How to build an AppSync API using a single table DynamoDB design


Rich

May 15, 2020

If you want to start a flame war in the AWS AppSync community then ask if you should use a single or multiple table design for DynamoDB. There's no shortage of opinions in both directions :fire::fire::fire:

In this article I'm going to show you how I built a generic single table DynamoDB solution for AWS AppSync.

Historical Context

Before I explain "how" it's important to remember that DynamoDB did not support on-demand capacity (pay per use pricing) when AppSync was originally released. Managing the read and write capacity across multiple tables was a lot of extra work. Under provisioning could result in your API failing and over provisioning was expensive.

By using a single table I could reduce the amount of time spent managing table capacity and save a lot of money. That is why began exploring how to build an AppSync API using only a single DynamoDB table.

Primary Key

For the tables primary key I used a generic attribute named `PK` that *does not* appear in the GraphQL schema. Keeping the primary key for the table out of the GraphQL schema allows me to use any field in the GraphQL type as the primary ID for a record.

Note: To keep my examples simple I will use `id` as the primary field but you can use a different field or fields.

Mapping the primary ID for the GraphQL type to `PK` is handled in the AppSync resolver request template. There are two patterns I recommend when mapping fields to `PK`.

1. `NODE#${id}` if you have globally unique ID's
2. `${__typename}#${id}` if you do not have globally unique ID's

Example: Let's start with a simple GraphQL schema for farms.

```
type Farm {
  id: ID!
  name: String
}

input CreateFarmInput {
  name: String!
}

type Mutation {
  createFarm(input: CreateFarmInput!): Farm
}

type Query {
  findFarm(id: ID!): Farm
}
```

The request resolver template for `createFarm` would be

```
#set( $id = $util.autoId() )
$util.qr($ctx.args.input.put("id", $id))
$util.qr($ctx.args.input.put("__typename", "Farm"))

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "PK": $util.dynamodb.toDynamoDBJson("NODE#$id"),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

This template generates an `id` for the new farm that is added to the input along with the `__typename`. It then prefixes that ID with `NODE#` to generate the `PK` value for the DynamoDB key.

When invoked with the following input

```
mutation {
  createFarm(input: { name: "Old MacDonald's" }) {
    id
  }
}
```

It will create a record in DynamoDB similar to

PK	__typename	id	name
NODE#1234	Farm	1234	Old MacDonald's

The farm can then be retrieved using `findFarm` that accepts an `id` which is mapped to the `PK` attribute in the request template:

```
{
  "version": "2017-02-28",
  "operation": "GetItem",
  "key": {
    "PK": $util.dynamodb.toDynamoDBJson("NODE#$ctx.args.id"),
  }
}
```

SECURITY WARNING

When using the `NODE#${id}` pattern you **must** check the `__typename` is one that you expect in **every** resolver that fetches, updates or deletes records. Failure to do this may allow someone to by pass security by using the ID from one type with the resolver for a different type that is more permissive.

The examples in this article **do not** include this check because I am focusing on the DynamoDB access pattern and not AppSync access controls.

Unique Secondary Fields

Requiring additional fields in our GraphQL type to be unique is a common problem. The `Farm` type contains an `id` field that uniquely identifies a record and never changes. It also has a `name` field that can change but must be unique.

With DynamoDB only the primary key on the table is unique. We can't add this constraint to other attributes. To solve this I can use `TransactionWriteItems` to write a record under different keys. The first key is the `NODE#${id}` as previously discussed. The second key uses the pattern `${__typename}#${fieldname}#${value}`.

The `createFarm` resolver request template will now look like

```
#set( $id = $util.autoId() )
$util.qr($ctx.args.input.put("id", $id))
$util.qr($ctx.args.input.put("__typename", "Farm"))

{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "YOUR-TABLE-NAME",
      "operation": "PutItem",
      "key": {
        "PK": $util.dynamodb.toDynamoDBJson("NODE#$id")
      },
      "attributeValues": $util.dynamodb.toMapValuesJson($ctx.args.input),
      "condition": {
        "expression": "attribute_not_exists(PK)",
        "returnValuesOnConditionCheckFailure": false
      }
    },
    {
      "table": "YOUR-TABLE-NAME",
      "operation": "PutItem",
      "key": {
        "PK": $util.dynamodb.toDynamoDBJson("Farm#name#$ctx.args.input.name.toLowerCase()")
      },
      "attributeValues": $util.dynamodb.toMapValuesJson($ctx.args.input),
      "condition": {
        "expression": "attribute_not_exists(PK)",
        "returnValuesOnConditionCheckFailure": false
      }
    },
  ]
}
```

Because I need to use the `2018-05-29` resolver template version for `TransactionWriteItems` and I only want to return one record my resolver response template is a little more complicated

```
#if ( $ctx.error )
  $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.args.input)
```

The main thing to understand is that I'm returning the `$ctx.args.input` instead of `$ctx.result`. This contains our `Farm` type with the `id` and `__typename` set correctly by the request template.

In DynamoDB the records will be stored as:

PK	__typename	id	name
NODE#1234	Farm	1234	Old MacDonald's
Farm#name#old macdonald's	Farm	1234	Old MacDonald's

If I call the `createFarm` mutation twice with the same name it will now fail because a record with the key `Farm#name#old macdonald's` already exists.

You may have noticed that I converted the name to lower case before using it in the `PK` attribute for the second record. This prevents people from getting around the unique name restriction by using different combinations of upper and lower case letters. The `name` attribute itself still retains the original case and that's the value the user will see when the record is queried via GraphQL.

As a side benefit I now have a fast way to retrieve a `Farm` record from the `name` using `GetItem`. This is useful if you have a natural key like email address or ISBN that you want to fetch records by.

One to One Relationships

One to One relationships are simple to implement. Let's add a `logo` to the `Farm` that returns an `Image` type.

```
type Image {
  id: ID!
  url: String
}

type Farm {
  id: ID!
  name: String
  logo: Image
}
```

I will also add a `logoId` field to the input for `createFarm`

```
mutation {
  createFarm(input: { name: "Old MacDonald's", logoId: "1111" }) {
    id
  }
}
```

In DynamoDB this will be saved as

PK	__typename	id	imageId	name	url
NODE#1234	Farm	1234	1111	Old MacDonald's	
NODE#1111	Image	1111			logo.gif

When the request resolver template for `logo` is executed the DynamoDB record for the farm is available in `$ctx.source`. This allows us to perform a `GetItem` with the correct `PK` value to retrieve the `Image`.

```
{
  "version": "2017-02-28",
  "operation": "GetItem",
  "key": {
    "PK": $util.dynamodb.toDynamoDBJson("NODE#$ctx.source.logoId"),
  }
}
```

Traversing both directions in a one to one relationship requires storing the ID for the opposite side in each record. In our example this means storing the `imageId` with the `Farm` and `farmId` with the `Image`.

Many to One Relationships

Many to One relationships work almost exactly the same as one to one relationships. On the many side you need to store the ID for the one side of the relationship.

Let's expand our farm to include cow's.

```
type Cow {
  id: ID!
  name: String
  farm: Farm
}

input CreateCowInput {
  name: String!
  farmId: ID!
}

type Farm {
  id: ID!
  name: String
}

input CreateFarmInput {
  name: String!
}

type Mutation {
  createCow(input: CreateCowInput!): Cow
  createFarm(input: CreateFarmInput!): Farm
}

type Query {
  findFarm(id: ID!): Farm
}
```

To create a new cow we would execute a mutation like

```
mutation {
  createCow(input: { name: "Bessie", farmId: "5678" }) {
    id
  }
}
```

The request resolver template for `createCow` is the same as `createFarm` with the `__typename` changed from `Farm` to `Cow`.

```
#set( $id = $util.autoId() )
$util.qr($ctx.args.input.put("id", $id))
$util.qr($ctx.args.input.put("__typename", "Cow"))

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "PK": $util.dynamodb.toDynamoDBJson("NODE#$id"),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

This will create a DynamoDB record similar to:

PK	__typename	id	farmId	name
NODE#1234	Cow	1234	5678	Bessie
NODE#5678	Farm	5678		Old MacDonald's

Our request resolver for the `farm` field on the `Cow` type would be:

```
{
  "version": "2017-02-28",
  "operation": "GetItem",
  "key": {
    "PK": $util.dynamodb.toDynamoDBJson("NODE#$ctx.source.farmId"),
  }
}
```

One to Many Relationships

One to Many relationships are more complicated and require writing additional attributes to the DynamoDB record. To implement these I use generic GSI's named `GSI01`, `GSI02`, `GSI03`, etc. Each of these will have a corresponding partition key `PK01`, `PK02`, `PK03`, etc and sort key `SK01`, `SK02`, `SK03`. The additional data for these attributes is added by the resolver request template during the create/update mutation.

In `PKxx` you typically write the value `${__typename}#${othersideId}` where `__typename` is the name of the type you're writing data for (the many side) and `othersideId` is the ID for the one side. The value for `SKxx` will depend on how you want sort the results when traversing from the one to the many. I'll explain this further down. All of this should be done in the resolver template so these details are not exposed in the GraphQL schema.

Let's make a tiny change to our farm schema adding a connection between `Farm` and `Cow`.

```
type Farm {
  id: ID!
  name: String
  cows: [Cow]
}
```

Everything else will remain the same.

When retrieving the cows from the farm I want them sorted by name. To do this I'll copy the `name` value into `SK01`. With the extra information our `createCow` request template now looks like

```
#set( $id = $util.autoId() )
$util.qr($ctx.args.input.put("id", $id))
$util.qr($ctx.args.input.put("__typename", "Cow"))
$util.qr($ctx.args.input.put("PK01", "Cow#$ctx.args.input.farmId"))
$util.qr($ctx.args.input.put("SK01", $ctx.args.input.name))

{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "PK": $util.dynamodb.toDynamoDBJson("NODE#$id"),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

When create cow is called with

```
mutation {
  createCow(input: { name: "Bessie", farmId: "5678" }) {
    id
  }
}
```

The cow DyanmoDB record is now written as:

PK	__typename	id	farmId	name	PK01	SK01
NODE#1234	Cow	1234	5678	Bessie	Farm#5678	Bessie
NODE#5678	Farm	5678		Old MacDonald's		

To get from farm to cow we want the `cows` resolver request template to `Query` the `GS01` index for a `PK01` value that is the farms ID prefixed by the return type. Our `cows` resolver request template is

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "index": "GS01",
    "query" : {
        "expression": "PK01 = :pk",
        "expressionValues" : {
            ":pk" : $util.dynamodb.toDynamoDBJson("Cow#$ctx.source.id")
        }
    }
}
```

Let's continue expanding the farm by adding chickens.

```
type Chicken {
  id: ID!
  name: String
  farm: Farm
}

type Cow {
  id: ID!
  name: String
  farm: Farm
}

input CreateChickenInput {
  name: String!
  farmId: ID!
}

input CreateCowInput {
  name: String!
  farmId: ID!
}

type Farm {
  id: ID!
  name: String
  chickens: [Chicken]
  cows: [Cow]
}

type Mutation {
  createChicken(input: CreateChickenInput!): Chicken
  createCow(input: CreateCowInput!): Cow
}

type Query {
  farm(id: ID!): Farm
}
```

You can copy the resolver request templates from `createCow` to `createChicken` and `cows` to `chickens`. Just remember to replace `Cow` with `Chicken` inside the templates.

What you might have noticed is that I did not need to create a new GSI. By prefixing the `farmId` with the `__typename` we are preventing a data collision which allows us to use the same GSI once for each GraphQL type. This slows the growth in GSI's because the number of GSI's for the table is determined by the GraphQL type using the largest number of GSI's.

The `${returnTypename}#${context.source.id}` pattern also works when a field returns multiple types. Let's modify the schema by adding `Animal` which can be a `Chicken` or `Cow`.

```
union Animal = Chicken | Cow

type Chicken {
  id: ID!
  name: String
  farm: Farm
}

type Cow {
  id: ID!
  name: String
  farm: Farm
}

input CreateChickenInput {
  name: String!
  farmId: ID!
}

input CreateCowInput {
  name: String!
  farmId: ID!
}

type Farm {
  id: ID!
  name: String
  chickens: [Chicken]
  cows: [Cow]
  animals: [Animal]
}

type Mutation {
  createChicken(input: CreateChickenInput!): Chicken
  createCow(input: CreateCowInput!): Cow
}

type Query {
  farm(id: ID!): Farm
}
```

To make this work I need a second GSI's. In `PK02` I'll store `Animal#${farmId}` using the `createChicken` and `createCow` resolver request templates. Here's what `createCow` now looks like

```
#set( $id = $util.autoId() )
$util.qr($ctx.args.input.put("id", $id))
$util.qr($ctx.args.input.put("__typename", "Cow"))
$util.qr($ctx.args.input.put("PK01", "Cow#$ctx.args.input.farmId"))
$util.qr($ctx.args.input.put("SK01", $ctx.args.input.name))
$util.qr($ctx.args.input.put("PK02", "Animal#$ctx.args.input.farmId"))
$util.qr($ctx.args.input.put("SK02", $ctx.args.input.name))

{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "PK": $util.dynamodb.toDynamoDBJson("NODE#$id"),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

The `animals` resolver request template on the `Farm` type will look similar to `cows` or `chickens` but it will use the prefix `Animal#` and the `GS02` index.

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "index": "GS02",
  "query" : {
      "expression": "PK02 = :pk",
      "expressionValues" : {
        ":pk" : $util.dynamodb.toDynamoDBJson("Animal#$ctx.source.id")
      }
  }
}
```

This will return data for both `Cow` and `Chicken` types sorted by the animal name. AppSync can tell which type each record is by looking at the `__typename` value that was written when it was created.

Before finishing I want to share a couple of tricks that you might find helpful.

1. Storing the lower case value of a field in `SKxx` gives case insensitive sorting while still being able to show the value as originally provided.
2. Using a default value like `AAAAAAAAAAA` or `ZZZZZZZZZZZ` in `SKxx` when a field is `null` allows you to group entries without a value at the top or bottom of a list that is otherwise sorted.
3. Adding prefixes to the `SKxx` value can be used to group sorted results. An example of this would be prefixing the name of active users with `A` and inactive with `I`.
4. Conditionally writing attributes allows you to create sparse indexes when you want to exclude records from the result. An example of this might be using `GS01` to access all cows that the farm has ever owned and `GS02` to access only cows that are still here.

Many to Many Relationships

Many to Many relationships with DynamoDB and AppSync can be problematic. Where possible you should avoid them. This can be done by:

1. Denormalizing data. For example: Store the state and country name as part of an address record instead of storing ID's then using resolvers to look up those records.
2. Perform the join in the application. For example: A CMS might have a many to many relationship between content and content types but it only has a small number of content types. You could load all of the content types through a list operation (using a DynamoDB `Query`) then match the content to the content type in the application.

If you cannot avoid many to many joins then options include:

1. Using 1-M and M-1 relationships with an intermediate type (recommended by the Amplify team). This works but it should come with a **big** warning because you are almost certainly introducing the N+1 problem. On the 1-M side you'll be using an efficient `Query` operation but on the M-1 side you're probably performing M x `GetItem` operations.
2. If you only have a small number of connections between two items and you only access it in one direction then you might be able to use a DynamoDB list attribute with `BatchGetItem`. Example: I might have an `Image` type that stores a larger number of images. Each user needs to pick 3 images. In the DynamoDB record for the user I could store the selected images ID's in a list attribute.

Data Migrations

As your GraphQL types change you will need to migrate data. Typically when this happens you need to find every record of a certain type then update it. Two hacks I've used to reduce the frequency of these updates are:

1. Returning `null` and forcing the client to deal with `null` in a sensible way.
2. Using the resolver response template to transform the DynamoDB data before sending it to the client.

At some point you're going to need to update data inside DynamoDB. To perform migrations I wrote a script that loads the entire table into memory as an array using the DynamoDB `Scan` operation. It then looped over every record passing it to a migration function based on the `__typename` attribute (if found). This function returns the DynamoDB operations that need to be performed to upgrade the record or `null` if there are none.

This is a simple solution but perfect for adding, removing and updating data.

Comparison to Multiple Table Design

So how does this compare to a multiple table solution like Amplify?

There's surprisingly very little difference.

1. The number and type of DynamoDB operations is almost the same. The only difference is that you should not `Scan` a single table solution to return all records of one type.
2. With on-demand capacity there is almost no price difference.
3. Both are likely to require updating items when making changes to the schema and can use the hacks I suggested.

The big differences are:

1. The number of tables you need to manage/monitor.
2. Multiple return types work better with single table.
3. Adding and removing DyanmoDB GSI's occurs less frequently with the single table design.

Can You Build a Better Single Table Solution?

Yes.

This solution is designed to be generic and contains compromises to achieve that. By looking at your access patterns you can design a solution that is more efficient.

What Next?

I've already started work on version 2 that will include:

1. Better support for many to many relations
2. Delta Sync support
3. Amplify data store support

I'm also working on a tool to generate a single table SAM backend.

If you want to know more then subscribe to my mailing list below or over at Graphboss.

What do you think?

2 Responses

Upvote Funny Love Surprised Angry Sad

2 Comments **Going Serverless** **Disqus' Privacy Policy** **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Mark Hayward • 2 months ago

Hi Rich
This is a fantastic article. I just wondered, 6 months on, is this something you would still recommend doing? I see most people are still going for the multiple-table approach Amplify defaults to. I am about to start a new project and I am considering the best approach.
Also, did you finish the SAM template for this design?
Many Thanks!

Reply • Share ›

Rich Buggy Mod ➦ Mark Hayward • 2 months ago

Hey **@Mark Hayward**

I absolutely still recommend single table for AWS AppSync. It feels like a lot of people are coming around to this now that a few people have documented how to do it.

Unfortunately I don't have have a SAM template for this code.

Good luck.

Reply • Share ›

Going Serverless
rich@goingserverless.com

Building serverless apps using TypeScript/JavaScript, AWS Amplify and React with an Amazon Web Services (AWS) backend (Lambda, AppSync, API Gateway, S3, DynamoDB, SNS and more).