**day33\_107856406\_dsdipt\_sudipto\_4september2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

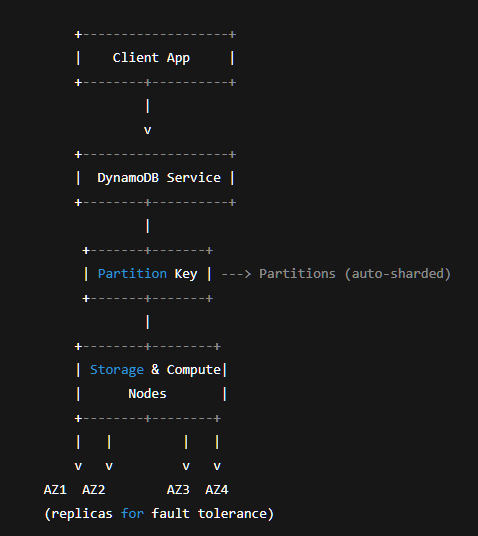
**Email:** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 4 September 2025 (Day 33)

### ***Task 1: Components of DynamoDB Architecture***

1. **Tables**
   * It’s like a collection of items (like rows in SQL), but flexible.
   * Every table has a **primary key** (partition key, optionally sort key).
2. **Items**
   * These are the records in the table.
   * Items are made of **attributes** (like columns in SQL).
3. **Partitions**
   * DynamoDB automatically **splits tables into multiple partitions** for scalability.
   * Partitioning is based on the **partition key**.
   * Each partition is stored across multiple **nodes** for redundancy. (the unnecessary duplication of the same data in multiple places within a database or storage system)
4. **Nodes**
   * These are servers that physically store data.
   * **Storage nodes** handle data storage.
   * **Compute nodes** handle read/write requests (like queries).
5. **Replication & Fault Tolerance**
   * DynamoDB replicates data **across multiple Availability Zones (AZs)**.
   * This ensures durability and high availability.



### ***Task 2: What are multivalue sorts or filters?***

**Multivalue Sorts/Filters** basically mean you’re sorting or filtering data based on multiple values or conditions at once, not just one.

**Example Scenario:**

Imagine we have a **Books table** with columns:

* BookID
* Title
* Author
* YearPublished

**Multivalue Filter Example**

Let’s say we want to get books by either **"George Orwell"** or **"Ray Bradbury"**. That’s a multivalue filter.

In DynamoDB CLI, we can do something like:

aws dynamodb scan \

--table-name Books \

--filter-expression "Author IN (:author1, :author2)" \

--expression-attribute-values '{":author1": {"S": "George Orwell"}, ":author2": {"S": "Ray Bradbury"}}'

This filters the records where the author is either Orwell or Bradbury – not just one specific value.

**Multivalue Sort (Concept)**

DynamoDB doesn’t support complex sorting directly (it sorts by sort key in ascending order), but in apps, you can simulate multivalue sorts by:

* Querying based on multiple sort key values
* Then ordering them yourself in the application code

Example:  
 Get books published in **1925, 1932, or 1949**, and sort by YearPublished.

Multivalue Sorts/Filters let you apply multiple values in a single filter or sort action instead of just one.  
Useful when searching for multiple options at once instead of making separate queries.

### ***Task 3: How do you lower the cost of DynomaDB?***

**Use On-Demand vs Provisioned Capacity Smartly**

**On-Demand** • Good for unpredictable workloads  
 • You pay per read/write request (expensive if lots of ops)

**Provisioned Capacity** (Preferred for steady workloads)  
 • You specify how many reads/writes per second you expect.  
 • Way cheaper if usage is predictable  
 • Example:

--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5

**Avoid Hot Partitions**

• Hot partition = too many requests hitting the same partition key → throttling  
 • Fix it by choosing a good partition key design → evenly distributed keys (e.g., use UUIDs instead of timestamps)

Avoid sequential IDs for partition key → makes access patterns predictable (bad for scaling)

**Use Smaller Item Sizes**

• DynamoDB charges by storage & read/write size  
 • Keep your records lean:

* Don’t store huge JSON blobs unless necessary
* Compress large attributes (if possible)

**Use Efficient Queries (Don’t Over-Scan)**

• Avoid full table scans  
 • Use **Query** instead of **Scan** whenever possible → Query is cheaper because it targets a partition key

Example:

aws dynamodb query --table-name Books --key-condition-expression "BookID = :id" --expression-attribute-values '{":id": {"N": "1"}}'

**Time-to-Live (TTL)**

Set a TTL on items that don’t need to live forever (like session logs)

aws dynamodb update-time-to-live \

--table-name YourTable \

--time-to-live-specification "Enabled=true, AttributeName=expiryTime"

Saves storage costs by auto deleting old data

**Use Global Secondary Indexes (GSI) Wisely**

• GSIs help query flexibility, but each GSI adds cost  
 • Only create GSIs you actually need  
 • Think twice before over-indexing

**Data Modeling Smarts**

• Design your table to reduce the need for expensive operations  
• Denormalize when needed → Store redundant data smartly instead of doing joins (DynamoDB isn’t relational)

**Summary :**

1. Prefer Provisioned + Auto Scaling
2. Design good partition keys (avoid hot partitions)
3. Keep item size small
4. Query > Scan for cost efficiency
5. Use TTL for auto cleanup
6. Avoid excessive GSIs
7. Smart data modeling → Pre-compute when possible