# DynamoDB

low latency, NoSQL DB service. Fully managed supports document and key-value data models.

- Good for apps that need consistent, single-digit ms latency at any scale.
- Good fit for mobile, web, gaming, ad-tech, IoT, etc.
- Stored on SSD - fast
- Avoids single points of failure - spread across 3 geo distinct AZs
- Serverless

## Consistency Models

- **Eventually Consistent** reads - consistent across all copies of data usually reached within a **second**. Repeating a read after a short time should return the updated data. (***Best Read Performance***)
- **Strongly Consistent** reads - any writes that occurred before the read will be reflected in the read. returns a result that reflects all writes that received a successful response prior to the read.

DynamoDB is made up of

1. Tables
2. Items (like a row)
3. Attributes (column)
4. Supports key-value and document data models.
5. Document can be JSON HTML or XML

DynamoDB stores and retrieves data based on Primary Key.

- The primary key can just consist of a simple partition key as a single attribute or a max of two attributes as a composite key consisting of partition key and sort key. 2 Types of Primary Keys.
    - **Partition Key** - unique attribute (e.g. user id, product ID, email address)
        - value of the partition key is input to an internal hash function. output of hash function determines the partition or physical location on which the data is stored.
        - *no two items can have same partition key is used as primary key*
    - **Composite Key** - (Partition Key + Sort Key) in combination
        - all items with the same Partition Key are sorted together then sorted according to the Sort Key value
        - allows you to store multiple items with the same Partition Key e.g. Same user posting multiple times to a forum.
            - Primary Key could use a Composite Key
                - Paritition Key - user ID
                - Sort key - timestamp of the post, usually a range like date.
            - 2 items may have the same Partition Key, but they must have a different Sort Key

    - All items with the same Partition Key are sorted together, then sorted according to the Sort Key value
    - Allows you to store multiple items with the same Parition Key

## Access Control

- Managed using IAM
- create IAM user with specific permissions to access/create DynamoDB
- Create an IAM role to enable you to obtain temporary access keys which can be used to access DynamoDB
- **IAM Condition** to retruct user access to only their own records. Uses the `dynamodb:LeadingKeys` parameter.

## Indexes

**Index** - a data structure which allows you perform fast queries on specific columns in a table. You select the columns that you want included in the index and run your searches on the index rather than the entire dataset.

- fast queries on a specific columns

  Two types supported:

- Local Secondary Index
- Global Secondary Index

### Local Secondary Index

- can only be created when creating a table - not later
- has same Partition Key but different Sort Key as original table
    - because you have a different alternative view of your data
- Queries based on Sort Key are much faster than the main table.

### Global Secondary Index

- *flexible*, create whenever you want
- different Partition Key and Different Sort Key
- Speeds up queries relating to alternative Part and Sort key

Index enable fast queries on specific data columns Give different view of your data, based on alternative Partition/Sort Keys

## Scan vs Query API

Query more efficient than Scan

Resume the impact of a query or scan by setting a smaller page size which uses fewer read operations.

### Query

Query operation finds items in a table based on the PK attribute and a distinct value to search for.

e.g. Select an item where the user ID is equal to 212, will select all the attributes for that item, e.g., first name, surname, email etc.

- Use optional Sort Key name and value to refind the results
- e.g. if your Sort Key is a temp stamp you can refund the query to only select items with a timestamp of last 7 days

### ProjectionExpression:

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.

By default a Query returns all attributes for items but you can use the `ProjectionExpression` parameter if you want the query to only return the specific attributes you want. e.g. only see email address rather than all attributes

*Results are always sorted by the Sort Key*

- Numeric ascending order by default
- ASCII character code values
- Can reverse the order by setting `ScanIndexForward` parameter to false. Relating to queries only, not scans.
- All queries are eventually consistent.

### Scan

- *Examines every item in the table.*
    - Scan dumps the entire table then filters out the values to provide the dresired result - removing unwanted items. Adds an extra step of removing the data you don't want.
    - As table grows, the scan operation takes longer.
    - Scan on a large table can use up the provisioned throughput for a large table in just a single operation.
- *By default returns all data attributes*
- Use `ProjectExpression` parameter to refine the scan to only return the attributes you want.

## Performance

Efficiency: Query > Scan

- Reduce impact of query or scan by setting a smaller page size which uses fewer read ops. e.g., set page size == 40 item return.

- Larger number of smaller ops will allow other requests to succeed without throttling.
- Avoid scan when possible. Design tables so you can use Query, Get, or BatchGetItem

Improve Scan Performance - Parallel Scans

- by default a scan processes data sequentially in returning 1 MB increments before moving on to retrieve the next 1 MB of data. It can only scan one partition at a time.
    - Configure Dynamo to use Parallel scans instead by logically dividing a table or index into segments and scanning each segment in parallel.
    - Best to avoid parallel scans if your table or index is already incurring heavy read/write activity from other applications.

## Provisioned Throughput

*measured in Capacity Units*

- on create, specify Read Capacity Units and Write Capacity Units
- **1x Write Capacity Unit = 1 x 1KB Write per second**
- **1x Read Capacity Unit = 1x Strongly Consistent Read of 4 KBps OR 2x Eventually Consistent Reads of 4 KBps (default)**

Example:

- table with 5x RCU and 5x WCU
- can perform 5 x 4KB Strongly Consistent reads = 20 KB per second
- Twice as many Eventually Consistent = 40 KB
- 5 x 1 KB Writes = 5 KB per second

```
#### Strongly Consistent Reads Calculation

e.g., you app needs to read 80 items (table rows) per second. Each item =
3KB in size. You need Strongly Consistent Reads.

1x RCU = 1x Strongly Consistent Read of 4 KBps

1. calculate how many RCUs needed for each read:
    size of each item / 4KB = 3 KB / 4 KB = 0.75 -> (round to nearest
whole) = 1 RCU per read operation.

#### Eventually Consistent Reads Calculation

1x RCU = 2x Eventually Consistent Reads of 4 KBps

2. you get twice the reads so divide the RCUs you needed for Strongly by 2
to get Eventually read number: 40 RCUs

#### Write Capacity Units Calculation
```

```
Want to write 100 items per second. Each item == 512 bytes in size.

1x WCU = 1x 1 KB Write per second.

1. calculate how many CUs for each write.
    Size of each item / 1 KB (for WCUs) = 512 bytes / 1 KB == 0.5 ->
rounded up to nearest whole 1x Write Capacity Unit per write operation.
2. Multipllied by number of writes per second = 100 x 1 = 100 Write
Capcity Units required.
```

## On Demand Capacity (new in 2018)

Charges apply for Reading Writing and Storing data

- auto scales up and down capacity units based on activity of your application.
- good for unpredictable workloads and serverless stacks
- pay for what you use (pay per request)

## DynamoDB Accelerator (DAX)

fully managed, clustered, in-memory cache for DynamoDB

- delivers up to 10x read performance improvement
- ms performance for millions of request per second

Write-through caching service (updates written to cache and backend store at the same time)

Allows you point your DynamoDB API calls at the DAX cluster.

on Cache miss - DAX performs an Event. Consistent GetItem operation against DynamoDB

Retrieval of data from DAX reduces the read load on tables

reduce Provisioned Read Capcity.

GOOD for retail during Black Friday, mobile games, ready-heavy bursty workloads. auction applications.

NOT good for :

- apps that require strongly consistent reads - only does eventually consistent reads
- write intensive applications
- application that do not perform many read operations
- applications that do not required ms response times.

## Elasticache

- in-memory caching - hitting a cache is faster than disk (db) reads
- `app <- elasticache -> database` - e.g., an app frequently requesting specific product information for best selling products.
- takes load off the databases. good if your database is particularly read-heavy and the data is not changing frequently.

Benefits:

- improves performance for read-heavy workloads (social networking, gaming, media sharing, Q&A portals)
- frequently access data stored in memory for low-latency access, improving the overall performance of your app
- also good for compute-heavy recommendation engines.
    - can be used to store results of I/O intensive database queries or output of compute-intensive calculations.

Types:

**Memcached**

- widely aadopted, mult-threaded, no Multi-AZ capability

**REDIS**

- open source key-value, supports more complex data structures, supports master/slave replication and Multi-AZ support

### Caching Strategies

**Lazy Loading** - only loads data into cache when necessary. if requested data is in cache, E returns data to app, if data not in cache or expired E returns a null.

PROS:

- only requested data cached. avoids filling up cache with useless data
- node failures are not fatal. a new empty node will just have a lot of cache misses initially

CONS:

- cache miss penalty on: initial request , query to database, writing of data to the cache
- stale data - if data only updated on cache miss, it can become stale. doesn't automatically update if the data in the database changes.

**Lazy Loading with TTL**

Time To Live:

- specifies number of seconds until the key (data) expires to avoid keeping stale data in the cache
- Lazy Loading treats an expired key as a cache miss, causes the app to retrieve the data from DB, subsequently write the data into the cache with a new TTL.
- doesn't eliminate stale data - helps avoid it.

**Write-Through** - adds or updates data to the cache whenever data is written to the DB

PROS:

- data in the cache is never stales
- Users are ghenerally more tolerant of additional latency when dating data than when retrieving it.

CONS:

- end up wasting resources (write penalty) since every write involves 1 write to the cache + 1 write to the database.
- if a node fails, a new one is spun up, data is missing in it. Have to wait until added or updated in the DB (mitigate by implementing Lazy Loading in conjunction with write-through)
- Wasted resources if most of the data is never read.

DAX vs Elasticache for DynamoDB:

- DAX was developed and optimized specifically for DynamoDB. Only lets you use the write-through strategy.
- Elasticache better for RDS

## Transactions (new in 2018. not on the exam)

ACID Transactions (Atomic, Consistent, Isolated, Durable)

- read or write multiple items across multiple tables as an all or nothing operation
- check for a pre-req condition before writing to a table.

## TTL

- an attribute that defines an expiry time for your data.
- good for removing irrelevant or old data no longer useful after time:
    - session data
    - event logs
    - temp data

- reduces cost for storage.

TTL is expressed as epoch time. Expiration of session data could be set to 2 hours after session begins.

```
#1) Check your IAM user permissions:

aws iam get-user

#2) Create SessionData table:

aws dynamodb create-table --table-name SessionData --attribute-definitions
\
AttributeName=UserID,AttributeType=N --key-schema \
AttributeName=UserID,KeyType=HASH \
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5

#3) Populate SessionData Table:

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