

# Lecture 8: AWS NoSQL Database

**Maharishi International University**

**Department of Computer Science**

**M.S. Thao Huy Vu**

# Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University (MIU).

# Agenda

- **Introduction to NoSQL Services in AWS**
- **Amazon DynamoDB**
- **Amazon DocumentDB**
- **Amazon ElastiCache**
- **Amazon Neptune**

# Introduction to NoSQL Services in AWS

- SQL Databases:
  - Relational Databases: Schema-based
  - Strong consistency
- NoSQL Databases:
  - Non-relational Databases: Schema-less
  - Eventual consistency.
- NoSQL & SQL map:
  - Collection – Table
  - Document – Row
  - Key – Column
  - Value – Column Value

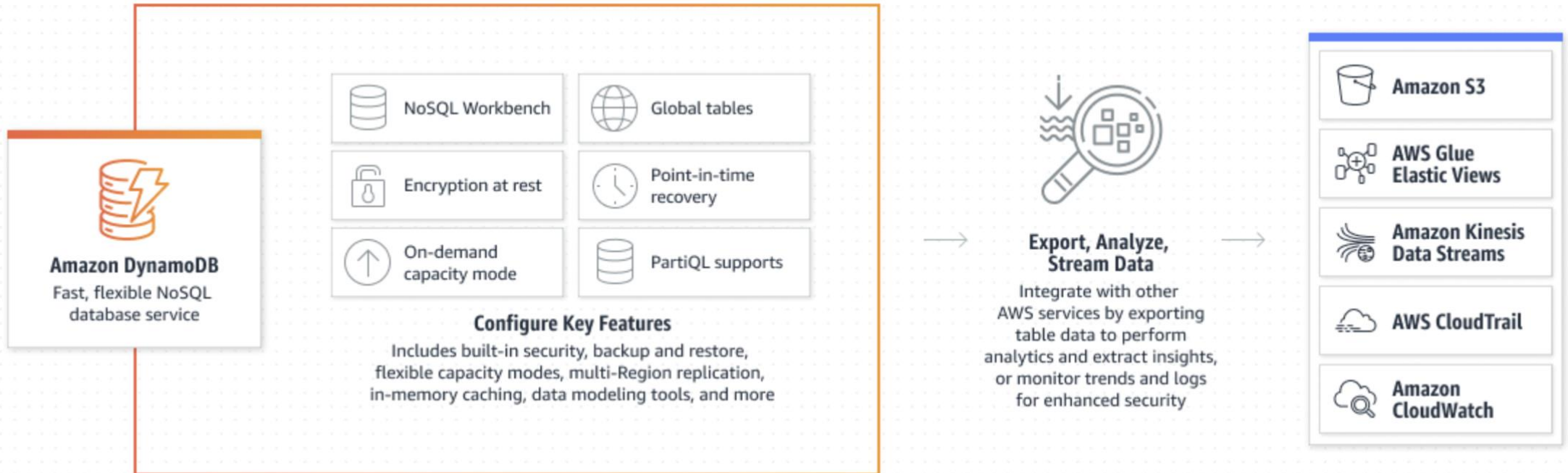
# Introduction to NoSQL Services in AWS

- Factors to choose NoSQL database
  - Model: Key-value, document, wide-column.
  - Access patterns.
  - Performance requirements.
  - Specific application needs.
- Use cases for NoSQL database:
  - Web-scale Applications: DynamoDB.
  - Content Management Systems: DocumentDB.

# Major Types of NoSQL Databases

- **Key-Value Databases:**
  - **Service:** Amazon DynamoDB
  - **Use Case:** Serverless apps, IoT, user profiles, shopping carts
- **Document Databases:**
  - **Service:** Amazon DocumentDB (with MongoDB compatibility)
  - **Use Case:** Content management, catalogs, user profiles
- **Graph Databases:**
  - **Service:** Amazon Neptune
  - **Use Case:** Social graphs, fraud detection, recommendation engines
- **In-Memory Databases: Amazon ElastiCache**
  - **Service:** Amazon ElastiCache (Redis & Memcached)
  - **Use Case:** Real-time analytics, session stores, gaming leaderboards
- **Wide-Column Databases**
  - **Service:** Amazon Keyspaces (for Apache Cassandra)
  - **Use Case:** Time-series data, event logs, telemetry
- **Search Databases: Amazon Elasticsearch Service**
  - **Service:** Amazon OpenSearch Service (formerly Elasticsearch Service)
  - **Use Case:** Full-text search, log analysis, observability

# Amazon DynamoDB



# Amazon DynamoDB

- A fully managed NoSQL database service designed for document and key-value store models.
- **High Performance:**
  - Delivers consistent single-digit millisecond latency.
  - Optimized for high-read and high-write workloads.
- **Scalability:**
  - Automatically scales to handle virtually unlimited requests or data size.
  - Suitable for applications ranging from small workloads to enterprise-scale systems.
- **Durability and Availability:**
  - Built-in multi-region replication and backup for fault tolerance.
  - Ensures high availability with no downtime for maintenance.
- **Serverless:**
  - No need to provision or manage servers.
  - Automatically adjusts capacity based on workload.



# Amazon DynamoDB

- **Data Model:**

- DynamoDB is a key-value and document database.
- Stores data in tables with items (rows) and attributes (columns), offering flexibility for unstructured or semi-structured data.

- **Indexing:**

- **Primary indexes:** Defined using a **partition key** or a combination of **partition key + sort key**.
- **Secondary indexes:**
  - **Global Secondary Indexes (GSI)**
  - **Local Secondary Indexes (LSI)**

- **Read and Write Consistency Models:**

- **Eventually Consistent Reads:** Fast, less resource-intensive.
- **Strongly Consistent Reads:** Guarantees the most recent data but may have higher latency.

# Amazon DynamoDB

- **Provisioned and On-Demand Capacity Modes:**
  - **Provisioned Mode:** Set specific read/write capacity units, suitable for predictable workloads.
  - **On-Demand Mode:** Automatically adjusts capacity for unpredictable or spiky workloads.
- **DynamoDB Streams:**
  - Captures logs of data changes (insert, update, delete) in real-time.
  - Useful for building event-driven applications or enabling cross-region replication.
- **Backup and Restore:**
  - Provides on-demand and continuous backups with point-in-time recovery.
  - Ensures data durability and protection against accidental deletes or corruption.
- **Security:**
  - Supports encryption at rest and in transit.
  - Integrates with AWS IAM for fine-grained access control and AWS KMS for encryption keys.

# DynamoDB – Flexible Table

- **Partition Key:**

- Determines the physical storage location of data.

- **Sort Key:**

- Orders items within a partition.
- Allows for additional granularity when querying items that share the same Partition Key.

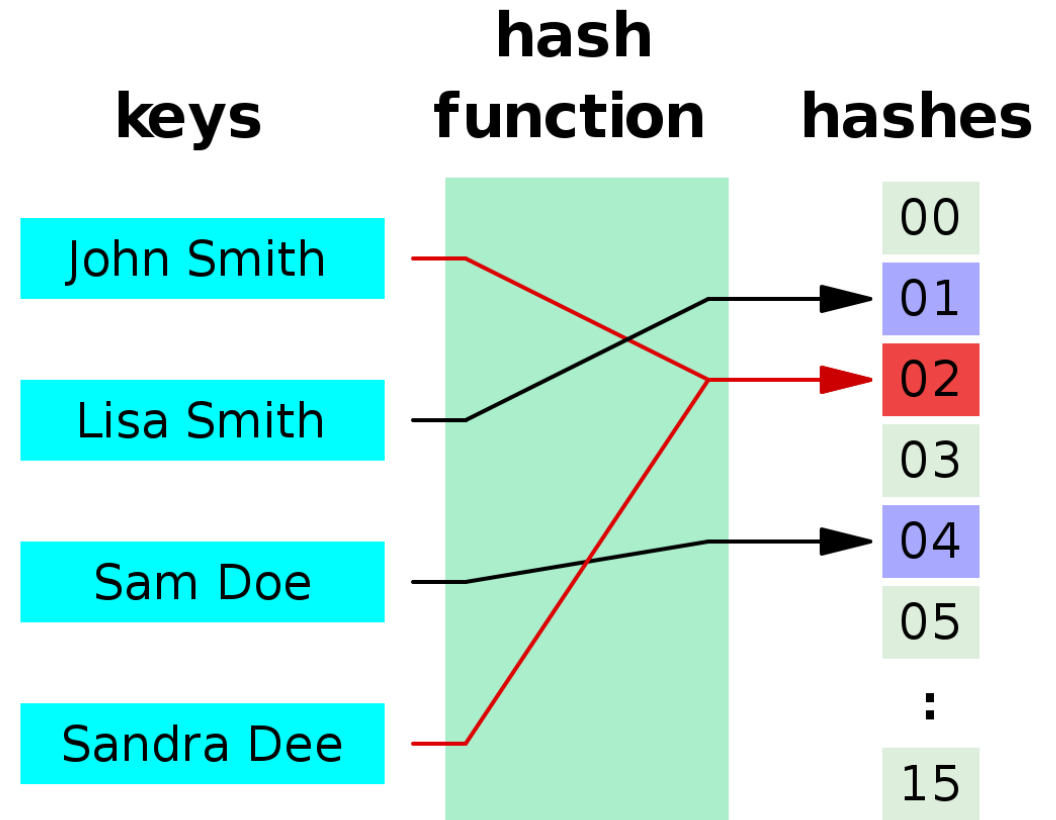
- **Primary Key:**

- Combines the **Partition Key** and **Sort Key** for composite key tables.
- Ensures uniqueness of each item in the table by combining both attributes.

# Partition Key

- **Hash Function:**

- Converts the **Partition Key** value into a numerical hash value.
  - This hash value maps to a specific partition in the database, corresponds to a physical **partition** (a storage location within DynamoDB).
- Items with the same partition key will hash to the same partition, ensuring efficient storage and retrieval.



# High Performance at Scale with Modern Databases

- Problem: Relational databases can become slow at scale due to:
  - The overhead of joins to combine data from multiple tables.
  - Disk I/O bottlenecks for complex queries.
  - Locking mechanisms for maintaining transaction consistency.
  - Rigid schemas that require costly migrations when data patterns change.
- Solution: Modern NoSQL databases like DynamoDB, Cassandra, and MongoDB use **denormalization**:
  - Related data is stored together in a single table or document.
  - Eliminates the need for joins, leading to faster performance at scale.
  - Designed for distributed systems to handle large-scale workloads efficiently.

# High Performance at Scale with Modern Databases

- DynamoDB: The **Sort (Range) Key** provides:
  - **Organization of related items** within the same Partition Key, enabling structured data storage.
  - **Efficient querying of related data** using range queries (e.g., BETWEEN, <, >, begins\_with).
  - Facilitates **one-to-many relationships**:
    - **Example:** A StudentID and CourseID combination allows tracking multiple students per course and multiple courses per student.
  - The **Partition Key and Sort Key together form a composite key** that uniquely identifies each item in the table.

# Composite primary key

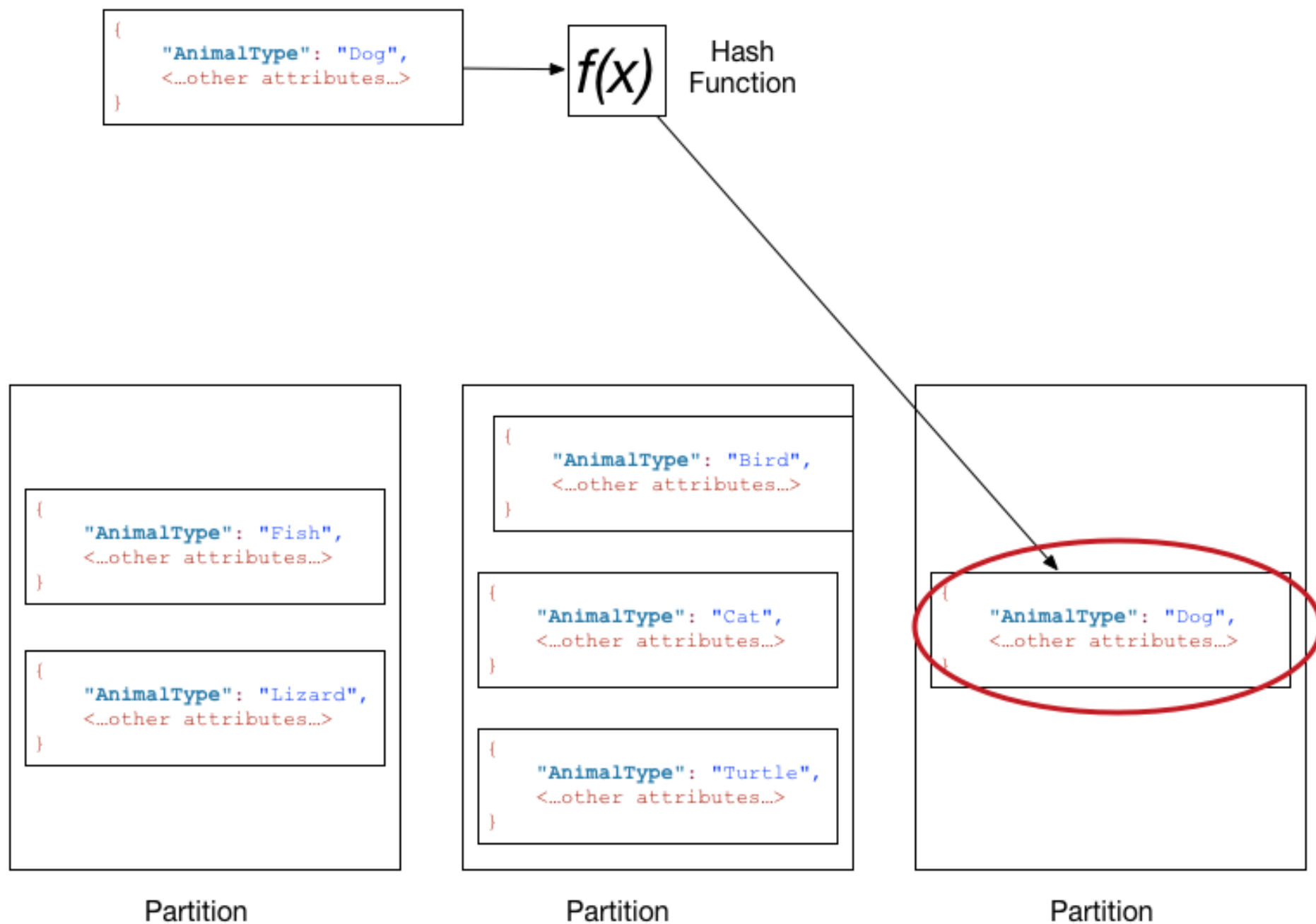
- **Composite Primary Key:**

- Consists of two attributes: **Partition Key** and **Sort Key**.
- To retrieve an item, both the **Partition Key** and **Sort Key** must be provided.

- **Partition Key:**

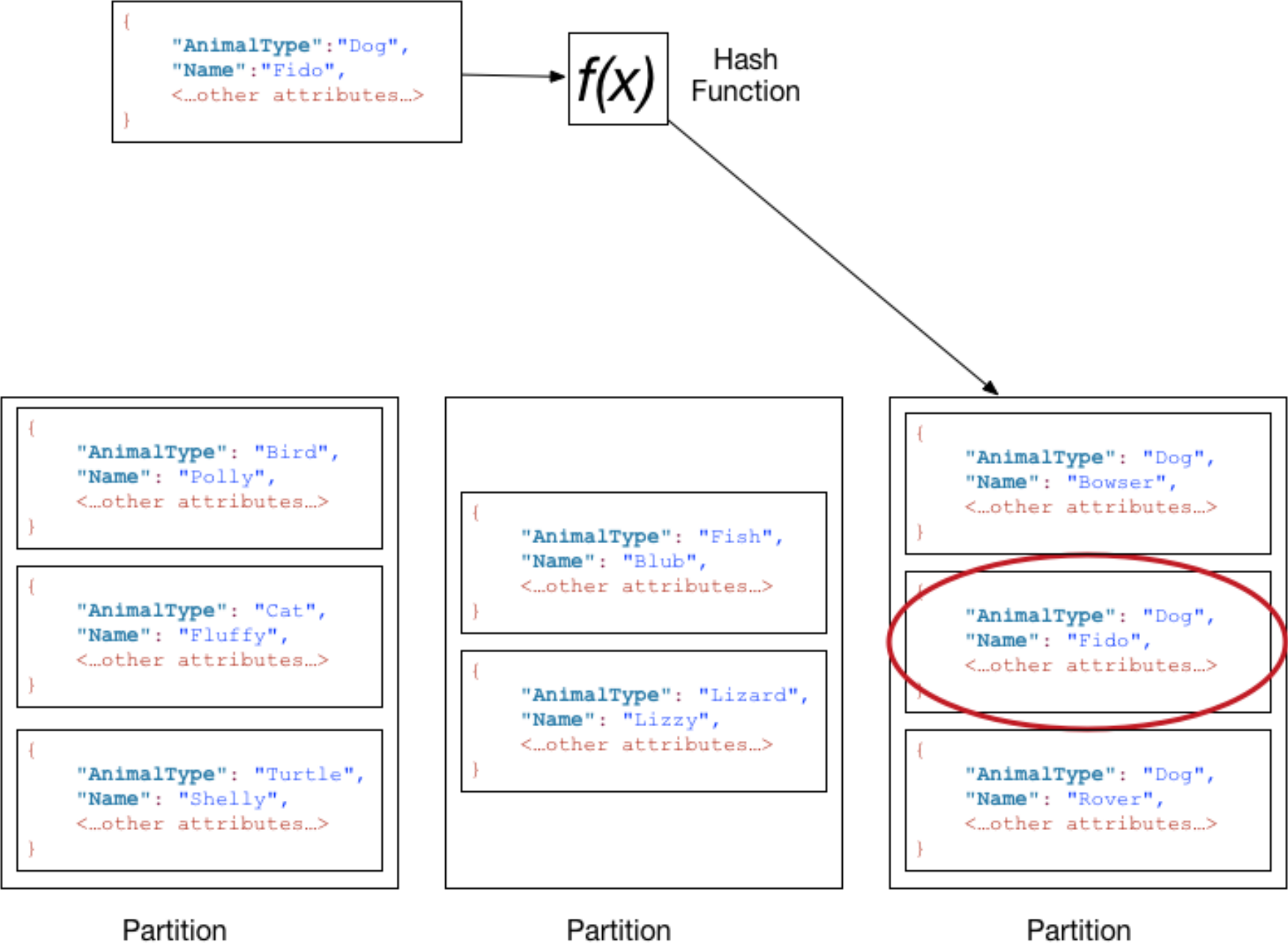
- Used as input to an internal hash function to determine the physical storage (partition).
- Items with the same **Partition Key** are stored together in the same partition.

- **Sort Key:** Organizes items within a partition in sorted order, enabling efficient querying and retrieval.





# With Sort Key (Name)



# Secondary Indexes

- **Secondary Indexes** allow you to query data using attributes other than the primary key. They provide flexibility for retrieving data efficiently based on alternate access patterns.
- **Global Secondary Index (GSI):**
  - Allows querying data using a **different partition key and optional sort key** than the primary key.
  - Supports **eventual consistency** for queries.
- **Local Secondary Index (LSI):**
  - Uses the **same partition key** as the primary key but a **different sort key**.
  - Limited to 5 LSIs per table and must be defined at table creation.
  - Supports **strongly consistent reads**.

## GameScores

UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
"101"	"Galaxy Invaders"	5842	"2015-09-15:17:24:31"	21	72	...
"101"	"Meteor Blasters"	1000	"2015-10-22:23:18:01"	12	3	...
"101"	"Starship X"	24	"2015-08-31:13:14:21"	4	9	...
"102"	"Alien Adventure"	192	"2015-07-12:11:07:56"	32	192	...
"102"	"Galaxy Invaders"	0	"2015-09-18:07:33:42"	0	5	...
"103"	"Attack Ships"	3	"2015-10-19:01:13:24"	1	8	...
"103"	"Galaxy Invaders"	2317	"2015-09-11:06:53:00"	40	3	...
"103"	"Meteor Blasters"	723	"2015-10-19:01:13:24"	22	12	...
"103"	"Starship X"	42	"2015-07-11:06:53:00"	4	19	...
...	...	...	...	...	...	

# Secondary Indexes

- GameScores tracks users and scores for a mobile gaming application. Each item is identified by a partition key (UserId) and a sort key (GameTitle).
- Problem: Querying TopScore by **GameTitle** requires scanning all records, which becomes inefficient as the table grows.
- To improve query performance, you can create a GSI with:
  - **GameTitle** as the partition key.
  - **TopScore** as the sort key.
- DynamoDB automatically includes the table's primary key attributes (UserId and GameTitle) in the GSI, making it easy to fetch full records.

# DynamoDB pricing Model

- On-Demand Capacity Mode
  - **Write:** \$1.25 per million write request units (WRUs)
  - **Read:** \$0.25 per million read request units (RRUs)
- Provisioned Capacity Mode
  - RDU: \$0.00013/RDU/hour
  - WDU: \$0.00065/WCU/hour

# Amazon DynamoDB

- Demo

DynamoDB

Dashboard

Tables

Explore items

PartiQL editor

Backups

Exports to S3

Imports from S3

Integrations New

Reserved capacity

Settings

▼ DAX

Clusters

Subnet groups

Parameter groups

Events

DynamoDB > Tables > Create table

Create table

Table details Info

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Enter name for table

Between 3 and 255 characters, containing only letters, numbers, underscores (\_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

Enter the partition key name

String

1 to 255 characters and case sensitive.

Sort key - optional

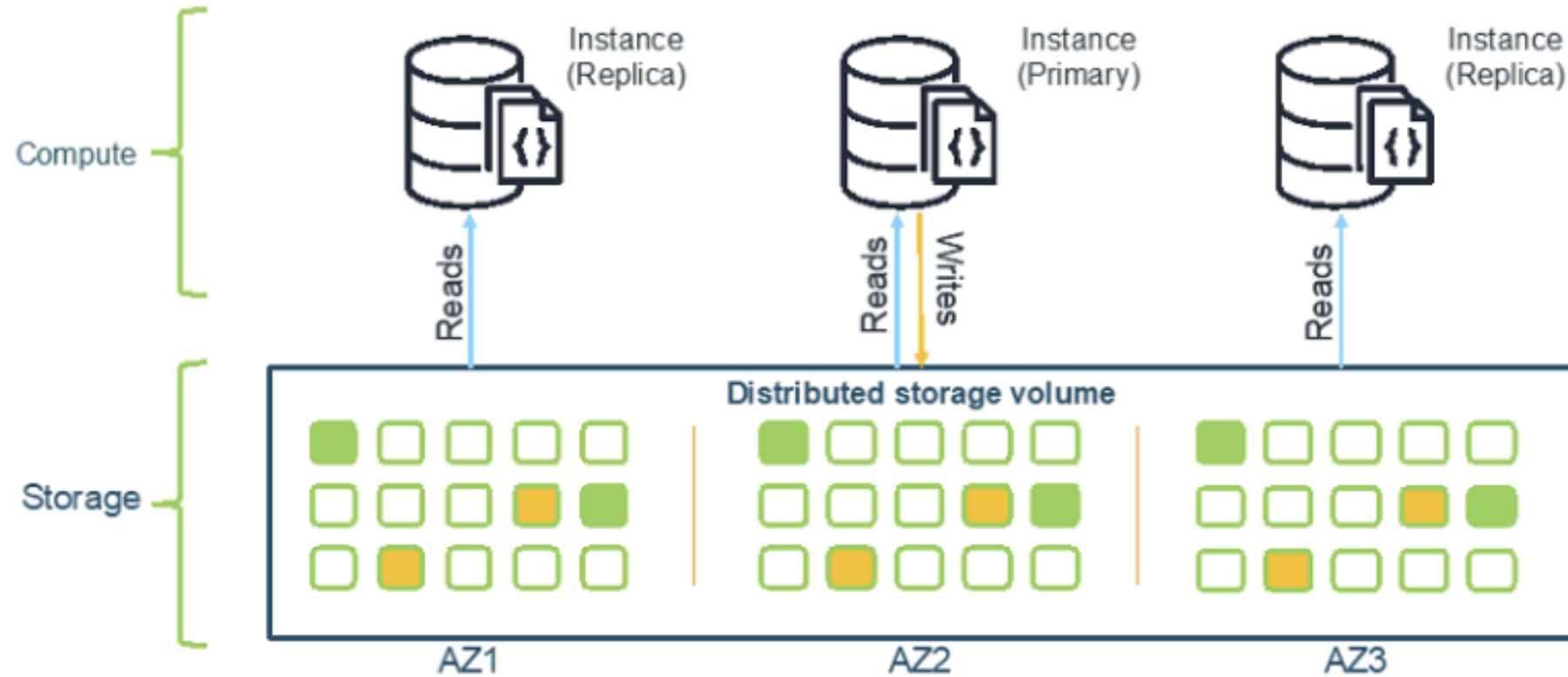
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

Enter the sort key name

String

1 to 255 characters and case sensitive.

# Amazon DocumentDB




# Amazon DocumentDB

- Amazon DocumentDB is a fully managed NoSQL document database service designed for scalable, secure, and high-performance applications.
- **Compatible with MongoDB:** Minimize changes to existing application code or tools.
- **Highly Scalable:**
  - Automatically scales storage up to **64 TB** per database cluster as your application grows.
  - Supports up to **15 read replicas** to scale read operations for high-traffic workloads.
- **Fault Tolerant:**
  - Stores six copies of your data across three AWS Availability Zones to ensure durability and high availability.
  - Provides automated failover to minimize downtime in the event of a failure.
- **Highly Secure:**
  - Provides encryption at rest and in transit using AWS Key Management Service (KMS).
  - Supports fine-grained access control using AWS Identity and Access Management (IAM).



# Amazon DocumentDB

- Demo



[DocumentDB](#) > [Clusters](#) > Create cluster

## Create Amazon DocumentDB cluster

### Cluster type

☒ **Instance Based Cluster**

Instance based cluster can scale your database to millions of reads per second and up to 128 TiB of storage capacity. With instance based clusters you can choose your instance type based on your requirements.

☐ **Elastic Cluster**

Elastic clusters can scale your database to millions of reads and writes per second, with petabytes of storage capacity. Elastic clusters support MongoDB compatible sharding APIs. With Elastic Clusters, you do not need to choose, manage or upgrade instances.

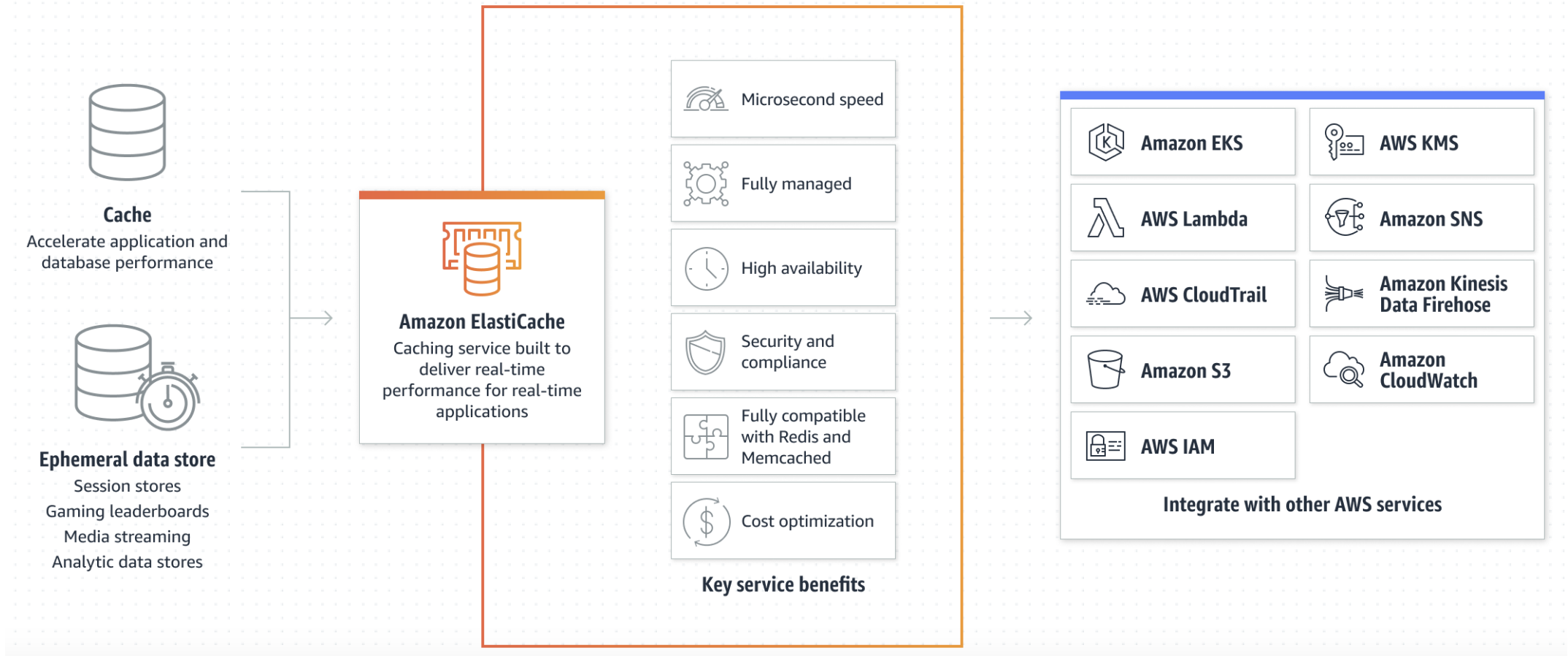
### Configuration

Cluster identifier [Info](#)

Specify a unique cluster identifier.

Engine version

# Amazon ElastiCache



# Amazon ElastiCache

- **Amazon ElastiCache** is a fully managed in-memory caching service that accelerates application performance by reducing data access latency.
- **Reducing Latency:** Stores frequently accessed data in memory, allowing applications to retrieve data in microseconds instead of querying slower storage systems.
- **Decreasing Database Load:** Offloads repetitive read operations from primary databases, reducing their workload and improving overall efficiency.
- **Improving Application Response Times:** Enables faster response times by serving cached results for high-demand or frequently accessed data.

# Amazon ElastiCache

- **Caching Strategies:**

- **Write-Through:** Automatically updates the cache when the underlying database is updated.
- **Lazy Loading:** Data is only cached when requested, reducing unnecessary cache population.
- **Time-to-Live (TTL):** Expiration policies to automatically evict stale data from the cache.

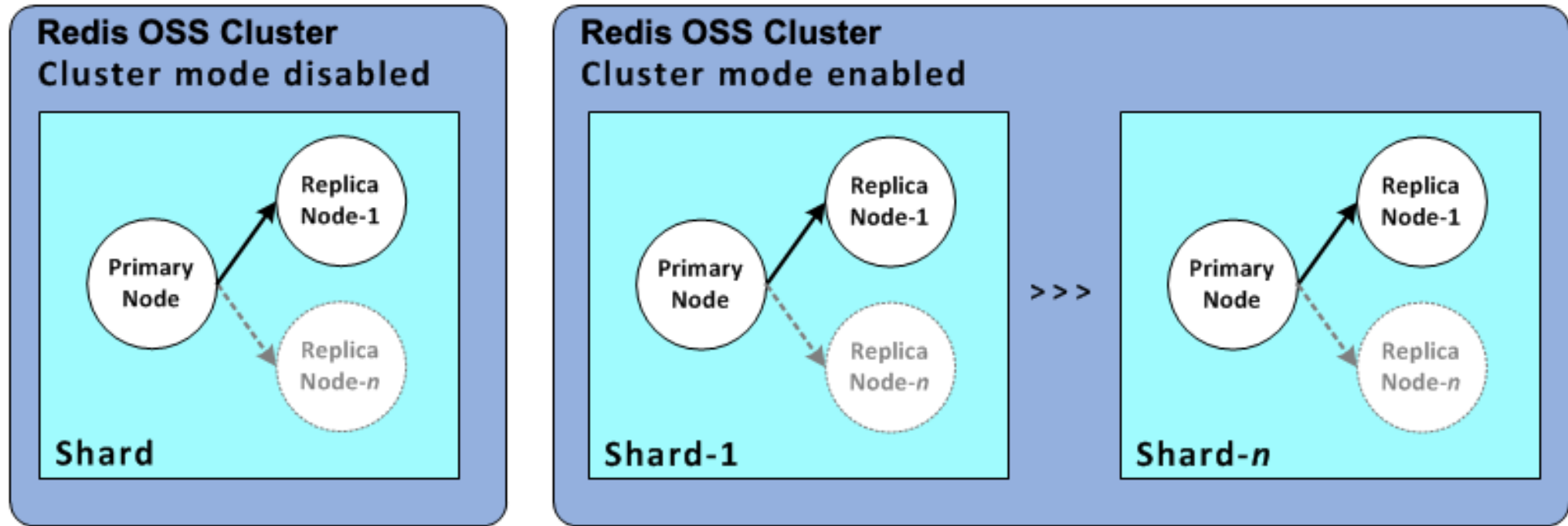
- **Cache Engines:**

- **Redis:**
  - Advanced features like data persistence, pub/sub messaging, and clustering.
  - Ideal for use cases requiring complex data structures (e.g., sorted sets, hashes).
- **Memcached:**
  - Simpler, high-performance caching layer for basic key-value storage.
  - Ideal for horizontally scaling workloads.

- **Performance:**

- **Low Latency:** Microsecond response times for read and write operations.
- **High Throughput:** Supports millions of requests per second.
- **Scalability:** Easily scales horizontally by adding nodes to meet application demands.

# Amazon ElastiCache - Redis



# Amazon ElastiCache - Redis

- **Components:**

- **Node:** The basic building block of an ElastiCache cluster, containing CPU, memory, and network resources.
- **Cluster:** A collection of nodes working together. Can be a single-node cluster or a multi-node cluster for scalability and redundancy.
- **Shard:**
  - A **subset of data** within a Redis cluster.
  - Redis clusters support **sharding** to distribute data across multiple shards, improving scalability.
  - Each shard can have one primary node (write operations) and multiple replica nodes (read operations).

- **Security:**

- **Encryption:** Supports encryption in transit (TLS) and at rest to secure sensitive data.
- **Authentication:** Redis offers password-based access control through Redis AUTH.
- **Access Control:** Integrates with AWS Identity and Access Management (IAM) and Amazon Virtual Private Cloud (VPC) for secure access and network isolation.

# Amazon ElastiCache - Memcached

- Components:
  - Node: The basic building block of an ElastiCache cluster, providing in-memory, key-value storage.
  - Cluster: A collection of separate nodes. Nodes do not share state, and data partitioning is handled by the client.
- Separates data across nodes in a cluster.
- Does not support internal sharding, replication, or failover
- Security:
  - Authentication: Memcached does not support built-in authentication; access is controlled through the network layer (e.g., VPC, security groups).
  - Encryption: Does not support encryption in-transit or at rest natively.
  - Access Control: Integrates with AWS Identity and Access Management (IAM) and Amazon VPC for secure access and isolation.

# Amazon ElastiCache

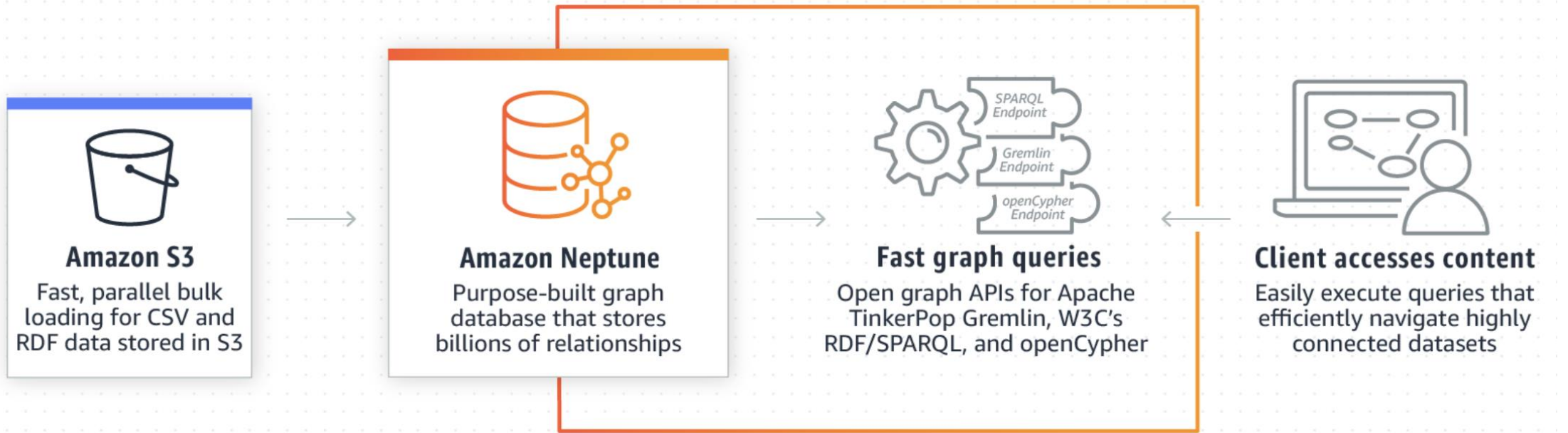
- **Cluster Configuration:**
  - **Single-Node Cluster:** Used for simple caching needs without redundancy.
  - **Multi-Node Cluster:**
    - Redis: Supports sharding and replication for scalability and fault tolerance.
    - Memcached: Supports horizontal scaling by adding nodes, but without replication.
  - **Parameter Groups:** Enable fine-tuning of cache engine settings for performance optimization. E.g: Evict the **least recently used (LRU)** keys when memory is full.



# Amazon ElastiCache – Use cases

- **Frequently Accessing Data:**
  - Cache frequently accessed data (e.g., user profiles, product catalogs) to reduce database queries and provide faster response times.
- **Session Management:**
  - Store user session data in memory to ensure low-latency access for applications like web portals and e-commerce platforms.
- **Database Query Optimization:**
  - Cache results of expensive or repetitive database queries to offload database load and improve application performance.
- **Real-Time Analytics:**
  - Perform lightning-fast computations and store real-time analytics data for use cases like gaming leaderboards, financial dashboards, or IoT telemetry.

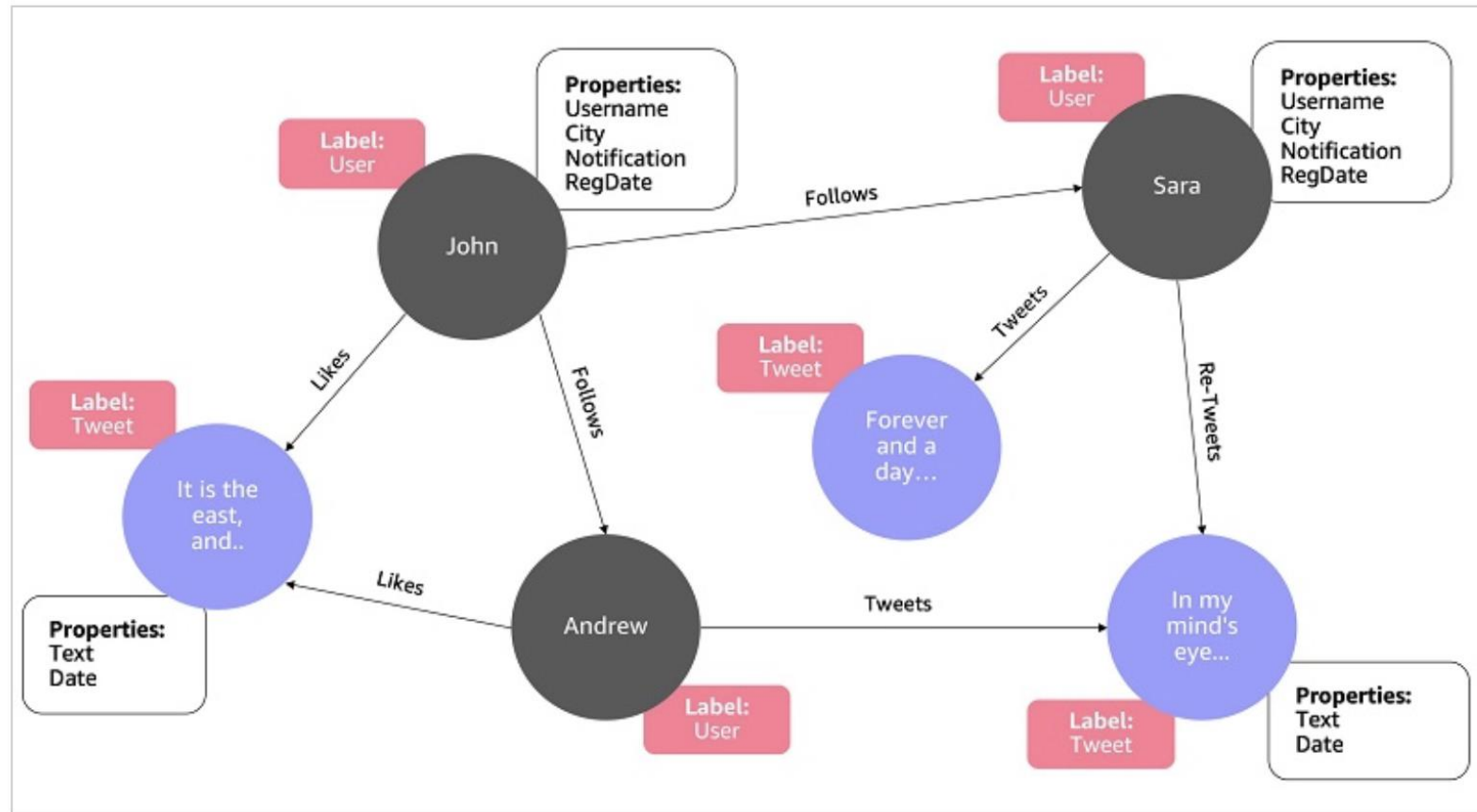
# Amazon Neptune



# Amazon Neptune

- **Amazon Neptune** is a fully managed graph database service designed for applications that work with highly connected datasets.
- **Graph Models:** Supports **Gremlin** and **SPARQL**.
- **High Performance:** Optimized for graph traversal, providing low-latency responses for complex relationship queries.
- **Scalability:** Scales to store billions of relationships and handle high query throughput.
- **Fault Tolerance:** Automatically replicates data across multiple Availability Zones with built-in backups.
- **Security:** Supports encryption at rest, in transit, and fine-grained access control via AWS IAM.

# Amazon Neptune



# Amazon Neptune - Components

- **Cluster Configuration:**

- **Instances:** Each Neptune cluster consists of a primary instance (read/write) and up to 15 read replicas for high availability and scalability.
- **Endpoints:**
  - **Cluster Endpoint:** Automatically redirects read and write traffic to the primary instance.
  - **Reader Endpoint:** Distributes read traffic across all replicas to optimize performance.
- **Replicas:** Enable fault tolerance and load balancing for read-heavy workloads.

# Amazon Neptune – Use cases

- **Knowledge Graphs:**

- Ideal for **AI applications** requiring a detailed view of relationships between customers, businesses, and other entities.
- Example: Building interconnected datasets for advanced insights and semantic search.

- **Fraud Detection:**

- Simplifies identifying complex fraudulent patterns that are challenging to detect with traditional relational databases.
- Example: Analyzing transactional relationships to uncover anomalies in financial systems.

- **Recommendation Engines:**

- Leverages relationships between data points to deliver **accurate and personalized recommendations**.
- Example: Suggesting products, content, or connections based on user interactions and preferences.

# References

- <https://docs.aws.amazon.com/>
- ChatGPT: <https://chatgpt.com/>
- Google AI: <https://gemini.google.com/app>
- Redis: <https://redis.io/docs/latest/commands>