# DynamoDB

Sunday, 8 October 2017 15:36

## DynamoDB Essentials

- Fully managed NoSQL database
  - Can scale up and down depending on demand without downtime or performance degradation
  - Manage data, not hardware or software
  - Built-in monitoring

- Consistent and fast performance
  - Data is stored on fast SSDs
  - You control performance through read/write capacity
  - Can spread out load across servers and tables
  - Replication across multiple availability zones in an AWS region (high availability)

## DynamoDB Features

- DynamoDB can be used via the AWS Console or API
  - Multi-language support through SDKs (JavaScript, PHP, Python, mobile, etc…)
  - Build-in features that speed up development
  - Command line interface

- Flexible data model with attributes and items

- Supports different levels of consistency
  - Eventually consistent
  - Strongly consistent

- Conditional updates and concurrency control
  - Atomic counter

## API Credentials with Access Keys

- Used to sign requests
  - The AWS SDKs use access keys
  - The Command Line Interfaces (CLIs)

- Can be disabled and deleted, but not retrieved.

- You can have temporary access keys which expire
- Useful when connecting from outside of AWS (like your computer or application)

**DynamoDB: Provisioned Throughput**

- Flexible read and write performance capacity
  - Set during table creation
  - Can be changed at anytime without downtime or performance degradation
- Automatically allocated machine resources
- Ability to reserve capacity

**DynamoDB: Important terms**

**Primary Key -** Unique identifier

- Partition Key (also known as hash attribute)
  - Simple primary key composed of one attribute
  - Used to retrieve data
  - Must be unique
- Partition and Sort (also known as range attribute) key
  - Composite primary key composed of two attributes: the partition key and sort key
  - Can have the same partition key, but must have a different sort key

**DynamoDB: Important terms**
**Secondary Indexes**

- Indexes let you query data using alternate keys (more flexibility)
- Can provide better performance
- Global Secondary Index
  - Partition key and Sort key can both be different from those on the table
  - Has its own provisioned throughput

- Has its own provisioned throughput
- Local Secondary Index
  - Partition key must be the same but the sort key is different
  - "Local" because every partition is scoped to a table partition with the same partition key value
  - Uses table's provisioned throughput

- Up to 5 global and local indexes per table

## DynamoDB: Important Limits

- 256 tables per region (can be increased on request)
- Partition key length: 2048 bytes maximum, 1 byte minimum
- Sort key length: 1024 bytes maximum, 1 byte minimum
- Item size: 400KB including attribute name and value

### API-specific limits

- Up to 10 **CreateTable**, **UpdateTable**, and **DeleteTable** actions running simultaneously
- A single **BatchGetItem** operation can get a maximum of 100 items
  - Must be < 16 MB in size
- A single **BatchWriteItem** operation can contain up to 25 **PutItem** or **DeleteItem** requests
  - Total size of all items must be < 16 MB
- **Query** and **Scan** result set is limited to 1 MB of data per call
  - **LastEvaluatedKey** in the response can be used to retrieve more

### DynamoDB Provisioned Capacity

- Unit of read capacity: 1 strongly consistent read per second or two eventually consistent reads per second for **items as large as 4 KB**
- Unit of write capacity: 1 write per second for **items up to 1KB**
- Key concepts needed:
  - Calculating required throughput
  - Understanding how secondary indexes affect throughput
  - Understanding what happens if your application's read/writes exceed

- Understanding what happens if your application's read/writes exceed throughput

## Calculating Read Capacity

- Round up to the nearest 4 KB multiplier
- Items that are 3 KB in size can still only do 1 strongly consistent or 2 eventually consistent reads per second
- Example:
  - Item size 3 KB
  - Want to read 80 items per second from the table
  - How many read capacity units are required?

## Calculating Read Capacity - Example

- Example: Your items are 3KB in size and you want to read 80 (strongly consistent read) items from a table per second
  - Item size 3KB
  - Want to read 80 items per second from the table
- Formula: (**ITEM SIZE** (rounded up to the next 4KB multiplier) / **4KB**) * # of items
- 80 * (3KB (round up to 4) / 4KB)
  - 80 * 1 = 80 required provisioned read throughput
- Bonus: Eventually consistent reads would cut that in half so:
  - (80 * 1) / 2 = 40 required read capacity

## Calculating Read Capacity - Example #2

- Example: Your items are 10KB in size and you want to read 80 (strongly consistent read) items from a table per second
  - Item size 10KB
  - Want to read 80 items per second from the table
- Formula: (**ITEM SIZE** (rounded up to the next 4KB multiplier) / **4KB**) * # of items
- 80 * (10KB (round up to 12) / 4KB)
  - 80 * 3 = 240 required provisioned read throughput
- Bonus: Eventually consistent reads would cut that in half so:

- (80 * 3) / 2 = 120 required read capacity

## Calculating Write Capacity

- Round up to the nearest 1 KB multiplier
- Example:
  - Item size 1.5 KB
  - Want to write 10 items per second from the table
  - How many write capacity units are required?

## Calculating Write Capacity - Example

- Example: Your items are 1.5KB in size and you want to write 10 items per second
- Formula: (**ITEM SIZE** (rounded up to the next 1KB multiplier) / 1**KB**) * # of items
- 10 * (1.5KB (round up to 2) / 1KB)
  - 10 * 2 = 20 required provisioned write throughput

## Read Throughput with Local Secondary Indexes

- Uses the same read/write capacity from parent table
- If you read only index keys and projected attributes, the calculations are the same
  - You calculate using the size of the index entry, not the table item size
  - Rounded up to the nearest 4KB
- If queried attributes aren't projected attributes or keys, we get extra latency and read capacity cost
  - You use read capacity from the index AND for every item from the table. Not just the attribute needed

## Write Throughput with Local Secondary Indexes

- Adding, updating, or deleting an item in a table also costs write capacity units to perform the action on the local index

- The cost of writing an item to a local secondary index depends on a few things:
  - If you write a new item to the table and that item defines an indexed attribute, or if you update an existing item and write an indexed attribute that was previously undefined, that will cost you **one write operation to put the item** in the index.
  - If you change the value of an indexed key attribute, **two writes are required**. One to delete the previous item from the index, and another to put the new item into the index.
  - If an update deletes an item that was in the index, **one write is required to delete the item** from the index.

## Read Throughput with Global Secondary Indexes

- Global indexes have their own throughput capacity, completely separate from that of the table's capacity.
- Global indexes support eventually consistent reads, which means that a single global secondary index query can get up to 8 KB per read capacity unit (because we take 4KB and multiply it by 2)
- Reads in global indexes are calculated the same as in tables, except that the size of the index entries is used instead of the size of the entire item.

## Write Throughput with Global Secondary Indexes

- Putting, Updating, or Deleting items in a table consumes the **index write capacity units**
- The cost of writing an item to a global index depends on a few things, and those are identical to the local secondary index rules

## Exceeding Throughput

- Requests exceeding the allocated throughput may be throttled
- With global secondary indexes, all indexes must have enough write capacity or the write might get throttled (even if the write doesn't affect the index!)
- You can monitor throughput in the AWS Console

## DynamoDB Queries

- Allow you to find items using only primary key values from a table or secondary index

- Benefits:
  - Returns the items matching the primary key search

- Much more efficient because it searches indexes only
- Returns all attributes of an item, or only the ones you want
- Is eventually consistent by default but can request a consistent read
- Can use conditional operators and filters to return precise results

## DynamoDB Scans

- Reads every item in the table and is operationally inefficient
- Looks for all items and attributes in a table by default

- Benefits:
  - Scans can apply filters to the results to refine values
  - Can return only specific attributes with the **ProjectionExpression** parameter

- Negatives:
  - The larger the data set in the table the slower performance of a scan
  - The more filters on the scan the slower the performance
  - Returns only filtered results
  - Only eventually consistent reads are available

## DynamoDB Scans - If you must use them

- What you should keep in mind:
  - You can reduce **Page Size** of an operation with the **Limit** parameter, to limit how much data you try to retrieve at the same time
  - Avoid performing Scans on mission-critical tables
  - Program your application logic to retry any requests that receives a response code saying that you exceeded provisioned throughput (or increase throughput)

## DynamoDB - Atomic Counters

- Allow you to increment or decrement the value of an attribute without interfering with other write requests
- Requests are applied in the order that they are received
- Updates are not idempotent - it will update the value each time it is called
- Use case: Increasing view count


## DynamoDB - Conditional Writes

- Helps coordinate writes
- Checks for a condition before proceeding with the operation
- Supported for **PutItem**, **DeleteItem**, **UpdateItem** operations

- Specify conditions in **ConditionExpression**:
  - Can contain attribute names, conditional operators, and built-in functions
- A failed conditional write returns **ConditionalCheckFailedException**