DynamoDB Transactions: Use Cases and Examples

Amazon's DynamoDB was released in 2012 and has been adding a drumbeat of new features ever since. It's hard to believe now, but the original version of DynamoDB didn't have DynamoDB Streams, parallel scans, or even secondary indexes.

One of the more exciting feature releases from DynamoDB in recent years has been the addition of DynamoDB Transactions at re:Invent 2018. With DynamoDB Transactions, you can write or read a batch of items from DynamoDB, and the entire request will succeed or fail together.

This feature release simplified a lot of workflows that involved complex versioning and multiple requests to accurately work across multiple items. In this post, we'll review how and why to use DynamoDB Transactions.

We'll cover:

- Background on DynamoDB Transactions, including differences with existing batch operations and idempotency with transactions
- Three common use cases for DynamoDB Transactions

This is part 1 of a two-part post on DynamoDB Transactions. Check out part 2 in this series for a look at performance testing DynamoDB transactions.

Let's get started!

Background on DynamoDB Transactions

To begin, let's look at a few details around DynamoDB Transactions. We'll cover two areas:

1. What are the transactional APIs and how do they differ from batch APIs?

2. Handling idempotency with transactional requests.

What are the transactional APIs?

There are two API calls that deal with transactions — `TransactWriteItems` and `TransactGetItems`. As you can guess from the name, the first is used to *write* multiple items in a single transaction. The second is used to *read* multiple items in a single transaction. Both transactional APIs allow you to operate on up to 25 items in a single request.

DynamoDB has long had batch-based APIs that operate on multiple items at a time. You could use `BatchGetItem` to read up to 100 items at once, or you could use `BatchWriteItem` to write up to 25 items at once.

There are two key differences between the Batch* APIs and the Transact* APIs. The first is around capacity consumption. When using the Transact APIs, you will be charged *twice* the capacity that would be consumed if you performed the operations without a transaction. Thus, if you have a `TransactWriteItem` request that inserts two items of less than 1KB, you will be charged for 4 write capacity units — 2 items of 1KB X 2 for transactions.

The second difference between the Transact APIs and the Batch APIs is around failure modes. With the Transact APIs, *all reads or writes will succeed or fail together*. In the Batch APIs, some requests can succeed and some can fail, and it's up to you to handle the errors.

A transactional request can fail for a number of reasons. First, one of the elements in a request could fail due to conditions in the request. For any of the write-based requests, you can include a condition expression in the request. If those conditions are not satisfied, the write will fail and the entire batch will fail.

Second, a transaction could fail if one of the items is being altered in a separate transaction or request. For example, if you make a `TransactGetItems` request on an item at the same time there is an open `TransactWriteItems` request being processed on the item, the `TransactGetItems` request will fail. This type of failure is known as a *transaction conflict*, and you can view CloudWatch Metrics on the number of transaction conflicts on your tables.

Finally, a transaction could fail for other reasons, such as your table not having enough capacity or the DynamoDB service being down generally.

Idempotency with transactional requests

For the `TransactWriteItem` API, DynamoDB allows you to pass a `ClientRequestToken` parameter with your request. Including this parameter will allow you to ensure your request is *idempotent* even if submitted multiple times.

To see how this is helpful, imagine you are doing a `TransactWriteItem` request that includes some write requests to increment an attribute on an item. If you had a network issue where you didn't know if this operation succeeded or failed, you could be in a bad state. If you assume the operation succeeded but it didn't, the value of your attribute will be lower than it should be. If you assume the operation failed when it didn't, you could submit the request again but the value of the attribute will be *higher* than it should be.

The `ClientRequestToken` handles this. If you submit a request with the same token and the same parameters within a 10 minute period, DynamoDB will ensure the request is idempotent. If the request succeeded when initially submitted, DynamoDB won't execute it again. If it failed the first time, DynamoDB will apply the writes in the request.

The `TransactWriteItems` API is the only DynamoDB API that allows for idempotency, so you could use the `TransactWriteItems` API even with a single item if you have a strong need for idempotency.

Common use cases for DynamoDB Transactions

Now that we know the basics about DynamoDB Transactions, let's see them in action. Remember that DynamoDB Transactions cost twice as much as a similar non-transaction operation, so we should be judicious and only use transactions when we really need them.

When are some good times to use transactions? I have three favorite examples that we'll walk through below:

- Maintaining uniqueness on multiple attributes
- Handling counts and preventing duplicates
- Authorizing a user to perform a certain action

I didn't include an example where I needed idempotency, as discussed in the previous section, but that's another example of a good use case for the transactional APIs.

Let's review each of the examples in turn.

Maintaining uniqueness on multiple attributes

In DynamoDB, if you want to ensure a particular attribute is unique, you need to build that attribute directly into your primary key structure.

An easy example here is a user signup flow for an application. You want the username to be unique across your application, so you build a primary key that includes the username.

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
USER#alexdebrie	USER#alexdebrie	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915

In the table above, our PK and SK values both include the `username` so that it will be unique.

But what if you also want to ensure that a given email address is unique across your system so that you don't have people signing up for multiple accounts to the same email address?

You might be tempted to add email into your primary key as well:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
USER#alexdebrie	USER#alex@debrie.com	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#hello@whitney.com	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#frankthetank@sinatra.com	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915

Now our table includes the username in the PK and the email in the SK.

However, this won't work. It's the *combination* of a partition key and sort key that makes an item unique within the table. Using this key structure, you're confirming that an email address will only be used once *for that username*. Now you've lost the original uniqueness properties on the username, as someone else could sign up with the same username and a different email address!

If you want to ensure that both a username and an email address are unique across your table, you need to create an item for each and add those items with a transaction.

The code to write such a transaction would be as follows:

```
response = client.transact_write_items(
    TransactItems=[
        {
            'Put': {
                'TableName': 'UsersTable',
                'Item': {
                    'PK': { 'S': 'USER#alexdebrie' },
                    'SK': { 'S': 'USER#alexdebrie' },
                    'Username': { 'S': 'alexdebrie' },
                    'FirstName': { 'S': 'Alex' },
                    ...
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        },
```

```
        {
            'Put': {
                'TableName': 'UsersTable',
                'Item': {
                    'PK': { 'S': 'USEREMAIL#alex@debrie.com' },
                    'SK': { 'S': 'USEREMAIL#alex@debrie.com' },
                },
                'ConditionExpression': 'attribute_not_exists(PK)
            }
        }
    ]
)
```

And now your table would look as follows:

Primary key		Attributes			
Partition key: PK	**Sort key: SK**				
USER#alexdebrie	USER#alexdebrie	Username	FirstName	LastName	DateOfBirth
		alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	Username	FirstName	LastName	DateOfBirth
		thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	Username	FirstName	LastName	DateOfBirth
		olblueeyes	Frank	Sinatra	December 12, 1915
USEREMAIL#alex@debrie.com	USEREMAIL#alex@debrie.com				
USEREMAIL#hello@whitney.com	USEREMAIL#hello@whitney.com				
USEREMAIL#frankthetank@sinatra.com	USEREMAIL#frankthetank@sinatra.com				

Notice that the item that stores a user by email address doesn't have any of the user's properties on it. You can do this if you will only access a user by a username and never by an email address. The email address item is essentially just a marker that tracks whether the email has been used.

If you *will* access a user by email address, then you need to duplicate all information across both items. Then your table might look as follows:

PK (Partition key) : String	SK (Sort key) : String	Username : String	FirstName : String	LastName : String	DateOfBirth : String
USER#alexdebrie	USER#alexdebrie	alexdebrie	Alex	DeBrie	May 26, 1988
USER#thevoice	USER#thevoice	thevoice	Whitney	Houston	August 9, 1963
USER#olblueeyes	USER#olblueeyes	olblueeyes	Frank	Sinatra	December 12, 1915
USEREMAIL#alex@...	USEREMAIL#alex@...	alexdebrie	Alex	DeBrie	May 26, 1988
USEREMAIL#hello@...	USEREMAIL#hello@...	thevoice	Whitney	Houston	August 9, 1963
USEREMAIL#frankth...	USEREMAIL#frankt...	olblueeyes	Frank	Sinatra	December 12, 1915

I'd avoid this if possible. Now every update to the user item needs to be a transaction to update both items. It will increase the cost of your writes and the latency on your requests.

Handling counts and preventing duplicates

A second place where transactions can be helpful is in storing counts for related items. Let's see how that works.

Imagine you have a social application with some sort of system for 'liking' items. It could be Twitter, as users like other tweets. It could be Reddit, where users can like or upvote particular posts or comments. Or it may be GitHub, where a user can add a reaction to an issue.

In all of these situations, you will want to store a record of the user that is upvoting a particular item in order to ensure that a user doesn't vote multiple times.

Additionally, when displaying the item that can be upvoted, you want to display the total number of upvotes. It's more efficient to denormalize this by storing a counter property on the item itself rather than doing a Query operation each time to fetch all the items that indicate the item has been upvoted.

Your table might look as follows:

Primary key		Attributes			
Partition key: PK	Sort key: SK				
POST#1caa5be06389	POST#1caa5be06389	Subreddit	Title	UpvotesCount	CreatedAt
		programming	How to use DynamoDB	2	2020-02-02 06:17:44
	USER#alexdebrie	Username			
		alexdebrie			
	USER#nothingbutnet	Username			
		nothingbutnet			
POST#bd69aa606274	POST#bd69aa606274	Subreddit	Title	UpvotesCount	CreatedAt
		personalfinance	OMG my student loans are huge!!!	1	2020-02-07 05:25:35
	USER#2cool4skool	Username			
		2cool4skool			

This table is a simplified version of Reddit. Users can create Posts, and other users can like the Posts. In this table, we have 5 items. Two of them are Post items (using the `POST<PostId>` pattern for both PK and SK), while three of them are UserLike Items (using the same `POST#<PostId>` pattern for PK and a `USER#<Username>` pattern for SK). Notice how each Post item has an `UpvotesCount` attribute that stores the total number of upvotes it has received.

When a user upvotes an item, you want to first ensure that the user hasn't previously upvoted the item, then increment the `UpvotesCount` attribute on the item. In a world without transactions, this would be a two-step process.

With transactions, you can do it in one step. The code to run this transaction would be as follows:

```
response = client.transact_write_items(
    TransactItems=[
        {
            'Put': {
                'TableName': 'RedditTable',
                'Item': {
                    'PK': { 'S': 'POST#1caa5be06389' },
                    'SK': { 'S': 'USER#alexdebrie' },
                    'Username': { 'S': 'alexdebrie' },
                },
                'ConditionExpression': 'attribute_not_exists(PK)
```

```
            }
        },
        {
            'Update': {
                'TableName': 'UsersTable',
                'Key': {
                    'PK': { 'S': 'POST#1caa5be06389' },
                    'SK': { 'S': 'POST#1caa5be06389' }
                },
                'UpdateExpression': 'SET UpvotesCount = UpvotesCour
                'ExpressionAttributeValues': {
                  ':incr': { 'N': '1' }
                }
            }
        }
    ]
)
```

Our transaction includes two write requests. The first one inserts a new item that indicates the user `alexdebrie` has upvoted the given Reddit post. This write request includes a condition expression that asserts that an item with the same key doesn't already exist, which would indicate that `alexdebrie` has already upvoted this item.

The second write request is an update expression to increment the `UpvotesCount` on the upvoted post.

If the first write request fails because `alexdebrie` has already upvoted this item, the `UpvotesCount` won't be updated either as the entire transaction will fail.

Note: you could handle this pattern in a different way, without transactions. You could do a simple `PutItem` request to add an item indicating that a user has upvoted a particular item, with the same condition expression as used above. Then, using DynamoDB streams, you could aggregate all of these new item insertions together and increment the parent item's `UpvotesCount` in batch.

These approaches are pretty similar, so you can go either way. I'd only recommend the streams-based approach if one of the following circumstances are true:

1. You want the upvote path to be as fast as possible, so you'll take the faster `PutItem` request over the slower `TransactWriteItems` request.

2. You have a small number of items that will be upvoted such that it's likely you can aggregate multiple individual upvotes together when incrementing the `UpvotesCount`.

Let's think about the second case a bit, using two different examples — a nationwide voting application and a social media site like Twitter. In the voting application, there are only a few options for users to select. You could save significant write capacity on your table by using the stream-based method to aggregate multiple votes together and increment the `UpvotesCount` in batch.

On the other hand, a social media site like Twitter has a huge cardinality of items that can be upvoted. Even with a batch size of 1000 records from your DynamoDB stream, it's unlikely you'll get multiple records that touch the same parent item. In that case, your write capacity usage will be the same, as you'll need to increment each parent item individually.

Access control

A third use case for transactions is in an authorization or access control setting.

Imagine you provide a SaaS application to large organizations. With a given organization's installation of your application, there are two levels of users: Admins and Members. Admins are allowed to add new Members to the installation but regular Members are not.

Without transactions, you would need to do a multi-step process to read from and write to DynamoDB to add a new member. With DynamoDB transactions, you can do it in one step.

Imagine you have the following table:

Primary key		Attributes		
Partition key: PK	Sort key: SK			
ORG#amazon	ORG#amazon	OrgName	Admins	CreatedAt
		Amazon	["Jeff Bezos","Andy Jassy"]	1994-07-13 03:01:27
	USER#charliebell	Username	Role	
		charliebell	Member	
	USER#jeffbezos	Username	Role	
		jeffbezos	Admin	
ORG#apple	ORG#apple	OrgName	Admins	CreatedAt
		Apple	["Tim Cook"]	1976-04-01 08:14:47

This table includes Organizations and Users. An Organization has a primary key pattern that uses `ORG#<OrgName>` for both PK and SK, while a User has a primary key pattern that uses `ORG#<OrgName>` for PK and `USER#<Username>` for SK.

Notice that the Organization items include an `Admins` attribute that is of type string set. This attribute stores which Users are Admins and thus can create new Users.

When someone tries to create a new User in an Organization, you can use a transaction to safely assert that the requesting member is an Admin.

The `TransactWriteItems` request would look as follows:

```
response = client.transact_write_items(
    TransactItems=[
        {
            'ConditionCheck': {
                'TableName': 'AccessControl',
                'Key': {
                    'PK': { 'S': 'ORG#amazon' },
                    'SK': { 'S': 'ORG#amazon' },
                },
                'ConditionExpression': 'contains(Admins, :user)',
                'ExpressionAttributeValues': {
                  ':user': { 'S': 'Charlie Bell' }
                }
            }
        },
```

```
        {
            'PutItem': {
                'TableName': 'AccessControl',
                'Item': {
                    'PK': { 'S': 'ORG#amazon' },
                    'SK': { 'S': 'USER#jeffbarr' },
                    'Username': { 'S': 'jeffbarr' }
                }
            }
        }
    ]
)
```

This example is slightly different than the previous. Notice that we're using a `ConditionCheck` operation. We don't actually want to modify the existing Organization item. We just want to assert a particular condition on it — that Charlie Bell is an Admin. If that is not correct, we want to fail the whole transaction.

There are a number of ways you can handle authorization in your application, and this is one option. The nice thing about it is that it's flexible — you can implement application-wide authorization using a singleton item in your table, or you can implement fine-grained, resource-based authorization by adding authorization information on a wide number of resources across your table.

Conclusion

In this post, we learned about DynamoDB Transactions. First, we covered the basics of DynamoDB Transactions — how do they work? What guarantees do they provide? How much do they cost?

Second, we saw some examples of DynamoDB Transactions in action. We went through three examples: maintaining uniquness on multiple attributes; handling counts and preventing duplicates; and managing access control.

DynamoDB Transactions are a great tool to add to your toolbelt. But you need to be aware of the performance impact as well. Be sure to read the follow-up post where we test the

performance impact of transactions in DynamoDB.

Want more content on DynamoDB Transactions? Check out the following:

- Edin Zulich gave a great presentation on using DynamoDB Transactions.
- After DynamoDB Transactions were announced at re:Invent 2018, Yossi Levanoni gave this talk on DynamoDB Transactions.

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

Published 12 Feb 2020

AWS DynamoDB

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

ALSO ON **ALEXDEBRIE**

Three ways to use AWS services from a …
a year ago • 5 comments
By default, Lambda functions in a VPC cannot access the public …

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

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

Ho Clo
2 ye
Ma for eas

What do you think?
4 Responses

Upvote Funny Love Surprised Angry Sad

18 Comments alexdebrie Disqus' Privacy Policy Login

Recommend Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Abhishek Gupta • 8 months ago

In "Handling counts and preventing duplicates" section wouldn't the first request in the write transaction check the existence of SK instead of PK to see if the user has commented on the given post earlier?

1 • Reply • Share ›

> **Alex D** Mod → Abhishek Gupta • 8 months ago
>
> Good question, **@Abhishek Gupta** , and it's one that trips up a lot of people.
>
> When you do a attribute_not_exists ConditionExpression on an element of the primary key, it checks to ensure the exact primary key does not exist. It doesn't matter whether you use the PK or the SK in your expression.
>
> Kind of confusing but it works either way :)
>
> 1 • Reply • Share ›
>
> > **Abhishek Gupta** → Alex D • 8 months ago
> >
> > Got it. Thanks Alex, learning a lot from your blog :)
> >
> > • Reply • Share ›
> >
> > > **Alex D** Mod → Abhishek Gupta • 8 months ago
> > >
> > > Glad to hear it! :)
> > >
> > > • Reply • Share ›

Ross Williams • 10 months ago

Great content, very useful in understanding how to insert data into an existing structure. What keeps me from using NoSQL more is mutating data. With relational 3NF changing a user's email address is trivial, but the demornalisation in this post worries me in terms of development time / cost, and potential for mistakes, especially as the app grows larger. Would love to see content on mutating denormalised data as a follow-up, or is it just a case of being very careful and doing daily batch "cleanup" scans?

1 • Reply • Share ›

Alex D Mod ➚ Ross Williams • 10 months ago

Thanks, **@Ross Williams** ! And good point. I've had more people asking me about mutations and updating data. I'll be working on some thoughts around that in the near future :).

1 • Reply • Share ›

Jim Ancona • 5 months ago

I'm using TransactWriteItems with a ConditionCheck to check a foreign key constraint. So I have two items in my transaction, the ConditionCheck, which checks for the existence of a parent item in the table, and a Put, which inserts a child item, which contains an attribute that references the parent. When I concurrently insert multiple children that all reference the same parent, I often get a TransactionCanceledException, where the CancellationReason is "TransactionConflict" and its position in the array corresponds to the condition check. As far as I can tell, the parent item isn't being written to concurrently. Is there some other reason I would be getting a conflict in this situation?

• Reply • Share ›

Alex D Mod ➚ Jim Ancona • 5 months ago

Hey **@Jim Ancona**, that's interesting that it appears to be the ConditionCheck that's failing. Is the item being checked one that's often written to in other operations? Here are the reasons a Transaction can fail, and most are when one of the items is being affected in another operation.
https://docs.aws.amazon.com...

• Reply • Share ›

Jim Ancona ➚ Alex D • 5 months ago

Thanks for the reply, **@Alex D** ! I've checked my logs, and the last write to the parent record appears to be more than 100 milliseconds before the failure. So I'm kind of at a loss. I've implemented retries as a workaround, but it seems odd that a simple existence check would fail in this way.

• Reply • Share ›

AdamH • 5 months ago

Thanks for this writeup. I have a question about the example provided for"Handling counts and preventing duplicates." Since the behavior of TransactWrite is to return an error "if one of the items is being altered in a separate transaction or request" then I'm wondering if the Update code where we increment a Post's UpvotesCount will generate transactional errors when one Post item receives multiple concurrent upvotes from multiple users (because they all attempt to update upvotesCount for the same Primary Key)? If my understanding there is correct, is there a good pattern to use to handle this scenario?

• Reply • Share ›

Alex D Mod ➚ AdamH • 5 months ago

Hey **@AdamH** ! Good question. That could happen, which would mess with your flow. One other pattern is to do your aggregations after the fact, such as hooking up to DynamoDB Streams and incrementing the UpvotesCount after the fact. This would ensure the Upvote succeeded without potentially blocking other transactions.

1 ^ | v • Reply • Share ›

Henrik Dohlmann • 7 months ago

Very useful information!
For the count example, how would you model: Top upvoted posts the user has voted upon?

I have thought about something with PK=USER, SK=<count>#POST to be able to perform the query.
When the UpvotesCount is updated on a post, I then need to write UpvotesCount#POST to an attribute on the PK=POST, SK=USER for all users that voted on that post.

Is that feasible? Or, should I do something else?

^ | v • Reply • Share ›

Alex D Mod ➚ Henrik Dohlmann • 7 months ago

If I'm understanding correctly, that's a pretty tricky one. You have a many-to-many relationship you're handling (each post can have many likes, and each user can like many posts). Additionally, the information you want in mutable (a running count of likes for the post).

It would require a lot of updates each time someone upvoted a post...

^ | v • Reply • Share ›

Henrik Dohlmann ➚ Alex D • 7 months ago

Precisely. That is why i asked :-)
I don't know if aggregate-on-write makes sense here, or if I should have a batch-job that adjust counts for each user.
Or, if there is some NoSQL pattern for this that I haven't read about yet.

^ | v • Reply • Share ›

Felix Bouaket Chanthapanya • 8 months ago

Hi ! awesome content. Will probably buy the book.

Quick question.

in your last example, any specific reason why the ConditionCheck is done on:
pk: ORG#amazon sk: ORG#amazon --> 'contains(Admins, :user)'

and not directly on:
pk: ORG#amazon sk: USER#charliebell --> 'eq(Role, :admin)

Reply • Share ›

Alex D Mod → Felix Bouaket Chanthapanya • 8 months ago

Glad you liked it!

There's no real reason :) You could use either one to check. Good question!

1 Reply • Share ›

Chris Shenton • 10 months ago

I didn't think I needed/wanted transactions. Your article convinced me otherwise. Thanks!

Reply • Share ›

Alex D Mod → Chris Shenton • 10 months ago

Glad to hear! :)

Reply • Share ›

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