All Articles

How to model one-to-many relationships in DynamoDB

DynamoDB is sometimes considered just a simple key-value store, but nothing could be further from the truth. DynamoDB can handle complex access patterns, from highly-relational data models to time series data or even geospatial data.

In this post, we'll see how to model one-to-many relationships in DynamoDB. One-to-many relationships are at the core of nearly all applications. In DynamoDB, you have a few different options for representing one-to-many relationships.

We'll cover the basics of one-to-many relationships, then we'll review five different strategies for modeling one-to-many relationships in DynamoDB:

1. Denormalization by using a complex attribute
2. Denormalization by duplicating data
3. Composite primary key + the Query API action
4. Secondary index + the Query API action
5. Composite sort keys with hierarchical data

Let's get started!

Basics of one-to-many relationships

A one-to-many relationship occurs when a particular object is the owner or source for a number of sub-objects. A few examples include:

- **Workplace:** A single office will have many employees working there; a single manager may have many direct reports.
- **E-commerce:** A single customer may make multiple orders over time; a single order may be comprised of multiple items.
- **Software-as-a-Service (SaaS) accounts:** An organization will purchase a SaaS subscription; multiple users will belong to one organization.

With one-to-many relationships, there's one core problem: how do I fetch information about the parent entity when retrieving one or more of the related entities?

In a relational database, there's essentially one way to do this—using a foreign key in one table to refer to a record in another table and using a SQL join at query time to combine the

two tables.

There are no joins in DynamoDB. Instead, there are a number of strategies for one-to-many relationships, and the approach you take will depend on your needs.

In this post, we will cover five strategies for modeling one-to-many relationships with DynamoDB:

- Denormalization by using a complex attribute
- Denormalization by duplicating data
- Composite primary key + the Query API action
- Secondary index + the Query API action
- Composite sort keys with hierarchical data

We will cover each strategy in depth below—when you would use it, when you wouldn't use it, and an example. The end of the post includes a summary of the five strategies and when to choose each one.

Denormalization by using a complex attribute

Database normalization is a key component of relational database modeling and one of the hardest habits to break when moving to DynamoDB.

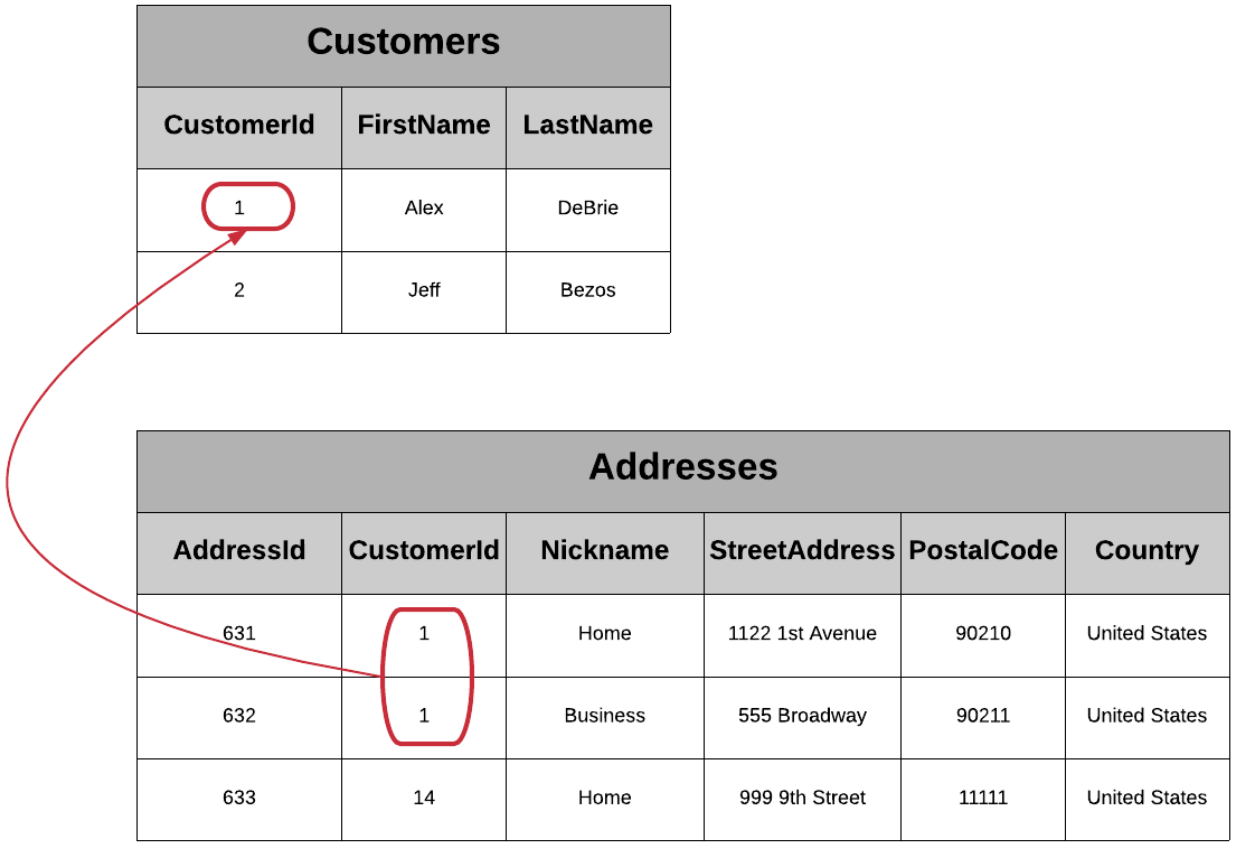
You can read the basics of normalization elsewhere, but there are a number of areas where *denormalization* is helpful with DynamoDB.

The first way we'll use denormalization with DynamoDB is by having an attribute that uses a complex data type, like a list or a map. This violates the first tenet of database normalization: to get into first normal form, each attribute value must be atomic. It cannot be broken down any further.

Let's see this by way of an example. Imagine we have an e-commerce site where there are Customer entities that represent people that have created an account on our site. A single Customer can have multiple mailing addresses to which they may ship items. Perhaps I have

one address for my home, another address for my workplace, and a third address for my parents (a relic from the time I sent them a belated anniversary present).

In a relational database, you would model this with two tables using a foreign key to link the tables together, as follows:



Customers

CustomerId	FirstName	LastName
1	Alex	DeBrie
2	Jeff	Bezos

Addresses

AddressId	CustomerId	Nickname	StreetAddress	PostalCode	Country
631	1	Home	1122 1st Avenue	90210	United States
632	1	Business	555 Broadway	90211	United States
633	14	Home	999 9th Street	11111	United States

Notice that each record in the `Addresses` table includes a `CustomerId`, which identifies the Customer to which this Address belongs. You can use the join operation to follow the pointer to the record and find information about the Customer.

DynamoDB works differently. Because there are no joins, we need to find a different way to assemble data from two different types of entities. In this example, we can add a `MailingAddresses` attribute on our Customer item. This attribute is a map and contains all addresses for the given customer:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
CUSTOMER #alexdebrie	CUSTOMER #alexdebrie	FirstName	LastName	MailingAddresses
		Alex	DeBrie	{ "Home": { "StreetAddress: "1122 1st Avenue", "PostalCode": "90210", "Country": "UnitedStates" }, "Business": { "StreetAddress: "555 Broadway", "PostalCode": "90211", "Country": "UnitedStates" } }
CUSTOMER #jeffbezos	CUSTOMER #jeffbezos	FirstName	LastName	MailingAddresses
		Jeff	Bezos	{ "Home": { "StreetAddress: "123 Spruce Drive", "PostalCode": "69361", "Country": "UnitedStates" } }

Because `MailingAddresses` contains multiple values, it is no longer atomic and thus violates the principles of first normal form.

There are two factors to consider when deciding whether to handle a one-to-many relationship by denormalizing with a complex attribute:

- **Do you have any access patterns based on the values in the complex attribute?**

  All data access in DynamoDB is done via primary keys and secondary indexes. You cannot use a complex attribute like a list or a map in a primary key. Thus, you won't be able to make queries based on the values in a complex attribute.

  In our example, we don't have any access patterns like "Fetch a Customer by his or her mailing address". All use of the `MailingAddress` attribute will be in the context of a Customer, such as displaying the saved addresses on the order checkout page. Given these needs, it's fine for us to save them in a complex attribute.

- **Is the amount of data in the complex attribute unbounded?**

  A single DynamoDB item cannot exceed 400KB of data. If the amount of data that is contained in your complex attribute is potentially unbounded, it won't be a good fit for denormalizing and keeping together on a single item.

In this example, it's reasonable for our application to put limits on the number of mailing addresses a customer can store. A maximum of 20 addresses should satisfy almost all use cases and avoid issues with the 400KB limit.

But you could imagine other places where the one-to-many relationship might be unbounded. For example, our e-commerce application has a concept of Orders and Order Items. Because an Order could have an unbounded number of Order Items (you don't want to tell your customers there's a maximum number of items they can order!), it makes sense to split Order Items separately from Orders.

If the answer to either of the questions above is "Yes", then denormalization with a complex attribute is not a good fit to model that one-to-many relationship.

Denormalization by duplicating data

In the strategy above, we denormalized our data by using a complex attribute. This violated the principles of first normal form for relational modeling. In this strategy, we'll continue our crusade against normalization.

Here, we'll violate the principles of second normal form by duplicating data across multiple items.

In all databases, each record is uniquely identified by some sort of key. In a relational database, this might be an auto-incrementing primary key. In DynamoDB, this is the primary key.

To get to second normal form, *each non-key attribute must depend on the whole key*. This is a confusing way to say that data should not be duplicated across multiple records. If data is duplicated, it should be pulled out into a separate table. Each record that uses that data should refer to it via a foreign key reference.

Imagine we have an application that contains Books and Authors. Each Book has an Author, and each Author has some biographical information, such as their name and birth year. In a relational database, we would model the data as follows:

Authors		
AuthorId	**AuthorName**	**AuthorBirthdate**
1	John Grisham	February 8, 1955
2	Stephen King	September 21, 1947
3	J.K. Rowling	July 31, 1965

Books			
BookId	**AuthorId**	**BookTitle**	**ReleaseYear**
1	2	The Shining	1977
2	2	It	1986
3	3	Harry Potter and the Sorcerer's Stone	1997

Note: In reality, a book can have multiple authors. For simplification of this example, we're assuming each book has exactly one author.

This works in a relational database as you can join those two tables at query-time to include the author's biographical information when retrieving details about the book.

But we don't have joins in DynamoDB. So how can we solve this? We can ignore the rules of second normal form and include the Author's biographical information on each Book item, as shown below.

Primary key		Attributes	
Partition key: AuthorName	**Sort key: BookName**		
Stephen King	It	AuthorBirthdate	ReleaseYear
		September 21, 1947	1986
	The Shining	AuthorBirthdate	ReleaseYear
		September 21, 1947	1977
J.K. Rowling	Harry Potter and the Sorcerer's Stone	AuthorBirthdate	ReleaseYear
		July 31, 1965	1997

Notice that there are multiple Books that contain the biographical information for the Author Stephen King. Because this information won't change, we can store it directly on the Book item itself. Whenever we retreive the Book, we will also get information about the parent Author item.

There are two main questions you should ask when considering this strategy:

- *Is the duplicated information immutable?*

- *If the data does change, how often does it change and how many items include the duplicated information?*

In our example above, we've duplicated biographical information that isn't likely to change. Because it's essentially immutable, it's OK to duplicate it without worrying about consistency issues when that data changes.

Even if the data you're duplicating does change, you still may decide to duplicate it. The big factors to consider are how often the data changes and how many items include the duplicated information.

If the data changes fairly infrequently and the denormalized items are read a lot, it may be OK to duplicate to save money on all of those subsequent reads. When the duplicated data does change, you'll need to work to ensure it's changed in all those items.

Which leads us to the second factor—how many items contain the duplicated data. If you've only duplicated the data across three items, it can be easy to find and update those items when the data changes. If that data is copied across thousands of items, it can be a real chore to discover and update each of those items, and you run a greater risk of data inconsistency.

Essentially, you're balancing the benefit of duplication (in the form of faster reads) against the costs of updating the data. The costs of updating the data includes both factors above. If the costs of either of the factors above are low, then almost any benefit is worth it. If the costs are high, the opposite is true.

Composite primary key + the Query API action

The next strategy to model one-to-many relationships—and probably the most common way—is to use a composite primary key plus the Query API to fetch an object and its related sub-objects.

A key concept in DynamoDB is the notion of *item collections*. Item collections are all the items in a table or secondary index that share the same partition key. When using the Query

API action, you can fetch multiple items within a single item collection. This can include items of different types, which gives you join-like behavior with much better performance characteristics.

Let's use one of the examples from the beginning of this section. In a SaaS application, Organizations will sign up for accounts. Then, multiple Users will belong to an Organization and take advantage of the subscription.

Because we'll be including different types of items in the same table, we won't have meaningful attribute names for the attributes in our primary key. Rather, we'll use generic attribute names, like `PK` and `SK`, for our primary key.

We have two types of items in our table—Organizations and Users. The patterns for the `PK` and `SK` values are as follows:

Entity	PK	SK
Organizations	`ORG#<OrgName>`	`METADATA#<OrgName>`
Users	`ORG#<OrgName>`	`User#<UserName>`

The table below shows some example items:

Primary key			
Partition key: PK	**Sort key: SK**		
ORG#MICROSOFT Item collection	METADATA#MICROSOFT Organization Item	OrgName	PlanType
		Microsoft	Enterprise
	USER#BILLGATES User items	UserName	UserType
		Bill Gates	Member
	USER#SATYANADELLA	UserName	UserType
		Satya Nadella	Admin
ORG#AMAZON	METADATA#AMAZON	OrgName	PlanType
		Amazon	Pro
	USER#JEFFBEZOS	UserName	UserType
		Jeff Bezos	Admin

In this table, we've added five items—two Organization items for Microsoft and Amazon, and three User items for Bill Gates, Satya Nadella, and Jeff Bezos.

Outlined in red is the item collection for items with the partition key of `ORG#MICROSOFT`. Notice how there are two different item types in that collection. In green is the Organization item type in that item collection, and in blue is the User item type in that item collection.

This primary key design makes it easy to solve four access patterns:

1. **Retrieve an Organization.** Use the GetItem API call and the Organization's name to make a request for the item with a PK of `ORG#<OrgName>` and an SK of `METADATA#<OrgName>`.

2. **Retrieve an Organization and all Users within the Organization.** Use the Query API action with a key condition expression of `PK = ORG#<OrgName>`. This would retrieve the Organization and all Users within it as they all have the same partition key.

3. **Retrieve only the Users within an Organization.** Use the Query API action with a key condition expression of `PK = ORG#<OrgName> AND begins_with(SK, "USER#")`. The use of the `begins_with()` function allows us to retrieve only the Users without fetching the Organization object as well.

4. **Retrieve a specific User.** If you know both the Organization name and the User's username, you can use the GetItem API call with a PK of `ORG#<OrgName>` and an SK of `USER#<Username>` to fetch the User item.

While all four of these access patterns can be useful, the second access pattern—Retrieve an Organization and all Users within the Organization—is most interesting for this discussion of one-to-many relationships. Notice how we're emulating a join operation in SQL by locating the parent object (the Organization) in the same item collection as the related objects (the Users). We are pre-joining our data by arranging them together at write time.

This is a pretty common way to model one-to-many relationships and will work for a number of situations.

Secondary index + the Query API action

A similar pattern for one-to-many relationships is to use a global secondary index and the Query API to fetch many. This pattern is almost the same as the previous pattern but it uses a secondary index rather than the primary keys on the main table.

You may need to use this pattern instead of the previous pattern because the primary keys in your table are reserved for another purpose. It could be some write-specific purpose, such as to ensure uniqueness on a particular property, or it could be because you have hierarchical data with a number of levels.

For the latter situation, let's go back to our most recent example. Imagine that in your SaaS application, each User can create and save various objects. If this were Google Drive, it might be a Document. If this were Zendesk, it might be a Ticket. If it were Typeform, it might be a Form.

Let's use the Zendesk example and go with a Ticket. For our cases, let's say that each Ticket is identified by an ID that is a combination of a timestamp plus a random hash suffix. Further, each ticket belongs to a particular User in an Organization.

If we wanted to find all Tickets that belong to a particular User, we could try to intersperse them with the existing table format from the previous strategy, as follows:

Primary key		Attributes	
Partition key: PK	**Sort key: SK**		
ORG#MICROSOFT	METADATA#MICROSOFT	OrgName	PlanType
		Microsoft	Enterprise
	USER#BILLGATES	UserName	UserType
		Bill Gates	Member
	USER#BILLGATES#TICKET#1569468714-JM14	CreatedAt	
		2019-09-25 22:31:54	
	USER#BILLGATES#TICKET#1570952398-MQR0	CreatedAt	
		2019-10-13 02:39:58	
	USER#SATYANADELLA	UserName	UserType
		Satya Nadella	Admin
ORG#AMAZON	METADATA#AMAZON	OrgName	PlanType
		Amazon	Pro
	USER#JEFFBEZOS	OrgName	UserType
		Amazon	Admin

Ticket items

Notice the two new Ticket items outlined in red.

The problem with this is that it really jams up my prior use cases. If I want to retrieve an Organization and all its Users, I'm also retrieving a bunch of Tickets. And since Tickets are likely to vastly exceed the number of Users, I'll be fetching a lot of useless data and making multiple pagination requests to handle our original use case.

Instead, let's try something different. We'll do three things:

1. We'll model our Ticket items to be in a separate item collection altogether in the main table. For the `PK` and `SK` values, we'll use a pattern of `TICKET#<TicketId>` which will allow for direct lookups of the Ticket item.

2. Create a global secondary index named `GSI1` whose keys are `GSI1PK` and `GSI1SK`.

3. For both our Ticket and User items, add values for `GSI1PK` and `GSI1SK`. For both items, the `GSI1PK` attribute value will be `ORG#<OrgName>#USER#<UserName>`.

For the User item, the `GSI1SK` value will be `USER#<UserName>`.

For the Ticket item, the `GSI1SK` value will be `TICKET#<TicketId>`.

Now our base table looks as follows:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
ORG#MICROSOFT	METADATA#MICROSOFT	OrgName	PlanType	GSI1PK & GSI1SK added to User items	
		Microsoft	Enterprise		
	USER#BILLGATES	UserName	UserType	GSI1PK	GSI1SK
		Bill Gates	Member	ORG#MICROSOFT#USER#BILLGATES	USER#BILLGATES
	USER#SATYANADELLA	UserName	UserType	GSI1PK	GSI1SK
		Satya Nadella	Admin	ORG#MICROSOFT#USER#SATYANADELLA	USER#SATYANADELLA
ORG#AMAZON	METADATA#AMAZON	OrgName	PlanType		
		Amazon	Pro		
	USER#JEFFBEZOS	UserName	UserType	GSI1PK	GSI1SK
		Jeff Bezos	Admin	ORG#AMAZON#USER#JEFFBEZOS	USER#JEFFBEZOS
TICKET#1569468714-JM14	TICKET#1569468714-JM14	CreatedAt	GSI1PK	GSI1SK	
		2019-09-25 22:31:54	ORG#MICROSOFT#USER#BILLGATES	TICKET#1569468714-JM14	
Ticket items					
TICKET#1570952398-MQR0	TICKET#1570952398-MQR0	CreatedAt	GSI1PK	GSI1SK	
		2019-10-13 02:39:58	ORG#MICROSOFT#USER#BILLGATES	TICKET#1570952398-MQR0	

Notice that our Ticket items are no longer interspersed with their parent Users in the base table. Further, the User items now have additional `GSI1PK` and `GSI1SK` attributes that will be used for indexing.

If we look at our GSI1 secondary index, we see the following:

Primary key		Attributes			
Partition key: GSI1PK	Sort key: GSI1SK				
ORG#MICROSOFT#USER#BILLGATES Bill Gates item collection	TICKET#1569468714-JM14	PK	SK	CreatedAt	
		TICKET#1569468714-JM14	TICKET#1569468714-JM14	2019-09-25 22:31:54	
	TICKET#1570952398-MQR0	PK	SK	CreatedAt	
		TICKET#1570952398-MQR0	TICKET#1570952398-MQR0	2019-10-13 02:39:58	
	USER#BILLGATES	PK	SK	UserName	UserType
		ORG#MICROSOFT	USER#BILLGATES	Bill Gates	Member
ORG#MICROSOFT#USER#SATYANADELLA	USER#SATYANADELLA	PK	SK	UserName	UserType
		ORG#MICROSOFT	USER#SATYANADELLA	Satya Nadella	Admin
ORG#AMAZON#USER#JEFFBEZOS	USER#JEFFBEZOS	PK	SK	UserName	UserType
		ORG#AMAZON	USER#JEFFBEZOS	Jeff Bezos	Admin

This secondary index has an item collection with both the User item and all of the user's Ticket items. This enables the same access patterns we discussed in the previous section.

One last note before moving on—notice that I've structured it so that the User item is the last item in the partition. This is because the Tickets are sorted by timestamp. It's likely that I'll want to fetch a User and the User's *most recent* Tickets, rather than the oldest tickets. As such, I order it so that the User is at the end of the item collection, and I can use the `ScanIndexForward=False` property to indicate that DynamoDB should start at the end of the item collection and read backwards.

Composite sort keys with hierarchical data

In the last two strategies, we saw some data with a couple levels of hierarchy—an Organization has Users, which create Tickets. But what if you have more than two levels of hierarchy? You don't want to keep adding secondary indexes to enable arbitrary levels of fetching throughout your hierarchy.

A common example in this area is around location-based data. Let's keep with our workplace theme and imagine you're tracking all the locations of Starbucks around the world. You want to be able to filter Starbucks locations on arbitrary geographic levels—by country, by state, by city, or by zip code.

We could solve this problem by using a composite sort key. This term is a little confusing, because we're using a composite primary key on our table. The term composite sort key

means that we'll be smashing a bunch of properties together in our sort key to allow for different search granularity.

Let's see how this looks in a table. Below are a few items:

Primary key		Attributes	
Partition key: Country	**Sort key: STATE#CITY#ZIP**		
USA	NE#OMAHA#68118	StreetAddress	SquareFeet
		15821 W Dodge Rd #100	921
	NY#NEWYORKCITY#10001	StreetAddress	SquareFeet
		875 6th Ave	1211
	NY#NEWYORKCITY#10019	StreetAddress	SquareFeet
		1500 Broadway	1924
FRANCE	ILE-DE-FRANCE#PARIS#75001	StreetAddress	SquareFeet
		26 Avenue de l'Opéra	2102

In our table, the partition key is the country where the Starbucks is located. For the sort key, we include the State, City, and ZipCode, with each level separated by a #. With this pattern, we can search at four levels of granularity using just our primary key!

The patterns are:

1. **Find all locations in a given country.** Use a Query with a key condition expression of `PK = <Country>`, where Country is the country you want.

2. **Find all locations in a given country and state.** Use a Query with a condition expression of `PK = <Country> AND begins_with(SK, '<State>#'`.

3. **Find all locations in a given country, state, and city.** Use a Query with a condition expression of `PK = <Country> AND begins_with(SK, '<State>#<City>'`.

4. **Find all locations in a given country, state, city, and zip code.** Use a Query with a condition expression of `PK = <Country> AND begins_with(SK, '<State>#<City>#<ZipCode>'`.

This composite sort key pattern won't work for all scenarios, but it can be great in the right situation. It works best when:

- You have many levels of hierarchy (>2), and you have access patterns for different levels within the hierarchy.
- When searching at a particular level in the hierarchy, you want all subitems in that level rather than just the items in that level.

For example, recall our SaaS example when discussing the primary key and secondary index strategies. When searching at one level of the hierarchy—find all Users—we didn't want to dip deeper into the hierarchy to find all Tickets for each User. In that case, a composite sort key will return a lot of extraneous items.

If you want a detailed walkthrough of this example, I wrote up the full Starbucks example on DynamoDBGuide.com.

Conclusion

In this post, we discussed five different strategies you can implement when modeling data in a one-to-many relationship with DynamoDB. The strategies are summarized in the table below.

Strategy	Notes	Relevant examples
Denormalize + complex attribute	Good when nested objects are bounded and are not accessed directly	User mailing addresses
Denormalize + duplicate	Good when duplicated data is immutable or infrequently changing	Books & Authors; Movies & Roles
Primary key + Query API	Most common. Good for multiple access patterns on the two entity types.	Most one-to-many relationships
Secondary index + Query API	Similar to primary key strategy. Good when primary key is needed for something else.	Most one-to-many relationships
Composite sort key	Good for very hierarchical data where you need to search at multiple levels of the hierarchy	Starbucks locations

Consider your needs when modeling one-to-many relationships and determine which strategy works best for your situation.

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

Published 11 Mar 2020

AWS DynamoDB

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

ALSO ON ALEXDEBRIE

How and Why to Use CloudFormation ...
2 years ago • 2 comments
Macros are a powerful tool for making CloudFormation easier to work with. ...

A Guide to S3 Batch on AWS
2 years ago • 2 comments
This post contains an overview and tutorial of AWS S3 Batch ...

Building a developer community
a year ago • 4 comments
In this post, learn which strategies work and which ones don't when you're ...

Us Re
2 ye
This cus wal

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Bernard Baltrusaitis • 7 months ago

Hey Alex, just wanted to thank you for extremely useful article with clear and concise examples. Well done, probably the best article related to DDB schema design I've found so far !

2 ^ | v • Reply • Share ›

> **Alex D** Mod ➚ Bernard Baltrusaitis • 7 months ago
>
> Thanks, Bernand! Glad it was helpful! :)
>
> ^ | v • Reply • Share ›

Jaad • 2 months ago

Fantastic article! What about modeling many-to-many relashionships in DynamoDB?

1 ^ | v • Reply • Share ›

> **Alex D** Mod ➚ Jaad • 2 months ago
>
> Thanks, **@Jaad** !
>
> Many-to-many relationships are a bit different, so I didn't cover them here. I do cover them in The DynamoDB Book if you want to check that out --> https://www.dynamodbbook.com/ :)
>
> 1 ^ | v • Reply • Share ›

James Gardner • 3 months ago

Thank you for the post Alex. I wanted to ask, in your example above, how exactly do you do a lookup for Ticket Items only? do you use the prefix or am I missing something conceptually? I don't see much in the docs about item collections and how they're used.

1 ^ | v • Reply • Share ›

> **Alex D** Mod ➚ James Gardner • 3 months ago
>
> Hey James! Presumably you would have the TicketId when you're doing a lookup, so you could do a GetItem where PK = TICKET#<ticketid> and SK = TICKET#<ticketid>.
>
> There's definitely not much in the AWS docs about item collections. I wish there was more there :)
>
> ^ | v • Reply • Share ›

Phil Herold • 6 months ago • edited

Hi Alex. This is a great article and the similar video I've seen as well. Is it possible to use the mapper annotatons (https://docs.aws.amazon.com... to map a one-to-many (in a single table)? It's not clear to me at all, but perhaps I'm missing something.

1 ^ | v • Reply • Share ›

> **Alex D** Mod → Phil Herold • 6 months ago
>
> Thanks, **@Phil Herold** ! Glad you liked it.
>
> I'm not that familiar with the Java DynamoDB Mapper library but my guess is that it does not help much here. I believe most of the mapper annotations help with attributes on a single item rather than tying together multiple items.
>
> ^ | v • Reply • Share ›
>
> > **Phil Herold** → Alex D • 6 months ago
> >
> > Thanks for the response. I was afraid that was the case. I suppose I could go with multiple tables with the mapper but it seems that is not best practice. The Java SDK has what I need to do this the "right" way.
> >
> > ^ | v • Reply • Share ›

Maximiliano Romay Figueroa • 8 months ago

Thanks for the clear explanation with examples for how to create one to many relationships, I needed this for my project.

1 ^ | v • Reply • Share ›

> **Alex D** Mod → Maximiliano Romay Figueroa • 8 months ago
>
> Glad to hear, **@Maximiliano Romay Figueroa** !
>
> ^ | v • Reply • Share ›

Michael Wolfenden • 9 months ago

For the User item, the GSI1SK value will be `+#USER#<username>+`

Should this be `USER#<username>`?

1 ^ | v • Reply • Share ›

> **Alex D** Mod → Michael Wolfenden • 9 months ago
>
> **@Michael Wolfenden** Good catch! You're absolutely right. Updated :)
>
> ^ | v • Reply • Share ›

Travis Harmon • 2 months ago

"Further, the User items now have additional GSI1PK and GSI1SK attributes that will be used for indexing."

I implemented a one-to-many relationship according to the pattern "Secondary index + the Query API action", but I did not add the "additional GSI1PK and GSI1SK attributes". How can those be used for indexing? What is the purpose of adding those?

^ | v • Reply • Share ›

Alex D Mod ➚ Travis Harmon • 2 months ago

Hey **@Travis Harmon**, they help unlock the additional access patterns we have in this use case -- get User and User Tickets -- without messing up our original access pattern. :)

^ | v • Reply • Share ›

HitHot • 4 months ago

Hi Alex,
I am currently reading your book, I have a question regarding partition and sort key that in which situation we need to define both the key as same and whats the benefit of it like here in this article PK & SK is defined as CUSTOMER#...

^ | v • Reply • Share ›

Alex D Mod ➚ HitHot • 4 months ago

Hey **@HitHot** , I don't quite understand the question. Could you elaborate on that? Also, feel free to DM me on Twitter or send me an email if that's easier than here :)

^ | v • Reply • Share ›

Vanathi MK • 4 months ago

Hi Alex. Im a fan of your articles. In section 4, where you are creating new partition for each ticket id. Is that really a good option? To create separate partition for a single item? Really looking forward to you rresponse

^ | v • Reply • Share ›

Alex D Mod ➚ Vanathi MK • 4 months ago

Hey **@Vanathi MK** , good question. I get the sentiment -- it does feel weird to use a new partition key for a single item.

However, it actually is a good idea. There's no additional overhead in DynamoDB to creating additional item collections (aka partition keys). All items with the same partition key are located together on the same node. If you won't be retrieving the items together, I give them different partition keys so that DynamoDB has more flexibility in where to allocate them across your database.

1 ^ | v • Reply • Share ›

Christos Dimitroulas • 5 months ago

This is a really great article, thanks for sharing this knowledge!

I was curious about the example with the Organizations in the second half of the article - what if we needed to support a query which could get a list of the Organizations in the system. Would that be something which can be supported by this data structure or would it be really problematic? If it's supported, what would be a good way of doing so?

^ | v • Reply • Share ›

Ahmed Mohiuddin • 5 months ago • edited

Hey Alex, what is the strategy if the attributes you are searching are optional (Any one or multiple can be provided when searching)?

Find all locations in a given country OR state OR city OR zip code

Do we create multiple GSI's?

Reply • Share ›

Boris Kuete • 6 months ago

Hi Alex, Thanks for the article.

I have two similar access patterns causing some issues:
1. Filter item by shop, status, category ordered by creationDate.
1. Filter item by shop, status, category and subcategory ordered by creationDate.

To Solve that I created a secondary index where:
GSI1PK=shop#shop_id_1 and
GSI1SK=AVAILABLE#CATEGORY_A#SUB_Category_1#CreatedDate

GSI1PK=shop#shop_id_1 and
GSI1SK=AVAILABLE#CATEGORY_A#SUB_Category_2#CreatedDate

Main Index:
PK=ITEM#item_1 and SK=SHOP#shop_id_1
PK=ITEM#item_2 and SK=SHOP#shop_id_1

But with such design, I will not have result ordered by creation date if the sort key condition is: GSI1SK begins_with AVAILABLE#CATEGORY_A#

Is there a way to achieve such case, without creating a new secondary Index

Thanks

Reply • Share ›

Air Fresh • 6 months ago

Hey Alex, why not just use a simply key for GSI to handle hierarchy data, for example, US#CA#San Fran#91134343, rather than PARTITION key + SORT key?

It seems that you use PARTITION key to group the stores, and let DynamoDB allocates them to same partition on a node. In this case, the data can stay together, and provide a good performance in retrieving them.

Is the above assumption right? Hope for your answer.

Thanks!

Reply • Share ›

Alex D Mod ➚ Air Fresh • 6 months ago

Hey @Air Fresh I. Just to clarify -- are you saying to have all items with the same

Hey @Air Fresh ! Just to clarify -- are you saying to have all items with the same partition key, or are you suggesting to use a simple primary key?

Assuming the first, I generally avoid putting all items in the same partition. You want to spread your items across multiple partitions to allow for better scalability. If you won't exceed the partition-level capacity limits (3000 RCUs / 1000 WCUs per second), then this isn't as big of a concern.

Reply • Share ›

Air Fresh → Alex D • 6 months ago

Hey Alex!

I means the second one

Sorry for the unclear question!

Reply • Share ›

dcboy • 8 months ago

Hi, what would you recommend for a case where you have categories and products and every product belongs to a category. So my table contains products (pk: id, sk: conf-prod) with a category as attribute and the same table contains categories (pk: id, sk: conf-category) and a name + description. I was thinking to duplicate the data but maybe I'm wrong? Should there be a relationship? A category won't change.

Reply • Share ›

Alex D Mod → dcboy • 8 months ago

First thing I'd think about is what are your access patterns around categories? Are you fetching products by category? Are you fetching products by category, and you need to get meta information about the category itself which is stored in the category item? Does the category information change or is it immutable?

If the answer to the third question is 'Yes', I'd probably use the second strategy of denormalization by duplicating data. Copy the information about the category into each product item.

If the third question is 'No' but the second question is 'Yes', then you'll probably want to use the Primary Key + Query or the Secondary Index + Query strategy. Put the category item in the same item collection as the product items.

Let me know if that helps! :)

Reply • Share ›

dcboy → Alex D • 8 months ago

Great help. It's immutable so I will go for the duplication. The access pattern is indeed fetching products by category

Reply • Share ›

Luka • 8 months ago

Hi, great article, thanks for sharing. My question is, by having as PK the organization name, aren't you afraid to have very large partitions? Also, if one organization is more active than the other, doesn't that mean that you could have throttling problems? Thanks in advance.

Reply • Share ›

Alex D Mod → Luka • 8 months ago

Thanks, Luka!

I'm not too afraid here. It depends on the size of these organizations, but if you had an organization with 10000 users and each user was 50kb (which is pretty big), that's still only half a GB. You shouldn't worry too much about partition size until you're in the 10GB range.

As far as hot key / throttling issues, DynamoDB Adaptive Capacity has mostly handled that. The big concern you'll have here is the 3000 RCU / 1000 RCU limit per partition. But that's pretty high -- 3000 reads or 1000 writes on a single partition per second.

Does that help?

Reply • Share ›

Nick Blair • 9 months ago • edited

Thanks for another article Alex, your posts are excellent! My question is regarding the use of `starts_with`. I can only find reference to `begins_with` in the dynamodb docs. Are they synonymous or am I missing a subtle difference?

Reply • Share ›

Alex D Mod → Nick Blair • 9 months ago

@Nick Blair there is a subtle difference between them: begins_with actually works, where as starts_with doesn't work!

D'oh, nice catch and thanks for asking. I always mess that up and yet I still forget to double-check every time. :)

I've updated to "begins_with" everywhere in the post. Thanks again!

2 Reply • Share ›

NM • 9 months ago • edited

A lot of this seems to go out the window if implementing fine-grained access control as described here: https://docs.aws.amazon.com... - would love to see a post from you **@Alex D** that takes this into account, as it seems to rule out using PK for much of anything, and require a lot of overloading of the SK.

Reply • Share ›

Alex D Mod → NM • 9 months ago

Good point! It definitely adds some complexity.

One thing to note: your data is often partitioned in a way that can handle this. For example, in our SaaS application example with Orgs & Users, the data is partitioned by Org Name. For each user within an Org, you could have an IAM policy where they could only read the data for their Organization.

I haven't done much with it before, but I'll add it to my list of potential blog posts!

Reply • Share ›

Code Rigger • 9 months ago

Why would you want to perform this modeling in Dynamo vs. doing an ETL from a Dynamo store to an RDBMS store for easier querying?

Reply • Share ›

Alex D Mod → Code Rigger • 9 months ago

If you're using DynamoDB as your primary, OLTP data store for an application, you may need to handle one-to-many relationships in your application.

For example, Amazon & AWS use DynamoDB for all Tier 1 applications (anything that makes money basically). For Amazon.com, this could mean fetching all Orders for a particular Customer. It's a one-to-many relationship they would need to handle in DynamoDB directly.

Reply • Share ›

Ross Williams • 9 months ago

For the Ticket example i'm not sure what you are gaining by adding the User to the PK/SK, what am I missing? Looks like it has a downside as you can no longer do a GetItem on just the Ticket ID.

Reply • Share ›

Alex D Mod → Ross Williams • 9 months ago

Hey **@Ross Williams** , adding the User to the Ticket item was only for the PK in the GSI. I need to add the User and the Org so that I could fetch all Tickets for a particular User within an Org.

For the base table, the Ticket only has the TicketId in its PK & SK. This would allow for GetItem on the TicketId.

Does that make sense?

Reply • Share ›

Ross Williams → Alex D • 9 months ago

Sorry I must be reading the image wrong, the one under the heading “ Now our base table looks as follows:” says primary key pk and has the format TICKET#id#USER#id, so I assumed it is more than just the ticket I’d.

1 Reply • Share ›

Alex D Mod ↗ Ross Williams • 9 months ago

Ahh, good catch! That's my mistake. I updated the images and the text to change it to TICKET#<ticketid>.

Thanks, Ross!

^ | v • Reply • Share ›