraphQL with single-table design.

### Background on DynamoDB

I'm not going to go deep on the details of DynamoDB, but I do want to discuss some high-level points in order to help the discussion later.

DynamoDB is a NoSQL database from AWS. It optimizes for extreme predictability and consistency of performance. It wants to provide the exact same performance for a query when you have 1 MB of data as when you have 1 GB, 1 TB, or even 1 PB of data.

To do that, DynamoDB intentionally restricts how you can use it. It forces you to use a primary key for most of your access patterns. This primary key is a combination of a hash key and a B-tree to allow for fast, predictable performance when operating on an individual item or when retrieving a range of contiguous items. Because of this primary key structure, you should see single-digit millisecond response times on individual items regardless of your database size.

Additionally, DynamoDB removes certain features provided by other databases. You can't do JOINs in DynamoDB. You can't do aggregations across multiple records in DynamoDB. Both of these operations can operate on an unbounded number of items with unpredictable performance characteristics. DynamoDB will not let you write a bad query and will not provide features that allow you to do so.

Finally, DynamoDB has clear, strict limits in certain areas. It won't let you create an item over 400KB as a way to force you into smaller, more targeted items. More importantly, it has a limit on the number of concurrent reads and writes against a subset of your data. If you exceed that limit for an individual second, your request will be throttled and you'll need to try again. Again, this is in pursuit of consistency and predictability. It turns performance into a binary decision -- single-digit response that either succeeded or failed -- rather than having slowly degrading application performance under load.

Because of the choices made by the DynamoDB team in order to provide that consistency, you need to model your data differently. Most importantly, you have to think about how you will actually use your data.

In a relational database, you design your data in an abstract, normalized way. Then you implement the queries and potentially add indexes to assist in performance.

With DynamoDB, this pattern is flipped. You consider your access patterns first, then design your table to handle those access patterns. Your data probably won't be normalized as it would in a relational database. It's going to be handcrafted for your specific requirements.

Let's think back to our GitHub repository example before. Because DynamoDB does not have joins, fetching a repository and the ten most recent issues in the repository would take at least two requests to the database if you had two different tables. Depending on how you modeled the primary keys for the tables, it could be two *sequential* requests, which will increase latency. We're back to the old N + 1 problem we discussed before (or at least the 1 + 1 problem :)).

This is where single-table design comes in. If you know you have an access pattern where you need the repository and the ten most recent issues, you can assemble the items next to each other in a single DynamoDB table. Then, you can use a Query operation to fetch both the repository and the issue items in a single request. You've turned your N + 1 query into a single query, and you've done it in a way that doesn't burn through CPU like you may with a SQL join operation. (For more on single-table design in the context of one-to-many relationships, check out this post on how to model one-to-many relationships in DynamoDB).

When I talk about single-table design, I'm primarily using it to mean using DynamoDB in a way that uses the Query operation to retrieve multiple, heterogenous items in a single request. You can also use a single table with DynamoDB in a way that never retrieves heterogenous items. I often do! However, in that case, the difference between using one table or multiple tables is less stark.

### The case for (and against) single-table design with GraphQL

Now that we know about the benefits of GraphQL and of single-table design, can we merge these two in a happy marriage?

It really depends on how invested you are in flexibility as a key benefit of using GraphQL. Remember that DynamoDB relies on designing for known access patterns, while one of GraphQL's benefits is the ability to quickly iterate on access patterns. By using single-table design, you are combining multiple entities in a single table to handle patterns where you need nested, related data in a single access pattern.

Earlier, we talked about how most GraphQL implementations use isolated, focused resolvers to fetch specific pieces of data. If a complex query with nested data is requested, the GraphQL will pass it to sequential resolvers that each focus on fetching their own data.

However, this keeps the N + 1 problem we discussed in an earlier section. We moved this problem to the backend rather than the frontend, but we haven't eliminated it entirely. And even moving this to the backend can be a performance win, as the backend is closer to the database, so multiple queries can be faster there.

We can *remove* or at least significantly reduce the N + 1 problem if we want. Rather than writing focused GraphQL resolvers, we can write broader, more complex resolvers. A top-level resolver could "lookahead" at other bits of the query to see if nested data has been requested. If it has, the resolver could fetch that nested data as part of its request to the database. In a relational database, that probably means a JOIN in your query. In DynamoDB, that means utilizing the tenets of single-table design to fetch multiple, heterogenous items in a single request.

However, lookaheads in your GraphQL resolvers are controversial, and some would even go to the point of calling them an anti-pattern.

Marc-Andre Giroux, the author of Production-Ready GraphQL and the implementor of large, public GraphQL APIs at GitHub and Shopify, has the following to say about lookaheads:

**Lookaheads and Order Dependent Fields**

Some libraries allow you to fetch information on what fields are being queried under the field that is currently being resolved by your function. Be very careful with this information. It may be tempting to preemptively load data for an order's products for example. There are problems with writing resolver logic that depends on the query shape or the child fields being queried under it:

- GraphQL fields on the query root can be executed in parallel; you probably don't want to lock yourself forever by having resolver logic that is not parallelizable.
- As your schema evolves, new queries that your resolvers did not expect can start making an appearance, meaning the logic you wrote for handling subqueries is now out of date and most likely wrong.

Trying to be too smart about the execution of a query will often make things worst. The beauty of the resolver pattern is that it does one thing, and does it really well. Try to keep its logic isolated. In general, don't assume where a type or field might be used in a query. Chances are it will be reused in a different context, and the main thing that you should rely on is the object it receives. Except for mutation fields, resolvers work best as **pure functions**; they should behave as consistently as possible and have no side effects when queried on the query root. This means avoiding mutating the query context at all costs and not depending on the execution order at all.

Note his recommendation to avoid lookahead queries and ensure that your resolvers are focused and narrow. If you follow this advice, most of the advantages of single-table design in DynamoDB are gone. You should still think about how your data is

accessed and design for that, but it will be on a much narrower scope. You will think about how specific entities will be accessed, but you won't think about collections of heterogenous entities.

## Making a choice

Given the discussion above, does that mean you should or should not use single-table design with GraphQL?

It depends! Software is about tradeoffs, and you need to find the right tradeoffs for your application.

From my very unscientific research, I'd say the majority of the GraphQL community agrees with Marc-Andre Giroux's advice. You should use isolated, focused resolvers that only fetch a specific piece of data. This gives you the benefit of GraphQL's query flexibility while taking the downside of reduced performance.

My guess is that Marc-Andre's advice is partly based on his specific experience. He helped design *public-facing* GraphQL APIs for GitHub and Shopify. Because these are to be used by the public rather than primarily by internal applications, they need significantly more flexibility.

Think of the resolver complexity in the GitHub GraphQL API if you implemented lookahead functionality with full query flexibility. I looked at the GitHub GraphQL schema, and the Repository object alone is immensely complex. I wanted to count all the different relations on the Repository object, but I gave up when I was at 11 and had only made it to `discussionCategories`.

Handling the variety of potential nested queries against a Repository would be near impossible. Even if your GraphQL API is smaller than the GitHub one, it can still be a lot of complexity to take on all the permutations of data.

Additionally, note that your problem at this point is not with GraphQL and DynamoDB. Your problem is with GraphQL, period. Marc-Andre's advice is not for a specific DynamoDB implementation. It's for all GraphQL APIs, regardless of the database used. Whether you're using DynamoDB, Postgres, or Neo4J, most GraphQL APIs are making multiple, sequential requests to the database to handle a nested query.

> Note that your problem at this point is not with GraphQL and DynamoDB. Your problem is with GraphQL, period.

That said, I have seen multiple smart people advocate for single-table design principles with GraphQL. Adam Elmore has done so on Twitter, and Rich Buggy has written a nice post about single-table design with GraphQL.

In these situations, I'm guessing most of the GraphQL usage comes from clients that they control. The access patterns are known and can be planned for. They're using GraphQL for the API type safety and the fewer network requests from the frontend, but they're less concerned about infinitely flexible APIs.

In this same vein, I saw a tweet recently describing BBC's usage of GraphQL from a QCon talk. The talk noted that GraphQL queries need to be registered in advance before they go live. This sounds like known, specific access patterns to me! I don't know whether the BBC is optimizing their resolvers in conjunction with that, but it can be a nice way to get some of the benefits of GraphQL without going all-in on flexible queries.

In sum, you need to decide what's more important to you -- system performance or flexibility. I can't tell you what's more important because it depends a lot on your context, team, and product.

## Did the Amplify team make a mistake in its implementation?

Now that I've (hopefully) convinced you that single-table and multi-table are both acceptable choices with GraphQL, I do want to discuss a final question that spurred this whole discussion.

AWS Amplify is a toolkit on top of AWS AppSync that makes it easy to turn a GraphQL schema into a fully managed GraphQL server, complete with databases and even the resolver code (or code-like configuration). As part of its implementation, it creates a separate DynamoDB table for each object (denoted by an `@model` directive) in the GraphQL schema.

Given that single-table and multi-table are both acceptable, was it right for the Amplify team to push the defaults toward multi-table? I believe yes, for two reasons.

> I believe the Amplify team made the correct decision to default toward multi-table design.

First, as mentioned above, isolated, focused resolvers appear to be the preferred option in the GraphQL community. It's not unanimous -- you can find examples of using lookaheads to enhance performance in GraphQL -- but that's the vibe I get.

Second, single-table design is playing on God-mode, and Amplify is designed to be a developer-friendly experience. Many people using Amplify are not building Amazon-scale services that require extreme performance. They're probably building MVPs and trying to iterate quickly to get traction. There may be a time for optimization later, but it's not on day 1.

Further, nothing precludes you from using single-table design with AppSync. As discussed, Rick has shown how to write a single-table resolver, and Rich Buggy has written a great post on this. There's a bit of asymmetry here -- someone that knows single-table design is knowledgeable to opt out of the defaults, whereas someone completely new to DynamoDB would be baffled by single-table design and would likely drop without knowing that multi-table is an option.

None of this is to excuse complaints of the difficulty of learning single-table design or new things generally. But given that I believe multi-table is a valid option for AppSync, it's a defensible choice to default to that mode.

Finally, if we're picking nits, my complaint about Amplify is that it makes it too easy to accidentally implement patterns using an inefficient DynamoDB Scan operation, which can blow up once an application goes to production.

## Conclusion

I hope this post has convinced you that both single-table and multi-table design are acceptable with GraphQL and DynamoDB. More broadly, I hope it's shown that GraphQL itself has a number of benefits and some downsides, and you need to pick what is important to you in your implementation.

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

**Recent posts**

How you should think about DynamoDB costs

Event-Driven Architectures vs. Event-Based Compute in Serverless Applications

Why I (Still) Like the Serverless Framework over the CDK

Key Takeaways from the DynamoDB Paper

Understanding Eventual Consistency in DynamoDB

Get the DynamoDB Book

**Tags:** AWS DynamoDB