Part 1: Foundational Concepts

1. What is NoSQL?

Definition:

NoSQL stands for **"Not Only SQL"** — it refers to a class of databases that **don't use traditional relational schemas**(tables, rows, and joins).
They are designed for **high scalability**, **flexibility**, and **speed**, especially for modern cloud-scale applications.

---

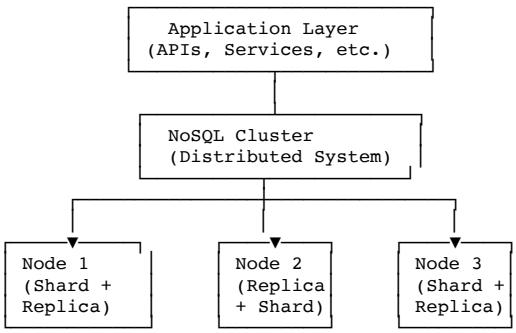
2. Relational Databases vs. NoSQL Databases

Feature	Relational Databases (RDBMS)	NoSQL Databases
Data Model	Tables with rows and columns	Key-Value, Document, Column, or Graph
Schema	Fixed schema (must define table structure before inserting data)	Schema-less (flexible attributes per item)
Scaling	Vertical (scale-up by adding CPU/RAM)	Horizontal (scale-out by adding more nodes)
Query Language	SQL (Structured Query Language)	Varies (e.g., JSON queries, key lookups, APIs)
Transactions	Strong ACID guarantees	Eventual consistency (some support ACID in limited form)
Examples	MySQL, PostgreSQL, Oracle	DynamoDB, MongoDB, Cassandra, Redis

---

NoSQL Architecture Overview

Typical **NoSQL architecture** is **distributed** — the database runs on multiple nodes (servers) in a cluster.
Each node stores part of the data (sharding) and replicates some data from others (replication).



Each node can serve read/write requests — depending on the replication strategy.

---

2. How NoSQL Manages Availability

a. Replication

Data is copied across multiple nodes.

- **Primary-Replica (Master-Slave)** – One primary node handles writes; replicas handle reads.
- **Multi-Master (Peer-to-Peer)** – Any node can accept reads and writes; conflicts are resolved later.

Example
In DynamoDB or Cassandra:

- Each item is replicated to **N nodes**.
- If one node fails, another replica can immediately serve the data.

---

b. Sharding (Data Partitioning)

Data is divided into smaller parts (shards) and distributed across multiple nodes.

- Each shard handles a subset of the data.
- A coordinator routes requests to the correct shard.

Example
A user table might be sharded by `user_id`:

- IDs 1–1000 → Node 1
- IDs 1001–2000 → Node 2

- IDs 2001–3000 → Node 3

If Node 2 goes down, replicas on Node 3 can take over.

---

c. Consistency and Availability Trade-Off (CAP Theorem)

NoSQL systems are often **AP (Available and Partition Tolerant)** — they prioritize staying online even if some nodes are unreachable.

- **Strong Consistency**: Wait for all replicas to agree before confirming writes.
- **Eventual Consistency**: Return success quickly and synchronize later (higher availability).

Example:
Cassandra, DynamoDB, and Riak use *eventual consistency* to maintain high uptime.

---

d. Automatic Failover and Self-Healing

- Nodes regularly send **heartbeat** signals.
- If a node fails, its partitions are automatically **reassigned** to healthy nodes.
- When the node comes back, **replication restores** its data.

---

e. Multi-Availability zone (Datacenter) or Multi-Region Replication

To ensure disaster recovery and regional availability:

- Data is replicated across **multiple regions**.
- Writes in one region are asynchronously replicated to others.

Example:
DynamoDB Global Tables replicate data automatically across regions — all are active and writable.

Example:

Relational:

```
CREATE TABLE Users (
  user_id INT PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(100)
);
```

NoSQL (JSON Document style):

```
{
  "user_id": 101,
  "name": "Vilas",
  "email": "vilas@example.com",
  "skills": ["AWS", "Kubernetes"]
}
```

NoSQL databases are ideal when:

- You need **massive scale** (millions of reads/writes per second)
- Data structure changes often
- You need **high availability and low latency**

Proved Usecase

1. Easily handle 1 million requests per second

2. 79.8 million requests/second - at constant speed (i.e. 10T requests per day)

3. NoSQL (handle Peta bytes of data)

4. Supports ACID transactions (c - serializable)

5. Eventual consistency

6. Durability

7. HA

8. Rate limiting

9. On-demand mode

9. Global replication

10. Point in time recovery

11. Backup to S3

Why Do We Need Secondary Indexes?

DynamoDB tables are designed for **fast lookups by the primary key** (partition key + optional sort key).

But what if you want to **query by a different attribute** (like "email" or "status")?
That's where **secondary indexes** come in — they allow **alternate query patterns** without scanning the entire table.

Two Types of Indexes

Type	Local Secondary Index (LSI)	Global Secondary Index (GSI)
Scope	Local to a partition key	Global across all partitions
Partition Key	Same as base table	Can be different
Sort Key	Different from base table	Can be different
Creation Time	Only at table creation	Can be added anytime
Consistency	Supports **strongly consistent** reads	Only **eventually consistent** reads
Item Size Limit	Shares 10GB limit with base partition	Independent size and throughput
Throughput	Uses base table's RCUs/WCUs	Has **its own RCUs/WCUs**
Use Case	When you need another way to sort within same partition	When you need to query using an entirely different attribute

What Are *Throttled Events* in DynamoDB?

DynamoDB is designed to deliver predictable performance based on the **provisioned throughput** (Read Capacity Units — RCUs and Write Capacity Units — WCUs) or **on-demand scaling limits**.

When your application sends more requests than the table (or a partition) can handle, DynamoDB starts **throttling** — i.e., **delaying or rejecting** requests to protect the underlying infrastructure.

---

3. What is DynamoDB?

Definition:

Amazon DynamoDB is a **fully managed NoSQL database service** provided by AWS.
It supports both **key-value** and **document** data models and is built for:

- High-performance
- Serverless operation
- Automatic scaling

Think of DynamoDB as:

> "A globally distributed, low-latency, zero-maintenance database."

---

4. DynamoDB Data Model

Concept	Description	Example
Table	Collection of items (like an RDBMS table)	`Users`
Item	A single record (like a row)	`{ "UserID": "U101", "Name": "Vilas" }`
Attribute	A field (like a column)	`"Name": "Vilas"`
Primary Key	Uniquely identifies each item	`UserID` or (`UserID`, `Timestamp`)
Partition Key	Determines the physical location of data	`UserID`
Sort Key	Optional, used for ordering/grouping data	`Timestamp`

Example: Key-Value and Document Model

Key-Value Example

```
{
```

```
  "UserID": "U101",
  "Name": "Vilas",
  "Email": "vilas@example.com"
}
```

Document Example

```
{
  "UserID": "U102",
  "Profile": {
    "Name": "Alex",
    "Skills": ["Python", "AWS", "Docker"]
  },
  "LoginCount": 10
}
```

Each item can have a **different structure** — perfect for dynamic applications.

---

5. Core Features of DynamoDB

Let's break down the most powerful features that make DynamoDB ideal for cloud-scale applications:

---

1. Fully Managed

- No need to manage servers, software patches, or storage.
- AWS automatically handles **hardware provisioning**, **replication**, and **backup**.
- Ideal for developers — focus on app logic, not infrastructure.

---

2. Serverless

- You don't manage EC2 instances or scaling clusters.
- You pay only for **read/write requests and storage**.
- **Auto scaling** adjusts capacity based on traffic.

Serverless = Zero administration + pay-as-you-go.

---

3. Fast and Predictable Performance

- Consistent **single-digit millisecond latency**.
- Uses **SSD storage** and **in-memory caching (DAX)** for ultra-fast reads.
- Supports both:
  - **Eventually consistent reads** (faster)
  - **Strongly consistent reads** (accurate but slower)

---

4. Highly Scalable

- Automatically scales horizontally to handle **millions of requests per second**.
- You can use **on-demand capacity** (auto scaling) or **provisioned capacity** (fixed read/write units).

---

5. Flexible Data Access

- **Primary Key Access** for fast lookups.
- **Secondary Indexes** (GSI, LSI) for additional query patterns.
- Supports rich queries using **Query** and **Scan** operations.

---

6. Secure and Reliable

- Data encrypted at rest and in transit.
- Integrates with **AWS IAM** for fine-grained access control.
- Automatically replicates data across multiple Availability Zones (AZs).

---

7. Integration with AWS Ecosystem

- Works with:
  - **Lambda** → serverless apps
  - **API Gateway** → REST APIs

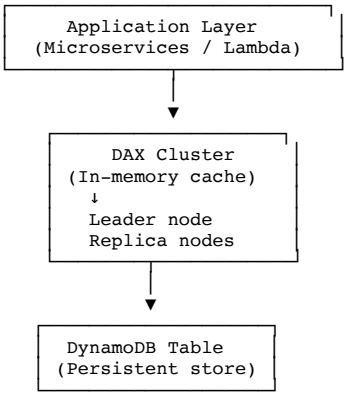
- **Kinesis** → real-time streaming
- **S3 / Glue** → analytics
- **CloudWatch** → monitoring and alerts

What is DAX (For in memory db)?

Amazon DAX (DynamoDB Accelerator) is a **fully managed, in-memory caching layer** for DynamoDB.
It's designed to **dramatically speed up read performance** — typically reducing response times from **milliseconds to microseconds**, even under heavy load.

Think of it as **"Redis for DynamoDB"**, but integrated, serverless, and fully managed by AWS.

DAX Architecture Overview



How it works:

1. Your application queries the **DAX endpoint** instead of DynamoDB directly.
2. DAX checks if the requested item exists in its **cache** (in-memory).
3. - If found → returns data instantly (microseconds latency).
   - If not found → fetches from DynamoDB, stores it in cache, then returns it.

Components of a DAX Cluster

Component	Description
DAX Cluster	A group of nodes managed as a unit. Usually 3+ nodes for HA.
Primary Node	Handles write-through operations to DynamoDB.
Replica Nodes	Serve read requests and replicate cache from the primary.
Endpoint	The cluster provides a single endpoint that your app connects to.

DAX vs. DynamoDB Table — Key Differences

Feature	DAX (DynamoDB Accelerator)	DynamoDB Table
Purpose	In-memory cache for faster reads	Persistent NoSQL database
Data Storage	In-memory (volatile)	SSD storage (durable)
Persistence	No – data is lost if cluster restarts	Yes – data is permanently stored
Latency	Microseconds	Milliseconds
Scalability	Horizontally scalable cluster	Horizontally scalable managed service
Fault Tolerance	Multi-node replication within DAX	Fully managed multi-AZ redundancy
Consistency	Eventually consistent reads	Strong or eventually consistent reads
Write Handling	Writes go through DAX → DynamoDB	Writes go directly to DynamoDB
Use Case	Read-heavy workloads needing low latency	General workloads (read/write balance)

The Two Models in Context

Both **Strongly Consistent Reads** and **Transactional Reads** guarantee correctness — but they operate at **different levels**:

Concept	Strongly Consistent Read	Transactional Read
Scope	One item (single record)	Multiple items (across one or more tables)
Guarantee Type	Latest committed version of an item	All-or-nothing (ACID) consistency across items
Atomicity	Not atomic across items	Fully atomic across all items

Read Timing	Real-time read after write	Snapshot isolation for all items in the transaction
Latency	Low (~milliseconds)	Higher (extra coordination)
Cost	1 RCU for 4 KB item	2× cost of regular operations
Use Case	When you must read the **latest** value of one item	When multiple reads/writes must be consistent together

---

1. Strongly Consistent Reads — "Always Get the Latest Data"

By default, DynamoDB provides **eventually consistent reads** (faster and cheaper).
But you can request a **strongly consistent read**, meaning:

> The read returns the most recent data, reflecting **all** successful writes prior to that read.

Example:

Suppose you just updated a user's balance:

```
dynamodb.update_item(
    TableName='Accounts',
    Key={'UserID': {'S': 'U1'}},
    UpdateExpression='SET Balance = :new',
    ExpressionAttributeValues={':new': {'N': '500'}}
)
```

Now you read the same item:

```
response = dynamodb.get_item(
    TableName='Accounts',
    Key={'UserID': {'S': 'U1'}},
    ConsistentRead=True   # Strongly consistent read
)
```

You are guaranteed to see `Balance = 500`.

If you omit `ConsistentRead=True`, the read might briefly return the **old value** (say 400) until replication catches up.

6. DynamoDB Hands-On (AWS Console)

Step-by-Step Example:

Goal: Create a table and insert data.

1. **Open** AWS Management Console → DynamoDB.
2. Click **Create Table**.
3. Enter:
   - Table name: `Users`
   - Partition key: `UserID` (String)
4. Click **Create Table**.

Table Created!

Now, **insert an item:**

```
{
  "UserID": "U001",
  "Name": "Vilas Varghese",
  "Role": "DevOps Engineer",
  "Skills": ["AWS", "Terraform", "EKS"]
}
```

You can now:

- Use **Query** → find users by `UserID`
- Use **Scan** → retrieve all items

---

7. DynamoDB Pricing Models

Mode	Description	When to Use
Provisioned Capacity	You specify read/write capacity units (RCUs/WCUs)	Predictable traffic
On-Demand Capacity	DynamoDB auto-scales up/down	Unpredictable or spiky traffic
Free Tier	25 GB storage + 25 RCU/WCU free per month	Testing / learning

---

8. Summary

Concept	Key Takeaway
NoSQL	Flexible, schema-less database for scale and speed
DynamoDB	AWS's fully managed NoSQL service
Data Models	Key-Value & Document
Core Strengths	Serverless, Fully Managed, Fast, Secure, Scalable
Use Cases	IoT apps, gaming, serverless APIs, e-commerce, mobile backends

---

9. Quick Example: Serverless App with DynamoDB

```
API Gateway → Lambda Function → DynamoDB Table
```

- API Gateway exposes an API.
- Lambda stores and reads data from DynamoDB.
- Fully serverless, infinitely scalable, and pay-per-use.

. Setting Up an AWS Account and DynamoDB Environment

Option 1: Using DynamoDB on AWS Cloud (Recommended for Production)

If you already have an AWS account:

Steps:

1. Go to https://aws.amazon.com/console/
2. Sign in or create a **new AWS account**.
3. In the AWS Console search bar, type **"DynamoDB"**.
4. Open the **DynamoDB dashboard**.
5. You can now:
   - Create tables
   - Insert items
   - Run queries and scans directly from the console UI

Tip:
All new AWS accounts get **25 GB of DynamoDB free tier** (good for practice).

Option 2: Running DynamoDB Locally (for offline testing)

AWS provides a **local version of DynamoDB** that you can run on your machine.
It behaves just like the cloud version — perfect for development and CI/CD pipelines.

A. Install DynamoDB Local

Option 1: Using AWS CLI & Java

1. Install **Java 8+** on your system.
2. Download DynamoDB Local:

   ```
   mkdir ~/dynamodb_local && cd ~/dynamodb_local
   wget https://s3.us-west-2.amazonaws.com/dynamodb-local/dynamodb_local_latest.zip
   unzip dynamodb_local_latest.zip
   ```

3. Run the local server:

   ```
   java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
   ```

4. The service runs on:
   `http://localhost:8000`

Option 2: Using Docker (Simplest Way)

If you have Docker installed:

```
docker run -p 8000:8000 amazon/dynamodb-local
```

Now DynamoDB Local runs on your system — accessible at:
`http://localhost:8000`

B. Configure AWS CLI (to talk to local or real DynamoDB)

```
aws configure
```

Enter dummy values for local testing:

```
AWS Access Key ID [None]: test
AWS Secret Access Key [None]: test
Default region name [None]: us-west-2
Default output format [None]: json
```

Now test connection:

```
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

You should get:

```
{ "TableNames": [] }
```

---

2. Understanding DynamoDB Core Components

DynamoDB organizes data into Tables, Items, and Attributes — similar to a database → table → row → column model, but far more flexible.

A. Table

A **Table** is the **top-level container** that stores data.

- Each table has a **Primary Key**.
- No fixed schema — items in a table can have **different attributes**.
- You can define **throughput capacity**, **indexes**, and **encryption** at table level.

Analogy:
Think of a **Table** as a **folder** that contains multiple JSON documents.

B. Item

An **Item** is a single record (similar to a row in SQL).

- Every item must have a **Primary Key** (Partition Key or Partition + Sort Key).
- Other attributes can vary per item.

Example — `Users` table:

UserID (PK)	Name	Role	Skills
U001	Vilas	DevOps	["AWS","K8s"]
U002	Anjali	Developer	["Python","React"]

C. Attribute

An **Attribute** is a single data field within an item.

Types of attributes:

- **Scalar** → single value
- **Document** → nested structures
- **Set** → collections of values

String set ["Admin", "Editor", "Viewer"]

Example item (JSON view):

```
{
  "UserID": "U001",
  "Name": "Vilas Varghese",
  "Role": "DevOps Engineer",
  "Skills": ["AWS", "Kubernetes"],
  "Profile": {
    "City": "Bangalore",
    "Experience": 5
  }
}
```

---

3. Understanding the Primary Key

Every DynamoDB table **must have a Primary Key**.
It uniquely identifies each item.

Key Type	Structure	Example	Use Case
Partition Key (Simple Key)	Single attribute	`UserID`	Unique user or product

Composite Key	Partition Key + Sort Key (`UserID`, `Timestamp`) Multiple items per user (e.g., user logs, orders)

Example

Users table (Simple Key):

```
{
  "UserID": "U001",
  "Name": "Vilas"
}
```

Orders table (Composite Key):

```
{
  "UserID": "U001",
  "OrderDate": "2025-11-12",
  "Item": "Laptop",
  "Price": 75000
}
```

Here, `UserID` = Partition Key
`OrderDate` = Sort Key
This allows multiple orders for one user.

---

4. DynamoDB Data Types

DynamoDB supports **three categories** of data types:

A. Scalar Types (single value)

Type	Description	Example
`String (S)`	Text data	`"Vilas"`
`Number (N)`	Integer or Float	`101`, `3.14`
`Binary (B)`	Binary data (images, files encoded)	`"QmluYXJ5RGF0YQ=="`
`Boolean (BOOL)`	True/False	`true`
`Null (NULL)`	Null value	`null`

Example item:

```
{
  "UserID": "U001",
  "Age": 30,
  "Active": true
}
```

B. Document Types (nested JSON-like structure)

Type	Description	Example
`List (L)`	Ordered collection of values	`["AWS", "DevOps"]`
`Map (M)`	Key-value pairs (like JSON object)	`{"City": "Bangalore", "Experience": 5}`

Example item:

```
{
  "UserID": "U001",
  "Profile": {
    "City": "Bangalore",
    "Experience": 5
  },
  "Skills": ["AWS", "Kubernetes", "Docker"]
}
```

Documents allow flexible nesting — perfect for semi-structured data.

C. Set Types (unique collections)

Type	Description	Example
`String Set (SS)`	Set of unique strings	`["AWS","Terraform","EKS"]`
`Number Set (NS)`	Set of unique numbers	`[101, 202, 303]`
`Binary Set (BS)`	Set of binary values	`["QmluYXJ5MQ==","QmluYXJ5Mg=="]`

Example:

```
{
  "UserID": "U002",
  "Certifications": ["AWS-SA", "CKA", "Terraform-Assoc"]
}
```

Sets automatically ensure **uniqueness** and allow **set operations** (union, intersection, etc.)

---

5. Hands-on Practice via AWS CLI (Local or Cloud)

Create a Table

```
aws dynamodb create-table \
    --table-name Users \
    --attribute-definitions AttributeName=UserID,AttributeType=S \
    --key-schema AttributeName=UserID,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --endpoint-url http://localhost:8000
```

---

2 Insert an Item

```
aws dynamodb put-item \
    --table-name Users \
    --item '{"UserID": {"S": "U001"}, "Name": {"S": "Vilas"}, "Skills": {"L": [{"S": "AWS"}, {"S": "Kubernetes'
    --endpoint-url http://localhost:8000
```

3 Retrieve the Item

```
aws dynamodb get-item \
    --table-name Users \
    --key '{"UserID": {"S": "U001"}}' \
    --endpoint-url http://localhost:8000
```

Output:

```
{
  "Item": {
    "UserID": {"S": "U001"},
    "Name": {"S": "Vilas"},
    "Skills": {"L": [{"S": "AWS"}, {"S": "Kubernetes"}]}
  }
}
```

---

6. Summary Table

Concept	Description	Example
Table	Top-level data container	`Users`
Item	Single record	`{ "UserID": "U001", "Name": "Vilas" }`
Attribute	Field within an item	`"Name": "Vilas"`
Primary Key	Uniquely identifies item	`UserID` or (`UserID`, `Timestamp`)
Scalar Type	Single value	String, Number
Document Type	Nested data	Map, List
Set Type	Unique collections	String Set, Number Set

-

- ## 1. The Importance of the Primary Key

Every item (row) in a DynamoDB table **must have a unique primary key** — this is how DynamoDB identifies and retrieves data efficiently.

➤ What is a Primary Key?

The **Primary Key** uniquely identifies each item in the table.
Unlike relational databases (where you can have many columns forming a composite key), DynamoDB uses one of two **primary key types**:

1. **Simple Primary Key** (only Partition Key)
2. **Composite Primary Key** (Partition Key + Sort Key)

➤ Why the Primary Key Is So Important

The Primary Key determines:

- How your data is **stored and distributed** across partitions.
- How your data can be **queried efficiently** (using `GetItem`, `Query`, etc.).
- How DynamoDB maintains **performance at scale**.

Because DynamoDB is a **distributed system**, the **primary key design** determines which **partition** (physical storage node) each item goes to.
So — choosing the right key structure is not just about uniqueness, but also about **even data distribution** and **query flexibility**.

---

2. The Partition Key (Hash Key)

➤ What it is

The **Partition Key** (also known as the **Hash Key**) is the first part of the primary key.
It is a **single attribute** whose value is **hashed** by DynamoDB to determine the **partition (storage node)** in which to store the item.

➤ Role in Data Distribution

The Partition Key controls **data distribution**.

- DynamoDB computes a **hash value** from the Partition Key.
- This hash value maps to a **specific physical partition**.
- All items with the same Partition Key end up in the same physical partition.

Example:

UserID (Partition Key)	Name	Age
`user1`	Alice	30
`user2`	Bob	25
`user3`	Charlie	28

If `UserID` is the partition key:

- Each user's data is distributed based on the hash of `user1`, `user2`, etc.
- Each user can be efficiently retrieved by its Partition Key.

➤ Problems of Poor Partition Key Design

If your Partition Key has **low cardinality** (few distinct values), it causes:

- **Hot partitions** — too much traffic on a single key.
- **Throttling** and **uneven data distribution**.
- **Poor performance**.

Example:

If your Partition Key is `Country`, and most users are from `India`, then the "India" partition becomes hot.

Best Practice:
Use a partition key that has **high cardinality** (many unique values) and **distributes read/write operations evenly**.

---

3. The Sort Key (Range Key)

➤ What it is

The **Sort Key** (also called **Range Key**) is the second part of a **composite primary key**.
Together, the Partition Key and Sort Key **uniquely identify** an item.

> Primary Key = (Partition Key + Sort Key)

➤ How it works

- All items with the same Partition Key are **stored together**, ordered by the Sort Key.
- This enables **sorted queries**, **range queries**, and **hierarchical data organization**.

Example:

UserID (Partition Key)	OrderDate (Sort Key)	OrderTotal
`user1`	`2024-11-01`	$100
`user1`	`2024-11-05`	$200
`user1`	`2024-11-10`	$50

Here:

- Partition Key = `user1`
- Sort Key = `OrderDate`

You can query all of user1's orders **sorted by date** or filter between specific dates:

```
Query: Get all orders for user1 between 2024-11-01 and 2024-11-07
```

➤ Benefits of Sort Key

- Enables **range queries** (`BETWEEN`, `BEGINS_WITH`, etc.)
- Maintains **ordered data** within a partition
- Useful for **time-series**, **versioned**, or **event log** data

4. Basic Data Modeling: Designing Your First Table

Let's go through a step-by-step example

Use Case: Store and retrieve user orders

Step 1: Identify Access Patterns

Ask:

- What questions will the application ask the database?

Examples:

1. Get a user by ID
2. Get all orders for a user
3. Get user's orders between two dates

Step 2: Define Table and Keys

We need to support "Get all orders for a user" and "filter by date".

So:

- **Partition Key** → `UserID`
  (groups all orders by user)
- **Sort Key** → `OrderDate`
  (enables ordering and range queries)

Step 3: Create Table

Attribute Name	Type
`UserID`	String
`OrderDate`	String (ISO-8601 date)
`OrderID`	String
`OrderTotal`	Number

Primary Key: (`UserID` [Partition Key], `OrderDate` [Sort Key])

This allows queries like:

```
Query all orders for user1 between 2024-11-01 and 2024-11-10
```

Step 4: Add Secondary Indexes (Optional)

If you also need to query by `OrderID`, create a **Global Secondary Index (GSI)**:

- GSI Partition Key: `OrderID`
- GSI Sort Key: (optional)

Example in AWS CLI

```
aws dynamodb create-table \
    --table-name UserOrders \
    --attribute-definitions \
        AttributeName=UserID,AttributeType=S \
        AttributeName=OrderDate,AttributeType=S \
    --key-schema \
        AttributeName=UserID,KeyType=HASH \
        AttributeName=OrderDate,KeyType=RANGE \
    --billing-mode PAY_PER_REQUEST
```

Summary

Concept	Description	Purpose
Primary Key	Unique identifier (Partition Key or Partition + Sort Key)	Ensures uniqueness & drives performance
Partition Key	Hash-based key	Distributes data evenly across partitions
Sort Key	Range-based key	Enables sorting, filtering, and complex queries
Good Data Model	Start with access patterns	Enables predictable and scalable performance

Core Operations and Querying (CRUD) in **complete detail**.
CRUD stands for:

- **C**reate → `PutItem`

- **R**ead → `GetItem`
- **U**pdate → `UpdateItem`
- **D**elete → `DeleteItem`

Each operation interacts directly with your DynamoDB table through the AWS CLI, SDKs, or Console.

Create an Item — `PutItem`

➤ Purpose

`PutItem` adds a **new item** to a DynamoDB table or **replaces** an existing item **with the same primary key**.

➤ Basic Syntax (AWS CLI)

```
aws dynamodb put-item \
    --table-name UserOrders \
    --item '{
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"},
        "OrderID": {"S": "ORD123"},
        "OrderTotal": {"N": "200"}
    }'
```

➤ Explanation:

- `--table-name`: Target DynamoDB table.
- `--item`: JSON structure defining the item's attributes.
- `"S"` indicates a **String** type, `"N"` a **Number**, `"BOOL"` for Boolean, etc.

➤ Key Details

- If the same **Primary Key** (`UserID + OrderDate`) already exists, DynamoDB **overwrites** the item by default.
- You can **prevent overwriting** using a **Condition Expression**.

Example (Prevent Overwrite):

```
aws dynamodb put-item \
    --table-name UserOrders \
    --item '{
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"},
        "OrderID": {"S": "ORD123"},
        "OrderTotal": {"N": "200"}
    }' \
    --condition-expression "attribute_not_exists(UserID)"
```

This ensures you **only create a new item** if `UserID` doesn't exist yet.

---

2 Read an Item — `GetItem`

➤ Purpose

`GetItem` retrieves a **single item** by its **Primary Key**.
You must specify **both the Partition Key and Sort Key** (if the table has one).

➤ Basic Syntax

```
aws dynamodb get-item \
    --table-name UserOrders \
    --key '{
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"}
    }'
```

➤ Example Output

```
{
    "Item": {
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"},
        "OrderID": {"S": "ORD123"},
        "OrderTotal": {"N": "200"}
    }
}
```

➤ Notes

- If no item exists, the output is **empty**.
- It always retrieves the **entire item**, unless you use **ProjectionExpression**.

Example (Get specific attributes only)

```
--projection-expression "OrderID, OrderTotal"
```

➤ Use Cases

- Quick lookup by key (fastest operation in DynamoDB).
- Ideal for user profile retrieval, product lookup, etc.

---

3 Update an Item — `UpdateItem`

➤ Purpose

`UpdateItem` modifies **one or more attributes** of an existing item — without replacing the entire record.

You can:

- Add new attributes
- Modify existing ones
- Increment numeric values
- Remove attributes

➤ Basic Syntax

```
aws dynamodb update-item \
    --table-name UserOrders \
    --key '{
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"}
    }' \
    --update-expression "SET OrderTotal = :val" \
    --expression-attribute-values '{":val": {"N": "250"}}'
```

➤ Explanation:

- `--update-expression` defines the operation.
  - `SET` → assign a new value.
  - `ADD` → increment numbers or append to sets.
  - `REMOVE` → delete attributes.
- `--expression-attribute-values` supplies the value placeholders.

➤ Example 1: Increment a Value

```
--update-expression "ADD OrderTotal :inc"
--expression-attribute-values '{":inc": {"N": "50"}}'
```

Adds +50 to the existing `OrderTotal`.

➤ Example 2: Add a New Attribute

```
--update-expression "SET PaymentStatus = :status"
--expression-attribute-values '{":status": {"S": "Paid"}}'
```

➤ Example 3: Conditional Update

Only update if the order is not already marked as Paid:

```
--condition-expression "PaymentStatus <> :status"
```

DynamoDB ensures **atomic** updates — no race conditions even with concurrent writes.

---

4 Delete an Item — `DeleteItem`

➤ Purpose

`DeleteItem` removes an item from a table based on its **primary key**.

➤ Basic Syntax

```
aws dynamodb delete-item \
    --table-name UserOrders \
    --key '{
        "UserID": {"S": "user1"},
        "OrderDate": {"S": "2024-11-05"}
    }'
```

➤ Key Points

- If the item doesn't exist, DynamoDB does **nothing** (no error).
- You can use a **condition expression** to delete only if a specific attribute matches.

Example (Conditional Delete)

```
--condition-expression "OrderTotal < :max"
--expression-attribute-values '{":max": {"N": "300"}}'
```

Deletes the item only if `OrderTotal` is less than 300.

Quick Comparison of CRUD Operations

Operation	Command	Purpose	Notes
PutItem	`aws dynamodb put-item`	Create/Replace an item	Overwrites existing item if same key
GetItem	`aws dynamodb get-item`	Retrieve one item	Fastest read, must specify full key
UpdateItem	`aws dynamodb update-item`	Modify attributes	Supports conditions and atomic operations
DeleteItem	`aws dynamodb delete-item`	Remove item	Supports conditions, safe delete

Dynamo db supports
High availability (% of time it is up - 99.999 - 5 min. yearly downtime)
Durability (level of redendency - no fail - separate node replaces) - 99.999999 durability

ideal for known access pattern

Part 3: Querying and Scanning in DynamoDB

Understanding Query vs Scan — Core Difference

Feature	Query	Scan
Access pattern	Retrieves items based on **Primary Key** (Partition Key, and optionally Sort Key)	Examines **every item** in the table
Performance	**Fast** — directly looks up partitions by key	**Slow** — sequentially reads every item
Read capacity usage	Uses **minimal read units** (only scanned items)	Uses **more read units** (reads entire table or index)
Scalability	Scales linearly with data	Becomes inefficient as data grows
Use case	When you know the Partition Key	When you must find data without knowing keys (only for small tables or admin tasks)

Key Point:

- **Query** = "Targeted search" → Fetches items matching a key.
- **Scan** = "Full table read" → Reads everything, filters in memory.

---

2 Why a Scan Is Generally Inefficient

A **Scan** reads **every item** in your table or secondary index.
For large tables, this means:

- High **read capacity unit (RCU)** consumption
- High **latency**

- Potential **throttling**
- Expensive cost

Even if you only need a few items, `Scan` still reads **all of them**, and then filters results **after reading** — meaning:

> DynamoDB charges for everything scanned, not just returned results.

Example of Scan Inefficiency

Suppose your table has **1 million items**, and you want only **100** that match a filter.

- **Query** → Reads only 100 relevant items.
- **Scan** → Reads all 1,000,000 items and filters out 999,900.

That's why `Scan` is only recommended for:

- **Testing / debugging**
- **Occasional maintenance**
- **When you truly can't design a key-based access pattern**

Best Practice:

> Avoid `Scan` in production queries.
> Design your **Partition Key** and **Sort Key** properly to use `Query` instead.

---

3 Using `Query` — The Right Way to Retrieve Data

`Query` retrieves **multiple items** with the **same Partition Key**,
and optionally **filters by Sort Key or other attributes**.

Example Table

Table Name: `UserOrders`
Primary Key:

- Partition Key: `UserID`
- Sort Key: `OrderDate`

➤ Example 1: Basic Query (By Partition Key)

```
aws dynamodb query \
    --table-name UserOrders \
    --key-condition-expression "UserID = :uid" \
    --expression-attribute-values '{":uid": {"S": "user1"}}'
```

What happens:

- DynamoDB looks up **only the partition** for `user1`.
- Returns **all items** (orders) for that user.

➤ Example 2: Query with Sort Key Condition

You can add conditions on the **Sort Key** (e.g., time range, greater than, begins with, etc.)

Get all orders for `user1` between two dates:

```
aws dynamodb query \
    --table-name UserOrders \
    --key-condition-expression "UserID = :uid AND OrderDate BETWEEN :start AND :end" \
    --expression-attribute-values '{
        ":uid": {"S": "user1"},
        ":start": {"S": "2024-11-01"},
        ":end": {"S": "2024-11-10"}
    }'
```

Supported Sort Key operators:

Operator	Description
=	Exact match
<, <=, >, >=	Range comparison
`BETWEEN`	Between two values
`BEGINS_WITH`	Prefix match for strings

➤ Example 3: Query Specific Attributes (Projection)

```
--projection-expression "OrderID, OrderTotal"
```

Retrieves only those attributes (reduces RCU usage).

---

4 Adding Filters — `FilterExpression`

A **Filter Expression** lets you filter results **after** DynamoDB finds items by key.
Important: **The filtering happens after retrieval**, so it **does not reduce RCU usage.**

➤ Example 1: Filter by Non-Key Attribute

Retrieve orders for user1, where OrderTotal > 100.

```
aws dynamodb query \
    --table-name UserOrders \
    --key-condition-expression "UserID = :uid" \
    --filter-expression "OrderTotal > :amount" \
    --expression-attribute-values '{
        ":uid": {"S": "user1"},
        ":amount": {"N": "100"}
    }'
```

DynamoDB first retrieves all orders for `user1`
then filters out those with `OrderTotal <= 100`.

➤ Example 2: Multiple Filters (Using AND/OR)

```
--filter-expression "OrderTotal > :amount AND PaymentStatus = :status"
--expression-attribute-values '{
    ":amount": {"N": "100"},
    ":status": {"S": "Paid"}
}'
```

Important Concepts

Concept	Description
Key Condition Expression	Filters items based on key attributes (Partition + Sort Key). Runs before data retrieval.
Filter Expression	Filters items based on non-key attributes. Runs after data retrieval.
Projection Expression	Limits which attributes are returned. Saves read cost.

---

5 Example — Full Query Command

```
aws dynamodb query \
    --table-name UserOrders \
    --key-condition-expression "UserID = :uid AND OrderDate BETWEEN :start AND :end" \
    --filter-expression "OrderTotal > :amount AND PaymentStatus = :status" \
    --projection-expression "OrderID, OrderDate, OrderTotal" \
    --expression-attribute-values '{
        ":uid": {"S": "user1"},
        ":start": {"S": "2024-11-01"},
        ":end": {"S": "2024-11-30"},
        ":amount": {"N": "100"},
        ":status": {"S": "Paid"}
    }'
```

This:

1. Fetches only `user1`'s partition
2. Filters orders between `2024-11-01` and `2024-11-30`
3. Returns only those with `OrderTotal > 100` and `PaymentStatus = Paid`
4. Returns only the three selected attributes

---

6 Understanding `Scan`

➤ What it does

Reads **every item** in a table or secondary index, optionally applying filters.

➤ Example

```
aws dynamodb scan \
    --table-name UserOrders \
    --filter-expression "OrderTotal > :amount" \
    --expression-attribute-values '{":amount": {"N": "100"}}'
```

Returns all items with `OrderTotal > 100`.
But it still scans every record first.

➤ When to Use `Scan`

- For **small tables** (few hundred items)
- For **admin tasks** (e.g., find missing attributes)
- For **data export / analysis** jobs
- When no key-based pattern exists (but design should fix that!)

➤ Improving Scan Efficiency

If you must use Scan:

1. Use **Parallel Scan** — divide table into segments.

   `--total-segments 4 --segment 0`

   (Run 4 scans in parallel to reduce latency)

2. Use **ProjectionExpression** to read only necessary attributes.
3. **Paginate** results — DynamoDB returns up to **1 MB per call**.

---

7 Summary Table

Operation	Uses Primary Key?	Efficient?	Filters Before Retrieval?	Filters After Retrieval?	Use Case
Query	Yes	Fast	KeyConditionExpression	FilterExpression (optional)	Fetch related items in a partition
Scan	No	Slow	No	FilterExpression	Full-table reads, small tables only

---

8 Best Practices Summary

Always **design your table** to support key-based queries (avoid scan).
Use **FilterExpression** only when you must — it does not reduce cost.
Use **ProjectionExpression** to limit data returned.
Use **Parallel Scan** for faster full-table scans when necessary.
Monitor with **CloudWatch Metrics** (`ConsumedReadCapacityUnits`, `ThrottledRequests`).

Pro Tip:

When you're planning queries:

> "If I can't write a KeyConditionExpression, my data model probably needs improvement."

Part 3: Advanced Topics and Optimization

- **Secondary Indexes:**
  - Understanding the need for secondary indexes.
  - When to use a **Local Secondary Index (LSI)**.
  - When to use a **Global Secondary Index (GSI)**.
  - Best practices for index design and **GSI overloading**.

Part 4: Secondary Indexes in DynamoDB

Why Do We Need Secondary Indexes?

DynamoDB's primary key design is **extremely efficient**,
but it comes with **one limitation**:

You can query only by **Partition Key** (and optionally **Sort Key**) of the **base table**.

This means:

- You can't query by non-key attributes (like email, status, product category, etc.).
- You can't have multiple "query patterns" using different keys on the same data.

Secondary Indexes solve this problem.

They allow you to **create alternate key structures** for the same data — just like creating extra "views" optimized for different queries.

Analogy:

Think of a DynamoDB table as a **primary book index** (by title).
Secondary Indexes are **extra indexes** (like by author or by year).

---

2 Types of Secondary Indexes

Type	Abbreviation	Key Attributes	Created	Query Scope
Local Secondary Index	LSI	Same Partition Key, different Sort Key	At table creation only	Limited to same partition
Global Secondary Index	GSI	Different Partition Key and Sort Key	Can be added anytime	Global (across all partitions)

---

3 Local Secondary Index (LSI)

➤ What It Is

A **Local Secondary Index (LSI)** allows you to:

- Keep the **same Partition Key** as the base table.
- Add a **different Sort Key**.

This means you can query the same "group" of items (same partition) but **sort or filter differently**.

➤ Use Case Example

Base Table: `UserOrders`

- Partition Key: `UserID`
- Sort Key: `OrderDate`

If you also want to query by **PaymentStatus** (e.g., list all "Pending" orders for a user),
you can create an **LSI** with:

- Partition Key: `UserID`
- Sort Key: `PaymentStatus`

➤ Query Example (Using LSI)

Get all orders for `user1` where `PaymentStatus = Pending`:

```
aws dynamodb query \
    --table-name UserOrders \
    --index-name UserPaymentStatusIndex \
    --key-condition-expression "UserID = :uid AND PaymentStatus = :status" \
    --expression-attribute-values '{
        ":uid": {"S": "user1"},
        ":status": {"S": "Pending"}
    }'
```

DynamoDB now uses the LSI to query by status efficiently,
instead of scanning all user orders.

➤ Key Characteristics

Feature	Description
Partition Key	Must match base table's Partition Key
Sort Key	Can be a different attribute
Consistency	Supports **strongly consistent reads**
Creation time	Must be defined when table is created (cannot be added later)
Query scope	Only items within same Partition Key

Limitations

- Max **5 LSIs** per table.
- Must be declared **at table creation**.
- Same partition = same 10 GB limit (across base + LSIs).
- Slightly higher write latency because LSI updates are synchronous.

---

4 Global Secondary Index (GSI)

➤ What It Is

A **Global Secondary Index (GSI)** lets you:

- Define a **completely new Partition Key** and an optional **Sort Key**.
- Query items **across all partitions**, globally.

It's like creating an **entirely new view** of your table.

➤ Use Case Example

Base Table: `UserOrders`

- Partition Key: `UserID`
- Sort Key: `OrderDate`

Now you want to query all orders by **OrderStatus**, regardless of user.
You can create a **GSI** with:

- Partition Key: `OrderStatus`
- Sort Key: `OrderDate`

➤ Example Query on GSI

```
aws dynamodb query \
    --table-name UserOrders \
    --index-name OrderStatusIndex \
    --key-condition-expression "OrderStatus = :status" \
    --expression-attribute-values '{
        ":status": {"S": "Pending"}
    }'
```

DynamoDB can now return **all pending orders** across all users.

➤ Key Characteristics

Feature	Description
Partition Key	Can be different from base table
Sort Key	Optional, can also differ
Consistency	Only **eventually consistent** reads supported
Creation time	Can be added anytime
Query scope	Global (spans all partitions)
Write cost	Slightly higher because updates are propagated asynchronously

---

Example: Table + GSI Definition (CLI)

```
aws dynamodb update-table \
    --table-name UserOrders \
    --attribute-definitions \
        AttributeName=OrderStatus,AttributeType=S \
        AttributeName=OrderDate,AttributeType=S \
    --global-secondary-index-updates '[
        {
            "Create": {
                "IndexName": "OrderStatusIndex",
                "KeySchema": [
                    {"AttributeName": "OrderStatus","KeyType": "HASH"},
                    {"AttributeName": "OrderDate","KeyType": "RANGE"}
                ],
                "Projection": {"ProjectionType": "ALL"},
```

```
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 5,
                    "WriteCapacityUnits": 5
                }
            }
        }
    ]'
```

---

5 Projection Types

Each index (LSI or GSI) must define which attributes it includes — called a **Projection**.

Type	Description	Example Use
KEYS_ONLY	Only index keys + table primary key	Minimal storage, light reads
INCLUDE	Include specific non-key attributes	Optimize for small queries
ALL	Include all attributes	Simple but higher storage & write cost

---

➤ Example

If you only need to query by `OrderStatus` and read `OrderTotal`,
you can use:

```
"Projection": {
  "ProjectionType": "INCLUDE",
  "NonKeyAttributes": ["OrderTotal"]
}
```

---

6 GSI Overloading — Advanced Design Pattern

GSI overloading is a **pro-level design technique** used to support **multiple query patterns** on a single GSI.

Concept:

You "overload" a GSI by storing **different types of items** in the same table (single-table design),
and control how they appear in the index using **attribute values**.

➤ Example

You have two entities in your table:

- `USER` items (user profiles)
- `ORDER` items (orders)

You can define one GSI:

- `GSI1PK`: General-purpose Partition Key
- `GSI1SK`: General-purpose Sort Key

Then populate them differently:

PK	SK	GSI1PK	GSI1SK	EntityType
USER#1	PROFILE	USER#1	PROFILE	USER
USER#1	ORDER#123	USER#1	ORDER#123	ORDER
USER#1	ORDER#124	USER#1	ORDER#124	ORDER

You can now query:

- All user's orders → `GSI1PK = USER#1 AND begins_with(GSI1SK, "ORDER#")`
- User profile → `GSI1PK = USER#1 AND GSI1SK = PROFILE`

One index supports multiple logical query patterns.

Benefits

- Fewer indexes → simpler maintenance and lower cost.
- Efficient multi-entity access (single-table design).

Caveats

- Requires disciplined attribute naming.

- Harder to understand/debug without proper documentation.
- Must plan queries in advance.

---

7 Best Practices for Index Design

Do's:

- Choose **access patterns first**, design indexes around them.
- Use **projection types** wisely — include only needed attributes.
- Monitor **index write capacity** (each write updates the index too).
- Use **GSI overloading** for complex data models.
- Prefer **Query** over **Scan** on indexes as well.

Don'ts:

- Don't create unnecessary indexes — every write incurs additional cost.
- Don't depend on LSI unless you need **strong consistency** or per-partition alternate sorting.
- Don't create GSIs that mirror your base table — wastes capacity.

---

8 Summary Table

Type	Partition Key	Sort Key	Scope	Consistency	Creation Time	Use Case
LSI	Same as table	Different	Local	Strong/Eventually	At table creation	Alternate sort/filter per partition
GSI	Different	Optional	Global	Eventually	Anytime	Alternate query pattern across all data

---

Real-world Example Recap

Table: `UserOrders`

- Partition Key: `UserID`
- Sort Key: `OrderDate`

Indexes:

1. **LSI:** `UserID + PaymentStatus` → Query orders per user by payment status
2. **GSI:** `OrderStatus + OrderDate` → Query all orders globally by status

Combined Access Patterns:

- Get all orders for user → `Query base table`
- Get all paid orders for user → `Query LSI`
- Get all pending orders globally → `Query GSI`

Summary of Key Takeaways

Secondary Indexes let you query data using alternate keys.
LSI = same Partition Key, different Sort Key (local scope).
GSI = new Partition Key, global scope (flexible).
Projections control storage vs cost trade-off.
GSI overloading enables multiple query patterns efficiently.
Always design indexes around **real-world access patterns**.

- **Capacity and Cost Management:**
  - **Provisioned Capacity vs. On-Demand Capacity**.
  - Understanding Read/Write Capacity Units (RCUs and WCUs).
  - How to use **Auto Scaling** to manage provisioned capacity.

1. Capacity Modes: Provisioned vs On-Demand

➤ On-Demand Capacity Mode

- In On-Demand mode, you **don't provision** read/write capacity ahead of time. You pay **per request** (per read/write) as your application uses it. Amazon Web Services, Inc.+2AWS Documentation+2
- It is ideal for workloads that are **unpredictable**, **bursty**, or when you don't know your traffic patterns. AWS Documentation+1
- Because you pay for actual usage, you avoid the cost of unused capacity.
- The trade-off: per-unit cost tends to be **higher** than for provisioned (for steady workloads). For example, one blog notes that on-demand can cost ~6–7× more than provisioned if you have very steady high usage. awsfundamentals.com+1

➤ Provisioned Capacity Mode

- In Provisioned mode, you **specify** how many read capacity units (RCUs) and write capacity units (WCUs) your table (or index) needs per second. Then you pay **hourly** for that provisioned amount — whether you use it or not. Amazon Web Services, Inc.+1
- Good for predictable, stable workloads where you can estimate usage and benefit from cost savings by provisioning "just enough".
- Risk: If you under-provision, you'll get throttled. If you over-provision (and don't use it), you pay for capacity wasted.
- You can combine with Auto Scaling (we'll cover below) to dynamically adjust capacity, which mitigates some of the risk.

---

2. Understanding Read/Write Capacity Units (RCUs & WCUs)

➤ Write Capacity Unit (WCU)

- 1 WCU allows **one standard write request per second** for items **up to 1 KB** in size. Amazon Web Services, Inc.+1
- If you write larger items (say 3 KB), you'll consume **3 WCUs** for that one write. (Standard write) Amazon Web Services, Inc.
- For transactional writes, it can require **2 WCUs** for a 1KB item (or more for larger item sizes). Amazon Web Services, Inc.

➤ Read Capacity Unit (RCU)

- 1 RCU allows **one strongly consistent read per second** for items **up to 4 KB**. Amazon Web Services, Inc.+1
- If you do **eventually consistent** reads, one RCU supports **two reads per second** (of items up to 4 KB). So eventual consistency is cheaper in terms of capacity units. Amazon Web Services, Inc.
- Larger item size (e.g., 8 KB) means you'll need more RCUs (rounded up) — e.g., reading 8 KB strongly consistent uses 2 RCUs. Amazon Web Services, Inc.

➤ Why this matters

- In Provisioned mode you pay **per RCU/WCU hour**, whether you use them or not. So sizing accurately matters. knowi.com
- In On-Demand mode you pay per **request unit** (WRU for write requests, RRU for reads) as you consume them. airbyte.com

---

3. Cost Example – US East (Ohio) Region

While exact region-specific values may change, here are typical numbers and how to calculate.

➤ Approximate unit rates (for provisioned mode)

According to sources:

- WCU cost ≈ **$0.00065 per WCU-hour** in cheaper regions like us-east-1, us-east-2. Cloudforecast+1
- RCU cost ≈ **$0.00013 per RCU-hour** in those regions. knowi.com+1
- Storage cost (Standard table class) ≈ **$0.25 per GB-month** after first 25 GB free. pump.co+1
- On-demand request cost example: write requests ~$0.625 per million WRUs, read ~$0.125 per million RRUs in those regions. pump.co+1

➤ Sample scenario

Scenario A: Steady workload (Provisioned)

Suppose your application needs:

- 100 writes per second (each up to 1 KB) → you need **100 WCUs**
- 200 reads per second (strongly consistent up to 4 KB) → you need **200 RCUs**

Cost per hour:

- Write capacity cost = 100 WCUs × $0.00065 = **$0.065 per hour**
- Read capacity cost = 200 RCUs × $0.00013 = **$0.026 per hour**
- Combined = **$0.091 per hour**

Cost per month (≈ 730 hours):

- $0.091 × 730 ≈ **$66.43** per month for capacity only.
- Add storage: say 100 GB of data → (100 GB – 25 GB free) = 75 GB charged × $0.25 = **$18.75/month**
- Total ≈ **$85.18/month**

Scenario B: Unpredictable, bursty workload (On-Demand)

Suppose you have:

- 2 million writes per month (up to 1 KB each)
- 4 million reads per month (strongly consistent 4 KB)

Using on-demand rates:

- Write cost ≈ 2 million × ($0.625 per million) = **$1.25**
- Read cost ≈ 4 million × ($0.125 per million) = **$0.50**
- Storage: same 100 GB → ~$18.75
- Total ≈ **$20.50/month**

So in this scenario, On-Demand is much cheaper — because usage is low and unpredictable.

Summary comparison

- If you have **steady, predictable** capacity needs (like Scenario A), provisioned mode gives cost-predictability and savings.
- If you have **variable, unpredictable traffic** (Scenario B), on-demand mode avoids waste and can be cheaper.

---

4. How to Use Auto Scaling with Provisioned Capacity

➤ Why Auto Scaling?

Because workloads often fluctuate (diurnal patterns, seasonal bursts), manually adjusting capacity is risky and laborious. Auto Scaling lets you set minimum/maximum capacity and target utilization (e.g., 70 %) and have AWS adjust your provisioned RCUs/WCUs within that envelope. awsfundamentals.com+1

➤ How it works

- Set initial “baseline” RCUs and WCUs (min capacity).
- Define maximum capacity you’re willing to allow.
- Define target utilization % (for example, 70%). That means when usage (consumed capacity) repeatedly hits ~70% of provisioned, scaling up will be triggered.
- DynamoDB auto-scales (up or down) within those bounds.
- You still pay for provisioned capacity, so you want the min to be as low as your lowest safe capacity, and max to handle spikes.

➤ Example

From earlier scenario with 100 WCUs / 200 RCUs baseline:

- Minimum: 50 WCUs / 100 RCUs
- Maximum: 150 WCUs / 300 RCUs
- Target utilisation: 70%

If an event causes writes to jump to 120/sec, auto scaling will raise WCUs toward 150 to keep you under throttle and near target utilisation. After traffic drops, capacity can scale down (but not below your min).

➤ Best practices

- Monitor metrics like `ConsumedCapacityUnits`, `ProvisionedCapacityUnits`, `ThrottledRequests`.
- Choose sensible lower bound (min) so cost isn't wasted when traffic is low.
- Choose upper bound (max) to avoid unexpected cost spikes.
- Use scheduled scaling if you know patterns (e.g., peak hours).
- Use eventually consistent reads when acceptable (half the RCU cost) — frees up capacity.
- Use caching layers (like DynamoDB Accelerator (DAX)) for read-heavy workloads to reduce RCUs.

---

5. Summary & Recommendations

- Choose **On-Demand** when: unpredictable traffic, new application, variable/spike heavy usage.
- Choose **Provisioned + Auto Scaling** when: steady predictable usage, cost-sensitive, you can monitor and adjust.
- Understand RCUs/WCUs: size items, choose consistency mode (strong vs eventual) to minimize consumption.
- Storage, backups, global tables, indexes also add cost — factor those in.
- Use cost-calculator tools (e.g., AWS Pricing Calculator) to estimate. calculator.aws

- **Advanced Features:**
  - **DynamoDB Streams**: Capturing item-level changes.
  - **Time-to-Live (TTL)**: Automatically expiring items.
  - **DAX (DynamoDB Accelerator)**: In-memory caching for faster reads.
  - **Global Tables**: Multi-region, multi-master replication

DynamoDB Advanced Features

DynamoDB Streams — *Capturing Item-Level Changes*

What It Is

DynamoDB Streams is a **time-ordered sequence of item-level changes** (insert, update, delete) that occur in your DynamoDB table.
It acts like a **change log**, allowing you to react to data modifications in real time.

Think of it as **"Change Data Capture" (CDC)** for DynamoDB.

How It Works

- Whenever an item in a table is **created, updated, or deleted**, the change is **recorded in the Stream**.
- Each record contains:
  - The **event type** (`INSERT`, `MODIFY`, `REMOVE`)
  - The **new and/or old images** (before and after data)
  - The **timestamp** and **sequence number**

You can enable streams with one of four options:

- `KEYS_ONLY` – Only primary keys
- `NEW_IMAGE` – Only the new item after modification
- `OLD_IMAGE` – Only the old item before modification

- `NEW_AND_OLD_IMAGES` – Both before and after images

Use Cases

Use Case	Description
Event-driven processing	Trigger AWS Lambda when new data arrives (e.g., send notification on new order)
Data replication	Stream changes to another DynamoDB table or S3 for analytics
Audit logs	Maintain a full history of changes for compliance
Search indexing	Update an external index (e.g., OpenSearch or Elasticsearch) automatically

---

Example: Trigger Lambda on New Orders

```
aws dynamodb update-table \
  --table-name Orders \
  --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Then in AWS Lambda console:

- Add a trigger: "DynamoDB Stream"
- Select the Orders table's stream
- Lambda will receive event objects like:

```
{
  "eventName": "INSERT",
  "dynamodb": {
    "NewImage": {
      "OrderID": {"S": "A100"},
      "Status": {"S": "CREATED"}
    }
  }
}
```

---

2 Time-to-Live (TTL) — *Automatically Expiring Items*

What It Is

TTL is a feature that lets you define a **timestamp attribute** (Unix epoch time) for each item.
When the current time passes that timestamp, DynamoDB **automatically deletes** the item — no manual cleanup needed.

How It Works

- You define a TTL attribute (e.g., `expireAt`)
- DynamoDB checks that attribute periodically
- When time passes, the item is **deleted asynchronously**
  (within 48 hours of expiration)

Example

ItemID	Data	expireAt
1	"session123"	1731465600 (11 Nov 2025, 00:00 UTC)

If TTL = `expireAt`, then this item will auto-delete after that time.

Enable TTL:

```
aws dynamodb update-time-to-live \
  --table-name SessionTable \
  --time-to-live-specification "Enabled=true, AttributeName=expireAt"
```

Benefits

- No cron jobs needed for cleanup.
- Reduces storage cost automatically.
- Great for **sessions, cache data, temporary tokens, expiring notifications**.

Notes

- Expired items are **not immediately deleted** — DynamoDB removes them *eventually*.

- Deletes from TTL **don't consume WCU**.
- TTL deletions can generate **Stream records**, so you can chain cleanup events if needed.

---

3 DAX (DynamoDB Accelerator) — *In-Memory Caching for Faster Reads*

What It Is

DAX (DynamoDB Accelerator) is a **fully managed, in-memory cache** for DynamoDB that reduces **read latency** from **milliseconds to microseconds** — up to 10× performance boost.

How It Works

- It's a **cluster** sitting between your application and DynamoDB.
- Your app queries the **DAX endpoint** instead of DynamoDB directly.
- DAX checks if data is cached:
  - If cached → returns instantly.
  - If not → fetches from DynamoDB and stores in cache.

Use Cases

Use Case	Description
Read-heavy workloads	Shopping carts, gaming leaderboards
Session management	Frequently read user sessions
Query caching	Common queries with infrequent updates

Example: How It Fits

```
App → DAX Cluster → DynamoDB Table
```

AWS SDKs support DAX natively:

```
from amazondax import AmazonDaxClient
from boto3.dynamodb.conditions import Key

dax = AmazonDaxClient(endpoint_url='mydaxcluster.abc123.clustercfg.dax.us-east-1.amazonaws.com')
table = dax.Table('UserSessions')

response = table.get_item(Key={'SessionID': '12345'})
print(response['Item'])
```

Benefits

- Reduces read latency (microseconds)
- Fully managed (AWS handles scaling, fault tolerance)
- Compatible with existing DynamoDB SDK APIs
- Reduces RCU consumption (reads served from cache)

Limitations

- **Write-through caching:** writes go to DAX → DynamoDB → DAX updated
- **Eventually consistent reads only**
- **Not global** (region-specific)
- Ideal for **read-heavy, write-light** workloads.

---

4 Global Tables — *Multi-Region, Multi-Master Replication*

What It Is

Global Tables allow you to replicate DynamoDB tables **across multiple AWS regions**, with **multi-master** write capability.
Every region can both **read and write**, and DynamoDB automatically replicates changes to all other regions.

How It Works

- You create a Global Table by linking identical tables in multiple regions.
- DynamoDB Streams + cross-region replication keep them in sync.
- Each replica is **fully functional** — local reads and writes, with **eventual consistency** across regions.

Example

Region	Table	Role
us-east-1 (N. Virginia)	Users	Active writer
eu-west-1 (Ireland)	Users	Active writer
ap-south-1 (Mumbai)	Users	Active writer

When a record is updated in Mumbai, DynamoDB Streams replicate it automatically to N. Virginia and Ireland.

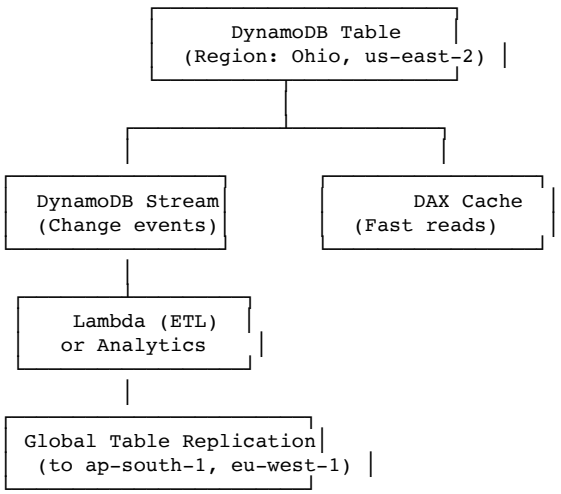
Benefits

- **Low latency** reads/writes for global users.
- **Active-active replication** — all regions can write.
- **Disaster recovery** — if one region fails, others continue.
- Fully managed and **conflict resolution handled automatically** (last-writer-wins model).

Considerations

- Each region has its **own capacity cost**.
- **Replication latency**: usually <1 second.
- If two regions update the same item at the same time, **last-writer-wins** (based on timestamp).
- Requires consistent schema across all replicas.

Example Architecture Overview



Summary Table

Feature	Purpose	Key Benefit	Example Use Case
Streams	Capture item-level changes	Real-time triggers, ETL	Lambda on data change
TTL	Auto-expire items	Storage cleanup	Session or token expiry
DAX	In-memory caching	Faster reads	Gaming leaderboard
Global Tables	Multi-region replication	Low-latency global access	Multi-region eCommerce

- **Security and Best Practices:**
  - IAM policies for fine-grained access control.
  - Encryption at rest and in transit.
  - Optimizing for single-table design.
  - Common anti-patterns to avoid.

DynamoDB Security and Best Practices

IAM Policies for Fine-Grained Access Control

Overview

DynamoDB integrates tightly with **AWS Identity and Access Management (IAM)** for authentication and authorization.
IAM controls **who** can perform **what actions** on **which resources** — at the **table, item, or attribute** level.

Levels of Access Control

Level	Description	Example
Table-level	Control who can read/write entire tables	Allow only read access to `OrdersTable`
Item-level	Control access to specific items via conditions	Allow users to read only their own items (using `UserID`condition)
Attribute-level	Control which attributes are visible	Allow read of all attributes except `Password`

Example: Fine-Grained IAM Policy (Table-Level)

Allow a user to **read and write items** in a specific table:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb:DeleteItem"
      ],
      "Resource": "arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable"
    }
  ]
}
```

Example: Item-Level Access (Using Condition Keys)

Allow a user to access **only their own records** in `Users` table:

```
{
  "Effect": "Allow",
  "Action": ["dynamodb:GetItem", "dynamodb:Query"],
  "Resource": "arn:aws:dynamodb:us-east-2:123456789012:table/Users",
  "Condition": {
    "ForAllValues:StringEquals": {
      "dynamodb:LeadingKeys": ["${aws:username}"]
    }
  }
}
```

This ensures each IAM user can query **only items where the Partition Key equals their username** — perfect for multi-tenant apps.

Example: Backup and Restore Role

You can create specific roles to **restrict backup-related access** only:

```
{
  "Effect": "Allow",
  "Action": [
    "dynamodb:CreateBackup",
    "dynamodb:DescribeBackup",
    "dynamodb:DeleteBackup",
    "dynamodb:ListBackups",
    "dynamodb:RestoreTableFromBackup"
```

```
  ],
  "Resource": "*"
}
```

Best Practices

- Use **least privilege principle** — only allow specific actions.
- Assign roles to **applications**, not users.
- Use **IAM roles for service accounts** (ECS/EKS/IAM roles for EC2 or Lambda).
- Combine with **AWS Organizations SCPs** for account-level guardrails.
- Rotate IAM keys regularly and **avoid hardcoding credentials** — use AWS SDK default providers or IAM roles.

---

2 Encryption — At Rest and In Transit

Encryption at Rest

DynamoDB **encrypts all data at rest by default** using **AWS KMS** (Key Management Service).

Type	Description
AWS owned CMK	Default free key managed by AWS (automatic)
AWS managed CMK	Key created automatically per account/service
Customer managed CMK	Your own key — gives control over key rotation, access, audit

---

Example: Customer-Managed CMK

When creating a table, you can specify:

```
aws dynamodb create-table \
  --table-name SecureTable \
  --sse-specification Enabled=true,SSEType=KMS,KMSMasterKeyId=arn:aws:kms:us-east-2:123456789012:key/abcd-1
```

DynamoDB then encrypts:

- Table data
- Indexes
- Streams
- Backups and restores

Encryption in Transit

All communication between clients and DynamoDB uses **TLS (HTTPS)** automatically.

When you use the AWS SDK, all API calls go through HTTPS:

```
dynamodb = boto3.client('dynamodb', region_name='us-east-2', use_ssl=True)
```

Protects against man-in-the-middle attacks and ensures confidentiality over the network.

Best Practices

- Use **Customer-managed KMS keys** for compliance or audit control.
- Restrict access to the KMS key via **key policies**.
- Rotate encryption keys periodically (KMS can auto-rotate yearly).
- Use **VPC Endpoints (AWS PrivateLink)** to keep DynamoDB traffic inside AWS network (not over public Internet).

---

3 Optimizing for Single-Table Design

Why Single Table?

Traditional relational design uses one table per entity (Users, Orders, Products, etc.).
In DynamoDB, **a single table can hold multiple entity types**, identified by key patterns.

This is the **Single-Table Design**, optimized for scalability and performance.

How It Works

You use:

- **Partition Key** = entity owner (e.g., `UserID`)
- **Sort Key** = entity type + identifier (e.g., `ORDER#123`, `PROFILE#INFO`)
- Possibly an `EntityType` attribute for readability.

PK	SK	EntityType	Details
user#1	PROFILE#INFO	User	name, email
user#1	ORDER#123	Order	total, date
user#1	ORDER#456	Order	total, date
user#1	ADDRESS#HOME	Address	city, pin

Query Example

Get all data for a user:

```
aws dynamodb query \
  --table-name AppTable \
  --key-condition-expression "PK = :pk" \
  --expression-attribute-values '{":pk":{"S":"user#1"}}'
```

Result → user profile, all orders, addresses — all retrieved in **one query**.

Faster and cheaper than scanning multiple tables.

Benefits

- Fewer tables → fewer management overheads.
- Query multiple entity types in one call.
- Easier to use Global Secondary Indexes across data.
- Ideal for microservices with shared access patterns.

Challenges

- Harder to design upfront (requires known access patterns).
- Harder to visualize schema for new developers.
- Must handle entity identification using prefixes (`ORDER#`, `USER#`).

Best Practices

- Start from **access patterns**, not schema.
- Use **prefixes** to distinguish entities (`ORDER#123`).
- Use **GSI overloading** to reuse indexes.
- Document entity relationships clearly.

---

4 Common Anti-Patterns to Avoid

Anti-Pattern	Why It's a Problem	Correct Approach
1. Using Scan for Queries	Scans read entire table — slow and costly.	Use `Query` with key condition.
2. Low-cardinality Partition Key	Causes hot partitions and throttling.	Choose high-cardinality key (userID, requestID).
3. Overusing GSIs	Each GSI costs more & duplicates data.	Use only for required access patterns.
4. Treating DynamoDB like RDBMS	Joins, complex filters don't exist; causes inefficiency.	Model data by access patterns (denormalize).
5. Unbounded Item Growth	Item exceeding 400KB limit breaks writes.	Split large attributes or use S3 for blobs.
6. Ignoring Auto Scaling	Over-provision or throttle during spikes.	Enable Auto Scaling or use On-Demand mode.
7. Hardcoding credentials	Security risk.	Use IAM roles and AWS SDK credential chain.
8. No TTL for temp data	Wasted storage cost.	Enable TTL for sessions and caches.
9. Ignoring hot partitions in GSIs	Uneven query load.	Use random suffixes or composite keys.

Summary

Category	Best Practice
Security	Use fine-grained IAM policies and least privilege

Encryption	Always use KMS (customer-managed if compliance needed)
Networking	Use VPC endpoints for private access
Schema	Design for access patterns (single-table preferred)
Performance	Avoid scans, use proper keys and indexes
Operations	Enable Auto Scaling, CloudWatch metrics, backups
Compliance	Enable encryption, audit with CloudTrail

Backup, Recovery, and Data Integrity

- **Continuous Backup and Recovery (Restoration):**
  - Enabling **Point-in-Time Recovery (PITR)**.
  - Understanding the 35-day window and automatic incremental backups.
  - **Restoring** a table to a specific point in time.

- **On-Demand Backup:**
  - Creating and managing **manual full backups** for audit/compliance.
  - Restoring from a manual backup.
  - Using **AWS Backup** for centralized policy management

DynamoDB Backup, Recovery, and Data Integrity

Overview — Why Backup & Recovery Matter

DynamoDB is a **managed NoSQL service**, which means AWS handles:

- Hardware failures
- Data replication across multiple AZs
- Automatic durability within a region

However, **logical data loss** can still occur — for example:

- You accidentally delete an item/table.
- An application bug overwrites data.
- Someone pushes a destructive update.

That's where **DynamoDB Backup and Restore** features come in — to **recover data safely and precisely**.

---

2 Continuous Backup and Recovery (Point-in-Time Recovery — PITR)

What is PITR?

Point-in-Time Recovery (PITR) provides **continuous backups** of your DynamoDB table data.
You can restore your table to **any second in the last 35 days**.

Key Highlights

Feature	Description
Enabled per table	You must enable PITR separately for each table
Retention window	Up to **35 days** of continuous backup history
Granularity	Restore to **any second** within the retention period
Type	Incremental backups (efficient and automatic)
Storage	Managed by AWS — no maintenance needed
Impact	No performance or latency impact on your table

How PITR Works

When you enable PITR:

1. DynamoDB starts **recording every item change** (insert, update, delete).
2. It continuously maintains **incremental backups**.
3. You can restore your table to **any exact second** within the 35-day window.

This gives **protection from accidental writes or deletes**, unlike standard backups that run once.

---

3 Enabling Point-in-Time Recovery (PITR)

You can enable PITR from the **AWS Console**, **AWS CLI**, or **SDKs**.

Example: Enable PITR using AWS CLI

```
aws dynamodb update-continuous-backups \
  --table-name OrdersTable \
  --point-in-time-recovery-specification PointInTimeRecoveryEnabled=true
```

Check PITR Status

```
aws dynamodb describe-continuous-backups \
  --table-name OrdersTable
```

Sample Output:

```
{
  "ContinuousBackupsDescription": {
    "PointInTimeRecoveryDescription": {
      "PointInTimeRecoveryStatus": "ENABLED",
      "EarliestRestorableDateTime": "2025-10-09T08:00:00Z",
      "LatestRestorableDateTime": "2025-11-13T08:00:00Z"
    }
  }
}
```

This shows:

- PITR is enabled
- The **earliest** and **latest restorable timestamps**
- You can restore to **any second** between these two

---

4 Restoring a Table to a Specific Point in Time

Important:

- Restores are done to a **new table** (you cannot overwrite the existing table).
- You can rename or swap the restored table later.

Example: Restore a Table using AWS CLI

```
aws dynamodb restore-table-to-point-in-time \
  --source-table-name OrdersTable \
  --target-table-name OrdersTable_Restore_2025_11_13 \
  --use-latest-restorable-time
```

Or, specify an **exact time**:

```
aws dynamodb restore-table-to-point-in-time \
  --source-table-name OrdersTable \
  --target-table-name OrdersTable_Restore_2025_11_13 \
```

```
  --restore-date-time 2025-11-12T10:15:00Z
```

This creates a new table (`OrdersTable_Restore_2025_11_13`) containing data as of that timestamp.

Verify Restoration Status

```
aws dynamodb describe-table --table-name OrdersTable_Restore_2025_11_13
```

It will show the **status: CREATING → ACTIVE** once completed.

Notes:

- The restored table has the same settings (indexes, provisioned capacity, encryption).
- PITR backups are region-specific.
- You can **restore to the same region** (cross-region restore is not supported directly for PITR — for that, use **Global Tables**).

---

5 Understanding the 35-Day Retention and Incremental Backups

Aspect	Explanation
Retention Period	PITR stores data change history for **35 rolling days**. After that, older data is automatically purged.
Incremental Storage	DynamoDB only stores **changes** (not full copies). This minimizes cost.
Cost Model	Based on table size and retention — AWS manages backups transparently.
Consistency	Backups are **eventually consistent**, not strongly consistent — but perfect for restoration.

Example Cost (Ohio Region – `us-east-2`)

From AWS pricing (as of 2025):

Backup Type	Cost
PITR Storage	**$0.20 per GB-month** of storage
On-demand backup storage	**$0.10 per GB-month**
Restore	**$0.15 per GB** restored

Example Calculation:

If your table is 100 GB:

- PITR monthly storage ≈ `100 GB × $0.20 = $20/month`
- Restoring the full table once costs `100 GB × $0.15 = $15`

So total ≈ `$35/month` for full protection — very affordable for production-grade safety.

---

6 Comparing PITR vs On-Demand Backups

Feature	PITR (Continuous)	On-Demand Backup
Backup frequency	Continuous	Manual / scheduled
Retention	35 days	Until deleted
Restore granularity	Any second in 35-day window	Entire backup snapshot
Restore destination	New table	New table
Use case	Protection from recent changes / deletes	Long-term compliance / archiving
Cost	$0.20/GB-month	$0.10/GB-month

Best practice:
Use **both** —

- PITR for **operational recovery**
- On-demand backups for **compliance and archiving**

---

7 Data Integrity Considerations

- DynamoDB automatically replicates data across **3 AZs** within a region.
- Use **Streams** to capture and validate changes externally.
- Enable **CloudWatch metrics** for "`ConsumedRead/WriteCapacityUnits`" and "`ThrottledRequests`".
- Periodically validate data via **checksums** or external validation jobs.

Best Practices Summary

Area	Best Practice
Backup Setup	Enable PITR for all production tables
Retention	Monitor 35-day window and archive older data via on-demand backups
Recovery	Restore to a new table and swap application pointer
Automation	Use Lambda + EventBridge to automate on-demand backup scheduling
Cost Optimization	Delete unused backups and tables after testing restore
Testing	Regularly perform **backup-restore drills** to ensure data integrity
Security	Backup data inherits encryption (KMS) and IAM controls

---

Example: Automated Backup Workflow (Recommended)

1. **Enable PITR** on all prod tables.
2. Schedule a **weekly on-demand backup** via Lambda.
3. Store metadata in S3 or DynamoDB "BackupsMetadata" table.
4. Validate backup integrity monthly by restoring a random sample.
5. Monitor backup size & cost via **AWS Cost Explorer**.

- **Data Export and Import:**
  - Exporting table data to Amazon S3 for analytics or long-term retention.
  - Importing data from S3 to a new or existing table.

DynamoDB Data Export and Import

(with S3 Archival Strategy — Hot/Warm/Cold Storage Explained)

Why Export/Import?

DynamoDB is a **real-time NoSQL database** optimized for speed, not long-term archival or big data analytics.
Sometimes, you need to:

- Archive old data for **compliance or retention**
- Run **analytics using Athena, Glue, or Redshift**
- **Migrate data** between accounts, regions, or environments (e.g., dev → prod)

Amazon S3 is the natural destination — durable, scalable, and inexpensive.

---

2 Exporting DynamoDB Data to S3

Overview

DynamoDB supports **direct export to S3 without consuming read capacity**.
Exports happen **server-side**, meaning DynamoDB reads its internal backups (not your live table).

Key Features

Feature	Description
No performance impact	Export runs from DynamoDB backups, not your live table
Granular control	Export entire table or specific time (PITR)
Formats	JSON or DynamoDB JSON
Destination	S3 bucket in same or different region
Encryption	Supports SSE-S3, SSE-KMS, or SSE-C
Cost	Pay for data exported (~$0.10/GB) + S3 storage cost

Example 1: Export Using AWS Console

1. Go to **DynamoDB Console → Tables → Exports and streams → Export to S3**
2. Choose:
   - **Export from:** Latest table snapshot or Point-in-time recovery
   - **Destination S3 bucket:** e.g., `my-dynamodb-exports`
   - **Encryption:** Choose KMS key if needed
3. Click **Export**.

Example 2: AWS CLI Command

Export data as of now:

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable \
  --s3-bucket my-dynamodb-exports \
  --s3-prefix orders-2025-11-13 \
  --export-format DYNAMODB_JSON
```

Export data at a specific time:

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable \
  --s3-bucket my-dynamodb-exports \
  --restore-date-time 2025-11-12T09:30:00Z
```

DynamoDB creates a folder in S3:

```
s3://my-dynamodb-exports/orders-2025-11-13/AWSLogs/123456789012/...
```

Each item from DynamoDB is exported as a **JSON record** — ready for Athena, Redshift, or Glue.

Cost Example (Ohio Region)

Component	Approx. Cost
Export operation	$0.10 per GB exported
S3 Standard storage	$0.023 per GB per month
S3 Glacier storage (for archive)	$0.004 per GB per month

Example: 100 GB table →

- Export cost: `100 × $0.10 = $10`
- S3 Standard for 1 month: `100 × $0.023 = $2.30`

Total: **≈ $12.30 for 100 GB** export and storage for a month.

---

3 Using the Exported Data

You can directly query DynamoDB export data in S3 using **Athena**.

Example: Query with Athena

1. Create a Glue table with schema:

```
CREATE EXTERNAL TABLE dynamodb_orders (
  orderid string,
  userid string,
  total int,
  date string
)
STORED AS JSON
LOCATION 's3://my-dynamodb-exports/orders-2025-11-13/';
```

2. Run SQL like:

```
SELECT userid, SUM(total) AS total_spent
FROM dynamodb_orders
GROUP BY userid;
```

This allows **analytics on DynamoDB data** without ETL.

---

4 Importing Data from S3 into DynamoDB

Overview

You can import DynamoDB data **directly from S3** — no custom code required.

This is ideal for:

- Restoring archived exports
- Migrating data across regions/accounts
- Seeding new environments

Key Notes

- Supported formats: **DynamoDB JSON** (from earlier exports)
- The import creates a **new table**
- Automatically sets up table schema
- No read/write capacity consumed on existing tables

Example: Import Using AWS CLI

```
aws dynamodb import-table \
  --table-name OrdersTable_Restore \
  --s3-bucket-source S3Bucket=my-dynamodb-exports,S3KeyPrefix=orders-2025-11-13 \
  --input-format DYNAMODB_JSON
```

To check status:

```
aws dynamodb describe-import \
  --import-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable_Restore/import/abc123
```

DynamoDB creates a **new table**, populates it, and logs progress in CloudWatch.

---

5 Data Archival in Amazon S3 — HOT, WARM, COLD Storage

S3 supports **multiple storage tiers** — each optimized for a different data access frequency.

Let's map DynamoDB's exported data to the **S3 lifecycle** model.

S3 Storage Classes Overview

Tier	Typical Name	Use Case	Cost (per GB/month, Ohio)	Retrieval Cost
Hot	`S3 Standard`	Frequently accessed (daily/weekly)	~$0.023	None
Warm	`S3 Standard-IA` (Infrequent Access)	Access < once a month	~$0.0125	Low retrieval fee
Cold	`S3 Glacier Instant Retrieval`	Rarely accessed, need instant recovery	~$0.004	Small retrieval fee
Deep Cold	`S3 Glacier Deep Archive`	Long-term compliance (rarely read)	~$0.00099	High retrieval latency (~12h)

Example Lifecycle Policy for DynamoDB Export Bucket

You can automate moving data across these tiers.

Step 1: Create an S3 Lifecycle Policy (JSON)

```
{
  "Rules": [
    {
      "ID": "ArchiveDynamoDBExports",
      "Prefix": "orders-",
```

```
      "Status": "Enabled",
      "Transitions": [
        { "Days": 30, "StorageClass": "STANDARD_IA" },
        { "Days": 90, "StorageClass": "GLACIER" },
        { "Days": 365, "StorageClass": "DEEP_ARCHIVE" }
      ],
      "Expiration": { "Days": 1825 }
    }
  ]
}
```

Step 2: Apply it to the bucket:

```
aws s3api put-bucket-lifecycle-configuration \
  --bucket my-dynamodb-exports \
  --lifecycle-configuration file://lifecycle.json
```

Now your data automatically:

- stays in **Standard** for 30 days,
- moves to **Infrequent Access** at 30 days,
- to **Glacier** at 90 days,
- and **Deep Archive** at 1 year,
- deleted after 5 years (if needed).

Example Cost for Archival (100 GB data)

Tier	Cost/GB	Cost/Month
Standard (Hot, 30 days)	$0.023	$2.30
IA (Warm, 60 days)	$0.0125	$1.25
Glacier (Cold, 9 months)	$0.004	$0.40
Deep Archive (Years 2–5)	$0.00099	$0.099

→ Average annual cost ≈ **$5.50/year for 100 GB**, vs. $27.60 if kept all year in Standard!

---

6 Best Practices

Area	Best Practice
Export	Automate periodic exports using EventBridge or Lambda
Import	Keep exports in DynamoDB JSON for direct import compatibility
Security	Use KMS encryption for S3 buckets storing DynamoDB exports
Lifecycle	Define clear transition rules to optimize S3 cost
Analytics	Use Athena/Glue to query exports — no need to reimport into DynamoDB
Monitoring	Track export/import jobs in CloudWatch and DynamoDB console
Compliance	Use Glacier/Deep Archive for long-term retention and audits

---

Example Real-Life Use Case

Scenario:
A fintech company keeps 30 days of active transactions in DynamoDB for performance but must retain 7 years of data for compliance.

Solution:

1. Enable daily export of `Transactions` table to S3 (automated via EventBridge + Lambda).
2. Apply S3 lifecycle:
   - 30 days: Standard
   - 90 days: Standard-IA
   - 1 year: Glacier
   - 7 years: Deep Archive
3. Use Athena to run ad-hoc queries on S3 exports when required by auditors.

Result:

- Live DynamoDB stays lean and fast.
- Historical data remains accessible and cheap to store.

Summary Table

Operation	DynamoDB Feature	Storage Tier	AWS Service	Typical Use
Continuous Backup	PITR	Hot	DynamoDB internal	Short-term recovery
Export to S3	Export to S3	Hot/Warm/Cold	S3	Analytics / Archive
Long-term Archive	Lifecycle to Glacier	Cold/Deep Archive	S3 Glacier	Compliance retention
Data Restore	Import from S3	Hot	DynamoDB	Rehydrate or migrate data

Project

DynamoDB + S3 + Athena + Lifecycle + Import — Full Hands-on Lab

---

Lab Prerequisites

Make sure you have:

- An **AWS account** with permissions for:
  - `dynamodb:*`
  - `s3:*`
  - `glue:*`
  - `athena:*`
- AWS CLI installed and configured (`aws configure`)
- DynamoDB table with sample data (we'll use `OrdersTable`)

Step 1: Create a Sample DynamoDB Table

Create a simple Orders table.

```
aws dynamodb create-table \
  --table-name OrdersTable \
  --attribute-definitions \
      AttributeName=OrderId,AttributeType=S \
  --key-schema \
      AttributeName=OrderId,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST
```

Insert some sample data:

```
aws dynamodb put-item \
  --table-name OrdersTable \
  --item '{"OrderId": {"S": "ORD1001"}, "UserId": {"S": "U001"}, "Total": {"N": "450"}, "Date": {"S": "2025
```

```
aws dynamodb put-item \
  --table-name OrdersTable \
  --item '{"OrderId": {"S": "ORD1002"}, "UserId": {"S": "U002"}, "Total": {"N": "999"}, "Date": {"S": "2025
```

Step 2: Create an S3 Bucket for Export

```
aws s3 mb s3://my-dynamodb-exports-$(date +%Y%m%d)
```

Check bucket creation:

```
aws s3 ls
```

Step 3: Export DynamoDB Table to S3

```
aws dynamodb export-table-to-point-in-time \
  --table-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable \
  --s3-bucket my-dynamodb-exports-20251113 \
  --s3-prefix orders-export \
  --export-format DYNAMODB_JSON
```

Note: Replace `arn:aws:dynamodb:us-east-2:123456789012` with your account + region.

Check export status:

```
aws dynamodb list-exports
```

You'll see output like:

```
{
  "ExportSummaries": [
    {
      "ExportArn": "arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable/export/01693000000000-abc123"
      "ExportStatus": "IN_PROGRESS"
    }
  ]
}
```

Wait until the status is **COMPLETED**.

Step 4: Verify Data in S3

List exported files:

```
aws s3 ls s3://my-dynamodb-exports-20251113/orders-export/ --recursive
```

You'll see a structure like:

```
s3://my-dynamodb-exports-20251113/orders-export/AWSLogs/123456789012/OrdersTable/export-001/...
```

Each item is stored as a JSON file.

Step 5: Configure S3 Lifecycle for Archival

Create a lifecycle configuration file (`lifecycle.json`):

```
{
  "Rules": [
    {
      "ID": "ArchiveDynamoDBExports",
      "Prefix": "orders-export/",
      "Status": "Enabled",
      "Transitions": [
        { "Days": 30, "StorageClass": "STANDARD_IA" },
        { "Days": 90, "StorageClass": "GLACIER" },
        { "Days": 365, "StorageClass": "DEEP_ARCHIVE" }
      ],
      "Expiration": { "Days": 1825 }
    }
  ]
}
```

Apply the lifecycle rule to the bucket:

```
aws s3api put-bucket-lifecycle-configuration \
  --bucket my-dynamodb-exports-20251113 \
  --lifecycle-configuration file://lifecycle.json
```

Now, your data will automatically move:

- to **Infrequent Access** after 30 days
- to **Glacier** after 90 days
- to **Deep Archive** after 1 year
- deleted after 5 years

Step 6: Query Exported Data using Athena

Step 6.1: Configure Glue Table

Go to **AWS Glue Console → Crawlers → Add Crawler**

Or do it manually with Athena:

```
CREATE EXTERNAL TABLE dynamodb_orders (
  OrderId string,
  UserId string,
  Total int,
  Date string
)
STORED AS JSON
LOCATION 's3://my-dynamodb-exports-20251113/orders-export/';
```

Now you can run queries directly:

```
SELECT UserId, SUM(Total) AS TotalSpent
FROM dynamodb_orders
GROUP BY UserId;
```

You've just queried DynamoDB data **without reloading it into DynamoDB** — this is cost-efficient analytics.

Step 7: Import Data Back into DynamoDB

If you need to rehydrate the data:

```
aws dynamodb import-table \
  --table-name OrdersTable_Restore \
  --s3-bucket-source S3Bucket=my-dynamodb-exports-20251113,S3KeyPrefix=orders-export \
  --input-format DYNAMODB_JSON
```

Check status:

```
aws dynamodb describe-import \
  --import-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersTable_Restore/import/abc123
```

Once **ACTIVE**, you can start using `OrdersTable_Restore` as a new table.

Step 8: Validate and Cleanup

List all DynamoDB tables:

```
aws dynamodb list-tables
```

Check record count:

```
aws dynamodb scan --table-name OrdersTable_Restore --select COUNT
```

Delete older exports if not needed:

```
aws s3 rm s3://my-dynamodb-exports-20251113/orders-export/ --recursive
```

Summary Workflow

Step	Action	AWS Service
1	Create table & add data	DynamoDB
2	Export to S3	DynamoDB + S3
3	Apply Lifecycle Policy	S3
4	Query via Athena	Athena + Glue
5	Import data back	DynamoDB Import
6	Archive for long-term	S3 Glacier / Deep Archive

Security & Cost Tips

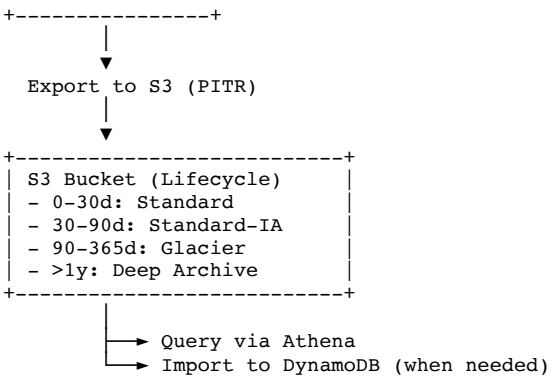
Aspect	Recommendation
Encryption	Enable KMS encryption for S3 bucket
Access Control	Use IAM roles instead of keys in CLI
Cost Optimization	Move to Glacier tiers for older data
Monitoring	Enable CloudWatch metrics for export/import jobs
Automation	Schedule exports using EventBridge and Lambda
Auditing	Use CloudTrail to log export/import activities

Example Real-World Cost (100 GB table in Ohio)

Component	Unit Cost	Approx. Monthly
Export	$0.10/GB	$10
S3 Standard (30 days)	$0.023/GB	$2.30
IA (next 60 days)	$0.0125/GB	$1.25
Glacier (next 9 months)	$0.004/GB	$0.40
Deep Archive (next 4 years)	$0.00099/GB	$0.099
Total Yearly (average)		**≈ $5–6 per 100 GB**

Diagram (Conceptual Flow)

```
+----------------+
| DynamoDB Table |
```



- **Change Data Capture (CDC):**
  - DynamoDB Streams: Enabling and configuring for **real-time event processing** (e.g., Lambda triggers).
  - Time-To-Live (TTL): Configuring automatic item expiration.

Change Data Capture (CDC) in **DynamoDB**, which is a key concept for real-time architectures, event-driven systems, and lifecycle automation.

We'll cover:

1. What CDC means in DynamoDB
2. DynamoDB Streams — setup, architecture, and Lambda integration
3. Time-To-Live (TTL) — automatic item expiry and cleanup
4. Best practices and real-world use cases

Change Data Capture (CDC) in DynamoDB

Change Data Capture (CDC) is a mechanism that detects and captures changes (inserts, updates, deletes) in your database in real time — enabling **event-driven processing** or **data replication**.

In DynamoDB, **CDC is implemented using DynamoDB Streams**.

---

1. DynamoDB Streams — Overview

What is a DynamoDB Stream?

A **DynamoDB Stream** captures changes to items in a table as a sequence of time-ordered events.

Whenever a table item is created, updated, or deleted, **a corresponding record** is added to the stream.

You can use these stream records to:

- Trigger **AWS Lambda** functions
- Feed data into **Kinesis Data Streams**, **SQS**, or **EventBridge**
- Replicate data to **Elasticsearch**, **Redshift**, or **another DynamoDB table**

Stream Record Types

Each record in the stream represents one of these operations:

Operation	Description
`INSERT`	A new item is added

`MODIFY`	An existing item is updated
`REMOVE`	An item is deleted

Stream Views (Configurations)

When enabling Streams, you choose what kind of data to capture:

StreamViewType	Captured Data
`KEYS_ONLY`	Only primary key attributes
`NEW_IMAGE`	The new state of the item after modification
`OLD_IMAGE`	The state of the item before modification
`NEW_AND_OLD_IMAGES`	Both before and after images

Enabling DynamoDB Streams

Let's create a table and enable a stream:

```
aws dynamodb create-table \
  --table-name OrdersCDC \
  --attribute-definitions AttributeName=OrderId,AttributeType=S \
  --key-schema AttributeName=OrderId,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Confirm the stream is active:

```
aws dynamodb describe-table --table-name OrdersCDC --query "Table.LatestStreamArn"
```

This ARN is crucial — it identifies your stream.

How DynamoDB Streams Work

1. You modify data in the table (Put, Update, or Delete)
2. The change is written to the **stream**
3. AWS keeps these records for **24 hours**
4. Consumers (like Lambda) can read the stream and process changes in real-time

---

Setting Up a Lambda Trigger for Real-Time Processing

Step 1: Create a Lambda Function

Example: process order changes

```
import json

def lambda_handler(event, context):
    for record in event['Records']:
        event_name = record['eventName']
        print(f"Detected {event_name} event")
        if event_name == 'INSERT':
            new_item = record['dynamodb']['NewImage']
            print("New order:", new_item)
        elif event_name == 'REMOVE':
            old_item = record['dynamodb']['OldImage']
            print("Deleted order:", old_item)
    return {'statusCode': 200}
```

Deploy it in AWS Lambda console or via CLI.

Step 2: Attach Lambda to the DynamoDB Stream

```
aws lambda create-event-source-mapping \
  --function-name ProcessOrderChanges \
  --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/OrdersCDC/stream/2025-11-13T09:00:00.000
  --starting-position LATEST
```

Now, whenever an item is added, updated, or deleted in `OrdersCDC`, the Lambda runs instantly.

Example Stream Event (INSERT)

Here's what a record looks like in the Lambda input event:

```
{
  "Records": [
```

```
    {
      "eventID": "1",
      "eventName": "INSERT",
      "dynamodb": {
        "Keys": { "OrderId": { "S": "ORD1001" } },
        "NewImage": {
          "OrderId": { "S": "ORD1001" },
          "UserId": { "S": "U001" },
          "Total": { "N": "450" }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "eventSource": "aws:dynamodb"
    }
  ]
}
```

⏳ 2. Time-To-Live (TTL)

Time-To-Live (TTL) lets you define an attribute (e.g., `ExpiresAt`) that determines when an item should be automatically deleted from your DynamoDB table.

TTL is perfect for:

- **Session data** (auto-expire user sessions)
- **Cache tables**
- **Temporary orders or tokens**
- **Soft-deleted items cleanup**

How TTL Works

- You choose an attribute name (e.g., `expiresAt`)
- Its value must be a **UNIX timestamp (epoch time in seconds)**
- DynamoDB deletes the item **within ~48 hours** after the timestamp passes
- Deletion is **non-chargeable** and **non-blocking**

Example: Enable TTL on a Table

```
aws dynamodb update-time-to-live \
  --table-name OrdersCDC \
  --time-to-live-specification "Enabled=true, AttributeName=expiresAt"
```

Verify TTL:

```
aws dynamodb describe-time-to-live --table-name OrdersCDC
```

Insert Item with Expiry

```
# Expires in 1 hour
EXPIRY=$(($(date +%s) + 3600))

aws dynamodb put-item \
  --table-name OrdersCDC \
  --item '{"OrderId":{"S":"ORD2001"}, "UserId":{"S":"U002"}, "Total":{"N":"999"}, "expiresAt":{"N":"'"$EXPI
```

DynamoDB will automatically delete this item after 1 hour.

TTL + Streams = Real-Time Cleanup Notifications

If Streams are enabled on the same table, you can capture TTL-deleted records in the stream.
This allows downstream systems (like S3, Elasticsearch, or audit logs) to react to deletions.

For example, you can:

- Trigger a **Lambda** when TTL deletes an item.
- Archive it to **S3** for audit compliance.

Use Cases for CDC (Streams + TTL)

Use Case	Description
Real-time analytics	Capture writes and send to Kinesis or Redshift
Event-driven apps	Trigger Lambdas on inserts/updates
Audit logs	Capture all CRUD events for compliance
Cross-region replication	Sync tables using Streams + Lambda
Session cleanup	Use TTL to expire temporary user sessions
Cache invalidation	Auto-remove stale cache entries using TTL

Best Practices

Area	Recommendation
Stream Retention	DynamoDB Streams retain data for 24 hours — process events quickly
Error Handling	Configure Lambda DLQ (SQS) for failed events
Idempotency	Make downstream consumers idempotent (avoid duplicate processing)
Monitoring	Use CloudWatch metrics like `IteratorAgeMilliseconds`
Cost Optimization	Turn off Streams when not in use
TTL Attribute Format	Always use epoch seconds (integer), not milliseconds

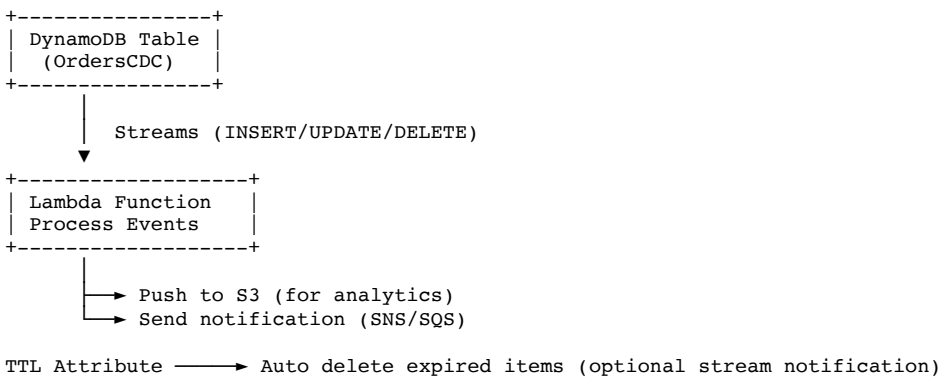
---

Summary

Feature	Purpose	Trigger Target	Key Benefit
DynamoDB Streams	Captures inserts, updates, deletes	Lambda, Kinesis, SQS	Enables real-time event-driven systems
TTL (Time-to-Live)	Automatically expires items	(Optional) Stream event	Simplifies cleanup, reduces costs

---

Example Architecture



High Availability and Cross-Region Replication

- **Database Replication (Global Tables):**
  - Introduction to **Global Tables (Multi-Region Replication)**.
  - Setting up and managing replica tables in different AWS Regions.

DynamoDB High Availability and Cross-Region Replication

Global Tables Deep Dive

---

1. What Are Global Tables?

A **DynamoDB Global Table** is a **multi-region, multi-master** setup that automatically replicates data between regions.

That means:

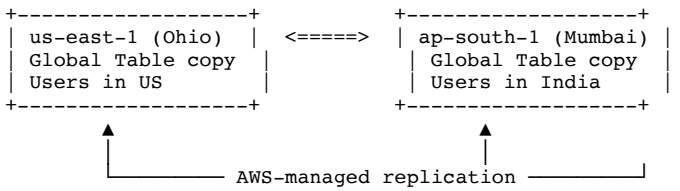
- You can **read and write** to the table in *any* region.
- DynamoDB **replicates** changes to all other participating regions.
- All regions hold **identical copies** of your data.

Benefits

Feature	Description
High Availability (HA)	If one region fails, others continue serving traffic
Low Latency Access	Users read/write to the region closest to them
Disaster Recovery (DR)	Built-in cross-region data redundancy
Automatic Conflict Resolution	Last writer wins (based on timestamps)
No Custom ETL or Streams Needed	DynamoDB manages replication transparently

---

How It Works (Architecture)



Each region hosts a full replica of the table and synchronizes changes asynchronously.

---

2. Global Tables Architecture Versions

Version	Description
Version 2017.11.29	Legacy version — manual regional table setup
Version 2019.11.21	Current version — easier, automatic management
Version 2023.11.30	Newest — supports import/export, on-demand capacity, and more flexible replication management

Always use the **latest version (2023.11.30)**.

---

3. Setting Up a Global Table (CLI or Console)

We'll create a **multi-region DynamoDB Global Table** between **Ohio (us-east-2)** and **Mumbai (ap-south-1)**.

Step 1: Create the Base Table

In the **primary region** (Ohio):

```
aws dynamodb create-table \
  --table-name OrdersGlobal \
  --attribute-definitions AttributeName=OrderId,AttributeType=S \
  --key-schema AttributeName=OrderId,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --region us-east-2
```

Step 2: Enable Global Table Replication

Add a **replica region** (Mumbai):

```
aws dynamodb update-table \
  --table-name OrdersGlobal \
  --replica-updates "[{\"Create\":{\"RegionName\": \"ap-south-1\"}}]" \
  --region us-east-2
```

This automatically creates a replica table in `ap-south-1` with the same schema and replication setup.

Step 3: Verify Replication

Check replication status:

```
aws dynamodb describe-table \
  --table-name OrdersGlobal \
  --region us-east-2 \
  --query "Table.Replicas"
```

You should see output like:

```
[
  {
    "RegionName": "us-east-2",
    "ReplicaStatus": "ACTIVE"
  },
  {
    "RegionName": "ap-south-1",
    "ReplicaStatus": "ACTIVE"
  }
]
```

---

4. How Replication Works Internally

1. When a write happens in **any region**, DynamoDB adds it to a replication log.
2. The change is asynchronously propagated to all other regions.
3. If multiple writes occur to the same item in different regions, DynamoDB applies **last-writer-wins** logic based on timestamps.

Conflict Resolution Example

If:

- Region A updates `OrderId=1001` at `T1`
- Region B updates same record at `T2`

Then, DynamoDB keeps the version with **later timestamp (T2)**.

---

5. Example: Testing Global Replication

Add an item in **Ohio**:

```
aws dynamodb put-item \
  --table-name OrdersGlobal \
  --item '{"OrderId":{"S":"ORD5001"},"UserId":{"S":"U100"},"Total":{"N":"450"}}' \
  --region us-east-2
```

Read the same item from **Mumbai**:

```
aws dynamodb get-item \
  --table-name OrdersGlobal \
  --key '{"OrderId":{"S":"ORD5001"}}' \
  --region ap-south-1
```

You'll see the replicated record appear within a few seconds.

---

6. Managing Global Tables

➕ Add a new replica region

```
aws dynamodb update-table \
  --table-name OrdersGlobal \
  --replica-updates "[{\"Create\":{\"RegionName\": \"eu-central-1\"}}]" \
  --region us-east-2
```

Remove a replica

```
aws dynamodb update-table \
  --table-name OrdersGlobal \
  --replica-updates "[{\"Delete\":{\"RegionName\": \"ap-south-1\"}}]" \
  --region us-east-2
```

---

7. Best Practices for Global Tables

Category	Best Practice
Schema Design	All replicas must share identical schema, GSI, and capacity settings
Conflict Handling	Application logic should tolerate eventual consistency
Write Operations	Use **unique item IDs** to minimize conflicts

Monitoring	Track `ReplicationLatency` and `ConflictCount` metrics in CloudWatch
Backup	Each region can have its own Point-in-Time Recovery (PITR)
IAM Roles	Ensure cross-region replication IAM permissions are configured properly
Network Latency	Keep write-heavy workloads localized per region if possible
DR Strategy	Test failover by redirecting traffic via Route 53 to secondary region

---

8. Common Use Cases

Use Case	Description
Global Applications	Users across continents read/write to the nearest region
Disaster Recovery	If one region fails, traffic automatically reroutes to another
Data Sovereignty	Keep data replicated across countries for compliance
Edge Caching	Use regional copies to reduce read latency
Cross-region Analytics	Run analytics in one region without impacting primary workloads

---

9. Example: Cost (Ohio + Mumbai)

Component	Cost Type	Example (100 GB, 1M writes/day)
DynamoDB storage	$0.25/GB	$25/month
Write units	$1.25/million writes	$37.50/month
Replication cost	$0.25/million replicated writes	$7.50/month
Total (2 regions)		≈ $70–75/month

> Replication doubles your write cost (since each write is duplicated in all regions). Reads are billed per region.

---

10. Monitoring and Maintenance

Use these tools for visibility:

- **CloudWatch Metrics**
  - `ReplicationLatency`
  - `WriteThrottleEvents`
  - `SystemErrors`
- **AWS CloudTrail** — for API activity logging
- **AWS Backup / PITR** — for recovery

---

11. High Availability Summary

Feature	Mechanism	Benefit
HA within a Region	DynamoDB automatically replicates data across 3 AZs	Fault tolerance within region
Cross-Region HA	Global Tables	DR and multi-region access
Replication Mode	Multi-active	All regions can read/write
Conflict Resolution	Last-writer-wins	Simple eventual consistency
Disaster Recovery	Region-level failover	Seamless continuity

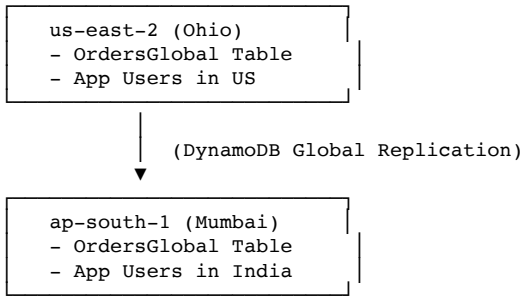
---

12. Example IAM Policy for Global Table Management

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:CreateTable",
        "dynamodb:UpdateTable",
        "dynamodb:DescribeTable",
        "dynamodb:ListTables"
      ],
      "Resource": "arn:aws:dynamodb:*:123456789012:table/OrdersGlobal"
    }
  ]
}
```

Attach this policy to the IAM role or user managing the Global Table.

Final Architecture Summary



Reads/Writes allowed in any region
Data replicated automatically
No downtime failover needed

- **Consistency Models:**
  - Understanding **Eventual Consistency** (default for replication).
  - Understanding **Strong Consistency** and its trade-offs.
  - Multi-Region Strong Consistency (MRSC) mode.

DynamoDB Consistency Models

When you read data from DynamoDB, you can choose how "fresh" that data must be — whether you're okay with a **slightly stale read** (eventual consistency) or you require **the latest committed data** (strong consistency).

This choice impacts:

- Performance (latency)
- Cost (read units)
- Global replication behavior (multi-region consistency)

---

1. Eventual Consistency (Default)

Definition

Eventual consistency means:

> When you write an item, DynamoDB **replicates** it to all storage nodes **asynchronously**.
> A read immediately after a write might not return the latest value, but **will eventually** once replication completes.

How It Works Internally

1. You issue a `PutItem` or `UpdateItem`.
2. DynamoDB writes to the **primary node** and returns success once acknowledged.
3. The data is then replicated across 3 Availability Zones in the region **eventually**.
4. If you perform a read immediately after the write, one of the replica nodes may not have the update yet.

Characteristics

Feature	Eventual Consistency
Default mode	Yes
Replication delay	Typically < 10ms
Latency	Lowest
Throughput cost	1 Read Capacity Unit (RCU) per 4KB
Use Case	When slightly stale data is acceptable (analytics, dashboards, caching)

Example

You `PutItem` → then immediately `GetItem`.

```
aws dynamodb get-item \
  --table-name Orders \
  --key '{"OrderId": {"S": "ORD1001"}}'
```

This may return the **old value** if replication hasn't finished yet.

But after a few milliseconds, all replicas converge to the same state.

Use Cases

- Product catalog
- Social feeds
- IoT sensor aggregation
- Reporting dashboards
- Large-scale analytics workloads

---

2. Strong Consistency

Definition

A **strongly consistent read** always returns the most recent committed value of the item — ensuring **read-after-write consistency**.

When you perform a read:

> DynamoDB waits until all replicas confirm that the write has been committed before returning data.

How It Works Internally

1. Write is sent to the leader node.
2. Leader ensures **replication to quorum** (usually 2/3 AZs).
3. When a client requests a **strongly consistent read**, DynamoDB only returns data from **up-to-date nodes**.

Characteristics

Feature	Strong Consistency
Default	(must be requested explicitly)
Replication delay	0 (always latest data)
Latency	Slightly higher
Throughput cost	1 RCU per 4KB (same as eventual, but fewer concurrent reads)
Read Availability	Only within a single region
Use Case	When you need up-to-date data (financial, transactional systems)

How to Request Strongly Consistent Reads (CLI Example)

```
aws dynamodb get-item \
  --table-name Orders \
  --key '{"OrderId": {"S": "ORD1001"}}' \
  --consistent-read
```

or in SDKs:

```
table.get_item(Key={'OrderId': 'ORD1001'}, ConsistentRead=True)
```

Use Cases

- Banking transactions
- Inventory deduction systems
- Payment gateways
- Session state management
- Order placement systems

---

3. Eventual vs Strong Consistency – Side-by-Side

Aspect	Eventual	Strong
Freshness	May be stale	Always up to date
Read latency	Lower	Slightly higher
Availability	Higher (can read from any replica)	Lower (must read from leader/quorum)
Throughput	Higher	Slightly lower
Cost	Lower	Slightly higher
Region scope	Supported globally	Single-region only
Recommended when	Data freshness not critical	Data correctness critical

---

4. Multi-Region Strong Consistency (MRSC)

Introduction

Previously, **strongly consistent reads** worked **only within a single region**.
With **Multi-Region Strong Consistency (MRSC)**, DynamoDB now supports *globally consistent reads and writes* across regions using **Global Tables**.

What It Does

- Enables **strong consistency across multiple AWS Regions**.
- A read from **any region** reflects the **most recent write** from any region.
- Uses advanced quorum-based consensus across replica regions.

How to Enable MRSC

When creating or updating a Global Table:

```
aws dynamodb create-table \
  --table-name OrdersGlobalStrong \
  --attribute-definitions AttributeName=OrderId,AttributeType=S \
  --key-schema AttributeName=OrderId,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST \
  --replication-group RegionName=us-east-2 RegionName=ap-south-1 \
  --replication-global-secondary-indexes \
      "[{\"IndexName\":\"UserIndex\",\"KeySchema\":[{\"AttributeName\":\"UserId\",\"KeyType\":\"HASH\"}],\"
  --global-table-replication-mode MULTI_REGION_STRONG
```

Or in the AWS Console:

> While creating a **Global Table**, select **"Multi-Region Strong Consistency"** option.

MRSC Characteristics

Feature	Multi-Region Strong Consistency
Consistency	Strong across all regions
Conflict resolution	Strongly consistent quorum (no "last-writer-wins")
Latency	Slightly higher (due to cross-region consensus)
Availability	Designed for 99.999%
Use Case	Mission-critical systems that require global correctness

---

Internal Mechanics

- DynamoDB synchronizes data using **consensus replication** (similar to Paxos/Raft).
- When a write happens in one region, it's committed only after **acknowledgment from quorum regions**.
- Ensures **no stale reads** across global replicas.

Use Cases for MRSC

- **Global Banking Apps** – synchronized balances across continents
- **Real-Time Inventory Systems** – global stock updates in sync
- **Multi-Region APIs** – consistent reads/writes worldwide
- **Critical IoT systems** – sensor state synchronization

MRSC Trade-offs

Pros	Cons
Global correctness	Higher write latency
Simplified global logic	Slightly increased replication cost
Zero conflict resolution needed	Requires multiple participating regions
Built-in durability	Requires premium pricing

> MRSC is optional — use it only if your app truly needs **strong global correctness**.
> For most apps, eventual consistency is cost-effective and sufficiently reliable.

Example Timing Comparison

Operation	Eventual	Strong	MRSC
Read latency (avg)	1–5 ms	3–10 ms	15–40 ms (cross-region)
Write latency (avg)	5–10 ms	10–15 ms	40–100 ms
Read accuracy	May lag	Always fresh (local)	Always fresh (global)

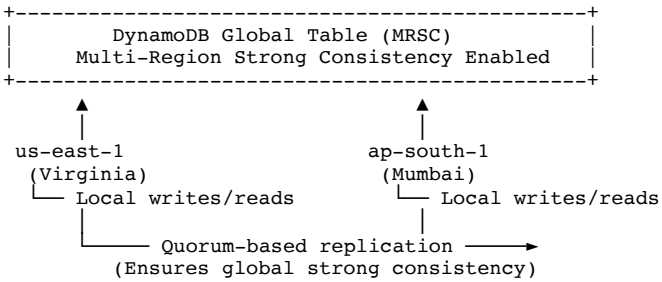
---

5. Choosing the Right Consistency Model

Application Type	Recommended Model
E-commerce product listing	Eventual
Shopping cart checkout	Strong
Global banking or wallet	MRSC
IoT analytics (sensor logs)	Eventual
Real-time inventory sync across regions	MRSC
Internal admin dashboards	Eventual or Strong (single region)

---

6. Example Architecture Diagram



Summary

Model	Scope	Latency	Read Accuracy	Use Case
Eventual Consistency	Global	Lowest	May be stale	High-volume, latency-sensitive reads
Strong Consistency	Single region	Medium	Always latest	Transactional systems
MRSC (Multi-Region Strong Consistency)	Multi-region	Highest	Always latest globally	Mission-critical, globally synchronized apps

---

Key Takeaways

- DynamoDB gives **consistency control per request** — you choose based on your SLA.
- **Strong reads** guarantee correctness but add latency.
- **Eventual reads** maximize performance and cost efficiency.
- **MRSC** brings **true global strong consistency**, but at a higher latency/cost.

- **Disaster Recovery and Failover:**
  - Designing the application layer for multi-region access (e.g., using Route 53).
  - Handling conflict resolution ("Last Writer Wins").

DynamoDB Disaster Recovery and Failover

DynamoDB is a **fully managed**, **multi-AZ**, and **serverless NoSQL database**, meaning it already provides **high availability** *within a single AWS Region*.
But for true **Disaster Recovery (DR)** — you must plan for **multi-region availability**.

Let's explore how that works.

---

1. Built-in High Availability (within a Region)

Even without any manual setup:

- DynamoDB automatically replicates data **synchronously** across **3 Availability Zones (AZs)** within the same region.
- This protects against:
  - Hardware failures
  - AZ outages
  - Disk corruption

You already have **durability** and **fault tolerance** inside one region.

But this doesn't protect against **Regional outages** (e.g., us-east-1 entire region down).
That's where **Disaster Recovery (DR)** and **Global Tables** come in.

---

2. Multi-Region Replication with Global Tables

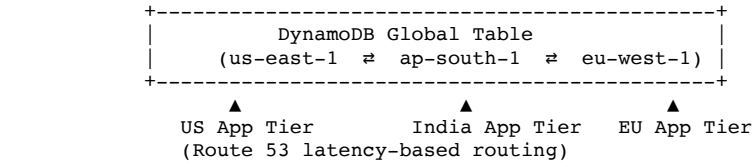
DynamoDB **Global Tables** are the core of its **Disaster Recovery strategy**.

What are Global Tables?

They are **fully managed multi-region, multi-master** DynamoDB tables that:

- Replicate data automatically between AWS regions.
- Allow **read and write operations** in all participating regions.
- Provide **eventual or strong global consistency** depending on configuration.

Example Architecture



Each region has:

- A **local DynamoDB replica**
- **Independent application stack** (EC2/ECS/Lambda)
- Shared DNS routing via **Route 53**

If one region goes down → traffic automatically shifts to another.

---

3. Designing Application Layer for Multi-Region Access

To achieve **seamless DR**, your **application** must be multi-region aware.

Components Involved

Component	Role

DynamoDB Global Table	Data replication
Amazon Route 53	DNS-based routing and health checks
Application Load Balancer / API Gateway	Front-end for each region
AWS Lambda / ECS / EC2	Application compute
CloudWatch / EventBridge	Failover monitoring and alarms

Step-by-Step: Multi-Region Setup Example

Step 1 — Create Global Table

Replicate between `us-east-1` and `ap-south-1`.

```
aws dynamodb create-global-table \
  --global-table-name Orders \
  --replication-group RegionName=us-east-1 RegionName=ap-south-1
```

Step 2 — Deploy Application Stack in Both Regions

You can use:

- **CloudFormation StackSets** or **Terraform**
- Identical Lambda / ECS services in both regions
- Both read/write to their local replica of the Global Table

Step 3 — Configure Route 53

Use **Latency-based routing** or **Health Check Failover Policy**.

Example:

- Primary endpoint → API in `us-east-1`
- Secondary endpoint → API in `ap-south-1`
- Route 53 monitors the health of the primary region's endpoint
- On failure, automatically reroutes traffic to secondary region

---

Step 4 — Handle Data Writes and Conflicts

Now both regions can accept writes simultaneously.
This introduces **conflict scenarios** if the same item is written in both regions before replication completes.

That's where **conflict resolution** comes in.

---

4. Conflict Resolution – "Last Writer Wins"

When DynamoDB Global Tables detect **concurrent updates** to the same item (same primary key) in multiple regions:

> The version with the **latest timestamp (Last Writer Wins)** takes precedence.

How It Works

- Each item has an **internal `LastUpdatedTime` metadata field**.
- DynamoDB uses this to determine which version is newer.
- The newer version overwrites the older one during replication.

Example Scenario

Time	Region	Operation	Result
10:00	`us-east-1`	Update Item `Order123` → `Status=Shipped`	Saved
10:00	`ap-south-1`	Update Item `Order123` → `Status=Delivered`	Saved
10:00:02	Sync occurs	Conflict detected → `Delivered` (latest timestamp) wins	

Final item across all replicas = **Delivered**

Limitations of "Last Writer Wins"

Concern	Explanation
No merge logic	Only one version kept; the other is overwritten
Timestamp sensitivity	Relies on clock synchronization (via NTP)
Possible lost updates	Simultaneous writes may lose intermediate states

> Tip: To handle merges manually, include a **version or revision ID** in your item schema and use **conditional writes** (`ConditionExpression`) to prevent overwriting stale data.

---

5. Conditional Writes to Prevent Conflicts

You can design your app to reject writes if a more recent version already exists:

```
response = table.put_item(
    Item={"OrderId": "123", "Status": "Delivered", "Version": 3},
    ConditionExpression="attribute_not_exists(Version) OR Version < :v",
    ExpressionAttributeValues={":v": 3}
)
```

Ensures no overwrite of newer versions
Write fails if concurrent update already applied

---

6. Testing Failover Scenarios

You should simulate failure to validate your setup:

Test	Expected Behavior
Primary region outage	Route 53 reroutes to secondary
Write in secondary	DynamoDB replicates automatically after primary recovers
Conflict write	Last Writer Wins applied
Network partition	Writes buffered; synced on reconnection

Use AWS **Fault Injection Simulator (FIS)** to test real failover events.

---

7. Recovery Strategies

Strategy	Description	Recovery Time Objective (RTO)	Recovery Point Objective (RPO)
Backup & Restore	Manual restore from snapshot	Hours	Minutes–hours
Global Table Active-Active	Multi-region replication	Seconds	Near-zero
Pilot Light	Minimal standby infrastructure in secondary region	Minutes	<1 minute
Warm Standby	Running secondary system with reduced capacity	Minutes	<1 minute

> **Recommended:** Use **Global Tables + Route 53** for **active-active DR**.

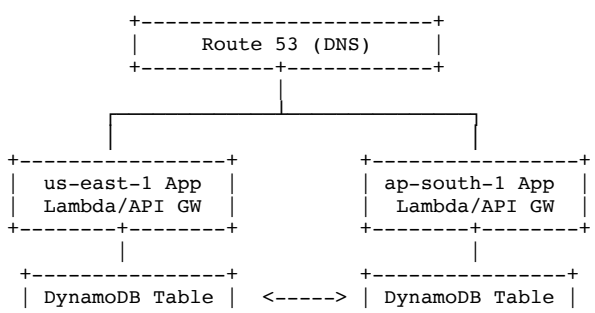
---

8. DR and Failover Summary

Layer	Service	Purpose
Data Layer	DynamoDB Global Tables	Multi-region replication
App Layer	ECS / Lambda / API Gateway	Region-local compute
Network Layer	Route 53	Global DNS failover
Monitoring	CloudWatch, EventBridge	Detect regional failures
Conflict Resolution	DynamoDB “Last Writer Wins”	Automatic version reconciliation

---

9. Example Multi-Region DR Diagram



```
| (Global Table) |          | (Replica)      |
+----------------+          +----------------+
          ⇄  Asynchronous Replication
```

Automatic read/write in both regions
Seamless DNS-based failover
Automatic data reconciliation via timestamps

---

10. Best Practices for DynamoDB DR Design

Best Practice	Description
Use Global Tables v2	Simplified setup, better performance
Enable PITR (Point-in-Time Recovery)	Protect against accidental deletes
Use Conditional Writes	Prevent conflict overwrites
Implement Route 53 health checks	Automatic failover
Simulate DR events	Validate architecture regularly
CloudWatch alarms	Detect replication lag or throttling

Summary

Concept	Description
Disaster Recovery (DR)	Ensures business continuity during region outages
Failover	Automatic rerouting of application traffic
Conflict Resolution	“Last Writer Wins” via timestamp
Recommended Design	Global Tables + Route 53 + Multi-region app stack
Optional Enhancements	Conditional writes + PITR + FIS testing

---

Monitoring, Tuning, and Cost Management

- **Performance Monitoring:**
  - Using **CloudWatch** metrics (Read/Write Capacity Unit utilization).
  - Monitoring latency, throttling, and errors.

Monitoring, Tuning, and Cost Management in DynamoDB

Part 1 — Performance Monitoring using CloudWatch

Why Monitoring DynamoDB Matters

DynamoDB is fully managed — meaning AWS automatically scales, patches, and maintains the infrastructure.
However, **you are still responsible** for:

- Provisioning the right capacity (if not using On-Demand)
- Avoiding throttling
- Optimizing read/write usage
- Understanding cost patterns

This is achieved via **Amazon CloudWatch**, which continuously collects DynamoDB performance metrics.

---

1. CloudWatch Integration Overview

DynamoDB automatically publishes key metrics to **Amazon CloudWatch** at **1-minute granularity**.

You can visualize these in:

- AWS Console → *DynamoDB* → *Metrics tab*

- AWS CloudWatch Console → *Metrics* → *DynamoDB namespace*
- AWS CLI or SDK

Each table (and global secondary index) has its own set of metrics.

---

2. Key CloudWatch Metrics for DynamoDB

Let's break down the **most important metrics** one by one.

---

a. ConsumedReadCapacityUnits

- Measures how many **Read Capacity Units (RCUs)** are being used per second.
- Each **RCU** = one strongly consistent read per 4KB item per second
  or two eventually consistent reads per 4KB per second.

Why it matters:

- If `ConsumedReadCapacityUnits` approaches `ProvisionedReadCapacityUnits`, your app is reading at its limit.
- If it frequently exceeds the provisioned limit → you'll see **throttling**.

Action: Scale up RCUs or enable **Auto Scaling**.

---

b. ConsumedWriteCapacityUnits

- Tracks how many **Write Capacity Units (WCUs)** are being used per second.
- Each **WCU** = one write per 1KB item per second.

Why it matters:

- High utilization = your app is close to its write throughput limit.
- Can cause throttling or latency spikes.

Action: Increase WCUs or move to **On-Demand** mode if spikes are unpredictable.

---

c. ProvisionedReadCapacityUnits / ProvisionedWriteCapacityUnits

- Shows the **configured capacity** for the table or index.
- Compare this with consumed metrics to determine **over-provisioning** or **under-provisioning**.

Action:
If provisioned >> consumed → reduce capacity (save cost).
If provisioned << consumed → scale up (avoid throttling).

---

d. ThrottledRequests

- Counts the number of read/write requests that were **rejected** due to exceeding capacity limits.
- Each throttled request results in an HTTP `ProvisionedThroughputExceededException`.

Why it matters:

- Throttling impacts latency and application reliability.
- Frequent throttling = under-provisioned table or inefficient access pattern (hot partition).

Action:

- Enable **Auto Scaling**.
- Review access pattern to ensure keys are well distributed.
- Consider using **DAX** for read-heavy workloads.

---

e. ReadThrottleEvents / WriteThrottleEvents

- Breaks down throttling by read and write operations.

Action:
Helps pinpoint whether throttling happens during reads, writes, or both.

---

f. SystemErrors (5XX)

- Number of internal service errors (rare for DynamoDB).
- Usually due to transient issues.

Action:
AWS SDK retries automatically handle this; monitor if rate increases.

---

g. UserErrors (4XX)

- Client-side errors like:
  - Invalid parameters
  - Access denied (IAM policy)
  - Conditional check failed

Action:
Review logs or enable **CloudWatch Logs Insights** for troubleshooting.

---

h. SuccessfulRequestLatency

- Measures **average latency** for successful requests.
- Available for reads, writes, queries, and scans.

Action:

- Sudden increase → check for throttling, network issues, or DAX cache miss.

---

i. ReturnedItemCount

- For `Query` and `Scan` operations, shows how many items are being returned.

Action:
Monitor this to understand **query efficiency** — large scans may indicate a poor key design.

---

3. Sample Dashboard Setup

You can create a **CloudWatch Dashboard** for DynamoDB like this:

Metric	Widget Type	Threshold
ConsumedReadCapacityUnits	Line chart	80% of provisioned
ConsumedWriteCapacityUnits	Line chart	80% of provisioned
ThrottledRequests	Line chart	>0 alert
ReadLatency	Number	>50 ms alert
WriteLatency	Number	>20 ms alert
SystemErrors (5xx)	Number	>0 alert

Example: Create Alarm for Throttling (CLI)

```
aws cloudwatch put-metric-alarm \
  --alarm-name "DynamoDBThrottlingAlert" \
  --metric-name ThrottledRequests \
  --namespace AWS/DynamoDB \
  --statistic Sum \
  --period 60 \
  --threshold 1 \
  --comparison-operator GreaterThanThreshold \
  --evaluation-periods 1 \
  --alarm-actions arn:aws:sns:us-east-1:123456789012:NotifyOps
```

This sends an SNS notification if throttling happens.

---

4. Tuning and Optimization Techniques

Monitoring is only the first step — here's how you tune based on metrics:

---

a. Auto Scaling

Automatically adjusts RCU/WCU based on usage trends.

```
aws application-autoscaling register-scalable-target \
  --service-namespace dynamodb \
  --resource-id table/Orders \
  --scalable-dimension dynamodb:table:ReadCapacityUnits \
  --min-capacity 5 \
  --max-capacity 100
```

> Target utilization (e.g., 70%) ensures capacity matches demand without overpaying.

---

b. Use On-Demand Mode for Variable Traffic

If traffic is unpredictable:

```
aws dynamodb update-table \
  --table-name Orders \
  --billing-mode PAY_PER_REQUEST
```

Automatically scales read/write throughput
Pay only for what you use
Slightly higher cost at constant, predictable workloads

---

c. Optimize Hot Partitions

If a few partition keys dominate traffic:

- Redistribute keys (e.g., add random suffix)
- Use **write sharding**
- Pre-compute hot keys and cache via DAX or ElastiCache

---

d. Tune Query Efficiency

- Use `Query` instead of `Scan` wherever possible.
- Use **ProjectionExpression** to return only necessary attributes.
- Use **Global Secondary Index (GSI)** for alternate access patterns.

---

e. Use DAX for Read Caching

DynamoDB Accelerator (DAX) reduces read latency from milliseconds → microseconds,
especially useful for read-heavy workloads.

---

f. Analyze with CloudWatch Contributor Insights

This tool identifies the **top N partition keys** causing throttling or high latency.

Example setup:

```
aws cloudwatch put-insight-rule \
  --rule-name "TopKeysDynamoDB" \
  --rule-definition '{
    "Schema": {
        "Name": "DynamoDBContributorInsights",
        "Version": 1
    },
    "Metrics": ["ThrottledRequests", "ConsumedReadCapacityUnits"],
    "GroupBy": ["partitionKey"]
}'
```

Helps detect uneven traffic distribution across partitions.

---

5. Cost Optimization Insights

Monitoring also helps **control DynamoDB cost**:

Optimization	Description
Use Auto Scaling	Match capacity to real demand
Prefer On-Demand for unpredictable workloads	No wasted capacity
Delete unused GSIs	GSIs have their own RCUs/WCUs cost
Use TTL for old data	Automatically remove stale items
Export to S3 for cold storage	Move analytics data to S3 Glacier / Infrequent Access
Monitor `Consumed*` vs `Provisioned*` metrics	Identify over-provisioned tables

Example: Cost Impact

Assume you provision:

- 200 RCUs and 100 WCUs
- Region: **US East (Ohio) (us-east-2)**

Capacity	Price per unit-hour	Monthly (30 days)
200 RCUs	$0.00013	~$18.72
100 WCUs	$0.00065	~$46.80
Total		**~$65.52/month**

If you enable **Auto Scaling** and reduce average usage to 50%,
you can cut cost to around **$32/month**.

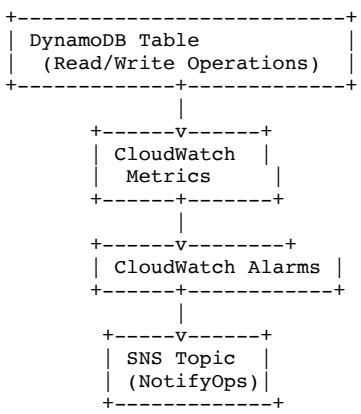
---

6. Common Monitoring Alerts to Configure

Alert	Reason
ThrottledRequests > 0	Capacity too low or hot partition
ConsumedRead/WriteCapacity > 80%	Approaching limit
ReadLatency > 50 ms	High query latency
SystemErrors > 0	Internal DynamoDB error
BillingEstimate > threshold	Unexpected cost increase

---

7. Example Monitoring Architecture



Fully automated monitoring
Instant alerting
Integration with Slack, PagerDuty, or email

Summary

Area	Metric/Tool	Goal
Read/Write performance	ConsumedRCU/WCU	Prevent throttling
Throttling	ThrottledRequests	Scale or redesign
Latency	SuccessfulRequestLatency	Detect app slowness
Error tracking	4xx/5xx metrics	Diagnose issues
Capacity optimization	Auto Scaling / On-Demand	Reduce cost
Hot partitions	Contributor Insights	Balance load
Cost	CloudWatch + Billing Dashboard	Budget control

- **Performance Tuning:**
  - Troubleshooting throttled requests.

- Adjusting **Auto Scaling** policies for Provisioned Capacity.

Performance Tuning in DynamoDB

Goal:

To ensure your DynamoDB tables and indexes are performing efficiently — **minimizing throttling**, maintaining low latency, and **optimizing cost** with the right capacity settings.

---

1. Troubleshooting Throttled Requests

Throttling occurs when your application exceeds the provisioned (or auto-scaled) **read or write capacity** for a DynamoDB table or index.

When throttling happens:

- DynamoDB returns a `ProvisionedThroughputExceededException`.
- Some requests fail or retry, increasing latency.

1.1. Common Causes of Throttling

Cause	Description	Example
Hot Partitions	Too many requests are sent to one partition key, overloading that partition.	Partition key = "US-East" for millions of users in that region.
Uneven traffic patterns	Burst of reads/writes beyond provisioned throughput.	Batch job floods the table at midnight.
Inadequate Provisioned Capacity	Not enough RCUs/WCUs allocated to match peak usage.	Table has 10 WCUs, but workload requires 30 WCUs.
Improper Auto Scaling settings	Scaling thresholds too conservative or cooldown too long.	Auto scaling set to react only when 90% capacity is used for 15 minutes.

1.2. How to Detect Throttling

Use **Amazon CloudWatch** metrics for your DynamoDB table:

Metric	Description
`ReadThrottleEvents`	Number of throttled read requests.
`WriteThrottleEvents`	Number of throttled write requests.
`ConsumedReadCapacityUnits` / `ConsumedWriteCapacityUnits`	Current capacity usage.
`ThrottledRequests`	Requests denied due to exceeding limits.

Check in AWS Console:
CloudWatch → Metrics → DynamoDB → TableName

If throttling is visible while usage is below 100% of provisioned capacity, it's a **data modeling issue** (e.g., hot partition).

1.3. How to Fix Throttling

Option 1: Use On-Demand Capacity Mode

If workloads are unpredictable:

- DynamoDB automatically scales up and down with demand.
- You pay per request, no need to manage RCUs/WCUs.

Example:

```
aws dynamodb update-table \
  --table-name Orders \
  --billing-mode PAY_PER_REQUEST
```

Pros: No throttling under normal circumstances, simple.
Cons: Slightly higher cost for constant high throughput.

Option 2: Tune Partition Keys

- Choose keys that evenly distribute traffic.
- Use a **composite key** (Partition + Sort Key) to spread writes.

Example:
Instead of partition key = "user123", use "user123#timestamp".

Option 3: Enable Auto Scaling for Provisioned Tables

Auto Scaling adjusts provisioned capacity automatically.

- Ensure target utilization is set to **70–80%**, not 100%.
- Set **cooldown period** to 60 seconds (default is often longer).

Example — Adjust Auto Scaling Policy

You can configure Auto Scaling in AWS Console or via CLI.

Example:

```
aws application-autoscaling register-scalable-target \
  --service-namespace dynamodb \
  --resource-id table/Orders \
  --scalable-dimension dynamodb:table:WriteCapacityUnits \
  --min-capacity 10 \
  --max-capacity 100

aws application-autoscaling put-scaling-policy \
  --service-namespace dynamodb \
  --resource-id table/Orders \
  --scalable-dimension dynamodb:table:WriteCapacityUnits \
  --policy-name WriteAutoScalingPolicy \
  --policy-type TargetTrackingScaling \
  --target-tracking-scaling-policy-configuration '{
    "TargetValue": 70.0,
    "PredefinedMetricSpecification": {
      "PredefinedMetricType": "DynamoDBWriteCapacityUtilization"
    },
    "ScaleInCooldown": 60,
    "ScaleOutCooldown": 60
  }'
```

Result:
The table “Orders” will automatically scale between **10 and 100 WCUs** to maintain 70% utilization.

1.4. Other Optimization Strategies

Strategy	Description
Batch Writes/Reads	Use BatchWriteItem or BatchGetItem to reduce API calls.
DAX (DynamoDB Accelerator)	In-memory cache for read-heavy workloads.
Exponential Backoff	Retry throttled requests with delays to smooth spikes.
Query instead of Scan	Scans consume more RCUs; use Queries when possible.

---

2. Adjusting Auto Scaling Policies (Provisioned Mode)

Auto Scaling uses **Application Auto Scaling** service behind the scenes.

Example Calculation:

Let’s say:

- Each write consumes **1 WCU**.
- You perform **2,000 writes/second**.
- Each WCU = **$0.00065 per hour** (Ohio region).

Baseline Cost:

```
2,000 WCU * $0.00065 * 24 * 30 ≈ $936/month
```

With Auto Scaling:

- You provision 500 WCU initially.
- Scale to 2,000 WCU during spikes for only 3 hours/day.

Optimized Cost:

```
(500 WCU * 21h/day + 2000 WCU * 3h/day) * $0.00065 * 30 ≈ $570/month
```

Savings: ~40%
Auto Scaling ensures you don’t overpay for idle time.

Best Practices Summary

Best Practice	Description
Use **Auto Scaling or On-Demand** for variable workloads.	Avoid manual capacity management.
Monitor **CloudWatch metrics** and set alarms.	Catch throttling early.
Use **DAX** or **caching layers** for high read throughput.	Reduce read RCUs.
Design **evenly distributed keys**.	Prevent hot partitions.
Combine **TTL** for data cleanup + Auto Scaling for cost.	Keep tables lean and efficient.

Would you like me to include a **diagram** showing how throttling, Auto Scaling, and CloudWatch monitoring interact in DynamoDB?

- **Cost Management:**
  - Comparing the pricing models of On-Demand vs. Provisioned Capacity.
  - Managing costs associated with Global Tables and Backups.

DynamoDB Cost Management

Effective cost management in DynamoDB involves understanding how AWS bills for read/write throughput, data storage, backups, and multi-region replication.

---

1. Comparing Pricing Models: On-Demand vs. Provisioned Capacity

DynamoDB offers **two billing modes**:

- **On-Demand Capacity Mode**
- **Provisioned Capacity Mode**

Let's explore both with **pricing examples (Ohio region – `us-east-2`)**.

1.1. On-Demand Capacity Mode

- **Fully managed scaling:** You don't have to specify RCUs or WCUs.
- **Pay per request** — best for unpredictable or spiky workloads.

Pricing (Ohio Region)

Operation Type	Cost per Million Requests
Write request unit (WRU)	**$1.25 per million writes**
Read request unit (RRU)	**$0.25 per million reads**

Example

Let's say your app performs:

- **2 million writes/day**
- **8 million reads/day**

Monthly Cost (approx 30 days):

```
Write: 2M * 30 * $1.25 / 1M = $75
Read:  8M * 30 * $0.25 / 1M = $60
---------------------------------
Total = $135/month
```

Advantages

- No need to plan capacity.
- Scales instantly with workload spikes.
- Great for new or unpredictable workloads.

Disadvantages

- More expensive for steady, predictable workloads.
- No reserved capacity discounts.

1.2. Provisioned Capacity Mode

You **predefine Read and Write Capacity Units (RCUs/WCUs)**.

Pricing (Ohio Region)

Type	Price per hour
1 RCU	**$0.00013/hour**
1 WCU	**$0.00065/hour**

Example

Let's say you provision:

- 100 RCUs and 50 WCUs.

Monthly cost:

```
RCU cost: 100 * 0.00013 * 24 * 30 = $9.36
WCU cost: 50 * 0.00065 * 24 * 30 = $23.40
-----------------------------------------
Total = $32.76/month
```

Advantages

- Cost-effective for consistent workloads.
- Eligible for **Auto Scaling** and **Reserved Capacity** discounts.
- Predictable billing.

Disadvantages

- Must plan throughput carefully (risk of over/under-provisioning).
- Slower adaptation to sudden spikes (unless Auto Scaling is tuned).

When to Use Which

Scenario	Recommended Mode
Unpredictable traffic	**On-Demand**
Steady, predictable usage	**Provisioned**
Early-stage apps or startups	**On-Demand**
Mature, stable production apps	**Provisioned with Auto Scaling**

---

2. Cost of Global Tables

Global Tables replicate data automatically across multiple AWS Regions.

Each region has **its own table replica**, and you pay for:

1. **Storage in each region**
2. **Write and read capacity (or on-demand charges)**
3. **Replication data transfer costs**

Pricing Components (Ohio Example)

Cost Component	Description	Ohio Region Pricing
Write capacity	Every replicated write is billed in each region.	$0.00065/WCU-hour
Read capacity	Reads billed per region (based on access pattern).	$0.00013/RCU-hour
Replication data transfer	Between regions (charged per GB).	~$0.02/GB

Example — 2 Region Setup (Ohio + Oregon)

You have a table that:

- Handles **2 million writes/day** (≈ 23 writes/sec)

- Each write = **1KB**
- Two regions replicated (Ohio ↔ Oregon)

Costs:

- **Write Cost (2 regions):**

```
23 WCUs * 2 regions * 0.00065 * 24 * 30 ≈ $21.53/month
```

- **Replication Data Transfer:**

```
2M writes/day * 1KB * 30 days = 60 GB/month
60 * $0.02 = $1.20/month
```

Total Global Table Overhead: ~$22.73/month

Tip: If you replicate across **3+ regions**, costs increase linearly per region.

Best Practices for Cost Control in Global Tables

Best Practice	Description
Limit replication to regions that need read/write access.	Avoid unnecessary replicas.
Use **DynamoDB Streams** + **Lambda** for selective replication.	Avoid full multi-region replication if partial is enough.
Optimize item size before replication.	Less data transfer = lower cost.

---

3. Managing Backup and Restore Costs

DynamoDB supports two backup types:

1. **On-demand backups**
2. **Point-in-time recovery (PITR)**

3.1. On-Demand Backup Pricing

Region	Price per GB/month
Ohio	**$0.10 per GB-month**

No performance impact during backup.
Stored until you delete it.

Example:
If your table is **100 GB**, backup storage costs:

```
100 GB * $0.10 = $10/month
```

3.2. Point-in-Time Recovery (PITR)

- Maintains continuous incremental backups for **up to 35 days**.
- Enables restore to any second within that window.

Pricing:

Region	PITR Cost
Ohio	**$0.20 per GB-month**

Example:
For the same 100 GB table:

```
100 GB * $0.20 = $20/month
```

Enables fast recovery from accidental deletes or corrupt data.
Cost adds up for large tables; disable PITR for non-critical data.

3.3. Restore Costs

When restoring from backup or PITR:

Operation	Cost
Restore data	**$0.15 per GB restored**

Example:

```
Restoring 50 GB → 50 * $0.15 = $7.50 one-time
```

Backup Cost Optimization Tips

Strategy	Benefit
Use **PITR** only for mission-critical tables.	Save storage costs.
Regularly delete old **on-demand backups**.	Avoid buildup of charges.
Export large historical data to **S3 Glacier**.	Cheaper long-term retention.
Use **Lifecycle policies in S3** after export.	Automate tiering (e.g., Glacier, Deep Archive).

Summary: Cost Management Cheatsheet

DynamoDB Feature	Cost Driver	Cost Control Tips
On-Demand Mode	Per million requests	Use for unpredictable workloads only
Provisioned Mode	RCUs/WCUs per hour	Enable Auto Scaling
Global Tables	Multi-region writes + storage	Limit to needed regions
Backups	$0.10–$0.20 per GB	Delete old backups regularly
PITR	Continuous 35-day backup	Disable on non-critical tables
Exports	Per GB transfer to S3	Compress and archive in Glacier

Recommendation for Production:

- Use **Provisioned + Auto Scaling** for stable traffic.
- Enable **PITR** on critical tables only.
- Use **Global Tables** selectively (based on latency and DR needs).
- Regularly audit costs in **AWS Cost Explorer → DynamoDB Usage Reports**.

———

Advanced information

1. What is DynamoDB & Why It Matters

- DynamoDB is a **fully managed NoSQL database service** by AWS. USENIX+1
- It supports massive scale, high throughput, and predictable low-latency (e.g., single-digit ms for 1 KB items). assets.amazon.science
- Designed for key design goals: availability, durability, scalability, predictable performance. distributed-computing-musings.com+1
- Use cases: high-traffic applications such as Amazon.com site, Alexa, fulfillment systems. USENIX+1

2. Architecture & Core Components

2.1 Table, Items, Partitions

- A DynamoDB **table** is a collection of items (rows). Each item has attributes. USENIX
- Primary key: either only partition key or partition + sort key. The partition key is hashed to determine which partition (i.e., storage node) the item resides on. assets.amazon.science
- Data is automatically **partitioned** horizontally: as data grows, more physical partitions are added. This enables "boundless" scale. assets.amazon.science

2.2 Replication & Availability

- To ensure durability and availability, partitions are replicated across multiple Availability Zones (AZs). muratbuffalo.blogspot.com+1
- For multi-region deployments (Global Tables), data can be geo-replicated across multiple AWS regions. USENIX

2.3 Request Routing & Performance

- The internal request router uses metadata to map the key's hash to the correct partition.
- The system aims to keep latencies stable even as tables scale (many terabytes). assets.amazon.science
- Internally uses mechanisms like leader election for writes, multi-node consensus in partitions, etc. (though details are abstracted). distributed-computing-musings.com

2.4 Capacity & Throughput

- DynamoDB has throughput models: provisioned capacity (RCUs and WCUs) or on-demand capacity. (From other AWS docs)
- The system must handle traffic spikes, hot partitions, uneven load. The paper discusses techniques for fairness and imbalance. distributed-computing-musings.com

---

3. How to Use DynamoDB in a Real-World Application

Let's say you're building a **global web service** with user profiles and user activity logs. You decide to use DynamoDB for the data store. Here's how you might apply the concepts:

3.1 Design Your Table

- Create a table `UserActivity` with partition key = `UserId` and sort key = `ActivityTimestamp`.
- This allows you to retrieve all activities for a user in chronological order.

3.2 Plan for Scale

- Expect heavy read traffic (e.g., retrieving recent activities) and moderate write traffic (activity logging).
- Use a provisioned capacity model initially (e.g., 100 RCUs, 50 WCUs) and enable Auto-Scaling.
- Use On-Demand if traffic is very spiky/unpredictable.

3.3 Choose Consistency & Throughput

- For user activity logs, eventually-consistent reads might suffice (slight delay OK).
- For user profile updates you might choose strongly consistent reads (immediately after write).
- Monitor the consumed capacity and throttling metrics; if you see hot partitions (e.g., one user with huge traffic), consider spreading by using a composite key or sharding.

3.4 Global Availability

- If you have users in US, Europe and Asia, you could use **Global Tables** so that each region has a local replica.
- This gives low latency reads/writes globally and high availability if a region fails.

3.5 Monitor & Maintain Performance

- Use CloudWatch to monitor latencies, throttling, read/write unit consumption.
- Avoid "hot keys" by distributing writes across many partition keys.
- Use TTL to expire old log entries to keep table size manageable.
- Export old data to archive (S3 + Athena) if appropriate.

---

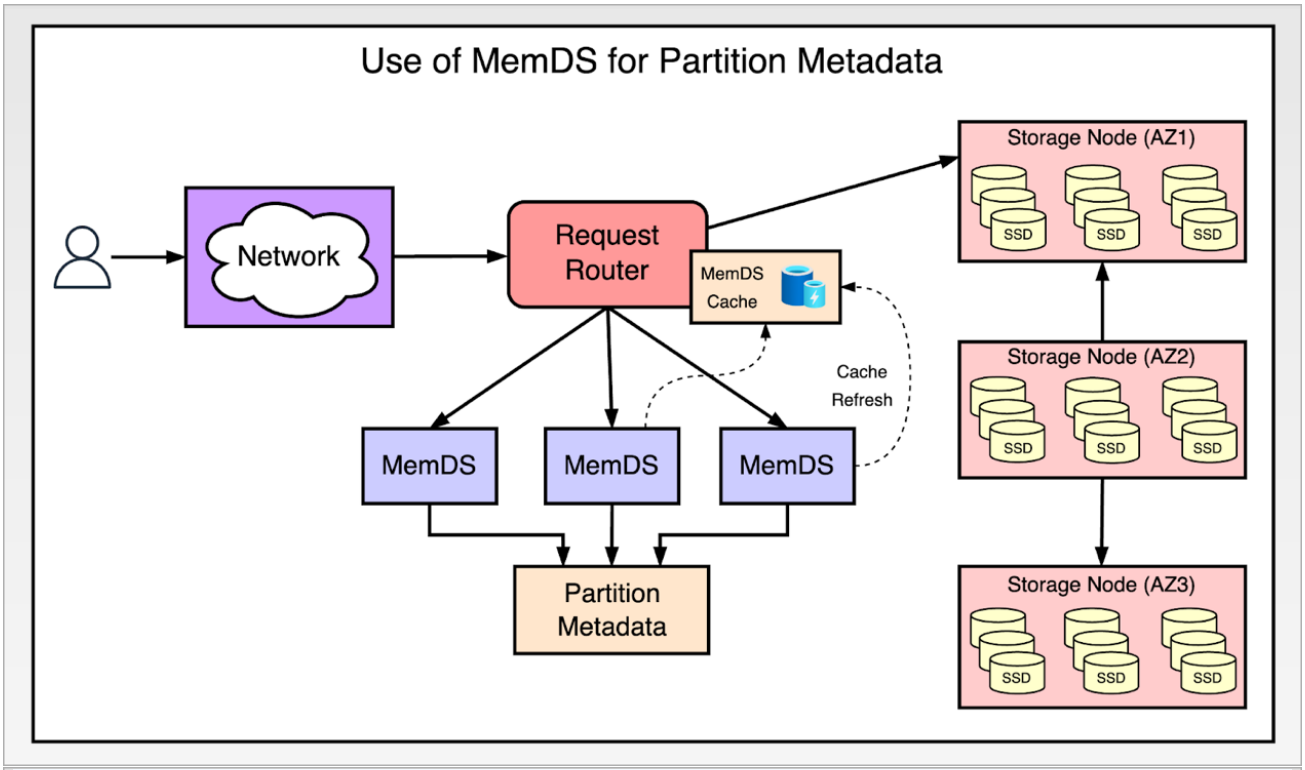
4. Key Lessons & Design Takeaways from the Video / Paper

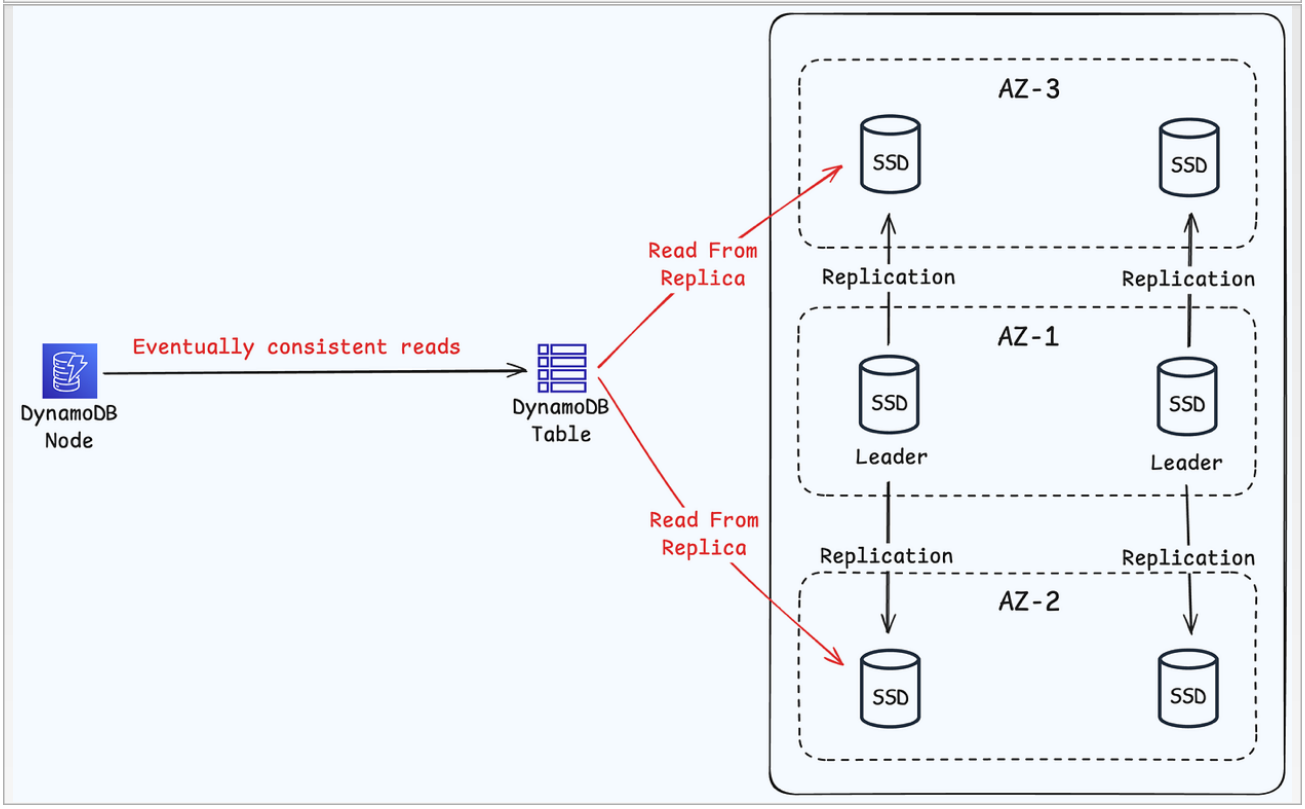
From the presentation and paper, here are major design lessons:

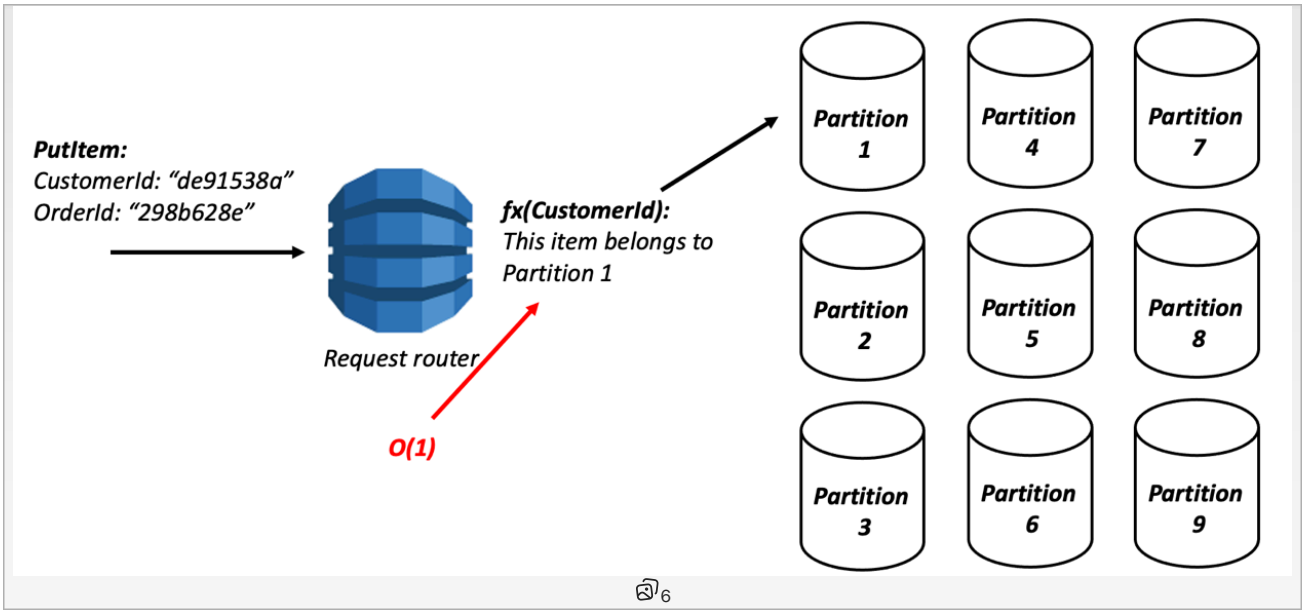
- **Predictability over absolute efficiency**: It's better to design for stable performance (even if slightly less efficient) than maximum throughput with surprises. muratbuffalo.blogspot.com
- **Partitioning is essential**: As data grows, the system must split partitions and reassess data distribution. Load imbalance must be addressed.
- **Operational excellence matters**: Multi-tenant architecture, monitoring, deployment safety, chaos testing are part of ensuring the service remains available at scale. distributed-computing-musings.com
- **No fixed limits for table size**: Tables should grow "elastically" and transparently to developers. assets.amazon.science
- **Flexible data model**: DynamoDB allows schema-less items, key/value or document style. Works for many use cases. USENIX

---

5. Diagram of Key Architecture


Use of MemDS for Partition Metadata
Network
Request Router
MemDS Cache
Cache Refresh
MemDS
MemDS
MemDS
Partition Metadata
Storage Node (AZ1)
Storage Node (AZ2)
Storage Node (AZ3)
SSD


DynamoDB Node
Eventually consistent reads
DynamoDB Table
Read From Replica
Read From Replica
AZ-3
AZ-1
AZ-2
SSD
Replication
Leader



(Use the above to visualize how partitions, nodes, replication and global tables fit together.)

---

6. Best Practices / Guidelines

- Choose a **good partition key** (high cardinality, evenly distributed).
- Avoid "hot partitions" — e.g., don't have many writes all going to same partition key value.
- Use **strongly consistent reads** only when required (they have higher cost/latency).
- Enable **Auto Scaling** or **On-Demand mode** for changing workloads.
- Enable **TTL** for deleting old unused items automatically.
- For global applications, use **Global Tables** to replicate data across regions.
- Use **backup & restore**, **export** capabilities for data protection and archiving.
- Monitor **latency (P50, P99)**, **throttling events**, **capacity usage**, **partition skew**.
- Build for **predictability**, not just peak efficiency.