All Articles

When to use (and when not to use) DynamoDB Filter Expressions

Over the past few years, I've helped people design their DynamoDB tables. For many, it's a struggle to unlearn the concepts of a relational database and learn the *unique* structure of a DynamoDB single table design. Primary keys, secondary indexes, and DynamoDB streams are all new, powerful concepts for people to learn.

Yet there's one feature that's consistently a red herring for new DynamoDB users — filter expressions. I'm going to shout my advice here so all can hear:

Filter Expressions won't save your bad DynamoDB table design.

Lots of people think they can use a filter expression in their Query or Scan operations to sift through their dataset and find the needles in their application's haystack. But filter expressions in DynamoDB don't work the way that many people expect.

In this post, we'll learn about DynamoDB filter expressions. We'll cover:

- The allure of filter expressions for DynamoDB novices
- How filter expressions actually work
- What to use instead of filter expressions
- When filter expressions should be used

This post contains some concepts from my Data Modeling with DynamoDB talk at AWS re:Invent 2019. Feel free to watch the talk if you prefer video over text.

Let's get started!

The allure of filter expressions

Proper data modeling is all about filtering. Your application has a huge mess of data saved. At certain times, you need this piece of data or that piece of data, and your database needs to quickly and efficiently filter through that data to return the answer you request.

If you're coming from a relational world, you've been spoiled. You've had this wonderfully expressive syntax, SQL, that allows you to add any number of filter conditions.

You can filter on the value of a column:

```
-- Find all orders over $100
SELECT *
FROM orders
WHERE amount >= 100
```

You can combine tables and filter on the value of the joined table:

```
-- Find all orders for user 'alexdebrie'
SELECT orders.*
```

```
FROM orders
JOIN users ON orders.user_id = users.id
WHERE users.username = 'alexdebrie'
```

You can use built-in functions to add some dynamism to your query

```
-- Find all orders in the last week
SELECT *
FROM orders
WHERE orderdate >= GETDATE() - 7
```

With this flexible query language, relational data modeling is more concerned about structuring your data correctly. Once you've properly normalized your data, you can use SQL to answer any question you want.

DynamoDB is not like that. With DynamoDB, you need to plan your access patterns up front, then model your data to fit your access patterns. This is because DynamoDB won't let you write a query that won't scale. As such, you will use your primary keys and secondary indexes to give you the filtering capabilities your application needs.

This can feel wrong to people accustomed to the power and expressiveness of SQL. At this point, they may see the `FilterExpression` property that's available in the Query and Scan API actions in DynamoDB. The FilterExpression promises to filter out results from your Query or Scan that don't match the given expression.

This sounds tempting, and more similar to the SQL syntax we know and love. However, filter expressions don't work as you think they would. In the next section, we'll take a look why.

How filter expressions actually work

Let's walk through an example to see why filter expressions aren't that helpful.

Imagine you have a table that stores information about music albums and songs. Your table might look as follows:

Primary Key		Attributes				
PK	**SK**					
ALBUM#PAUL MCCARTNEY#FLAMING PIE	ALBUM#PAUL MCCARTNEY#FLAMING PIE	**AlbumName**	**Artist**	**Sales**	**RecordLabel**	
		Flaming Pie	Paul McCartney	1,604,904	Capitol Records	
	SONG#1	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Flaming Pie	The Song We W	Paul McCartney	841,040	Capitol Records
	SONG#2	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Flaming Pie	The World Tonigl	Paul McCartney	1,109,418	Capitol Records
ALBUM#KATY PERRY#TEENAGE DREAM	ALBUM#KATY PERRY#TEENAGE DREAM	**AlbumName**	**Artist**	**Sales**	**RecordLabel**	
		Teenage Dream	Katy Perry	6,124,098	Capitol Records	
	SONG#1	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Teenage Dream	Teenage Dream	Katy Perry	2,556,981	Capitol Records
	SONG#2	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Teenage Dream	Last Friday Night	Katy Perry	3,714,905	Capitol Records

In your table, albums and songs are stored within a collection with a partition key of `ALBUM#<artistName>#<albumName>`. Albums have a sort key of `ALBUM#<artistName>#<albumName>` while songs have a sort key of `SONG#<songNumber>`. In addition to information about the album and song, such as name, artist, and release year, each album and song item also includes a `Sales` attribute which indicates the number of sales the given item has made.

Imagine you wanted to find all songs that had gone platinum by selling over 1 million copies. You might think you could use the Scan operation with a filter expression to make the following call:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

const results = await client.scan({
  TableName: 'MusicTable',
  FilterExpression: 'Sales >= :platinum' AND begins_with(SK, :song)
  ExpressionAttributeValues: {
    ':platinum': 1000000,
    ':song': 'SONG#'
  }
})
```

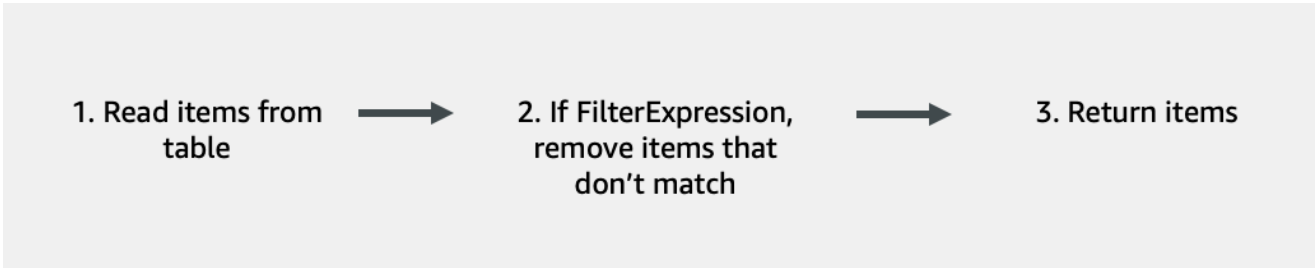
The example above is for Node.js, but similar principles apply for any language.

In the operation above, we're importing the AWS SDK and creating an instance of the DynamoDB Document Client, which is a client in the AWS SDK for Node.js that makes it easier for working with DynamoDB. Then, we run a `Scan` method with a filter expression to run a scan query against our table. The filter expression states that the `Sales` property

must be larger than 1,000,000 and the SK value must start with `SONG#`. This will return all songs with more than 1 million in sales.

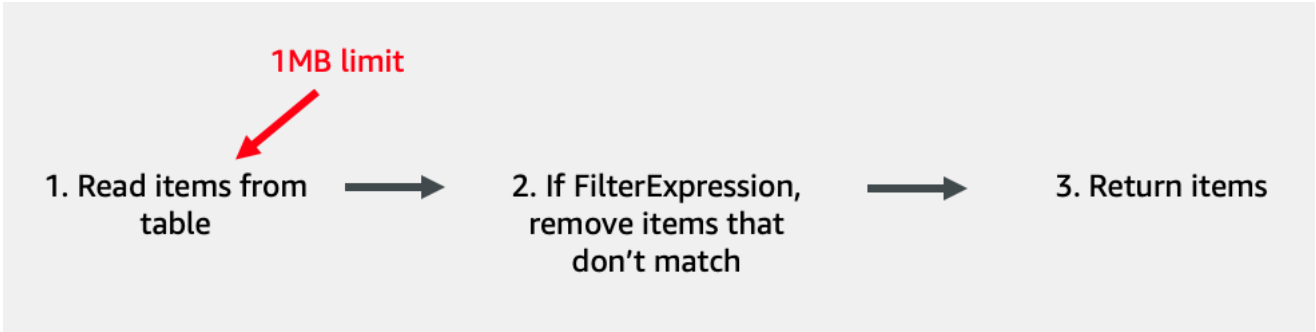
To see why this example won't work, we need to understand the order of operations for a Query or Scan request.

When you issue a Query or Scan request to DynamoDB, DynamoDB performs the following actions in order:



First, it reads items matching your Query or Scan from the database. Second, if a filter expression is present, it filters out items from the results that don't match the filter expression. Third, it returns any remaining items to the client.

However, the key point to understand is that the Query and Scan operations will return a maximum of 1MB of data, and this limit is applied in step 1, **before the filter expression is applied.**



Imagine your music table was 1GB in size, but the songs that were platinum were only 100KB in size. You might expect a single Scan request to return all the platinum songs, since it is under the 1MB limit. However, since the filter expression is not applied until after the items are read, your client will need to page through 1000 requests to properly scan your table. Many of these requests will return empty results as all non-matching items have been filtered out.

A 1GB table is a pretty small table for DynamoDB — chances are that yours will be much bigger. This makes the Scan + filter expression combo even less viable, particularly for OLTP-like use cases.

So what should you use to properly filter your table? We'll cover that in the next section.

How to properly filter your data in DynamoDB

Now that we know filter expressions aren't the way to filter your data in DynamoDB, let's look at a few strategies to properly filter your data. We'll walk through a few strategies using examples below, but the key point is that **in DynamoDB, you must use your table design to filter your data**.

There are a number of tools available to help with this. We'll look at the following two strategies in turn:

- Partition key
- Sparse index

Filtering with a partition key

The most common method of filtering is done via the partition key. The partition key is used to separate your items into *collections*, and a single collection is stored together on a single node in DynamoDB.

In the example portion of our music table, there are two different collections:

Primary Key		Attributes				
PK	**SK**					
ALBUM#PAUL MCCARTNEY#FLAMING PIE **Item collection #1**	ALBUM#PAUL MCCARTNEY#FLAMING PIE	**AlbumName**	**Artist**	**Sales**	**RecordLabel**	
		Flaming Pie	Paul McCartney	1,604,904	Capitol Records	
	SONG#1	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Flaming Pie	The Song We W	Paul McCartney	841,040	Capitol Records
	SONG#2	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Flaming Pie	The World Tonigl	Paul McCartney	1,109,418	Capitol Records
ALBUM#KATY PERRY#TEENAGE DREAM **Item collection #2**	ALBUM#KATY PERRY#TEENAGE DREAM	**AlbumName**	**Artist**	**Sales**	**RecordLabel**	
		Teenage Dream	Katy Perry	6,124,098	Capitol Records	
	SONG#1	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Teenage Dream	Teenage Dream	Katy Perry	2,556,981	Capitol Records
	SONG#2	**AlbumName**	**SongName**	**Artist**	**Sales**	**RecordLabel**
		Teenage Dream	Last Friday Night	Katy Perry	3,714,905	Capitol Records

The first collection is for Paul McCartney's Flaming Pie, and the second collection is for Katy Perry's Teenage Dream.

If we have access patterns like "Fetch an album by album name" or "Fetch all songs in a given album", we are filtering our data. We don't want all songs, we want songs for a single album. We can use the partition key to assist us.

When designing your table in DynamoDB, you should think hard about how to segment your data into manageable chunks, each of which is sufficient to satisfy your query. This is how DynamoDB scales as these chunks can be spread around different machines.

The value used to segment your data is the "partition key", and this partition key must be provided in any Query operation to DynamoDB. The requested partition key must be an exact match, as it directs DynamoDB to the exact node where our Query should be performed.

Imagine we want to execute this a Query operation to find the album info and all songs for the Paul McCartney's Flaming Pie album. We could write a Query as follows:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

const results = await client.query({
  TableName: 'MusicTable',
  KeyConditionExpression: 'PK = :pk',
  ExpressionAttributeValues: {
    ':pk': 'ALBUM#PAUL MCCARTNEY#FLAMING PIE'
  }
})
```

The key condition expression in our query states the partition key we want to use — `ALBUM#PAUL MCCARTNEY#FLAMING PIE`. This filters out all other items in our table and gets us right to what we want.

Sparse index

In the last example, we saw how to use the partition key to filter our results. In this section, we'll look at a different tool for filtering — the sparse index.

With DynamoDB, you can create secondary indexes. Secondary indexes are a way to have DynamoDB replicate the data in your table into a new structure using a different primary key schema. This makes it easy to support additional access patterns.

When creating a secondary index, you will specify the key schema for the index. The key schema is comparable to the primary key for your main table, and you can use the `Query` API action on your secondary index just like your main table. DynamoDB will handle all the work to sync data from your main table to your secondary index.

DynamoDB will only include an item from your main table into your secondary index if the item has **both** elements of the key schema in your secondary index. This is where you notion of sparse indexes comes in — you can use secondary indexes as a way to provide a global filter on your table through the presence of certain attributes on your items.

Let's see how this might be helpful. The most common way is to narrow down a large collection based on a boolean or enum value. In our music example, perhaps we want to find all the songs from a given record label that went platinum. A single record label will have a huge number of songs, many of which will not be platinum. We can use the sparse secondary index to help our query.

First, let's design the key schema for our secondary index. Our access pattern searches for platinum records for a record label, so we'll use `RecordLabel` as the partition key in our secondary index key schema.

For the sort key, we'll use a property called `SongPlatinumSalesCount`. The value for this attribute is the same as the value for `SalesCount`, but our application logic will only include this property if the song has gone platinum by selling over 1 million copies.

Our base table looks as follows:

Primary Key		Attributes					
PK	SK						
ALBUM#PAUL MCCARTNEY#FLAMING PIE	ALBUM#PAUL MCCARTNEY#FLAMING PIE	AlbumName	Artist	Sales	RecordLabel		
		Flaming Pie	Paul McCartney	1,604,904	Capitol Records		
	SONG#1	AlbumName	SongName	Artist	Sales	RecordLabel	
		Flaming Pie	The Song We W	Paul McCartney	841,040	Capitol Records	
	SONG#2	AlbumName	SongName	Artist	Sales	RecordLabel	SongPlatinumSalesCount
		Flaming Pie	The World Tonigl	Paul McCartney	1,109,418	Capitol Records	1,109,418
ALBUM#KATY PERRY#TEENAGE DREAM	ALBUM#KATY PERRY#TEENAGE DREAM	AlbumName	Artist	Sales	RecordLabel		
		Teenage Dream	Katy Perry	6,124,098	Capitol Records		
	SONG#1	AlbumName	SongName	Artist	Sales	RecordLabel	SongPlatinumSalesCount
		Teenage Dream	Teenage Dream	Katy Perry	2,556,981	Capitol Records	2,556,981
	SONG#2	AlbumName	SongName	Artist	Sales	RecordLabel	SongPlatinumSalesCount
		Teenage Dream	Last Friday Night	Katy Perry	3,714,905	Capitol Records	3,714,905

The table is the exact same as the one above other than the addition of the attributes outlined in red. There are three songs that sold more than 1,000,000 copies, so we added a `SongPlatinumSalesCount` for them.

Now, our secondary index is as follows:

Primary Key		Attributes				
RecordLabel	SongPlatinumSalesCount					
Capitol Records		AlbumName	SongName	Artist	Sales	RecordLabel
	3,714,905	Teenage Dream	Last Friday Night (T.G.I.F)	Katy Perry	3,714,905	Capitol Records
		AlbumName	SongName	Artist	Sales	RecordLabel
	2,556,981	Teenage Dream	Teenage Dream	Katy Perry	2,556,981	Capitol Records
		AlbumName	SongName	Artist	Sales	RecordLabel
	1,109,418	Flaming Pie	The World Tonight	Paul McCartney	1,109,418	Capitol Records

Notice that our secondary index is sparse — it doesn't have all the items from our main table. It doesn't include any `Album` items, as none of them include the `SongPlatinumSalesCount` attribute. Further, it doesn't include any `Song` items with fewer than 1 million copies sold, as our application didn't include the `PlatinumSalesCount` property on it.

Now I can handle my "Fetch platinum songs by record label" access pattern by using my sparse secondary index. I can run a `Query` operation using the `RecordLabel` attribute as my partition key, and the platinum songs will be sorted in the order of sales count.

The query would be:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

const results = await client.query({
  TableName: 'MusicTable',
  IndexName: 'PlatinumSongsByLabel',
  KeyConditionExpression: 'PK = :pk',
  ExpressionAttributeValues: {
```

```
    ':pk': 'Capitol'
  }
})
```

This essentially gives me the following pattern in SQL:

```
-- Fetch all platinum songs from Capital records
SELECT songs.*
FROM songs
JOIN albums ON songs.album_id = albums.id
JOIN labels ON albums.label_id = labels.id
WHERE labels.name = 'Capitol'
AND songs.copies_sold >= 1000000
ORDER BY songs.copies_sold DESC
```

When filter expressions should be used

We've now seen why filter expressions don't work as you think they would and what you should use instead. But it raises the question — when are filter expressions useful? Surely we don't think that the DynamoDB team included them solely to terrorize unsuspecting users!

The three examples below are times where you might find filter expressions useful:

1. Reducing response payload size;
2. Easier application filtering;
3. Better validation around time-to-live (TTL) expiry.

Let's look at each of these in turn.

Reducing response payload size

The first reason you may want to use filter expressions is to reduce the size of the response payload from DynamoDB. DynamoDB can return up to 1MB per request. This is a lot of data

to transfer over the wire. If you know you'll be discarding a large portion of it once it hits your application, it can make sense to do the filtering server-side, on DynamoDB, rather than in your client.

Easier application filtering

A second reason to use filter expressions is to simplify the logic in your application. If you're immediately going to filter a bunch of items from your result, you may prefer to do it with a filter expression rather than in your application code.

To make it real, let's say you wanted to fetch all songs from a single album that had over 500,000 sales.

You could fetch all the songs for the album, then filter out any with fewer than 500,000 sales:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

const results = await client.query({
  TableName: 'MusicTable',
  KeyConditionExpression: 'PK = :pk',
  ExpressionAttributeValues: {
    ':pk': 'ALBUM#PAUL MCCARTNEY#FLAMING PIE'
  }
})

const filteredResults = results.Items.filter(item => item.Sales >=
```

Or, you could use a filter expression to remove the need to do any client-side filtering:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

const results = await client.query({
  TableName: 'MusicTable',
  KeyConditionExpression: 'PK = :pk',
```

```
  FilterExpression: 'Sales >= :threshold',
  ExpressionAttributeValues: {
    ':pk': 'ALBUM#PAUL MCCARTNEY#FLAMING PIE',
    ':threshold': 500000
  }
})
```

You've saved the use of `filter()` on your result set after your items return.

This one comes down to personal preference. I prefer to do the filtering in my application where it's more easily testable and readable, but it's up to you.

Better validation around time-to-live (TTL) expiry

The last example of filter expressions is my favorite use — you can use filter expressions to provide better validation around TTL expiry.

DynamoDB allows you to specify a time-to-live attribute on your table. This attribute should be an epoch timestamp. DynamoDB will periodically review your items and delete items whose TTL attribute is before the current time.

The TTL attribute is a great way to naturally expire out items. The canonical use case is a session store, where you're storing sessions for authentication in your application. This session expires after a given time, where the user must re-authenticate.

Your table might look as follows:

Primary key	Attributes		
Partition key: SessionId			
d96d4fa6-2a20-48e0-a6cf-676397597a81	Username	CreatedAt	ExpiresAt
	alexdebrie	1577902318	1578507118
44893d1d-1793-401c-9f67-0f2e17e049b9	Username	CreatedAt	ExpiresAt
	donaldduck	1577988718	1578593518
c2c15407-c461-4986-befb-9dd5c00eee8a	Username	CreatedAt	ExpiresAt
	nedstark	1578075118	1578679918

In this table, my partition key is `SessionId`. I also have the `ExpiresAt` attribute, which is an epoch timestamp. I'll set my TTL on this attribute so that DynamoDB will remove these items when they're expired.

While I've found the DynamoDB TTL is usually pretty close to the given expiry time, the DynamoDB docs only state that DynamoDB will typically delete items within 48 hours of expiration. As such, there's a chance our application may read an expired session from the table for 48 hours (or more!) after the session should have expired.

To simplify our application logic, we can include a filter expression on our `Query` to the session store that filters out any sessions that have already expired:

```
const AWS = require('aws-sdk');
const client = new AWS.DynamoDB.DocumentClient({apiVersion: '2012-0

// Find the current timestamp.
const time = Date.now() / 1000

client.query({
  TableName: 'SessionStore',
  KeyConditionExpression: 'SessionId = :session',
  FilterExpression: 'ExpiresAt >= :currentTime',
  ExpressionAttributeValues: {
    ':session': 'd96d4fa6-2a20-48e0-a6cf-676397597a81'
    ':currentTime': time
  }
})
```

Now our application doesn't have to perform an additional check to ensure the returned item has expired. If our query returns a result, then we know the session is valid. The TTL is still helpful is cleaning up our table by removing old items, but we get the validation we need around proper expiry.

Conclusion

In this article, we saw why DynamoDB filter expressions may not help the way you think. First we saw why filter expressions trick a lot of relational converts to DynamoDB. Then we explored how filter expressions actually work to see why they aren't as helpful as you'd expect.

We then saw how to model your data to get the filtering you want using the partition key or sparse secondary indexes. We also saw a few ways that filter expressions can be helpful in your application.

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

Thanks to Jeremy Daly for his assistance in reviewing this post. All mistakes are mine.

Published 13 Jan 2020

AWS DynamoDB

Alex DeBrie on Twitter

ALSO ON ALEXDEBRIE

A Detailed Overview of AWS API Gateway

2 years ago • 23 comments

Look inside the black box of AWS API Gateway to understand authorization, …

Using Custom Resources to …

2 years ago • 2 comments

This CloudFormation custom resources tutorial walks you though when …

Three Projects to Get You Started with …

2 years ago

Serverless is one of the hottest programming trends in 2019. Check out three …

Bu co

a ye

In t stra one

What do you think?

27 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments alexdebrie Disqus' Privacy Policy 1 Login

Recommend 1 Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data