AMAZON AWS SOLUTIONS ARCHITECT

DYNAMODB OVERVIEW
OVERVIEW

WHAT IS DYNAMO DB?

- Fully managed NoSQL database service that provides fast and predictable performance with seamless scalability
- Offloads the administrative burdens of operating and scaling a distributed database, so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling
- Offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data
- You can scale up or scale down your tables' throughput capacity without downtime or performance degradation
- Allows for full backups of your tables for long-term retention and archival for regulatory compliance needs
- Automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance
- Automatically replicated across multiple Availability Zones in an AWS region

CORE COMPONENTS

- Tables, Items, and Attributes
  - A *table* is a collection of data
  - An *item* is a group of attributes that is uniquely identifiable among all of the other items.
  - An *attribute* is a fundamental data element, something that does not need to be broken down any further.
- Primary Key
  - Unique identifier, or primary key, that distinguishes the item from all of the others in the table
- Secondary Indexes
  - Lets you query the data in the table using an alternate key, in addition to queries against the primary key
- DynamoDB Streams
  - An optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near real time, and in the order that the events occurred.

People


{
"PersonID": 101,
"LastName": "Smith",
"FirstName": "Fred",
"Phone": "555-4321"
}
{
"PersonID": 102,
"LastName": "Jones",
"FirstName": "Mary",
"Address": {
"Street": "123 Main",
"City": "Anytown",
"State": "OH",
"ZIPCode": 12345
}
}
{
"PersonID": 103,
"LastName": "Stephens",
"FirstName": "Howard",
"Address": {
"Street": "123 Main",
"City": "London",
"PostalCode": "ER3 5K8"
},
"FavoriteColor": "Blue"
}

Music

```
{
    "Artist": "No One You Know",
    "SongTitle": "My Dog Spot",
    "AlbumTitle": "Hey Now",
    "Price": 1.98,
    "Genre": "Country",
    "CriticRating": 8.4
}
```

```
{
    "Artist": "No One You Know",
    "SongTitle": "Somewhere Down The Road",
    "AlbumTitle": "Somewhat Famous",
    "Genre": "Country",
    "CriticRating": 8.4,
    "Year": 1984
}
```

```
{
    "Artist": "The Acme Band",
    "SongTitle": "Still in Love",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 2.47,
    "Genre": "Rock",
    "PromotionInfo": {
        "RadioStationsPlaying": [
            "KHCR",
            "KQBX",
            "WTNR",
            "WJJH"
        ],
        "TourDates": {
            "Seattle": "20150625",
            "Cleveland": "20150630"
        },
        "Rotation": "Heavy"
    }
}
```

```
{
    "Artist": "The Acme Band",
    "SongTitle": "Look Out, World",
    "AlbumTitle": "The Buck Starts Here",
    "Price": 0.99,
    "Genre": "Rock"
}
```

PRIMARY KEY

- When you create a table, in addition to the table name, you must specify the primary key of the table.
- The primary key uniquely identifies each item in the table, so that no two items can have the same key.
- Each primary key attribute must be a scalar (meaning that it can hold only a single value). The only data types allowed for primary key attributes are string, number, or binary.
- Two Types of Primary Keys
  - **Partition key** – A simple primary key, composed of one attribute known as the *partition key*.
  - **Partition key and sort key** – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.
- In a table that has a partition key and a sort key, it's possible for two items to have the same partition key value. However, those two items must have different sort key values.
- The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions, based on their partition key values.
- The sort key of an item is also known as its *range attribute*. The term *range attribute* derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

SECONDARY INDEXES

- A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key
- You can create one or more secondary indexes on a table. You can define up to 5 global secondary indexes and 5 local secondary indexes per table.
- DynamoDB doesn't require that you use indexes, but they give your applications more flexibility when querying your data
- Two Types of Indexes
  - **Global secondary index** – An index with a partition key and sort key that can be different from those on the table
  - **Local secondary index** – An index that has the same partition key as the table, but a different sort key

"GENREALBUMTITLE" EXAMPLE

- Every index belongs to a table, which is called the *base table* for the index. In the preceding example, *Music* is the base table for the *GenreAlbumTitle* index.
- DynamoDB maintains indexes automatically
- When you create an index, you specify which attributes will be copied, or *projected*, from the base table to the index

DYNAMODB STREAMS

- Is optional feature that captures data modification events in DynamoDB tables
- The data about these events appear in the stream in near real time, and in the order that the events occurred.
- Each event is represented by a *stream record*. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:
  - A new item is **added** to the table: The stream captures an image of the entire item, including all of its attributes.
  - An item is **updated**: The stream captures the "before" and "after" image of any attributes that were modified in the item.
  - An item is **deleted** from the table: The stream captures an image of the entire item before it was deleted.
- Each stream record also contains the name of the table, the event timestamp, and other metadata.
- Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

TABLE CRUD OPERATIONS

- **CreateTable**
  - Requirements
    - **Table name-** The name must conform to the DynamoDB naming rules, and must be unique for the current AWS account and region.
    - **Primary key-** The primary key can consist of one attribute (partition key) or two attributes (partition key and sort key). You need to provide the attribute names, data types, and the role of each attribute: HASH (for a partition key) and RANGE (for a sort key).
    - **Throughput settings-** You must specify the initial read and write throughput settings for the table. You can modify these settings later, or enable DynamoDB auto scaling to manage the settings for you.
  - The TableStatus element indicates the current state of the table (CREATING).
  - It might take a while to create the table, depending on the values you specify for ReadCapacityUnits and WriteCapacityUnits.
  - Larger values for these will require DynamoDB to allocate more resources for the table.

TABLE CRUD OPERATIONS

- **DescribeTable**
  - To view details about a table, use the DescribeTable operation.
  - The output from DescribeTable is in the same format as that from CreateTable, including:
    - the timestamp when the table was created
    - its key schema, its provisioned throughput settings
    - its estimated size
    - any secondary indexes that are present.
  - If you issue a DescribeTable request immediately after aCreateTable request, DynamoDB might return an error (ResourceNotFoundException), due to an eventually consistent query
  - For billing purposes, your DynamoDB storage costs include a per-item overhead of 100 bytes
    - This extra 100 bytes per item is not used in capacity unit calculations or by the DescribeTable operation.

TABLE CRUD OPERATIONS

- **UpdateTable**
  - Modify a table's provisioned throughput settings.
  - Manipulate global secondary indexes on the table
  - Enable or disable DynamoDB Streams on the table
- **DeleteTable**
  - You can remove an unused table with the DeleteTable operation
  - Deleting a table is an unrecoverable operation.
  - When you issue a DeleteTable request, the table's status changes from ACTIVE to DELETING. It might take a while to delete the table, depending on the resources it uses
  - When the DeleteTable operation concludes, the table no longer exists in DynamoDB
- **ListTables**
  - Returns the names of the DynamoDB tables for the current AWS account and region.
- **DescribeLimits**
  - Returns the current read and write capacity limits for the current AWS account and region

THROUGHPUT

- When you create a new table in DynamoDB, you must specify its *provisioned throughput capacity*
- This is the amount of read and write activity that the table will be able to support.
- DynamoDB uses this information to reserve sufficient system resources to meet your throughput requirements
- You can optionally allow DynamoDB auto scaling to manage your table's throughput capacity; however, you still must provide initial settings for read and write capacity when you create the table
- DynamoDB auto scaling uses these initial settings as a starting point, and then adjusts them dynamically in response to your application's requirements
- DynamoDB automatically distributes your data across partitions, which are stored on multiple servers in the AWS Cloud
- As your application data and access requirements change, you might need to adjust your table's throughput settings
- ***Capacity units***—the amount of data your application needs to read or write per second

CAPACITY UNITS

- A ***read capacity unit*** represents one strongly consistent read per second, or two eventually consistent reads per second, for an item **up to 4 KB in size**
  - For example, suppose that you create a table with 10 provisioned read capacity units. This will allow you to perform 10 strongly consistent reads per second, or 20 eventually consistent reads per second, for items up to 4 KB.
  - Reading an item larger than 4 KB consumes more read capacity units. For example a strongly consistent read of an item that is 8 KB (4 KB × 2) consumes 2 read capacity units.
  - An eventually consistent read on that same item consumes only 1 read capacity unit.
  - Item sizes for reads are rounded up to the next 4 KB multiple. For example, reading a 3,500-byte item will consume the same throughput as reading a 4 KB item.

CAPACITY UNITS

- A ***write capacity unit*** represents one write per second, for an item **up to 1 KB** in size
  - For example, suppose that you create a table with 10 write capacity units. This will allow you to perform 10 writes per second, for items up to 1 KB size per second.
  - Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item will consume the same throughput as writing a 1 KB item.
- If your application performs reads or writes at a higher rate than your table can support, DynamoDB will begin to ***throttle*** those requests
- In some cases, DynamoDB will use ***burst capacity*** to accommodate reads or writes in excess of your table's throughput settings.
  - With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled.
  - Burst capacity is available on a best-effort basis, and DynamoDB does not guarantee that this capacity is always available.

INITIAL SETTINGS

- Take the following into account when configuring DynamoDB
  - **Item sizes.** Some items are small enough that they can be read or written using a single capacity unit. Larger items will require multiple capacity units
  - **Expected read and write request rates.** In addition to item size, you should estimate the number of reads and writes you need to perform per second
  - **Read consistency requirements.** Read capacity units are based on strongly consistent read operations, which consume twice as many database resources as eventually consistent reads.
    - You should determine whether your application requires strongly consistent reads, or whether it can relax this requirement and perform eventually consistent reads instead. (Read operations in DynamoDB are eventually consistent, by default; you can request strongly consistent reads for these operations if necessary.)
- If you have enabled DynamoDB auto scaling for a table, then its throughput capacity will be dynamically adjusted in response to actual usage.
- You can modify your table's provisioned throughput settings using the AWS Management Console or the UpdateTable operation

DATA TYPE SIZES

- **Strings** are Unicode with UTF-8 binary encoding.
  - Size = *(length of attribute name) + (number of UTF-8-encoded bytes)*.
- **Numbers** are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed.
  - Size = *(length of attribute name) + (1 byte per two significant digits) + (1 byte)*.
- A **binary** value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size.
  - Size = *(length of attribute name) + (number of raw bytes)*.
- The size of a **null** attribute or a **Boolean** attribute is *(length of attribute name) + (1 byte)*.
- An attribute of type **List** or **Map** requires 3 bytes of overhead, regardless of its contents.
  - Size = *(length of attribute name) + sum (size of nested elements) + (3 bytes)* .
  - The size of an **empty** List or Map is *(length of attribute name) + (3 bytes)*.

RESERVED WORDS

- On some occasions, you might need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word.
- To work around this, you can replace Comment with an expression attribute name such as #c
- The # (pound sign) is required and indicates that this is a placeholder for an attribute name
- If an attribute name begins with a number or contains a space, a special character, or a reserved word, then you *must* use an expression attribute name to replace that attribute's name in the expression.

```
aws dynamodb get-item \
     --table-name ProductCatalog \
     --key '{"Id":{"N":"123"}}' \
     --projection-expression "#c" \
     --expression-attribute-names '{"#c":"Comment"}'
```

PROJECTION EXPRESSIONS

- A *projection expression* is a string that identifies the attributes you want
- To retrieve a single attribute, specify its name
- For multiple attributes, the names must be comma-separated
- You can use any attribute name in a projection expression, provided that the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. If an attribute name does not meet this requirement, you will need to define an expression attribute name as a placeholder.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key file://key.json \
--projection-expression "Description, RelatedItems[0], ProductReviews.FiveStar"
```

NESTED ATTRIBUTES

- Suppose that you wanted to access the nested attribute ProductReviews.OneStar

```
aws dynamodb get-item \
     --table-name ProductCatalog \
     --key '{"Id":{"N":"123"}}' \
     --projection-expression "ProductReviews.OneStar"
```

- But what if you decided to use an expression attribute name instead?

```
aws dynamodb get-item \
     --table-name ProductCatalog \
     --key '{"Id":{"N":"123"}}' \
     --projection-expression "#pr1star" \
     --expression-attribute-names '{"#pr1star":"ProductReviews.OneStar"}'
```

- DynamoDB would return an empty result, instead of the expected map of one-star reviews.
- The correct approach would be to define an expression attribute name for each element in the document path

```
aws dynamodb get-item \
     --table-name ProductCatalog \
     --key '{"Id":{"N":"123"}}' \
     --projection-expression "#pr.#1star" \
     --expression-attribute-names '{"#pr":"ProductReviews", "#1star":"OneStar"}'
```

EXPRESSION ATTRIBUTE VALUES

- If you need to compare an attribute with a value, define an expression attribute value as a placeholder
- *Expression attribute values* are substitutes for the actual values that you want to compare — values that you might not know until runtime
- An expression attribute value must begin with a :, and be followed by one or more alphanumeric characters

READ OPERATIONS

- **GetItem**
  - Reads a single item from a table
  - Capacity unit calculation is simply the item size and round it up to the next 4 KB boundary
  - If you specified a strongly consistent read, this is the number of capacity units required
  - For an eventually consistent read (the default), take this number and divide it by two.
  - Performs an eventually consistent read, by default. You can use the ConsistentRead parameter to request a strongly consistent read instead.
  - Returns all of the item's attributes. You can use a *projection expression* to return only some of the attributes.

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--consistent-read \
--projection-expression "Description, Price, RelatedItems" \
--return-consumed-capacity TOTAL
```

READ OPERATIONS

- **BatchGetItem**
  - Reads up to 100 items, from one or more tables
  - Processes each item in the batch as an individual GetItem request, so DynamoDB first rounds up the size of *each item* to the next 4 KB boundary, and then calculates the total size
  - The result is not necessarily the same as the total size of all the items.
  - DynamoDB performs the individual read or write operations in parallel. Your applications benefit from this parallelism without having to manage concurrency or threading
  - The batch operations are essentially wrappers around multiple read or write requests

```
aws dynamodb batch-get-item --request-items file://request-items.json
```

```
{ "Thread": {
      "Keys": [
            { "ForumName":{"S": "Amazon DynamoDB"}, "Subject":{"S": "DynamoDB Thread 1"} },
            { "ForumName":{"S": "Amazon S3"}, "Subject":{"S": "S3 Thread 1"} }
      ],
      "ProjectionExpression":"ForumName, Subject, LastPostedDateTime, Replies"
      }
}
```

READ OPERATIONS

- **Query**
  - Reads multiple items that have the same partition key value
  - All of the items returned are treated as a single read operation, where DynamoDB computes the total size of all items and then rounds up to the next 4 KB boundary
  - A filter expression is applied after a Query finishes, but before the results are returned. Therefore, a Query will consume the same amount of read capacity, regardless of whether a filter expression is present.
  - To specify the search criteria, you use a *key condition expression*—a string that determines the items to be read from the table or index.
  - You must specify the partition key name and value as an equality condition.
  - You can optionally provide a second condition for the sort key (if present)

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name" \
--expression-attribute-values '{":name":{"S":"Amazon DynamoDB"}}'
```

READ OPERATIONS

- If you perform a read operation on an item that does not exist:
  - DynamoDB will still consume provisioned read throughput:
  - A strongly consistent read request consumes one read capacity unit
  - An eventually consistent read request consumes 0.5 of a read capacity unit

WRITE OPERATIONS

- **PutItem**
  - Writes a single item to a table
  - If an item with the same primary key exists in the table, the operation replaces the item.
  - For calculating provisioned throughput consumption, the item size that matters is the larger of the two

```
aws dynamodb put-item --table-name Thread file://request-items.json
```

```
{
      "ForumName": {"S": "Amazon DynamoDB"},
      "Subject": {"S": "New discussion thread"},
      "Message": {"S": "First post in this thread"},
      "LastPostedBy": {"S": "fred@example.com"},
      "LastPostDateTime": {"S": "201603190422"}
}
```

WRITE OPERATIONS

- **UpdateItem**
  - Modifies a single item in the table
  - DynamoDB considers the size of the item as it appears before and after the update
  - The provisioned throughput consumed reflects the larger of these item sizes.
  - Even if you update just a subset of the item's attributes, UpdateItem will still consume the full amount of provisioned throughput (the larger of the "before" and "after" item sizes).

```
aws dynamodb update-item \
--table-name Thread \
--key file://key.json \
--update-expression "SET Answered = :zero,
Replies = :zero, LastPostedBy = :lastpostedby" \
--expression-attribute-values
     file://expression-attribute-values.json \
--return-values ALL_NEW
```

```
{
     ":zero": {"N":"0"},
     ":lastpostedby": {"S":"barney@example.com"}
}
```

```
{
     "ForumName": {"S": "Amazon DynamoDB"},
     "Subject": {"S": "New discussion thread"}
}
```

WRITE OPERATIONS

- **DeleteItem**
  - Removes a single item from a table.
  - The provisioned throughput consumption is based on the size of the deleted item
- **BatchWriteItem**
  - Writes up to 25 items to one or more tables.
  - DynamoDB processes each item in the batch as an individual *PutItem* or *DeleteItem* request (updates are not supported), so DynamoDB first rounds up the size of each item to the next 1 KB boundary, and then calculates the total size.
  - The result is not necessarily the same as the total size of all the items
- For PutItem, UpdateItem, and DeleteItem operations, DynamoDB rounds the item size up to the next 1 KB. For example, if you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.
- PutItem, UpdateItem, and DeleteItem allow *conditional writes*, where you specify an expression that must evaluate to true in order for the operation to succeed. If the expression evaluates to false, DynamoDB will still consume write capacity units from the table

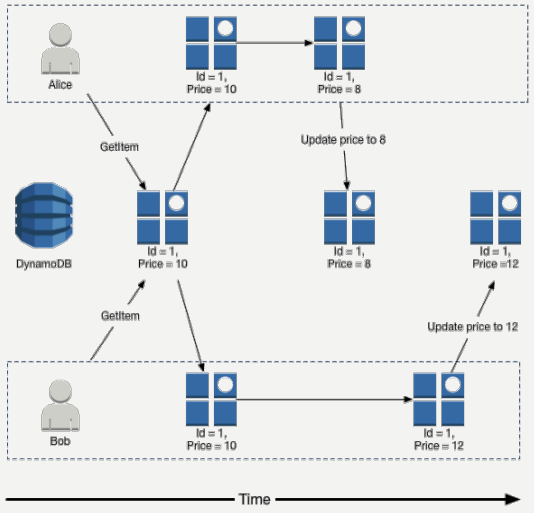
RETURN VALUES

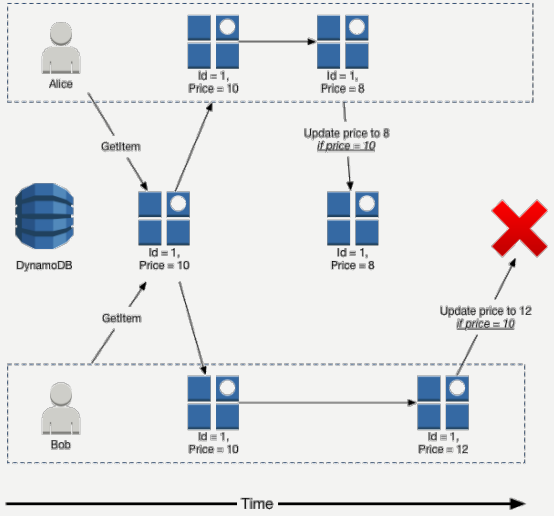
- In some cases, you might want DynamoDB to return certain attribute values as they appeared before or after you modified them
- The PutItem, UpdateItem, and DeleteItem operations have a ReturnValues parameter that you can use to return the attribute values before or after they are modified
- The default value for ReturnValues is NONE

Operation	Return Values	Description
PutItem	ALL_OLD	If you overwrite an existing item, ALL_OLD returns the entire item as it appeared before the overwrite. If you write a nonexistent item, ALL_OLD has no effect.
UpdateItem	ALL_OLD	If you update an existing item, ALL_OLD returns the entire item as it appeared before the update. If you update a nonexistent item (upsert), ALL_OLD has no effect.
UpdateItem	ALL_NEW	If you update an existing item, ALL_NEW returns the entire item as it appeared after the update. If you update a nonexistent item (upsert), ALL_NEW returns the entire item.
UpdateItem	UPDATED_OLD	If you update an existing item, UPDATED_OLD returns only the updated attributes, as they appeared before the update. If you update a nonexistent item (upsert), UPDATED_OLD has no effect.
UpdateItem	UPDATED_NEW	If you update an existing item, UPDATED_NEW returns only the affected attributes, as they appeared after the update. If you update a nonexistent item, UPDATED_NEW returns only the updated attributes, as they appear after the update.
DeleteItem	ALL_OLD	If you delete an existing item, ALL_OLD returns the entire item as it appeared before you deleted it. If you delete a nonexistent item, ALL_OLD does not return any data.

CONDITIONAL WRITES

- By default, the DynamoDB write operations (PutItem, UpdateItem, DeleteItem) are *unconditional*: each of these operations will overwrite an existing item that has the specified primary key.
- A conditional write will succeed only if the item attributes meet one or more expected conditions. Otherwise, it returns an error.
- Conditional writes are *idempotent*. This means that you can send the same conditional write request to DynamoDB multiple times, but the request will have no further effect on the item after the first time DynamoDB performs the update.





SCAN OPERATIONS

- A Scan operation reads every item in a table or a secondary index
- By default, a Scan operation returns all of the data attributes for every item in the table or index
- You can use the ProjectionExpression parameter so that Scan only returns some of the attributes, rather than all of them
- Scan always returns a result set
  - If no matching items are found, the result set will be empty.
- A single Scan request can retrieve a maximum of 1 MB of data applied before the filter expression is evaluated
- If you need to further refine the Scan results, you can optionally provide a filter expression.
  - A *filter expression* determines which items within the Scan results should be returned to you.
  - All of the other results are discarded
- With Scan, you can specify any attributes in a filter expression—including partition key and sort key attributes
- A Scan operation performs eventually consistent reads, by default. This means that the Scan results might not reflect changes due to recently completed PutItem or UpdateItem operations.

RESULT PAGINATION

- DynamoDB *paginates* the results from Scan operations
- With pagination, the Scan results are divided into "pages" of data that are 1 MB in size (or less)
- An application can process the first page of results, then the second page, and so on
- The LastEvaluatedKey from a Scan response should be used as the ExclusiveStartKey for the next Scan request.
  - If there is not a LastEvaluatedKey element in a Scan response, then you have retrieved the final page of results. (The absence of LastEvaluatedKey is the only way to know that you have reached the end of the result set.)

```
2017-07-07 12:19:14,389 - MainThread - botocore.parsers - DEBUG - Response body:
b'{"Count":7,"Items":[{"title":{"S":"Monster on the Campus"}},{"title":{"S":"+1"}}, {"title":{"S":"100
Degrees Below Zero"}},{"title":{"S":"About Time"}},{"title":{"S":"After Earth"}}, {"title":{"S":"Age of
Dinosaurs"}},{"title":{"S":"Cloudy with a Chance of Meatballs 2"}}],
"LastEvaluatedKey":{"year":{"N":"2013"},"title":{"S":"Curse of Chucky"}},"ScannedCount":100}'
```

SECONDARY INDEXES

- Many applications might benefit from having one or more secondary (or alternate) keys available, to allow efficient access to data with attributes other than the primary key
- A *secondary index* is a data structure that contains a subset of attributes from a table, along with an alternate key to support Query operations
- You can retrieve data from the index using a Query, in much the same way as you use Query with a table
- A table can have multiple secondary indexes, which gives your applications access to many different query patterns
- Every secondary index is associated with exactly one table, from which it obtains its data
  - This is called the *base table* for the index.
  - When you create an index, you define an alternate key for the index
- You also define the attributes that you want to be *projected*, or copied, from the base table into the index
- Every secondary index is automatically maintained by DynamoDB
  - When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes

SECONDARY INDEXES

- Global secondary index
  - An index with a partition key and a sort key that can be different from those on the base table
  - A global secondary index is considered "global" because queries on the index can span all of the data in the base table, across all partitions
- Local secondary index
  - An index that has the same partition key as the base table, but a different sort key
  - A local secondary index is "local" in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value
- For maximum query flexibility, you can create up to 5 global secondary indexes and up to 5 local secondary indexes per table
- To get a detailed listing of secondary indexes on a table, use the DescribeTable operation.
  - DescribeTable will return the name, storage size and item counts for every secondary index on the table
  - These values are not updated in real time, but they are refreshed approximately every six hours

SECONDARY INDEXES

Characteristic	Global	Local
Key Schema	The primary key of a global secondary index can be either simple (partition key) or composite (partition key and sort key).	The primary key of a local secondary index must be composite (partition key and sort key).
Key Attributes	The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary.	The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary.
Size Restrictions Per Partition Key Value	There are no size restrictions for global secondary indexes.	For each partition key value, the total size of all indexed items must be 10 GB or less.

SECONDARY INDEXES

Characteristic	Global	Local
Queries and Partitions	A global secondary index lets you query over the entire table, across all partitions.	A local secondary index lets you query over a single partition, as specified by the partition key value in the query.
Read Consistency	Queries on global secondary indexes support eventual consistency only.	When you query a local secondary index, you can choose either eventual consistency or strong consistency.
Provisioned Throughput Consumption	Every global secondary index has its own provisioned throughput settings for read and write activity. Queries or scans on a global secondary index consume capacity units from the index, not from the base table. The same holds true for global secondary index updates due to table writes.	Queries or scans on a local secondary index consume read capacity units from the base table. When you write to a table, its local secondary indexes are also updated; these updates consume write capacity units from the base table.

SECONDARY INDEXES

Characteristic	Global	Local
Projected Attributes	With global secondary index queries or scans, you can only request the attributes that are projected into the index. DynamoDB will not fetch any attributes from the table.	If you query or scan a local secondary index, you can request attributes that are not projected in to the index. DynamoDB will automatically fetch those attributes from the table.

SECONDARY INDEXES

- The following must be specified on creation:
  - The **type** of index to be created – either a **global** secondary index or a **local** secondary index.
  - A **name** for the index. The naming rules for indexes are the same as those for tables. The name must be unique for the base table it is associated with, but you can use the same name for indexes that are associated with different base tables.
  - The **key schema** for the index. Every attribute in the index key schema must be a top-level attribute of type String, Number, or Binary. Other data types, including documents and sets, are not allowed. Other requirements for the key schema depend on the type of index:
  - For a global secondary index, **the partition key** can be any scalar attribute of the base table. A sort key is optional, and it too can be any scalar attribute of the base table.
  - For a local secondary index, the **partition key must be the same** as the base table's partition key, and the sort key must be a non-key base table attribute.
  - **Additional attributes**, if any, to project from the base table into the index. These attributes are in addition to the table's key attributes, which are automatically projected into every index. You can project attributes of any data type, including scalars, documents, and sets.
  - The provisioned throughput settings for the index, if necessary:
    - For a **global** secondary index, you must specify read and write capacity unit settings. These provisioned throughput settings are independent of the base table's settings.
    - For a **local** secondary index, you do not need to specify read and write capacity unit settings. Any read and write operations on a local secondary index draw from the provisioned throughput settings of its base table.

BACKUP AND RESTORE

- **Backup**
  - When you create an on-demand backup, a time marker of the request is cataloged
  - The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot
  - Backup requests are processed instantaneously and become available for restore within minutes
  - Each time you create an on-demand backup, the entire table data is backed up.
    - There is no limit to the number of on-demand backups that can be taken
  - All backups in DynamoDB work without consuming any provisioned throughput on the table
  - Along with data, the following are also included on the backups:
    - Global secondary indexes (GSIs)
    - Local secondary indexes (LSIs)
    - Streams
    - Provisioned read and write capacity

BACKUP AND RESTORE

- **Restore**
  - A table is restored without consuming any provisioned throughput on the table
  - The destination table is set with the same provisioned read capacity units and write capacity units as the source table, as recorded at the time the backup was requested
  - The restore process also restores the local secondary indexes and the global secondary indexes.
  - You can only restore the entire table data to a new table from a backup
  - Restore times vary based on the size of the DynamoDB table that's being restored
  - You can write to the restored table only after it becomes active
  - You can't overwrite an existing table during a restore operation.

POINT-IN-TIME RECOVERY

- Amazon DynamoDB point-in-time recovery (PITR) provides automatic backups of your DynamoDB table data
- You can enable point-in-time recovery using the AWS Management Console, AWS Command Line Interface (AWS CLI), or the DynamoDB API
  - When it's enabled, point-in-time recovery provides continuous backups until you explicitly turn it off
- The point-in-time recovery process always restores to a new table
- The retention period is a fixed 35 days (five calendar weeks) and can't be modified
- Any number of users can execute up to four concurrent restores (any type of restore) in a given account
- If you disable point-in-time recovery and later re-enable it on a table, you reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the LatestRestorableDateTime.
- Along with data, the following are also included on the new restored table using point in time recovery:
  - Global secondary indexes (GSIs)
  - Local secondary indexes (LSIs)
  - Provisioned read and write capacity
  - Encryption settings

QUESTIONS?