Mastering Amazon DynamoDB ACID transactions:

When and how to use the new transactional APIs

Arturo Hinojosa, Senior Product Manager, Amazon DynamoDB

April 25, 2019

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Learning Objectives

1. Learn when to use DynamoDB transactions to simplify your code.

2. Learn how the new DynamoDB transactional APIs work and how to implement them in your applications through examples demonstrating transactional design patterns.

3. Learn best practices for using DynamoDB transactions and how the new API calls interact with other DynamoDB features.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Agenda

1. Introduction to Amazon DynamoDB transactions and atomicity, consistency, isolation, and durability (ACID) in Amazon DynamoDB.

2. Modern application use cases for transactional APIs.

3. Important things to know when integrating the transactional API into your application: metering, concurrency control, and interaction with other features.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Prerequisites

You should be familiar with basic DynamoDB concepts, such as tables, items, partition keys and sort keys, indexes, Time To Live (TTL), and DynamoDB Streams.

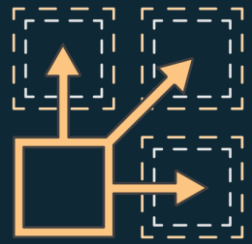
We will only introduce these concepts briefly in this presentation before using them in the examples.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

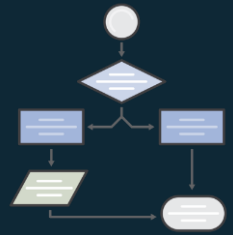
aws

How do developers use transactions?

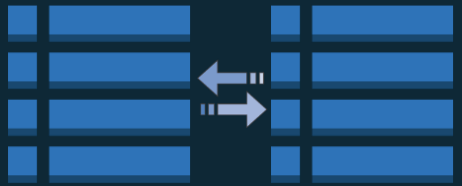
Transactions help developers maintain data correctness by coordinating actions across multiple items and tables.



Make simultaneous writes and updates to multiple items.



Check multiple conditions before making changes.



Update data across tables consistently.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Developer challenges

Traditionally, developers using non-relational databases faced challenges implementing transactions in their applications:

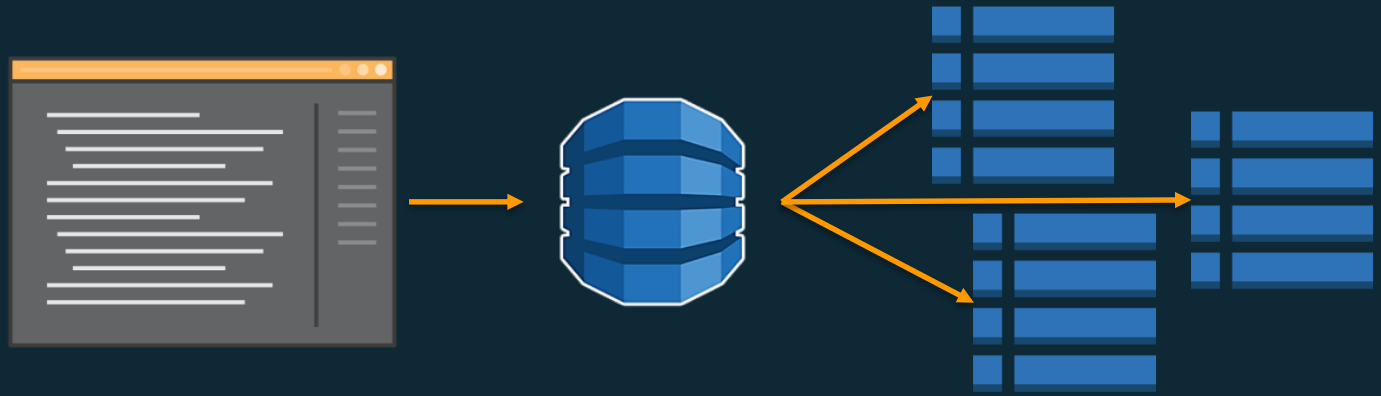
- No built-in support, had to depend on custom transactions client libraries or create and maintain their own client library.
- Added complexity of modifying their application logic to work around the lack of built-in transactions support.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Amazon DynamoDB Transactions

Simplify your code by executing multiple, all-or-nothing actions within and across tables with a single API call.



© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Benefits of DynamoDB transactions

Simplifies the developer experience of making coordinated, all-or-nothing changes to multiple items.

Faster performance and lower latency by using server-side transactions library.

Easier to extend the scale, performance, and enterprise features of DynamoDB to a broader set of workloads.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Use cases for transactions

Execute financial transactions.

Process orders.

Coordinate actions in multi-player games.

Update distributed data.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Transactional API at a glance

TransactWriteItems

- Include up to 10 unique items per transaction.
- Evaluate conditions.
- If all conditions are simultaneously true, perform write operations.
- Consumes 2 WCUs for every 1 KB of data.

TransactGetItems

- Include up to 10 unique items per transaction.
- Return a consistent, isolated snapshot of all items.
- Consumes 2 RCUs for every 4 KB of data.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

DynamoDB flexibility

The same transaction can include items:

- With the same or different partition keys.
- Within the same table or across different DynamoDB tables.

All items within a single transaction must be:

- Within the same AWS Region.
- Within the same AWS account.
- Within DynamoDB.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

DynamoDB scale

Transactions scale like the rest of DynamoDB:

- Virtually limitless concurrent transactions.
- Low latency, high availability.
- Never deadlocks.
- Request/response-based implementation.
- Scales horizontally.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Reading and writing data

	Non-transactional (Singleton)	Transactional
Reads	BatchGetItem Query Scan GetItem	TransactGetItems
Writes	BatchWriteItem PutItem UpdateItem DeleteItem	TransactWriteItems

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Examples

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

User profile management (example #1)

Requirements

- Users identified by user name and optionally email.
- No two users can share the same identifier (user name or email).
- Users can change their email.

What are we demonstrating

- Enforcing unique constraints.
- Strongly consistent global indexing.
- Heterogeneous tables.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

User profile management

Solution: Users table containing two kinds of records:

- Profile records, keyed by opaque user name with user details.
- Alias records, keyed by user email and referencing the profile record.

Id	UserNameRef	Email	PhoneNum
“John123”		“john@example.com”	“444-555-0110”
“john@example.com”	“John123”		
“jane456”			“444-555-0199”

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Insert new profile

```
data = await dynamoDb.transactWriteItems({ TransactItems: [

  { Put: {

       TableName: 'Users',

       Item: { Id: { S: 'John123' }, Email: {S: 'john123@example.com'}, ...},

       ConditionExpression: 'attribute_not_exists(Id)‘ }},
  { Put: {

       TableName: 'Users',

       Item: { Id: { S: 'john123@example.com' }, UserNameRef: {S: 'John123'}},

       ConditionExpression: 'attribute_not_exists(Id)',}}

   ] }).promise();
```

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Update email address

```
data = await dynamoDb.transactWriteItems({ TransactItems: [
{ Put: { TableName: 'Users',
    Item: { Id: { S: 'John123' }, Email: {S: 'john456@newemail.org'}, ...},
    ConditionExpression: 'attribute_exists(Id) and Email = :old_email',
    ExpressionAttributeValues: {':old_email' :{'S': 'john123@example.com'}}}},
{ Put: { TableName: 'Users',
    Item: { Id: { S: 'john456@newemail.org' }, UserNameRef: {S: 'John123'}},
    ConditionExpression: 'attribute_not_exists(Id)', }},
{ Delete: { TableName: 'Users',
    Item: { Id: { S: 'john123@example.com' }}}).promise();
```

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management (use case #2)

Requirements

- Guests create room reservations.
- Guests check in to rooms, fulfilling a reservation.
- Guests check out of rooms, closing a reservation.

What are we demonstrating?

- Idempotency – Operations with side effects happen at most once.
- Consistency – Multi-item changes are only triggered when valid.
- Atomicity – Ensuring multi-item changes happen all together.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management data model

Guests

- Id (primary key)
- Name, Email, etc.
- Reservations (set)
- OccupiesRooms (set)

Reservations

- Id (primary key)
- GuestId
- FulfilledByRoom
- State

Rooms

- Id (primary key)
- RentedToReservation
- State

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management data model

Record Type	Id (PK)	Attributes
Guest	“John”	Reservations : { “501”, “502” } OccupiesRooms : { “20014” }
Reservation	“501”	GuestId: “John” State: “FULFILLED” FulfilledByRoom: “20014”
Reservation	“502”	GuestId: “John” State: “PENDING”
Room	“20014”	State: “OCCUPIED” RentedToReservation : “501”

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Creating a reservation

Use idempotency to ensure that only a single reservation is created, even with multiple retries.

Client generates a unique reservation ID.

Client generates TransactWriteItems request with the following operations:

- Conditional Put for the new reservation.
- Update the guest reservations set and add the new reservation.

Client transaction is idempotent and can be retried without double booking.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management – Check-in

Record Type	Id (PK)	Attributes
Guest	“John”	Reservations : { “501” } OccupiesRooms : {}
Reservation	“501”	GuestId: “John” State: “PENDING” FulfilledByRoom: null
Room	“20014”	State: “FREE” RentedToReservation: null

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management – Check-in

Record Type	Id (PK)	Attributes
Guest	“John”	Reservations : { “501” } OccupiesRooms : { “20014” }
Reservation	“501”	GuestId: “John” State: “FULFILLED” FulfilledByRoom: “20014”
Room	“20014”	State: “OCCUPIED” RentedToReservation : “501”

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management – Checkout

Record Type	Id (PK)	Attributes
Guest	“John”	Reservations : { “501” } OccupiesRooms : { “20014” }
Reservation	“501”	GuestId: “John” State: “FULFILLED” FulfilledByRoom: “20014”
Room	“20014”	State: “OCCUPIED” RentedToReservation : “501”

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Hotel reservation management – Checkout

Record Type	Id (PK)	Attributes
Guest	“John”	Reservations : { ~~“501”~~ } OccupiesRooms : { ~~“20014”~~ }
Reservation	“501”	GuestId: “John” State: “CLOSED” FulfilledByRoom: “20014”
Room	“20014”	State: “FREE” RentedToReservation : ~~“501”~~

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Attachment management (use case #3)

Requirements

- A social media site allows users to attach media to posts.
- Posts need to share media efficiently.
- Posts can have multiple attachments.
- Unreferenced media needs to be deleted.

What are we demonstrating?

- External resource metadata and lifetime management.
- Using Time To Live (TTL) and streams.
- Reference counting using transactions.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Attachment management data model

Attachment item

- S3Ref (PK) – reference to an object in Amazon Simple Storage Service (Amazon S3).
- RefCount – # of posts referencing this attachment.
- TTL – time when this attachment is eligible to be deleted.

Post item

- Id (PK) – post ID.
- Attachments (set) – set of attachment IDs.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Attachment management invariants

1. RefCount = number of posts referencing the attachment.
2. If RefCount > 0, then S3Ref refers to a valid Amazon S3 object.
3. TTL is set if, and only if, RefCount = 0.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Create attachment and link it to a post

- Generate unique Amazon S3 object name.
- Create attachment record in DynamoDB.
  - Set RefCount = 0
  - Set TTL = 24 hours from now.
- Upload object to S3.
- In a TransactWriteItems Request.
  - Check that the attachment is not already referenced by the post.
  - Increase RefCount by 1, remove TTL.
  - Add attachment reference to post.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Clone attachment reference

- Copy the reference of an attachment from one post to another.
- In a TransactWriteItems Request.
  - Check that the attachment is not already referenced by the new post.
  - Increase RefCount by 1, remove TTL.
  - Add attachment reference to new post.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Delete unused external resources

- Attachment records will be deleted a day after their reference count drops to zero.
- In AWS Lambda, process stream records representing items deleted by TTL.
- Delete the corresponding objects from Amazon S3.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Integrating DynamoDB transactions in your applications

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Concurrency control

Optimistic

- No deadlocks possible.
- Low latency.
- Optimized for scale out.

Your responsibilities:

- Design for scale out.
- Avoid unnecessary conflicts.
- Design to avoid hot keys.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Why didn’t my transaction succeed?

Item-based failure reasons:

- Condition failure.
- Concurrent transaction on the same item is in-progress.

Other reasons:

- Insufficient capacity.
- Internal server error (HTTP 500).
- Malformed request.
- Insufficient permissions.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Retrying transactions: ClientRequestToken

- Ensures SDK can retry without “replaying” the same intent multiple times.
- ClientRequestToken – idempotency token at the SDK/API level
  - Autogenerated, you can override.
  - Applies for 10 minutes.
  - Provides information about transaction state and outcome.
  - Sometimes doesn’t remove the need for app-level idempotency/intent design.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

App-level retries

Necessary when SDK retries have been exhausted.

- Build a new view of the world.
- Retry with new preconditions and desirable end state.

How can you get a snapshot of the world? Three options:

- Set ReturnValuesOnConditionCheckFailure = ALL_OLD.
- Perform TransactGetItems.
- Work from an eventually consistent view (results of queries, etc.).

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Integrating with other DynamoDB features

- General rule about DynamoDB data
  - Committed state only
  - But may not be isolated
- Examples
  - Streams – Committed writes, sharded
  - Backups and point-in-time recovery (PITR) – Subset of a committed transaction is possible
  - Global secondary index (GSI) – Committed writes are indexed individually
  - Global tables – Transactional API disabled by default; writes are replicated individually.
  - DynamoDB local – Transactional API calls supported, but TransactionConflictExceptions need to be simulated using testing framework.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Access control

- Individual operations are authorized separately
  - (NEW) dynamodb:ConditionCheck
  - dynamodb:UpdateItem
  - dynamodb:PutItem
  - dynamodb:DeleteItem

- No separate permissions for top level new APIs

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Metering transactions

Transactions uses a two-phase commit protocol.

1 read/write to prepare the transaction and 1 read/write to commit the transaction.

Reads consume 2 RCUs for every 4 KBs of data read.

Writes consume 2 WCUs for every 1 KB of data written.

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

Thank you!

Arturo Hinojosa

Contact us through the AWS forums or follow us on Twitter at @dynamodb

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.