The DynamoDB Book

Glossary

- **Attributes**: Properties on a DynamoDB item. Comparable to a column in a relational database, except attributes do not need to be defined on the table. Attributes have a name and a value. Each attribute value has a specific type. There are ten different types of attributes: String, Number, Binary, Boolean, Null, Map, List, String Set, Number Set, Binary Set

- **Composite Primary Key**: A primary key consisting of two elements: a partition key and a sort key. Used to model complex relationships between items.

- **Consistency**: When reading an item, consistency refers to the guarantee as to whether you're reading the most recent version of an item. By default, DynamoDB is eventually-consistent, meaning your read may not reflect all writes to an item. You can opt into strongly-consistent reads, which will guarantee you have received the most recent write. Eventually-consistent reads consume half as many read capacity units as strongly-consistent reads.

- **Global Secondary Index**: A secondary index that can use any primary key combination. A global secondary index can be added or removed from your table at any time. It uses provisioned capacity separately from that of your underlying table. You cannot make strongly-consistent reads from a global secondary index. Global secondary indexes are preferred due to their flexibility.

- **Item**: An individual record in DynamoDB. Comparable to a row in a relational database.

- **Item Collection**: All items with the same partition key in a base table or secondary index. Item collections are critical for grouping your data and for handling relationships with DynamoDB.

- **Local Secondary Index**: A secondary index that uses the same partition key and a different sort key as your base table. A local secondary index must be created when your table is created, and it uses the same provisioned throughput as your base table. You can make strongly-consistent reads on your local secondary index. Local secondary indexes are disfavored due to their inflexibility.

- **Overloading keys and indexes**: In a single-table design, you include multiple items of different types in a single table. With overloaded keys and indexes, the attributes in the primary keys mean different things for different item types.

Glossary

- **Partition**: A storage concept underlying DynamoDB. Your data in DynamoDB is split across multiple partitions. This allows you to receive consistent performance as your data scales.

- **Partition Key**: Part of the primary key. A required element of each item. Used to determine which storage node holds a particular item.

- **Primary key**: Unique identifier for an item. A primary key is set on a table, and each item must contain the primary key. The primary key uniquely identifies each item in the table. Almost all access patterns rely on the primary key. There are two types of primary keys: simple primary keys and composite primary keys.

- **Provisioned throughput**: If using provisioned throughput billing mode, you set a specific number of read capacity units and write capacity units that are available for your table and secondary indexes. You pay an hourly cost for those units.

- **Read capacity units**: Read capacity units are how you are billed for reads in DynamoDB. Each read of up to 4KB of data consumes one read capacity unit. Eventually consistent reads consume half as much capacity, and transactions consume twice as much capacity as normal.

- **Secondary index**: A copy of your data with a different primary key. You create a secondary index on a table by specifying the primary key of the secondary index. DynamoDB will replicate your table's data to that secondary index with the new primary key format. There are two types of secondary index: local secondary index and global secondary index.

- **Simple primary key**: A primary key consisting of just one element: a partition key. Used for key-value access.

- **Single-table design**: The recommended data modeling strategy in DynamoDB. In single-table design, all entities in your application are kept in a single table. You use overloaded keys and indexes to group your items into item collections designed for specific access patterns.

- **Sort key**: Part of a composite primary key. Used to sort items within a particular item collection

Glossary

- **Sparse index**: A strategy for modeling data with secondary indexes. A secondary index will only copy an item from a base table if the item has all elements of the secondary index's primary key. You can use this to strategically filter data in your secondary index.

- **Time-to-live (TTL)**: An attribute set on your table to automatically expire items. If you set the attribute, DynamoDB will remove any item in your table that has the attribute and whose value is before the current time. The property value must be a number representing the epoch timestamp in seconds when the item should be deleted. Items are generally deleted within 48 hours, though there is no SLA.

- **Transaction**: Transactional operations allow you to combine multiple read or write requests in one operation. The requests will succeed or fail together. Transactions consume twice as much capacity as their corresponding non-transactional requests.

- **Write capacity units**: Write capacity units are how you are billed for writes in DynamoDB. Each 1KB write of data consumes one write capacity unit. Transactions consume twice as much capacity as normal.

One-to-Many Relationships

WHAT

A one-to-many relationship is when one entity (the 'parent') has a number of related entities

EXAMPLES

- **Office >> Employees** (Many employees work in one office building)
- **Customer >> Orders** (A customer makes multiple orders)

STRATEGIES

- **Denormalization + complex attribute**: Store the related entities on the parent items as a complex attribute, such as a list or a map..

    - Good when:

        - you have no direct access patterns on the related entities directly; AND
        - the number of related entities is limited (because of 400KB limit on DDB item)

    Example: Customers and mailing addresses

The DynamoDB Book

Strategies Cheatsheet

One-to-Many Relationships

- **Denormalization + duplication**: Duplicate information about the parent entity into each related item.
  - Good when:
    - The duplicated information is immutable; OR
    - The duplicated information changes infrequently and is not spread across many items

    Example: Books and authors (store an author's biographical information on each book)

- **Primary key + query:** Use a composite primary key to include both the parent and related entities in the same item collection. This is the most common one-to-many relationship pattern.
  - Good when:
    - You have access patterns on both the parent and the related items
    - Have one access pattern where you want to grab both the parent and all related items (similar to a SQL join)

    Example: Customer and Orders in e-commerce store

One-to-Many Relationships

- **Secondary index + query**: Use a secondary index to include parent and related entities in the same item collection.
    - Good when:
        - same as previous strategy, but the primary key is used for something else

      Example: Orders and Order Items in e-commerce store
- **Composite sort key:** Encode multiple levels of data in sort key
    - Good when:
        - Multiple levels (2+) of hierarchy to model
        - When fetching a hierarchy, you want to get all subitems of that hierarchy

      Example: Starbucks store locations

The DynamoDB Book

Strategies Cheatsheet

Many-to-Many Relationships

WHAT

A many-to-many relationship is when multiple objects of one type are related to multiple objects of a different type

EXAMPLES

- **Students <<->> Classes** (A student enrolled in multiple classes; a class has multiple students)
- **Movies <<->> Actors** (A movie has multiple actors; an actor has a role in multiple movies)

CHALLENGE

How to get fresh data on both sides of the relationship

Many-to-Many Relationships

STRATEGIES

- **Shallow duplication:** If one side of the relationship only needs a little data about the other side, you can add just that data in a complex attribute on the related item.
  - Good when:
    - One side of the relationship only needs a little bit of data about the related items
    - Limited number of related items (due to 400KB item size limit)

    > Example: Students in class if you don't need all information about the students when showing the class

- **Adjacency list:** Model each entity as an item type and model the relationship as an item type. One side of the relationship is in the same item collection for the primary key, and the other side is in the same item collection in a secondary index.
  - Good when:
    - You have access patterns to fetch the parent and all related items on both sides of the relationship
    - Information about the relationship itself is immutable

    > Example: Movies, Actors, and Role

Many-to-Many Relationships

- **Materialized Graph**: Break nodes into individual relationships with other nodes. Use a secondary index to find all items that have relationships with a given node.
  - Good when:
    - Lots of different node types and relationship types
    - You have a Ph.D in graph theory

- **Normalization + multiple requests**: You store a record about related items. After fetching an item and all relations, you then make a follow-up request to get information about the related items.
  - Good when:
    - Information about the related items is highly mutable and highly duplicated

Example: Social network friendship relationships

Filtering Strategies

KEY

The majority of filtering should be done with your primary key

STRATEGIES

- **Filtering with your partition key:**
  - All data with the same partition key is located together
  - Find a high-cardinality value to segment your data
    - e.g., UserID, OrderID
  - Most important concept in DynamoDB data modeling

- **Filtering with the sort key**:
  - Data within the same partition is sorted in order of UTF-8 bytes
  - You can fetch a range of items within a partition by including conditions on the sort key in a Query
  - Two ways to use sort keys:
    - Sort key itself is meaningful

      Example: Sort key is timestamp. Use sort key conditions on Query to find the time you want.
    - Sort key is important due to how you've arranged data.

      Example: Multiple item types in item collection. Use sort key to "join" data.
- **Composite sort key**:
  - Encode multiple values into the sort key

    Example: OrderStatusDate (combination of OrderStatus + Date of Order)
  - Use when you always want to filter on multiple attributes and at least one of the attributes is an enum-like value

Filtering Strategies

- **Sparse indexes:**
    - Items are only copied to secondary index if the item has all elements of the secondary index's primary key
        - Can be used as a global filter on your dataset
    - Two patterns:
        - Filtering within an entity type based on a condition
            - E.g.: Unread messages only (see Chapter 20)
            - Can be used with an overloaded secondary index
        - Projecting a single type of entity into a secondary index
            - E.g.: Only Users into index (see Chapter 20)
            - Cannot be used with an overloaded index

- **Filter expressions**:
    - Expression applied to filter Query or Scan results after items are read but before they are returned to the client
    - DO NOT rely on these to compensate for bad table design:
        - 1 MB read limit is applied before filter
        - Still charged for all items read before filter
    - Use cases:
        - Reduce payload size to client
        - Easier application filtering
        - Better validation around TTL expiry

- **Client-side filtering:**:
    - Do advanced filtering & sorting in the client (e.g., browser) rather than in the database
    - Good when:
        - Many complicated filtering and sorting patterns to support
        - Data size is not large

Strategies Cheatsheet

Sorting

KEY

You must consider sorting when modeling

BASICS

- Within a partition, sort keys are sorted
    - No discernible ordering across partition keys (sorted after partition key is hashed)

- Sorted in order of UTF-8 bytes:
    - Alphabetical
    - Uppercase before lowercase
    - Numbers and symbols relevant, too

- Can use epoch timestamps or ISO 8601 format for timestamps
    - ISO 8601 is human-readable so preferred
    - Use epoch timestamp if you need to perform math or use for TTL

- For unique, sortable IDs, use KSUIDs
    - K-Sortable Unique IDentifier
    - Provides low chance of collision, like a UUID
    - Chronologically-sortable to fetch oldest or most-recent items

Strategies Cheatsheet

Sorting

OTHER STRATEGIES

- Sorting on changing attributes:
    - If an element in your primary key changes, you can't update the item. You need to delete the existing item and create a new one.
    - Thus, if you're using an attribute that updates for sorting (e.g., UpdatedAt timestamp), model it in your secondary index. DynamoDB will handle delete + create.

- Ascending vs. descending
    - By default, Query and Scan reads in ascending order
    - If you want descending (e.g., Fetch most recent items), then you need to use ScanIndexForward=False in Query or Scan operation.

- Two relational access patterns in one item collection
    - If you have an entity that is the parent in two one-to-many relationships, you can sometimes model both relationships in the same item collection.
    - Put parent entity between the two related entity types in the item collection and filter accordingly.
    - Access patterns must sort in different directions.

- Zero-padding when sorting with numbers
    - If using numbers in your sort key but in string format, be sure to zero-pad your numbers.
    - String numbers evaluated from left to right, so "10" comes before "2", unless you zero-pad: "00010" vs "00002"

Migration Strategies

KEY QUESTION

Does my migration require modifying existing items? If not, it's additive (much easier)

MIGRATION STRATEGIES

- Adding attributes to an existing entity:
  - Modify your application code to write the new attribute
  - Handle defaults for those without it
  - Doesn't require changing existing items

- Adding a new entity without relations
  - Usually your new entity is related to something, but might not have a relational (e.g., "join") access pattern.
  - Write new entity into its own item collection.
  - Does not require changing existing items.

- Adding a new entity type into an existing item collection
  - New entity that does have a 'join' relationship with an existing entity
  - No room in data model to put into existing item collection
  - Add new entity and add new attributes to existing entity (see below on performing a migration)

- Joining existing items into a new item collection
  - Combining two items or adding new access pattern to an entity
  - Migration to add new attributes to existing entity (see below on performing a migration)

Migration Strategies

PERFORMING A MIGRATION

1. Update application code to start writing new attributes to all entities going forward.
2. Run an ETL operation to update existing entities
    a. Scan table to find all entity types that need updating.
    b. For each item found, add new attributes to the item.
3. Add new secondary index if needed

PARALLEL SCAN

- Scanning an entire table can be slow. To speed it up, use a parallel scan.
- For each worker in the scan, include two parameters:
    - TotalSegments: the total number of workers to process the scan
    - Segment: the number to identify this specific worker
- DynamoDB will handle state management for the parallel scan

Additional Strategies

- **Uniqueness on 2+ attributes**:
    - Must have an item for each attribute that you want to keep unique
    - When adding a new entity, use a transaction to create a new item type for each attribute you want to keep unique

- **Handling sequential IDs**:
    - Avoid if possible; try to use more meaningful identifiers
    - If you need it, two step process:
        1. Update parent items to increment counter and return UPDATED_NEW attributes
        2. Use returned counter value as ID for new item

- **Pagination**:
    - Use Limit to return the number of items you want
    - In API design, be sure to pass back the 'last seen' item so that you can start your Query there

- **Singleton items**:
    - Most items have a parameter in the primary key to segment across different instances
    - Sometimes you need a single item that is used across the entire application
        - E.g., Overall Jobs status; Content for a specific page in your application.
    - Use static values for primary key without any parameters
    - Watch out for hot key issues

- **Reference counts**
    - Use transaction to add related item and increment count of total relations on parent item.

Expressions Cheatsheet

Basics

Expressions are statements written as strings that alter existing items or API actions in some way. They are comparable to mini-SQL statements.

Five types of expressions:

- **Key condition expressions**: Used in the Query API call to describe which items you want to retrieve in your query
- **Filter expressions**: Used in the Query and Scan operations to describe which items should be returned to the client after finding items that match your key condition expression
- **Projection expressions**: Used in all read operations to describe which attributes you want to return on items that were read
- **Condition expressions**: Used in write operations to assert the existing condition (or non-condition) of an item before writing to it
- **Update expressions**: Used in the UpdateItem call to describe the desired updates to an existing item

Expression Names and Values

- ExpressionAttributeNames and ExpressionAttributeValues are used to refer to the names and values of attributes you use in your expressions.
- ExpressionAttributeValues are required when referencing values.
- ExpressionAttributeNames are not required when referencing attribute names, unless using a reserved word.

ExpressionAttributeValues:

- Start with a colon (:)
- Must include a type (unless using DocumentClient).

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: 'Actor = :actor',
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' }
     }
})
```

ExpressionAttributeNames:

- Start with a pound sign (#)
- Not required, unless using a reserved word for attribute name
- There are over 500 reserved words, including commonly-used names like Count, Month, Name, & Timestamp
- RECOMMENDATION: Prefer using ExpressionAttributeNames in all requests for consistency and simplicity

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: '#actor = :actor',
     ExpressionAttributeNames: {
          '#actor': 'Actor'
     },
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' }
     }
})
```

Key Condition Expressions

- Key condition expressions are used with the Query operation to describe the items you want
- A key condition expression must include an exact match for a partition key

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: '#actor = :actor',
     ExpressionAttributeNames: {
          '#actor': 'Actor'
     },
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' }
     }
})
```

- A key condition expression may include conditions on the sort key as well
- A key condition expression may not specify conditions on attributes that are not in the primary key

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: '#actor = :actor'
     AND #movie BETWEEN :a AND :m,
     FilterExpression: "#year > :year",
     ExpressionAttributeNames: {
          '#actor': 'Actor',
          '#movie': 'Movie'
     },
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' },
          ':a: { 'S': 'A' },
          ':m' { 'S': 'M' }
     }
})
```

The DynamoDB Book

Expressions Cheatsheet

Filter Expressions

Filter expressions allow you to filter items based on non-key attributes

Filter expressions will not save your bad table design.

When performing a Query or Scan operation, the following three steps occur:

1. DynamoDB reads items from your table
2. If there is a filter expression, all items not matching the filter expression are removed
3. The results are returned to you

Important: the 1MB item size limit for Query and Scan applies in Step 1, before the filter expression is performed.

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: '#actor = :actor',
     FilterExpression: "#year > :year",
     ExpressionAttributeNames: {
          '#actor': 'Actor',
          '#year': 'Year'
     },
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' },
          ':year: { 'S': '2000' }
     }
})
```

Projection Expressions

- Projection expressions can be used with any read operation to limit the attributes that are returned with items
- Like filter expressions, projection expressions are applied after items are read from the table

```
const response = dynamodb.query({
     TableName: 'MoviesAndActors',
     KeyConditionExpression: '#actor = :actor',
     ProjectionExpression: "#actor, #movie, #role",
     ExpressionAttributeNames: {
          '#actor': 'Actor',
          '#year': 'Year',
          '#role': 'Role'
     },
     ExpressionAttributeValues: {
          ':actor': { 'S': 'Tom Hanks' }
     }
})
```

Expressions Cheatsheet

Condition Expressions

Condition expressions can be used on all operations that alter data (PutItem, UpdateItem, DeleteItem, etc.).

The condition expression is evaluated before the write. If the expression is false, the write will not be performed.

This example uses the attribute_not_exists ( ) function to avoid overwriting an item with the same key.

```
const response = dynamodb.putItem({
  TableName: 'Users',
  ConditionExpression: "attribute_not_exists(#username)",
  Item: {
       "Username": { "S": "bountyhunter1" },
       "Name": { "S": "Boba Fett" },
       "CreatedAt": { "S":(new Date()).toISOString()
  },
  ExpressionAttributeNames={
    "#username": "Username"
  }
})
```

This example uses the contains ( ) function to only update the item if the requesting user is in the Admins set attribute for the item.

```
const results = dynamodb.updateItem({
  TableName: 'BillingDetails',
  Key: {
    "PK": { "S": 'Amazon' }
  }
  ConditionExpression: "contains(#a, :user)",
  UpdateExpression: "Set #st :type",
  ExpressionAttributeNames: {
    "#a": "Admins",
    "#st": "SubscriptionType"
  },
  ExpressionAttributeValues: {
    ":user": { "S": 'Jeff Bezos' },
    ":type": { "S": 'Pro' }
  }
})
```

Expressions Cheatsheet

Update Expressions

Update expressions are used in UpdateItem operations to indicate the updates to be performed.
Update expressions save extra requests to DynamoDB as you don't need to retrieve the item before changing it.

In this example, the update expression is updating the user's profile picture URL to a new value.

```
const response = dynamodb.updateItem(
   TableName='Users',
   Key: {
      "Username": { "S": "dynamodb_fan" }
   }
   UpdateExpression: "SET #picture :url",
   ExpressionAttributeNames: {
      "#picture": "ProfilePictureUrl"
   },
   ExpressionAttributeValues: {
      ":url": { "S": <https://....> }
   }
)
```

There are four verbs that can be used with Update Expressions

SET: Set the value of an attribute or increment an existing number.
REMOVE: Remove an existing attribute.
ADD: Increment an existing number or add element to a set
DELETE: Remove element from a set

JavaScript Cheatsheet

Setup and Usage

Installation:

```
npm install aws-sdk
```

Creating a Client:

```
const AWS = require('aws-sdk')
const dynamodb = new AWS.DynamoDB();
```

Customizing your Client:

- Use the 'region' parameter to set your region.
- Use the 'endpoint' parameter when using with DynamoDB Local

```
const dynamodb = new AWS.DynamoDB({
  region: 'us-east-1',
  endpoint: 'http://localhost:8000'
});
```

Promise Support:

- Add a ".promise()" to the end of your command to return an awaitable Promise.

```
const item = await dynamodb.getItem({
  TableName: 'MyTable',
  Key: {
    'Id': { 'S': '1234' }
  }
}).promise()

console.log(item)
```

The DynamoDB Book

JavaScript Cheatsheet

Basic Operations

PutItem:

- Used for writing an item to a table
- Will completely overwrite any existing item
- Use UpdateItem to only modify certain attributes

```
const item = await dynamodb.putItem({
  TableName: 'MyTable',
  Item: {
    "Username": { "S": "myuser" },
    "Name": { "S": "Bob G. User" },
    "Location": { "S": "Phoenix, AZ" },
  }
 }
*/
```

Use ConditionExpression to prevent overwriting existing items.

```
// This will throw a ConditionalCheckFailedException
// if an item with the given Username exists
const item = await dynamodb.putItem({
  TableName: 'MyTable',
  Item: {
    "Username": { "S": "myuser" },
    "Name": { "S": "Bob G. User" },
    "Location": { "S": "Phoenix, AZ" },
  },
  ConditionExpression: "attribute_not_exists(Username)"
}).promise()
```

GetItem:

- Used for reading a single item from a table
- Will throw an exception if the item does not exist (TODO: Confirm)

```
const item = await dynamodb.getItem({
  TableName: 'MyTable',
  Key: {
    "Username": { "S": "myuser" }
  }
}).promise()

console.log(item)
/* Output:
  data ={
    Item: {
      "Username": { "S": "myuser" },
      "Name": { "S": "Bob G. User" },
      "Location": { "S": "Phoenix, AZ" },
    }
  }
*/
```

Basic Operations

Update Item:

- Used for writing an item to a table or updating an item if it exists
- If the item exists, it will only modify the attributes given in the UpdateExpression

```
const item = await dynamodb.updateItem({
  TableName: 'MyTable',
  Key: {
    "Username": { "S": "myuser" }
  },
  UpdateExpression: "SET #location = :location",
  ExpressionAttributeNames: {
    "#location": "Location"
  },
  ExpressionAttributeValues: {
    ":location": { "S": "Omaha, NE" }
  }
}).promise()
```

Use ReturnValues to control the information that is sent back

Options for ReturnValues:

- **NONE:** No attributes are returned (Default setting)
- **ALL_OLD:** The entire item before the update operation is returned
- **UPDATED_OLD:** The previous values for only the attributes updated in the operation are returned
- **ALL_NEW:** The entire item after the update operation is returned
- **UPDATED_NEW:** The current values for only the attributes updated in the operation are returned

```
const item = await dynamodb.updateItem({
  TableName: 'MyTable',
  Key: {
    "Username": { "S": "myuser" }
  },
  UpdateExpression: "SET #location = :location",
  ExpressionAttributeNames: {
    "#location": "Location"
  },
  ExpressionAttributeValues: {
   ":location": { "S": "Omaha, NE" }
  }
  ReturnValues: 'UPDATED_NEW'
}).promise()

console.log(item)
/* Output:
  data ={
    Attributes: {
      "Location": { "S": "Omaha, NE" }
    }
  }
*/
```

DeleteItem:

- Used for reading a single item from a table
- Idempotent operation. Will not throw an exception if the item does not exist
- Can add ConditionExpression to only delete under certain conditions
- Can add ReturnValues to control what is returned

```
const item = await dynamodb.deleteItem({
  TableName: 'MyTable',
  Key: {
    "Username": { "S": "myuser" }
  }
}).promise()
```

Options for ReturnValues:

- **None:** No attributes are returned (Default Setting)
- **ALL_OLD:** The entire item before the delete operation is returned

```
const item = await dynamodb.deleteItem({
  TableName: 'MyTable',
  Key: {
    "Username": { "S": "myuser" }
  },
  ReturnValues: "ALL_OLD"
}).promise()
```

Batch Operations

The two batch operations - BatchReadItem and BatchWriteItem - are used to combine multiple operations in a single request. All reads or writes in a batch will succeed or fail independently.

The response from a batch operation will include an "UnprocessedKeys" property (for BatchGetItem) or an "UnprocessedItems" property (for BatchWriteItem) that includes any items that were not processed due to size limits, throttling errors, or other errors. Retry these requests to ensure all items are processed.

BatchGetItem:

- Can request up to 100 items and receive up to 16MB of data
- Can read items from multiple tables
- Unhandled items are returned in the "UnprocessedKeys" property

```
const items = await dynamodb.batchGetItem({
  RequestItems: {
    "MyFirstTable": {
      Keys: [
        { "Username": { "S": "user123" },
        { "Username": { "S": "user567" }
      ]
    }
    "MySecondTable": {
      Keys: [
        { "OrderId": { "S": "dbdf628e" },
        { "OrderId": { "S": "2b55f660" }
      ]
    }
  }
}).promise()
```

Batch Operations

BatchWriteItem:

- Can write up to 25 items and up to 16MB of data
- Can write items to multiple tables
- Only puts and deletes allowed - no updates to existing items
- No ConditionExpressions allowed
- Unhandled items are returned in the "UnprocessedItems" property

```
const items = await dynamodb.batchWriteItem({
  RequestItems: {
    "MyFirstTable": [
      {
        PutRequest: {
          Item: {
            "User": { "S": "user123" }
          }
        }
      },
      {
         PutRequest: {
           Item: {
             "User": { "S": "user123" }
           }
         }
       }
    ]
  }
  }).promise()
```

Transactions

The two transactional operations - TransactGetItems and TransactWriteItems - are used to combined multiple operations in a single request. All reads or writes in a transaction will succeed or fail together.

TransactGetItems:

- Can request up to 25 items and receive up to 4MB of data
- Can read items from multiple tables
- Will throw an error if any of the requested items are being updated in another operation

```
const items = await dynamodb.transactGetItems({
  TransactItems: [
    Get: {
      Key: { "Username": { "S": "user123" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "Username": { "S": "user567" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "OrderId": { "S": "dbdf628e" },
      TableName: "MySecondTable"
    }
  ]
}).promise()
```

The DynamoDB Book

JavaScript Cheatsheet

Transactions

TransactWriteItems:

- Can write up to 25 items with up to 4MB of data
- Can write items from multiple tables
- Four operation types:
    - Put
    - Update
    - Delete
    - ConditionCheck
- ConditionCheck does not change an item but asserts a condition about an item

```
const items = await dynamodb.transactWriteItems({
  TransactItems: [
    {
      Put: {
        Item: {
          "Username": { "S": "user123" },
          "Name": { "S": "Bob G. User" }
        },
        TableName: "MyFirstTable"
      }
    },
    {
      Put: {
      Delete: {
        Key: { "Username": { "S": "user567" },
        TableName: "MyFirstTable"
    }
  ]
}).promise()
```

Idempotence:

Ensure idempotency of a TransactWriteItem request by specifying the "CleintRequestToken."

If a TransactWriteItem request was made with the same ClientRequestToken and payload within the last 10 minutes, DynamoDB will ensure it was only processed once.

```
const params = {
  TransactItems: [ ... ],
  ClientRequestToken: "MyIdempotencyToken"
}
const first = await dynamodb.transactWriteItems(params).promise()

// This will be an idempotent request
const second = await dynamodb.transactWriteItems(params).promise()
```

JavaScript Cheatsheet

Query

Query is used to read multiple items in a single item collection. An item collection refers to all items with the same partition key.

You must provide the partition key of your table or index when using a Query.

```
const response = dynamodb.query({
  TableName: 'MoviesAndActors',
  KeyConditionExpression: '#actor = :actor',
  ExpressionAttributeNames: {
    '#actor': 'Actor'
  },
  ExpressionAttributeValues: {
    ':actor': { 'S': 'Tom Hanks' }
  }
})
```

You may provide conditions on the sort key of your Query to filter the results you get back.

```
const response = dynamodb.query({
  TableName: 'MoviesAndActors',
  KeyConditionExpression: '#actor = :actor AND #movie BETWEEN :a
AND :m'
  KeyConditionExpression: '#actor = :actor',
  ExpressionAttributeNames: {
    '#actor': 'Actor',
    '#movie': 'Movie
  },
  ExpressionAttributeValues: {
    ':actor': { 'S': 'Tom Hanks' },
    ':a': { 'S': 'A' },
    ':m': { 'S': 'M'
  }
})
```

You may use a FilterExpression to filter on attributes other than your sort key.

When using a FilterExpression, you will still pay for all items read that match your KeyConditionExpression.

```
const response = dynamodb.query({
  KeyConditionExpression: '#actor = :actor',
  FilterExpression: '#release >= :year',
  ExpressionAttributeNames: {
    '#actor': 'Actor',
    '#release': 'ReleaseYear'
  },
  ExpressionAttributeValues: {
    ':actor': { 'S': 'Tom Hanks' },
    ':year': { 'S': '2000' }
  }
})
```

JavaScript Cheatsheet

Scan

The Scan operation reads all items in your table.

The Scan operation is limited to 1MB of results per request, so you will need to paginate over your data to receive it all.

```
const response = dynamodb.scan({
  TableName: 'MoviesAndActors'
})
```

```
let has_more = true
const args = { TableName: 'MoviesAndActors' }
const items = []

while has_more:
  const response = dynamodb.scan(args)
  items.concat(response.Items)
  if (!response.LastEvaluatedKey) {
    args.ExclusiveStartKey = response.LastEvaluatedKey
  } else {
    has_more = false
  }
```

JavaScript Document Client Cheatsheet

Setup and Usage

Installation:

```
npm install aws-sdk
```

Creating a Client:

```
const AWS = require('aws-sdk')
const docClient = new AWS.DynamoDB.DocumentClient();
```

Customizing your Client:

- Use the 'region' parameter to set your region.
- Use the 'endpoint' parameter when using with DynamoDB Local

```
const docClient = new AWS.DynamoDB.DocumentClient({
  region: 'us-east-1',
  endpoint: 'http://localhost:8000'
});
```

Promise Support:

- Add a ".promise()" to the end of your command to return an awaitable Promise.

```
const item = await docClient.get({
  TableName: 'MyTable',
  Key: {
    'Id': '1234'
  }
}).promise()

console.log(item)
```

JavaScript Document Client Cheatsheet

Basic Operations

PutItem:

- Used for writing an item to a table
- Will completely overwrite any existing item
- Use UpdateItem to only modify certain attributes

```
const item = await dynamodb.put({
   TableName: 'MyTable',
   Item: {
      "Username": "myuser",
      "Name": "Bob G. User",
      "Location": "Phoenix, AZ",
   },
   ReturnValues: "ALL_NEW"
}).promise()

console.log(item)
/* Output:
   data ={
      Item: {
         "Username": "myuser",
         "Name": "Bob G. User",
         "Location": "Phoenix, AZ",
      }
   }
*/
```

Use ConditionExpression to prevent overwriting existing items.

```
// This will throw a ConditionalCheckFailedException
// if an item with the given Username exists
const item = await dynamodb.put({
   TableName: 'MyTable',
   Item: {
      "Username": "myuser",
      "Name": "Bob G. User",
      "Location": "Phoenix, AZ",
   },
   ConditionExpression: "attribute_not_exists(Username)"
}).promise()
```

JavaScript Document Client Cheatsheet

Basic Operations

GetItem:

- Used for reading a single item from a table
- Response will be empty if the item does not exist

```
const item = await dynamodb.get({
  TableName: 'MyTable',
  Key: {
    "Username": "myuser"
  }
}).promise()

console.log(item)
/* Output:
  data ={
    Item: {
      "Username": "myuser",
      "Name": "Bob G. User",
      "Location": "Phoenix, AZ",
    }
  }
*/
```

Basic Operations

Update Item:

- Used for writing an item to a table or updating an item if it exists
- If the item exists, it will only modify the attributes given in the UpdateExpression

```
const item = await dynamodb.update({
  TableName: 'MyTable',
  Key: {
    "Username": "myuser"
  },
  UpdateExpression: "SET #location = :location",
  ExpressionAttributeNames: {
    "#location": "Location"
  },
  ExpressionAttributeValues: {
    ":location": "Omaha, NE"
  }
}).promise()
```

Use ReturnValues to control the information that is sent back

Options for ReturnValues:

- **NONE:** No attributes are returned (Default setting)
- **ALL_OLD:** The entire item before the update operation is returned
- **UPDATED_OLD:** The previous values for only the attributes updated in the operation are returned
- **ALL_NEW:** The entire item after the update operation is returned
- **UPDATED_NEW:** The current values for only the attributes updated in the operation are returned

```
const item = await dynamodb.update({
  TableName: 'MyTable',
  Key: {
    "Username": "myuser"
  },
  UpdateExpression: "SET #location = :location",
  ExpressionAttributeNames: {
    "#location": "Location"
  },
  ExpressionAttributeValues: {
    ":location": "Omaha, NE"
  },
  ReturnValues: 'UPDATED_NEW'
}).promise()

console.log(item)
/* Output:
  data ={
    Attributes: {
      "Location": "Omaha, NE"
    }
  }
*/
```

Basic Operations

DeleteItem:

- Used for reading a single item from a table
- Idempotent operation. Will not throw an exception if the item does not exist
- Can add ConditionExpression to only delete under certain conditions
- Can add ReturnValues to control what is returned

```
const item = await dynamodb.delete({
  TableName: 'MyTable',
  Key: {
    "Username": "myuser"
  }
}).promise()
```

Options for ReturnValues:

- **None:** No attributes are returned (Default Setting)
- **ALL_OLD:** The entire item before the delete operation is returned

```
const item = await dynamodb.delete({
  TableName: 'MyTable',
  Key: {
    "Username": "myuser"
  },
  ReturnValues: "ALL_OLD"
}).promise()
```

JavaScript Document Client Cheatsheet

Batch Operations

The two batch operations - BatchReadItem and BatchWriteItem - are used to combine multiple operations in a single request. All reads or writes in a batch will succeed or fail independently.

The response from a batch operation will include an "UnprocessedKeys" property (for BatchGetItem) or an "UnprocessedItems" property (for BatchWriteItem) that includes any items that were not processed due to size limits, throttling errors, or other errors. Retry these requests to ensure all items are processed.

BatchGetItem:

- Can request up to 100 items and receive up to 16MB of data
- Can read items from multiple tables
- Unhandled items are returned in the "UnprocessedKeys" property

```
const items = await dynamodb.batchGet({
  RequestItems: {
    "MyFirstTable": {
      Keys: [
        { "Username": "user123" },
        { "Username": "user567 }"
      ]
    }
    "MySecondTable": {
      Keys: [
        { "OrderId": "dbdf628e" },
        { "OrderId": "2b55f660 }"
      ]
    }
  }
}).promise()
```

JavaScript Document Client Cheatsheet

Batch Operations

BatchWriteItem:

- Can write up to 25 items and up to 16MB of data
- Can write items to multiple tables
- Only puts and deletes allowed - no updates to existing items
- No ConditionExpressions allowed
- Unhandled items are returned in the "UnprocessedItems" property

```
const items = await dynamodb.batchWrite({
  RequestItems: {
    "MyFirstTable": [
      {
        PutRequest: {
          Item: { S: 'user123' }
        }
      },
      {
        PutRequest: {
          Item: { S: 'user123' }
        }
      }
    ]
  }
}).promise()
```

JavaScript Document Client Cheatsheet

Transactions

The two transactional operations - TransactGetItems and TransactWriteItems - are used to combine multiple operations in a single request. All reads or writes in a transaction will succeed or fail together.

TransactGetItems:

- Can request up to 25 items and receive up to 4MB of data
- Can read items from multiple tables
- Will throw an error if any of the requested items are being updated in another operation

```
const items = await dynamodb.transactGet({
  TransactItems: [
    Get: {
      Key: { "Username": "user123" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "Username": "user567" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "OrderId": "dbdf628e" },
      TableName: "MySecondTable"
    }
  ]
}).promise()
```

JavaScript Document Client Cheatsheet

Transactions

TransactWriteItems:

- Can write up to 25 items with up to 4MB of data
- Can write items from multiple tables
- Four operation types:
  - Put
  - Update
  - Delete
  - ConditionCheck
- ConditionCheck does not change an item but asserts a condition about an item

```
const items = await dynamodb.transactWrite({
  TransactItems: [
    {
      Put: {
        Item: {
          "Username": "user123",
          "Name": "Bob G. User"
        },
        TableName: "MyFirstTable"
      }
    },
    {
      Delete: {
        Key: { "Username": "user567" },
        TableName: "MyFirstTable"
    }
  ]
}).promise()
```

Idempotence:

Ensure idempotency of a TransactWriteItem request by specifying the "CleintRequestToken."

If a TransactWriteItem request was made with the same ClientRequestToken and payload within the last 10 minutes, DynamoDB will ensure it was only processed once.

```
const params = {
  TransactItems: [ ... ],
  ClientRequestToken: "MyIdempotencyToken"
}
const first = await dynamodb.transactWrite(params).promise()

// This will be an idempotent request
const second = await dynamodb.transactWrite(params).promise()
```

Query

Query is used to read multiple items in a single item collection. An item collection refers to all items with the same partition key.

You must provide the partition key of your table or index when using a Query.

```
const response = dynamodb.query({
  TableName: 'MoviesAndActors',
  KeyConditionExpression: '#actor = :actor',
  ExpressionAttributeNames: {
    '#actor': 'Actor'
  },
  ExpressionAttributeValues: {
    ':actor': 'Tom Hanks'
  }
})
```

You may provide conditions on the sort key of your Query to filter the results you get back.

```
const response = dynamodb.query({
  TableName: 'MoviesAndActors',
  KeyConditionExpression: '#actor = :actor AND #movie BETWEEN :a
AND :m'
  KeyConditionExpression: '#actor = :actor',
  ExpressionAttributeNames: {
    '#actor': 'Actor',
    '#movie': 'Movie
  },
  ExpressionAttributeValues: {
    ':actor': 'Tom Hanks',
    ':a': 'A',
    ':m': 'M'
  }
})
```

You may use a FilterExpression to filter on attributes other than your sort key.

When using a FilterExpression, you will still pay for all items read that match your KeyConditionExpression.

```
const response = dynamodb.query({
  KeyConditionExpression: '#actor = :actor',
  FilterExpression: '#release >= :year',
  ExpressionAttributeNames: {
    '#actor': 'Actor',
    '#release': 'ReleaseYear'
  },
  ExpressionAttributeValues: {
    ':actor': 'Tom Hanks',
    ':year': '2000'
  }
})
```

Scan

The Scan operation reads all items in your table.

```
const response = dynamodb.scan({
  TableName: 'MoviesAndActors'
})
```

The Scan operation is limited to 1MB of results per request, so you will need to paginate over your data to receive it all.

```
let has_more = true
const args = { TableName: 'MoviesAndActors' }
const items = []

while has_more:
  const response = dynamodb.scan(args)
  items.concat(response.Items)
  if (!response.LastEvaluatedKey) {
    args.ExclusiveStartKey =
    response.LastEvaluatedKey
  } else {
    has_more = false
  }
```

Setup and Usage

Installation:

```
pip install boto3
```

Creating a Client:

```
import boto3
client = boto3.client('dynamodb')
```

Customizing your Client:

- Use the 'region' parameter to set your region.
- Use the 'endpoint' parameter when using with DynamoDB Local

```
client = boto3.client(
   'dynamodb',
   region_name='us-east-1',
   endpoint_url='http://localhost:8000'
)
```

Basic Operations

PutItem:

- Used for writing an item to a table
- Will completely overwrite any existing item
- Use UpdateItem to only modify certain attributes

```
item = dynamodb.put_item(
  TableName='MyTable',
  Item={
    "Username": { "S": "myuser" },
    "Name": { "S": "Bob G. User" },
    "Location": { "S": "Phoenix, AZ" },
  }
)

print(item)
/* Output:
  data ={
    Item: {
      "Username": { "S": "myuser" },
      "Name": { "S": "Bob G. User" },
      "Location": { "S": "Phoenix, AZ" },
    }
  }
*/
```

Use ConditionExpression to prevent overwriting existing items.

```
// This will throw a ConditionalCheckFailedException
// if an item with the given Username exists
item = dynamodb.put_item(
  TableName='MyTable',
  Item={
    "Username": { "S": "myuser" },
    "Name": { "S": "Bob G. User" },
    "Location": { "S": "Phoenix, AZ" },
  },
  ConditionExpression="attribute_not_exists(Username)"
)
```

Basic Operations

GetItem:

- Used for reading a single item from a table
- Response will be empty if the item does not exist

```
item = client.get_item(
  TableName='MyTable',
  Key={
    "Username": { "S": "myuser" }
  }
)

print(item)
/* Output:
  data ={
    Item: {
      "Username": { "S": "myuser" },
      "Name": { "S": "Bob G. User" },
      "Location": { "S": "Phoenix, AZ" },
    }
  }
*/
```

Basic Operations

Update Item:

- Used for writing an item to a table or updating an item if it exists
- If the item exists, it will only modify the attributes given in the UpdateExpression

```
item = client.update_item(
  TableName='MyTable',
  Key={
    "Username": { "S": "myuser" }
  },
  UpdateExpression="SET #location = :location",
  ExpressionAttributeNames={
    "#location": "Location"
  },
  ExpressionAttributeValues={
    ":location": { "S": "Omaha, NE" }
  }
)
```

Use ReturnValues to control the information that is sent back

Options for ReturnValues:

- **NONE:** No attributes are returned (Default setting)
- **ALL_OLD:** The entire item before the update operation is returned
- **UPDATED_OLD:** The previous values for only the attributes updated in the operation are returned
- **ALL_NEW:** The entire item after the update operation is returned
- **UPDATED_NEW:** The current values for only the attributes updated in the operation are returned

```
item = client.update_item(
  TableName='MyTable',
  Key={
    "Username": { "S": "myuser" }
  },
  UpdateExpression="SET #location = :location",
  ExpressionAttributeNames={
    "#location": "Location"
  },
  ExpressionAttributeValues={
    ":location": { "S": "Omaha, NE" }
  },
  ReturnValues="UPDATED_NEW"
)

print(item)
/* Output:
  data ={
    Attributes: {
      "Location": { "S": "Omaha, NE" }
    }
  }
*/
```

Basic Operations

DeleteItem:

- Used for reading a single item from a table
- Idempotent operation. Will not throw an exception if the item does not exist
- Can add ConditionExpression to only delete under certain conditions
- Can add ReturnValues to control what is returned

```
item = client.delete_item(
   TableName='MyTable',
   Key={
      "Username": { "S": "myuser" }
   }
)
```

Options for ReturnValues:

- **NONE:** No attributes are returned (Default Setting)
- **ALL_OLD:** The entire item before the delete operation is returned

```
item = client.delete_item(
   TableName='MyTable',
   Key={
      "Username": { "S": "myuser" }
   },
   ReturnValues="ALL_OLD"
)
```

Batch Operations

The two batch operations - BatchReadItem and BatchWriteItem - are used to combine multiple operations in a single request. All reads or writes in a batch will succeed or fail independently.

The response from a batch operation will include an "UnprocessedKeys" property (for BatchGetItem) or an "UnprocessedItems" property (for BatchWriteItem) that includes any items that were not processed due to size limits, throttling errors, or other errors. Retry these requests to ensure all items are processed.

BatchGetItem:

- Can request up to 100 items and receive up to 16MB of data
- Can read items from multiple tables
- Unhandled items are returned in the "UnprocessedKeys" property

```
resp = client.batch_get_item(
  RequestItems={
    "MyFirstTable": {
      Keys: [
        { "Username": { "S": "user123" },
        { "Username": { "S": "user567" }
      ]
    }
    "MySecondTable": {
      Keys: [
        { "OrderId": { "S": "dbdf628e" },
        { "OrderId": { "S": "2b55f660" }
      ]
    }
  }
)
```

Batch Operations

BatchWriteItem:

- Can write up to 25 items and up to 16MB of data
- Can write items to multiple tables
- Only puts and deletes allowed - no updates to existing items
- No ConditionExpressions allowed
- Unhandled items are returned in the "UnprocessedItems" property

```
resp = dynamodb.batch_write_items(
  RequestItems={
    "MyFirstTable": [
      {
        "PutRequest": {
          "Item": {
            "User": { "S": "user123"
          }
        }
      },
      {
        "PutRequest": {
          "Item": {
            "User": { "S": "user123"
          }
        }
      }
    ]
  }
)
```

Transactions

The two transactional operations - TransactGetItems and TransactWriteItems - are used to combine multiple operations in a single request. All reads or writes in a transaction will succeed or fail together.

TransactGetItems:

- Can request up to 25 items and receive up to 4MB of data
- Can read items from multiple tables
- Will throw an error if any of the requested items are being updated in another operation

```
resp = client.transact_get_items(
  TransactItems=[
    Get: {
      Key: { "Username": { "S": "user123" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "Username": { "S": "user567" },
      TableName: "MyFirstTable"
    },
    Get: {
      Key: { "OrderId": { "S": "dbdf628e" },
      TableName: "MySecondTable"
    }
  ]
)
```

TransactWriteItems:

- Can write up to 25 items with up to 4MB of data
- Can write items from multiple tables
- Four operation types: Put, Update, Delete, or ConditionCheck
- ConditionCheck does not change an item but asserts a condition about an item

```
resp = client.transact_write_items(
  TransactItems=[
    {
      "Put": {
        "Item": {
          "Username": { "S": "user123" },
          "Name": { "S": "Bob G. User" }
        },
        "TableName": "MyFirstTable"
      }
    },
    {
      "Delete": {
        "Key": { "Username": { "S": "user567" },
        "TableName": "MyFirstTable"
      }
    ]
)
```

Transactions

Idempotence:

Ensure idempotency of a TransactWriteItem request by specifying the "CleintRequestToken."

If a TransactWriteItem request was made with the same ClientRequestToken and payload within the last 10 minutes, DynamoDB will ensure it was only processed once.

```
params = {
  "TransactItems": [ ... ],
  "ClientRequestToken": "MyIdempotencyToken"
}
first = client.transact_write_items(params)

// This will be an idempotent request
second = client.transact_write_items(params)
```

The DynamoDB Book

Python Cheatsheet

Query

Query is used to read multiple items in a single item collection. An item collection refers to all items with the same partition key.

ou must provide the partition key of your table or index when using a Query.

```
response = client.query(
   TableName='MoviesAndActors',
   KeyConditionExpression='#actor = :actor',
   ExpressionAttributeNames={
      '#actor': 'Actor'
   },
   ExpressionAttributeValues={
      ':actor': { 'S': 'Tom Hanks' }
   }
)
```

You may provide conditions on the sort key of your Query to filter the results you get back.

```
response = client.query(
   TableName='MoviesAndActors',
   KeyConditionExpression='#actor = :actor AND #movie BETWEEN :a
  AND :m' ",
  ExpressionAttributeNames={
     '#actor': 'Actor',
     '#movie': 'Movie'
   },
   ExpressionAttributeValues={
     ':actor': { 'S': 'Tom Hanks' },
     ':a': { 'S': 'A' },
     ':m': { 'S': 'M'
   }
)
```

You may use a FilterExpression to filter on attributes other than your sort key.

When using a FilterExpression, you will still pay for all items read that match your KeyConditionExpression.

```
response = client.query(
   TableName='MoviesAndActors',
   KeyConditionExpression='#actor = :actor',
   FilterExpression="#year > :year",
   ExpressionAttributeNames={
     '#actor': 'Actor',
     '#year': 'Year'
   },
   ExpressionAttributeValues={
     ':actor': { 'S': 'Tom Hanks' },
     ':year': { 'N': '2000' }
   }
)
```

The DynamoDB Book

Python Cheatsheet

Scan

The Scan operation reads all items in your table.

```
response = client.scan(
  TableName='MoviesAndActors'
)
```

The Scan operation is limited to 1MB of results per request, so you will need to paginate over your data to receive it all.

```
has_more = True
args = { "TableName": 'MoviesAndActors' }
items = []

while has_more
  response = client.scan(args)
  Items += response.Items
  if response['LastEvaluatedKey']:
    args['ExclusiveStartKey'] =
    response['LastEvaluatedKey']
  else:
    has_more = false
  }
```