1. Google File System (GFS)

Google File System (GFS) is a **scalable distributed file system** developed by Google to handle large data sets using commodity hardware.

Key Characteristics:

- **Fault-tolerant**: Designed to work efficiently even when hardware fails.
- **High throughput**: Optimized for batch processing and large streaming reads.
- **Append-mostly operations**: Supports append-heavy workloads instead of random writes.
- **Scalability**: Easily scales to thousands of nodes and petabytes of data.

Architecture Components:

- **Master Server**:
  - Manages **metadata** (file system tree, chunk locations, access control).
  - Controls **file-to-chunk** mapping and chunk replication.
- **Chunk Servers**:
  - Store actual **data chunks** (default size 64 MB).
  - Handle **read/write** requests from clients.
  - Each chunk is replicated across **multiple chunkservers** (default: 3 replicas).
- **Client**:
  - Requests metadata from **Master** and data directly from **Chunk Servers**.

Working:

- When a file is created, it's split into **chunks** and stored across chunkservers.
- Clients read/write by first contacting the **master** for metadata and then interacting with chunkservers.
- Data consistency is maintained using **version numbers** and **lease mechanisms**.

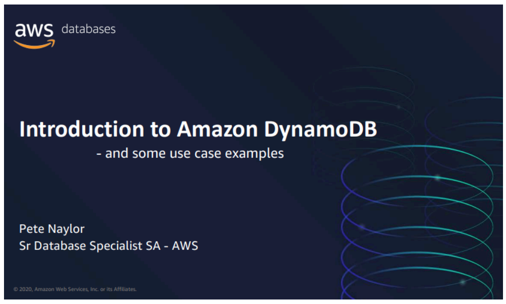
Fault Tolerance:

- Replication ensures data availability.
- Master periodically **logs and checkpoints metadata**.
- Clients automatically retry on failure.

Limitations:

- Single Master can be a **bottleneck**, though mitigated using **shadow masters** and **checkpointing**.

DYNAMO DB



Amazon’s relational databases (SQL) struggled with **scale, cost, and complexity** around 2004.

Response:

- In 2007, Amazon released the **Dynamo** paper (core idea of a distributed key-value store).
- In 2012, AWS launched **DynamoDB**, a managed, serverless version of it.

Feature	SQL (Relational DB)	DynamoDB (NoSQL)	Explanation / Keyword Meaning
Data Structure	**Normalized**	**Denormalized**	*Normalized*: Data split into multiple tables to avoid redundancy (uses joins). *Denormalized*: Data kept together for fast access, even with redundancy.
Scaling	**Vertical Scaling**	**Horizontal Scaling**	*Vertical*: Add more CPU/RAM to a single server. *Horizontal*: Add more servers (sharding).
Query Flexibility	**Ad hoc queries supported**	**Requires pre-defined access patterns**	*Ad hoc*: Run flexible queries any time (e.g., SQL `SELECT * WHERE ...`). *Pre-defined*: Design data upfront for specific queries.
Use Case Fit	Good for **OLAP**	Built for **OLTP**	*OLAP*: Analytics-heavy systems (e.g., dashboards). *OLTP*: High-speed transactional systems (e.g., shopping cart).

Why not just use SQL everywhere?

→ SQL breaks when traffic, data size, or app complexity explodes.

Scaling Mechanism in DynamoDB

Traditional SQL Scaling (Vertical Scaling):

- In SQL databases, scaling involves upgrading a server’s resources (CPU, RAM, disk), which can lead to bottlenecks, downtime, and cost inefficiencies.
- This approach becomes less effective for large-scale, high-traffic applications.

NoSQL Scaling (Horizontal Scaling):

- NoSQL databases like DynamoDB scale horizontally by distributing data across multiple nodes (servers).

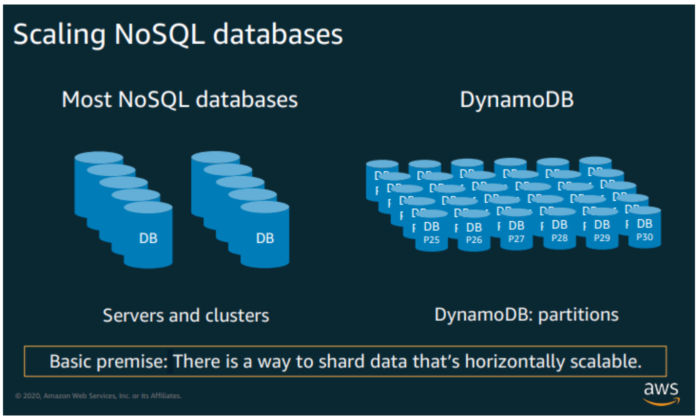
- This method is more efficient and handles large, dynamic workloads without bottlenecks.

DynamoDB's Scaling Mechanism:

1. **Workload Management**:
    - Scaling is based on data volume and read/write units (RUs and WUs).
    - Each partition can handle up to 1,000 WUs, 3,000 RUs, and 10GB of data.
2. **Automatic Incremental Scaling**:
    - DynamoDB automatically adjusts the number of partitions based on data access patterns and volume.
    - Users interact with tables, while partitioning is handled behind the scenes.
3. **Partitioning**:
    - DynamoDB splits tables into partitions, each assigned to different servers for load distribution.
    - This partitioning is transparent to users, who only manage tables.

Key Takeaway:

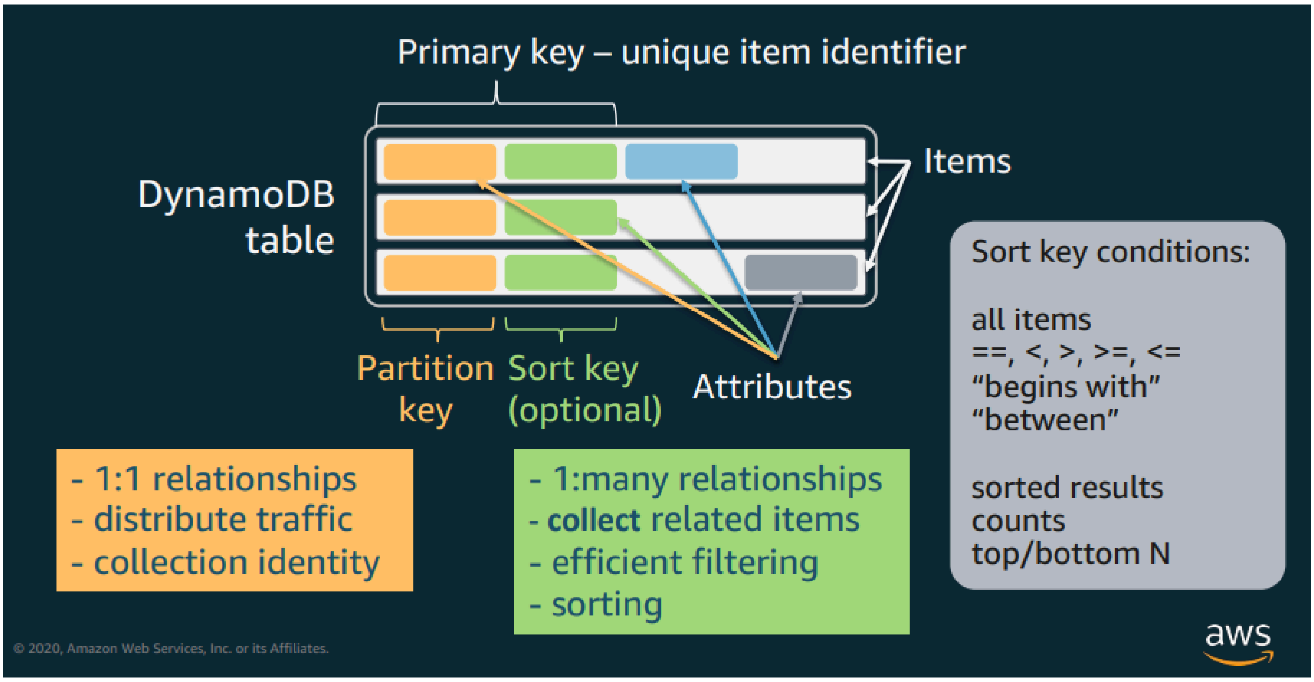
DynamoDB provides automatic horizontal scaling using a partition-based model. This allows it to efficiently handle high traffic and large data volumes without manual intervention.



While most NoSQL systems rely on scaling clusters, **DynamoDB scales horizontally by sharding data into partitions** — making it easier to scale with **high availability, low latency**, and **automatic load balancing**.

How does it handle spiky loads or high traffic on a single key?

→ Adaptive Capacity & DAX (DynamoDB Accelerator)



DynamoDB Table Structure

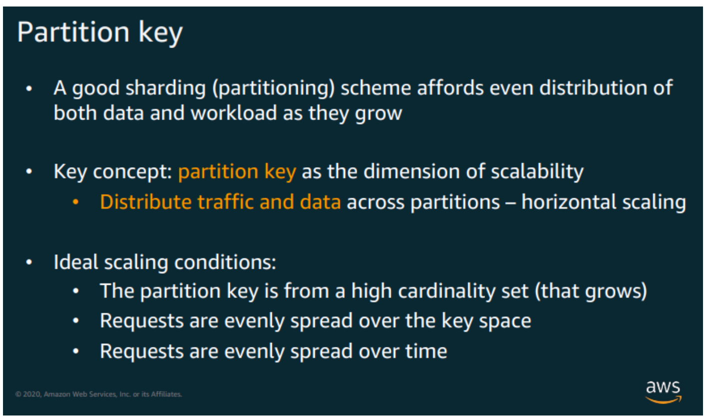
- A DynamoDB table stores **items** (similar to rows in a relational database).
- Each item contains **attributes** (similar to columns).

1. Primary Key

- The **primary key** uniquely identifies each item.
- It can be:
  - **Partition Key only** (Simple Primary Key)
  - **Partition Key + Sort Key** (Composite Primary Key)

Purpose

- **Partition Key**:
  - Used to determine the partition where data is stored.
  - Best for one-to-one relationships.
- **Sort Key**:
  - Helps store and retrieve related data.



    - Enables querying within a partition (e.g., time range, top N).

Key Conditions

- Query operations can use conditions like `=`, `<`, `>`, `BETWEEN`, `BEGINS_WITH`, etc., on sort keys.

---

2. Global Secondary Index (GSI)

- A GSI provides an **alternate way to query data** using different attributes than the primary key.
- It has its own **Partition Key and optional Sort Key**.
- DynamoDB maintains the GSI separately from the main table.

Key Concepts

- **Online Indexing**: The index is updated as new data is written.
- **Projected Attributes**:
    - You can choose to include specific attributes from the main table.
    - Three types:
        - KEYS_ONLY: only partition and sort keys
        - INCLUDE: specific non-key attributes
        - ALL: all attributes
- You can define **up to 20 GSIs per table**.
- **Capacity is provisioned separately** from the main table. [Chatgpt Example]

Feature	Primary Index	Global Secondary Index (GSI)
Defined when	Table is created	Created optionally anytime
Based on	Table's Partition & Sort Key	Custom attributes you choose
Can be changed?	No	Yes, can be added later
Capacity	Uses table's capacity	Has its own capacity settings
Use case	Default, most frequent query	Alternate access/query patterns

Sharding / Partitioning

- DynamoDB automatically shards data across multiple partitions for scalability and performance.

How It Works

- Each item's **Partition Key** is hashed.
- The hash output determines which **partition** the item will go to.
- Related items (with the same partition key) are stored together.

Benefits

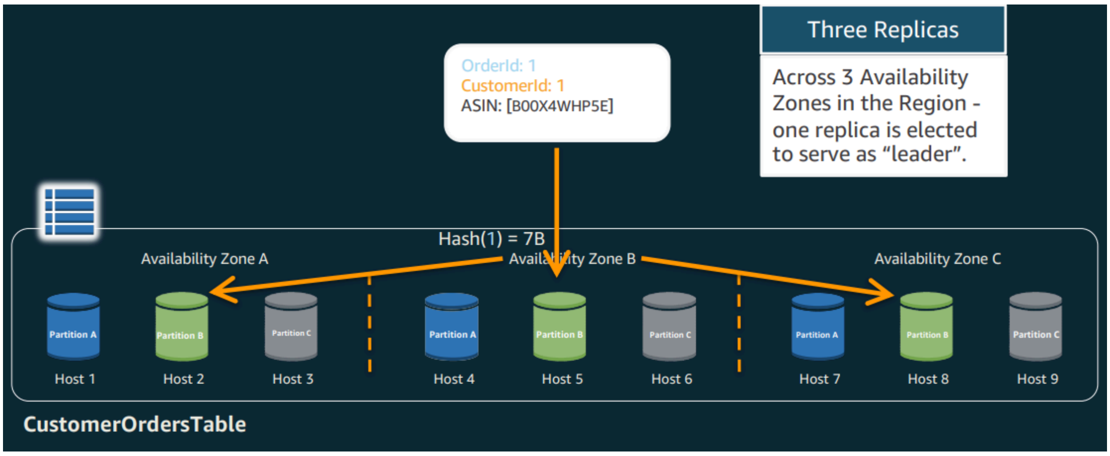
- Prevents uneven load (avoids “hot partitions”).
- Ensures data is spread across nodes for parallelism.
- Supports millions of reads/writes per second.

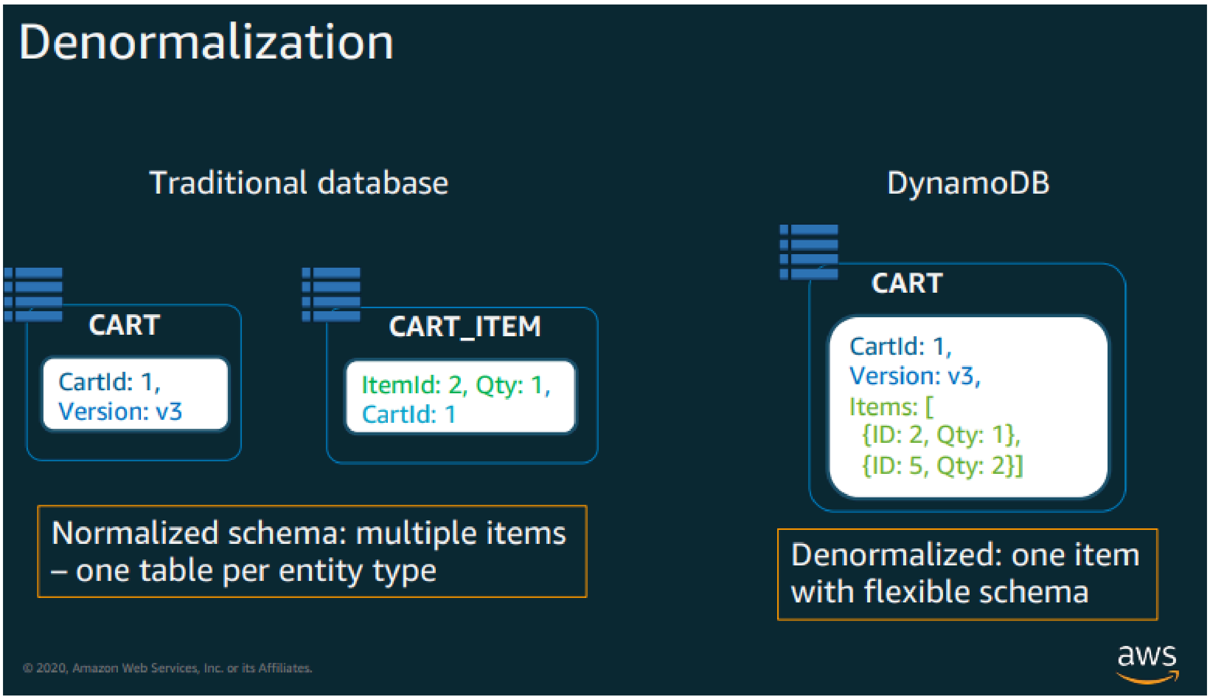
A View from a Different Angle – Replication and Availability

- Each DynamoDB partition is **replicated across 3 Availability Zones (AZs)** within an AWS Region.

Key Points

- **Three replicas** per partition ensure fault tolerance.
- One replica acts as the **leader**, handling all writes.
- The other two are **followers**, available for reads or takeover if needed.
- This architecture ensures:
    - **High Availability**
    - **Durability**
    - **Automatic failover** in case of an AZ outage





Denormalization is the process of combining related data into a single structure instead of splitting it across multiple related tables.

Comparison with Traditional Databases

- Traditional databases use normalization with multiple related tables (e.g., `Cart` and `Cart_Item`)
- Enforce strict schemas and use joins

DynamoDB Approach

- Uses **denormalization**: stores all related data in a single record
- Schema-less structure allows faster lookups

Traditional SQL Queries:

sql Copy Edit

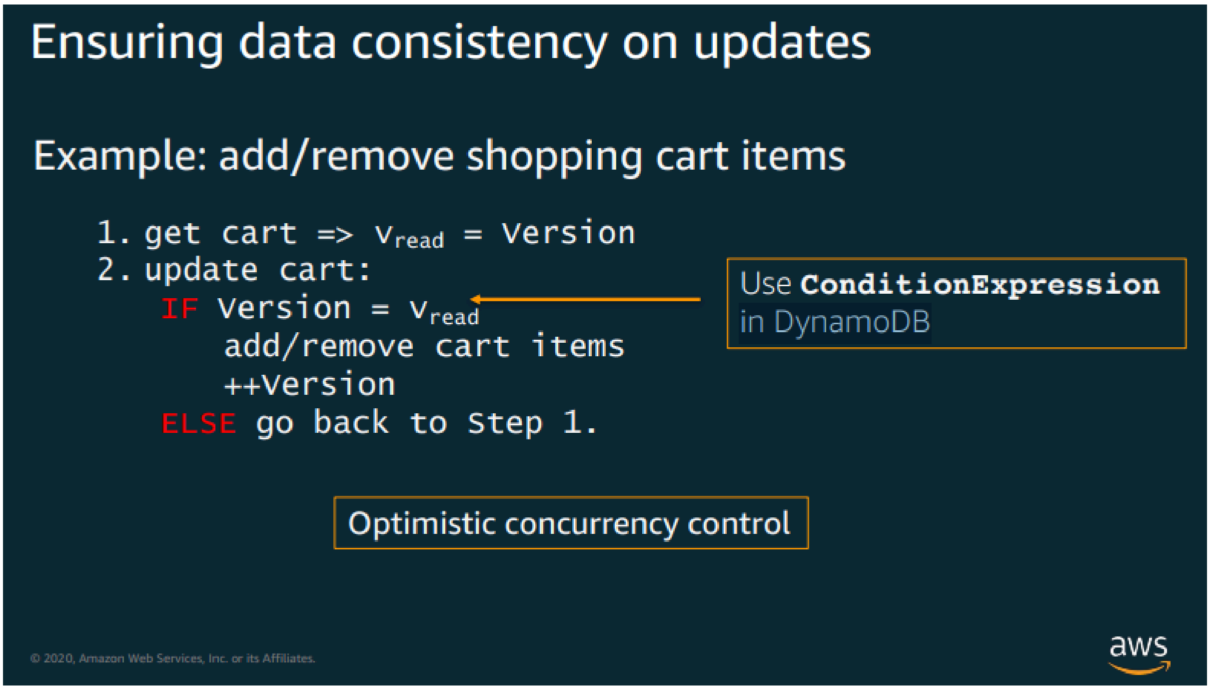
```sql
SELECT * FROM cart WHERE id=1;
SELECT * FROM cart_items WHERE cart_id=1;
```

Equivalent DynamoDB Record:

json Copy Edit

```json
{
  "CartID": 1,
  "Version": 1,
  "Items": [
    {"ID": "X", "Qty": 1},
    {"ID": "Y", "Qty": 2}
  ]
}
```

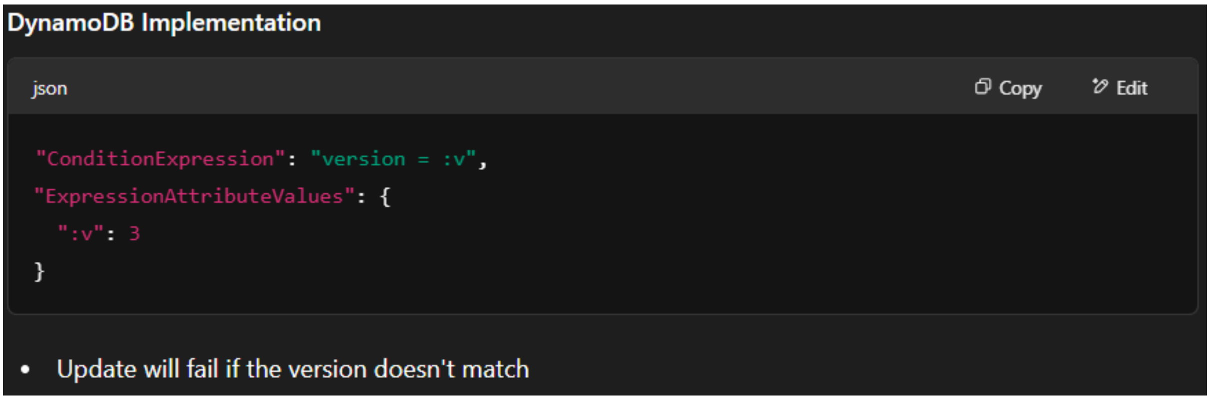
- All data is fetched and updated together

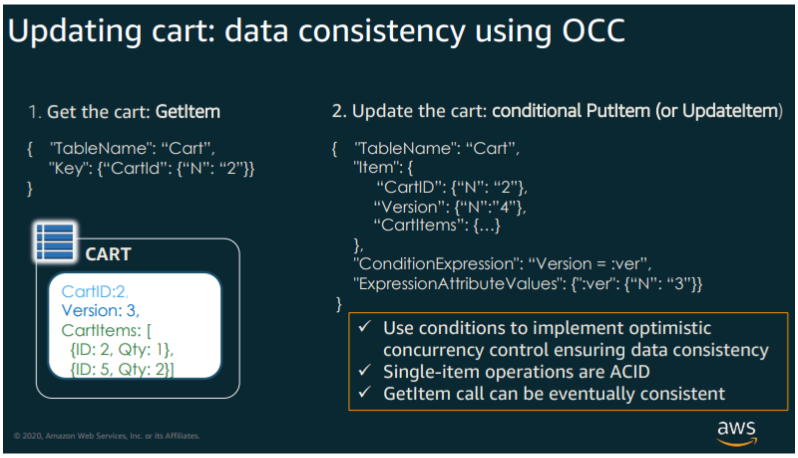


Problem
Concurrent updates (e.g., multiple users updating a cart) can lead to **race conditions**.

Solution: Optimistic Concurrency Control (OCC)

- Read the current version (e.g., version = 3)
- Update only if the version hasn't changed
- If the version has changed, retry





Yes, you **can have strongly consistent read-after-write and concurrency control** with DynamoDB — here's the justification:

1. Strongly Consistent Read-after-Write

By default, DynamoDB provides **eventual consistency**, meaning you might get stale data right after a write.
But if you need **immediate accuracy** (like verifying a recent update), **DynamoDB supports *strongly consistent reads*** as an option.

How?

- When you make a read request, you can set:

json Copy Edit

```json
"ConsistentRead": true
```

- This ensures the read always reflects **the latest successful write** (from the same region).

Example Use Case:

- After updating a cart, you immediately fetch the cart again and want to ensure you get the updated version — strongly consistent read guarantees that.

---

2. Concurrency Control via Optimistic Locking (OCC)

DynamoDB allows **safe concurrent writes** using **Optimistic Concurrency Control**, which avoids race conditions.

How it works:

- You store a `version` field in your item.
- When updating, you include a **ConditionExpression** that checks if the version hasn't changed.

Example:

json Copy Edit

```json
"ConditionExpression": "version = :oldv",
"ExpressionAttributeValues": {
  ":oldv": 3
}
```

- If someone else updated the item and the version is now 4, your update **fails gracefully**, preventing conflicts.
- You can then **retry** by re-fetching the latest item and attempting the update again.

Feature	Support in DynamoDB	Purpose
Strongly Consistent Read	Yes (`ConsistentRead: true`)	Ensures accurate read after write
Concurrency Control (OCC)	Yes (using `ConditionExpression`)	Prevents conflicting writes

✅ By Default: Eventually Consistent

- When you **read data** after a write, you **might not see the latest value immediately**.
- This is called **eventual consistency** — the system guarantees that **all replicas will be updated eventually**, but **not instantly**.

🧠 Why Use Eventual Consistency?

- It's **faster** and more **scalable**, especially for global-scale applications.
- It reduces the load on the system and helps maintain high availability.

🔒 But... Strong Consistency is Also Available!

If your use case needs **immediate consistency**, DynamoDB lets you **explicitly request** it:

json Copy Edit

```json
"ConsistentRead": true
```

This makes the read **strongly consistent**, meaning:

- You'll **always get the latest committed data** from a successful write.
- But this works **only within the same AWS Region** (not across regions).

Yes, you can model complex data relationships with DynamoDB

🗂️ DynamoDB Data Modeling Patterns

1. One-to-One (Key-Value Lookup):

- Use **Partition Key** (e.g., `UserId` or `Email`)
- Access via `GetItem` / `BatchGetItem`
- Example: Get user details by email.

2. One-to-Many (Parent-Child):

- Use **Partition Key + Sort Key**
- Access via `Query`
- Example: Get all device readings between two timestamps.

3. Many-to-Many:

- Use a table and a **GSI with PK and SK swapped**
- Access via `Query`
- Example: Find all games for a user or all users for a game.

Challenges

- Growing datasets
- Variable or spiky throughput
- Imbalanced workloads (hot partitions)
- Poor traffic distribution across shards

🔸 Solutions

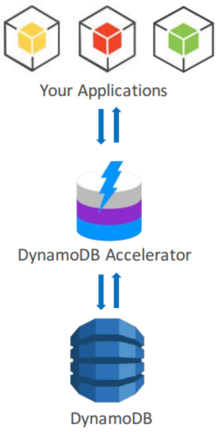
1. **Time-To-Live (TTL):**
    - Automatically delete expired data to reduce storage.
2. **Provisioned Mode + Auto Scaling:**

    ○ Automatically adjusts capacity based on usage patterns.

3. **On-Demand Mode:**
    ○ Ideal for unpredictable or spiky workloads. No capacity planning needed.

4. **Adaptive Capacity:**
    ○ DynamoDB shifts capacity to "hot" partitions to handle uneven workloads.

5. **DynamoDB Accelerator (DAX):**



    ○ In-memory caching layer for **read-heavy** applications
    ○ Fully managed, fault-tolerant, multi-AZ
    ○ Compatible with DynamoDB API (no code changes needed)
    ○ Supports **write-through caching**

Integrating DynamoDB into your data flow

1. Complex Queries and Analytics

Problem

Amazon DynamoDB is a high-performance NoSQL database. It is optimized for key-value access and simple queries, but it is **not designed for complex analytics or full-text search**. For these use cases, you may need other services.

Solution

To handle complex analytics and queries:

- **Use services best suited for those operations**, such as:
    ○ **Amazon Athena** – for interactive SQL queries on data stored in S3.

- **Amazon Redshift** – a data warehouse optimized for analytical queries.
- **Amazon Elasticsearch Service (now OpenSearch Service)** – for full-text search and log analytics.

Data Sync Using DynamoDB Streams

To **ensure data consistency** across these services and maintain up-to-date information:

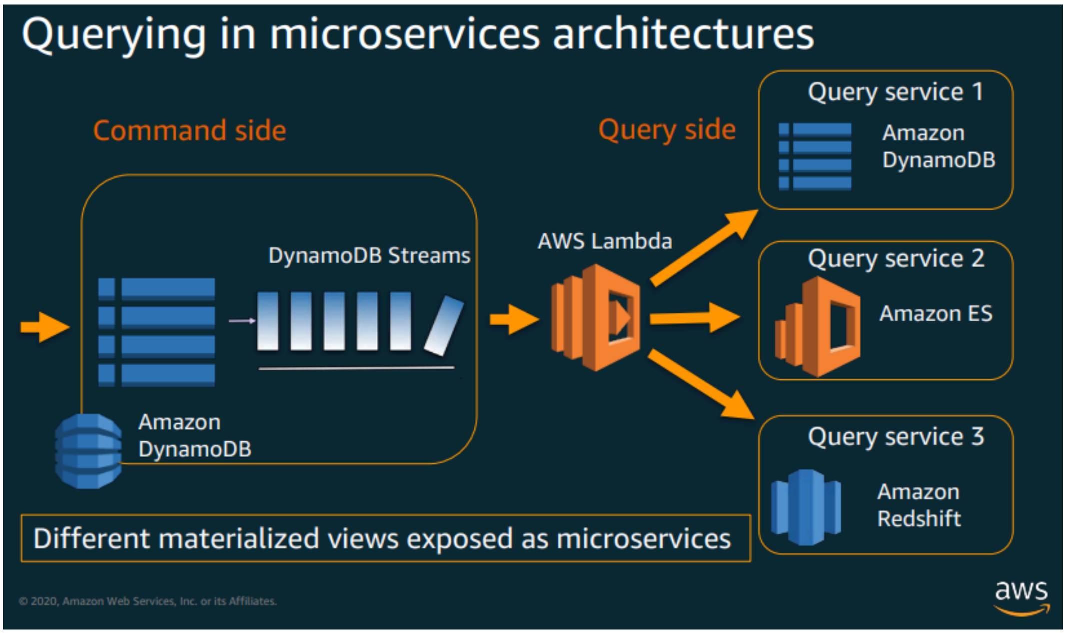
- **DynamoDB Streams** captures real-time changes (insert/update/delete) in a DynamoDB table.
- These changes can be **consumed by an AWS Lambda function**.
  - AWS Lambda is a serverless compute service that can execute code in response to DynamoDB stream events.
  - It allows for **end-to-end serverless architecture** with no manual intervention or servers to manage.

Workflow

- Data is written to the **DynamoDB table**.
- Changes are pushed to **DynamoDB Streams**.
- **AWS Lambda** consumes these changes and processes them (e.g., updates an analytics system or another database).
- Final analytics results can be stored or queried from a more appropriate service.

---

2. Querying in Microservices Architectures



Architecture Breakdown

This section shows how **DynamoDB Streams + AWS Lambda** can be used to enable **multiple microservices to work with different views of the same data**.

Separation of Concerns

- **Command Side** (Write Operations):
  - All writes (create/update/delete) are performed on the **Amazon DynamoDB** table.
  - These changes are captured by **DynamoDB Streams** in real time.
- **Query Side** (Read Operations):
  - Using **AWS Lambda**, changes from the stream are processed and sent to different **query services**.
  - Each query service has its own database or system optimized for its specific query requirements.

Examples of Query Services

- **Query Service 1** – continues to use **Amazon DynamoDB** if fast key-based lookups are sufficient.
- **Query Service 2** – sends data to **Amazon Elasticsearch Service (ES)** for full-text search capabilities.
- **Query Service 3** – sends data to **Amazon Redshift** for complex analytical queries.

Materialized Views as Microservices

Each query service can be thought of as a **materialized view** – a precomputed dataset tailored for a specific use case or query pattern.

- These views are maintained **automatically** through the stream + lambda mechanism.
- This pattern is commonly referred to as **CQRS (Command Query Responsibility Segregation)**.

Summary: Key Takeaways

1. **DynamoDB Streams + AWS Lambda** enables **real-time data propagation** from your core DynamoDB database to other specialized services.

2. **Decouple writes and reads** – you write once to DynamoDB and create many **query-optimized views** in different services.

3. Supports **serverless architecture** – no infrastructure management needed.

4. Enables **scalable microservices design** where different parts of your application (or different teams) can independently scale and optimize their data usage.

Polyglot Persistence – in simple terms:

Definition:
It means **using different types of databases** (SQL, NoSQL, graph, etc.) for different parts of an application — **"the right tool for the right job."**

🔹 Why it's used?

Because **no single database** is perfect for every use case. For example:

- **DynamoDB** for high-scale key-value or document data
- **RDS/MySQL** for relational, transactional data
- **Elasticsearch** for full-text search
- **Redis** for caching
- **Neo4j** (graph DB) for relationship-heavy data

🔹 Example:

An **e-commerce site** might use:

- **DynamoDB** for user sessions (fast, scalable)
- **MySQL** for orders and payments (ACID transactions)
- **Redis** for quick product lookups
- **Elasticsearch** for search functionality