Amazon DynamoDB: AWS NoSQL Database for Massive Scale

It is not a relational database and does not use SQL. Instead, it is designed for key-value and document-based data models.

DynamoDB is a fully managed NoSQL database service that is highly available and automatically replicated across three Availability Zones (AZs).

⚡ Key Features:

Serverless architecture: No need to manage servers or infrastructure.

Handles millions of requests per second, supporting trillions of rows and hundreds of terabytes of data.

Ensures fast and consistent performance with single-digit millisecond latency for real-time applications.

Integrated with AWS IAM for fine-grained access control and secure administration.

Supports auto-scaling to adjust to workload demands and reduce costs.

💰 Storage Classes:

Two table types are available:

Standard: For frequently accessed data.

Infrequent Access (IA): For cost-effective storage of rarely accessed data.

✅ DynamoDB is ideal for applications needing low-latency access at massive scale, such as mobile apps, IoT, and gaming backends.

Understanding Data Modeling in DynamoDB (Key-Value Structure)

🔑 Primary Key Structure

A **Primary Key** in DynamoDB consists of:

Partition Key (e.g., Product ID) – Used to distribute data data across storage nodes.

Sort Key (optional, e.g., Type) – Used to organize related items related items under the same partition.

🧩 Flexible Schema per Item

Each item in a DynamoDB table can have **a different set of set of attributes**—a key advantage of NoSQL.

The **schema is defined at the item level**, not at the table level. table level.

Different types of content (books, albums, movies) can be can be stored in the same table without rigid structure. structure.

🧠 What This Means

The table behaves like a collection of items where:

Each item is uniquely identified by its key combination.

DynamoDB is ideal for applications needing **flexible data models**, **fast key-based access**, and **scalable storage**.

DynamoDB is a key-value and document database, where each item (record) is stored with a **primary key** and optional **attributes**.

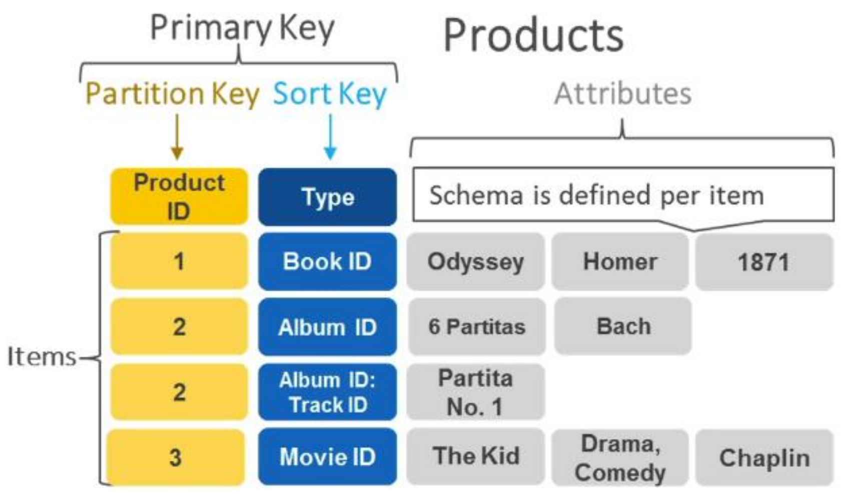
In the diagram:

Product ID = 1, Type = Book ID has attributes like "Odyssey", "Homer", 1871.

Product ID = 2 is reused for two different types of items (Album and Track), grouped under the same partition.

Product ID = 3, Type = Movie ID has different attributes such as genre and director.

✅ DynamoDB is ideal for applications needing **flexible data models**, **fast key-based access**, and **scalable storage**.



Product ID	Type	Schema is defined per item		
1	Book ID	Odyssey	Homer	1871
2	Album ID	6 Partitas	Bach	
2	Album ID: Track ID	Partita No. 1		
3	Movie ID	The Kid	Drama, Comedy	Chaplin

Amazon DynamoDB Accelerator (DAX): Speed Boost for NoSQL

Application Request

Applications first send requests to **DAX**.

Cache Check

If data is cached, DAX responds instantly.

Database Fetch

If not, DAX fetches the data from **DynamoDB**, caches it, and returns it.

DAX (DynamoDB Accelerator) is a **fully managed in-memory caching layer** designed specifically for DynamoDB.

It improves read performance by up to **10x**, reducing latency from **milliseconds to microseconds**.

⚡ Key Benefits:

In-memory cache: Reduces the need to repeatedly query the main DynamoDB table.

Highly scalable and secure: Built to handle large-scale traffic and integrated with AWS security services.

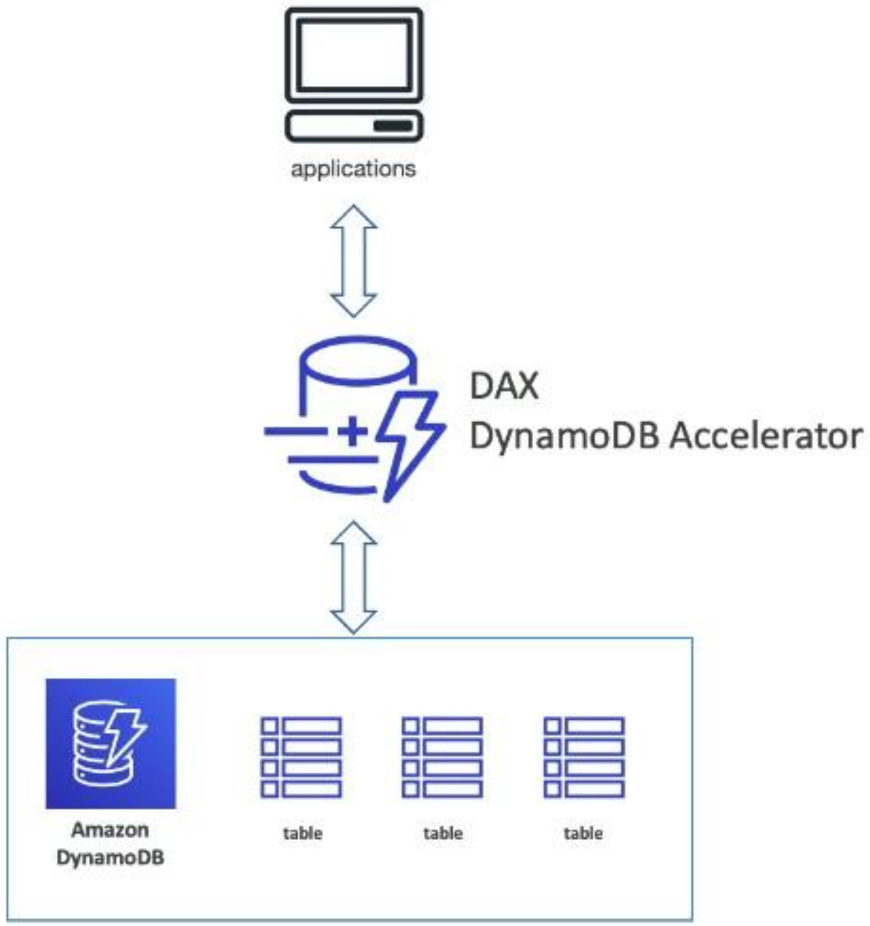
Drop-in replacement for DynamoDB SDK: Applications use DAX without major code changes.

🔍 DAX vs. ElastiCache:

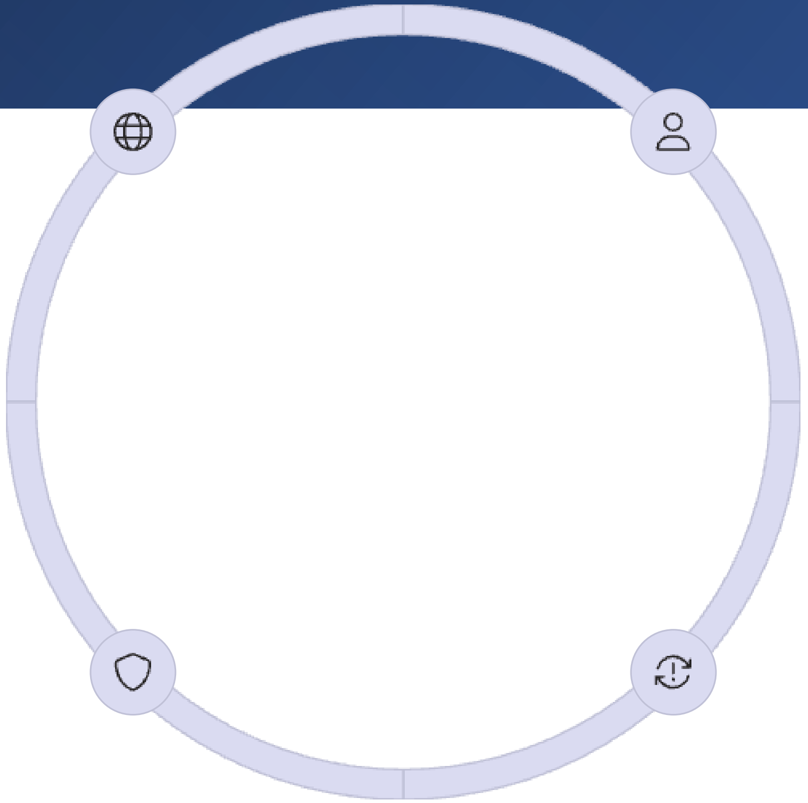
DAX is only used with DynamoDB and tightly integrated into its architecture.

ElastiCache is more general-purpose and can cache results from any type of database (e.g., RDS, Redshift, etc.).

✅ Ideal for **read-heavy workloads** where speed is critical, such as gaming, ad tech, and mobile apps.



DynamoDB Global Tables: Real-Time Multi-Region Access



Global Deployment

A **global table** is deployed in multiple regions (e.g., **us-east-1** and **eu-west-3**).

Low-Latency Access

Any **user request** (read or write) is served by the **nearest DynamoDB DynamoDB region**, reducing response time.

Disaster Recovery

If one region fails, traffic can **seamlessly switch to another region**— **region**—ensuring high availability and disaster recovery.

Automatic Synchronization

Data is **automatically synchronized** between all regions using **two-way replication**.

Global Tables in DynamoDB allow data to be stored and **accessed in multiple AWS regions** with **low latency**.

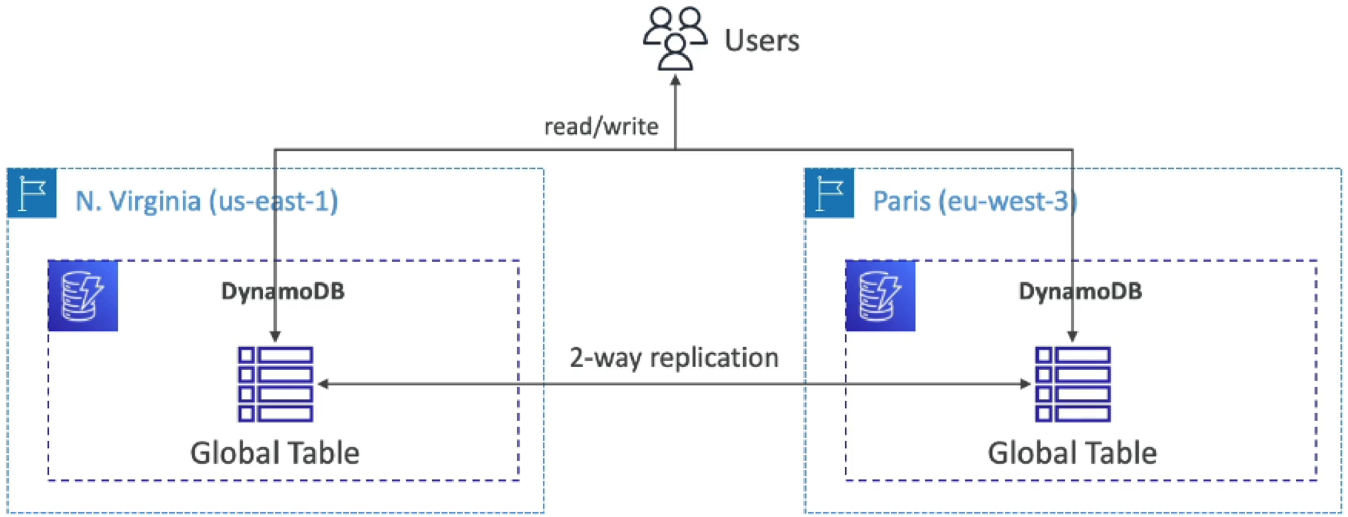
This enables **read and write operations from any region** using a model called **Active-Active replication**.

✅ **Key Benefits:**

Low-latency access for globally distributed applications.

Active-Active architecture: all regions can serve writes.

Built-in replication keeps data consistent across all locations.



📌 Ideal for global applications like e-commerce, gaming, and social platforms where users access data from various continents.

What Is Docker? — A Simple Overview

Containerization

Containers bundle everything the app needs: code, libraries, settings, system tools.

Consistent Environments

Apps run the **same everywhere** — on laptops, servers, or cloud.

Fast Deployment

Fast deployment — containers can start or stop in **seconds**.

Easy Scaling

Easier to scale applications up or or down as needed.

Docker is a platform that helps **developers package and run applications** inside lightweight, portable units called **containers**.

📦 What Are Containers?

Containers bundle everything the app needs: code, libraries, settings, system tools.

They can be **run on any system** — no matter the underlying hardware or operating system.

✅ Why Docker Is Powerful:

No compatibility issues — eliminates "it works on my machine" problems.

Predictable performance and **fewer bugs** due to consistent environments.

Compatible with **any language, OS, or tech stack**.

💡 Docker simplifies the development-to-production journey and enables rapid, reliable software delivery.

Docker on an Operating System (OS)

A **Docker container** is a lightweight unit that includes an application and all its dependencies.

Containers run on top of a **host operating system**—for example, a Linux OS on an **EC2 instance** (virtual server).

🧱 What the Diagram Shows:

A **server (EC2 instance)** is running **multiple containers** side by side.
Each container runs a **different application or service**, such as:

- 🟦 Java-based apps (Java logo)
- 🟢 Node.js apps (Node logo)
- 🐬 MySQL database (MySQL logo)

All containers **share the host OS** but remain **isolated** from each other.

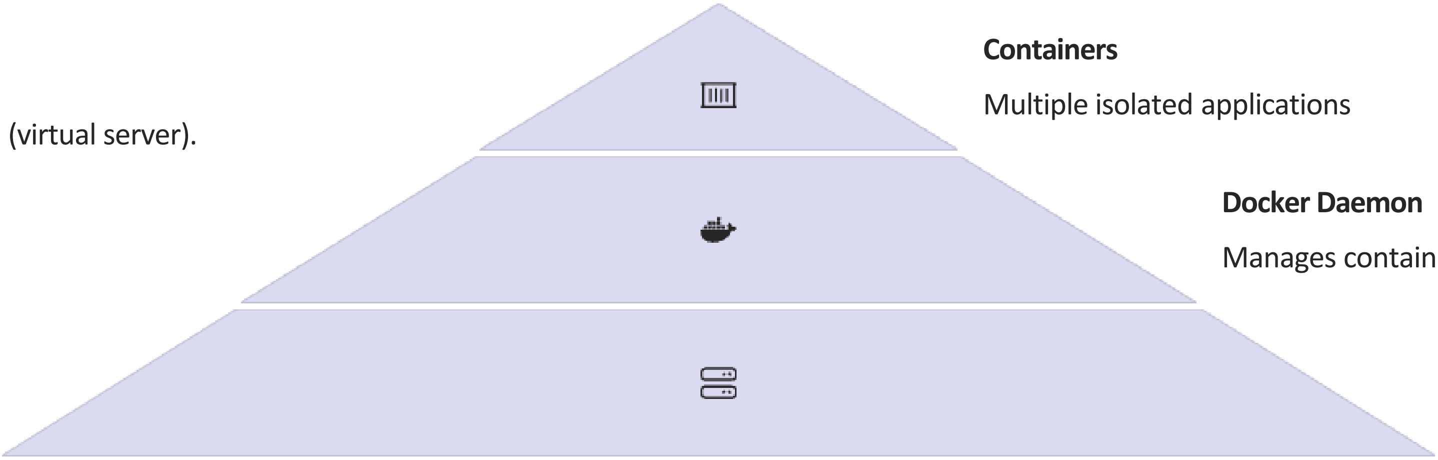
🔄 Benefits of This Setup:

Applications are **independent** and don't conflict, even if they use different languages.

Resources are used **efficiently**—no need to spin up separate VMs for each service.

Faster startup time, **easy to scale**, and **easy to manage**.

✅ This architecture is commonly used in microservices environments to run multiple services on a single machine.



Containers

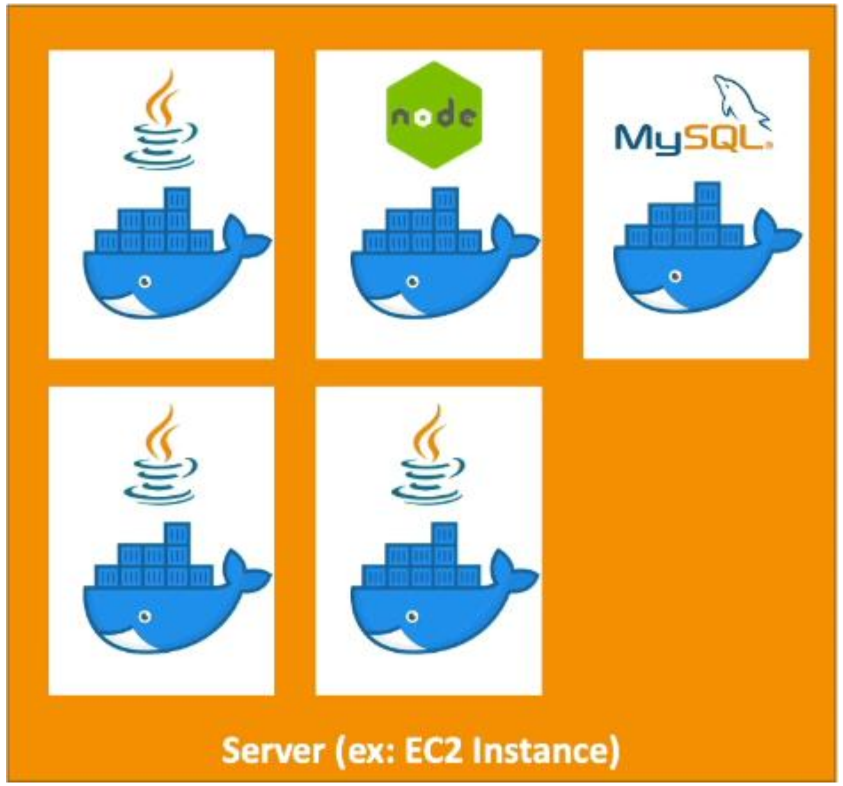
Multiple isolated applications

Docker Daemon

Manages containers

Host OS

Shared operating system



Where Are Docker Images Stored?

Public Repositories
Most popular: **Docker Hub**

Contains **base images** like:

•Ubuntu (Operating System)

•MySQL (Database)

•NodeJS, Java (Runtime environments)

Useful for quickly starting common applications.

Docker images are saved in **repositories**, which are locations that store, share, and manage images.

📤 **Two Main Types of Repositories:**

1. **Public Repositories**

2. **Private Repositories**

🧠 **Summary:**

Docker images = packaged applications.

Repositories = cloud storage places where images live.

Public (for sharing) vs. Private (for security & control).

📦 When running a container, Docker pulls the image from the specified repository first.

Private Repositories

Example: **Amazon ECR (Elastic Container Registry)**

Used in enterprise settings to securely store custom or sensitive Docker images.

Offers **access control** and **integration** with AWS services.

Virtual Machines

Virtual Machines vs. Docker Containers

Each VM runs **its own Guest OS** (e.g., Linux, Windows).

A **hypervisor** (like VirtualBox or VMware) sits between the Host the Host OS and VMs.

Heavier and slower: More memory and CPU are used due to due to multiple OS layers.

Docker Containers

All containers **share the same Host OS**.

The **Docker Daemon** runs directly on the Host OS to manage manage containers.

Lightweight and fast: No need for full OS for each app.

Clustering

- Main Clustering Ideas:
  - Use features to decide which points are most similar to other points.
  - Realize that there is no final correct **y** label to compare cluster results to.
  - We can think of clustering as an unsupervised learning process that “discovers” potential labels.

Clustering

- Unsupervised Learning Paradigm Shift:
    - *How do we assign a new data point to a cluster?*
        - Different approaches depending on the unsupervised learning algorithm used.
        - Use features to assign most appropriate cluster.

Clustering

- Unsupervised Learning Paradigm Shift:
    - *How do we assign a new data point to a cluster?*
        - Just as before, no way to measure if this was the “correct” assignment.