NoSQL : Not only SQL

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Outline

- What is NoSQL?
- How is it different from SQL?
- Why do we need NoSQL?
- NoSQL Database types
- How to choose between SQL and NoSQL?
- Case Study: Amazon DynamoDB

Introduction to NoSQL

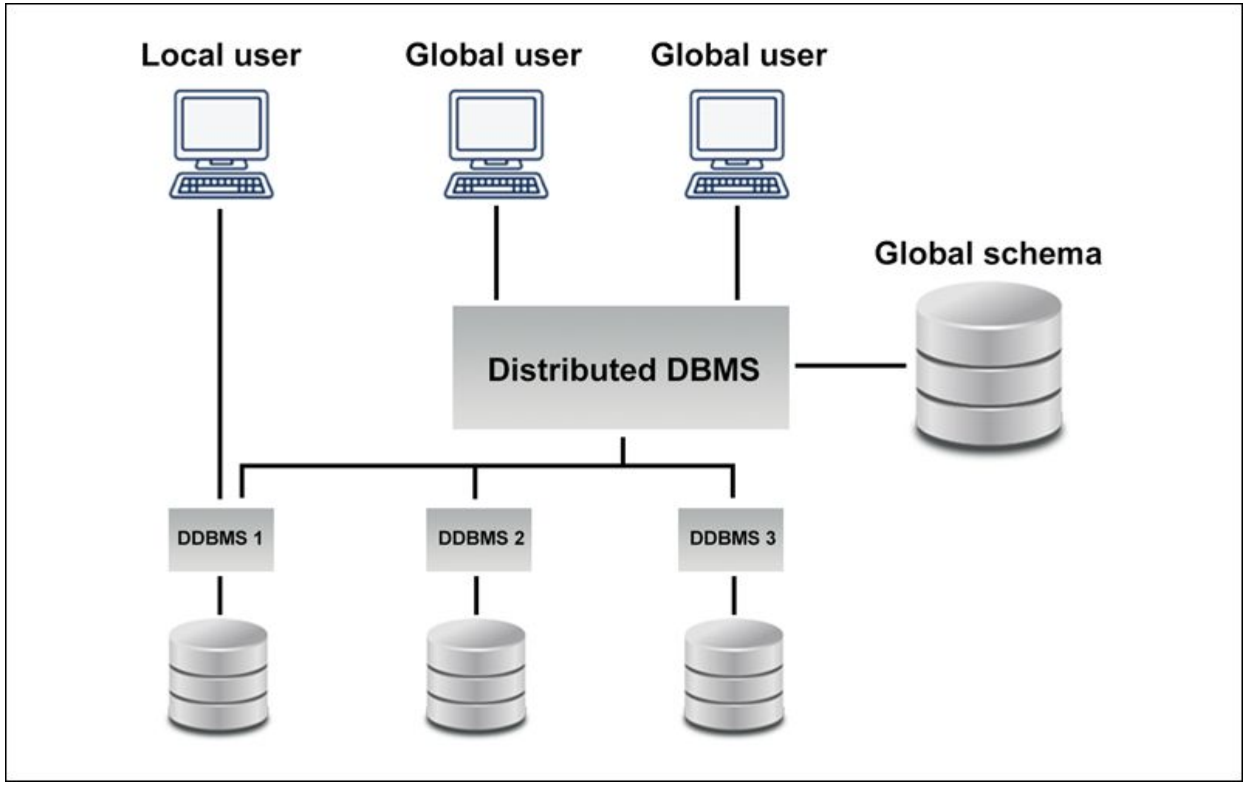
NoSQL

- Stands for “Not Only SQL”
- Basically a non-relational, schema-less and largely distributed database
- Developed in late 2000s to deal with limitations of SQL databases

Umm … What is a Distributed Database ?

Distributed Database

- Ever wondered how companies like Amazon manage their DB?
- Basically, Database is logically divided and distributed across multiple computers
- All these computers are connected in a network



Need of Distributed Databases

- Scalability
  - What if your database size exceeds 100GB?
  - Is read/write speed still same?
- Fault Tolerance and High Availability
  - What if your database system fails? Can it recover by itself ?
- Geographic Distribution
  - What if network latency increases b/w geographically distributed nodes?

NoSQL vs SQL

SQL	NoSQL
Supports Relationships and Joins	No support for Joins and relationships
High Maintenance Cost	Low Maintenance Cost
Predefined Schema	Dynamic Schema
Vertically Scalable	Horizontally Scalable
Follows ACID property	Does not follow ACID property
Eg: PostgreSQL, MySQL etc.	Eg: Cassandra, Neo4j etc.

But … Why should you choose NoSQL?

Benefits of NoSQL

- Agility
  - SQL has a fixed data model hence, does not support agile development
  - A key principle of agile development is the ability to adapt to changing application requirements
  - NoSQL being able to support dynamic schema, supports agile development

- Handling Unstructured Data
  - NoSQL supports dynamic Schema, hence can handle unstructured data
  - SQL needs relationship between different data to be able perform ‘Joins’

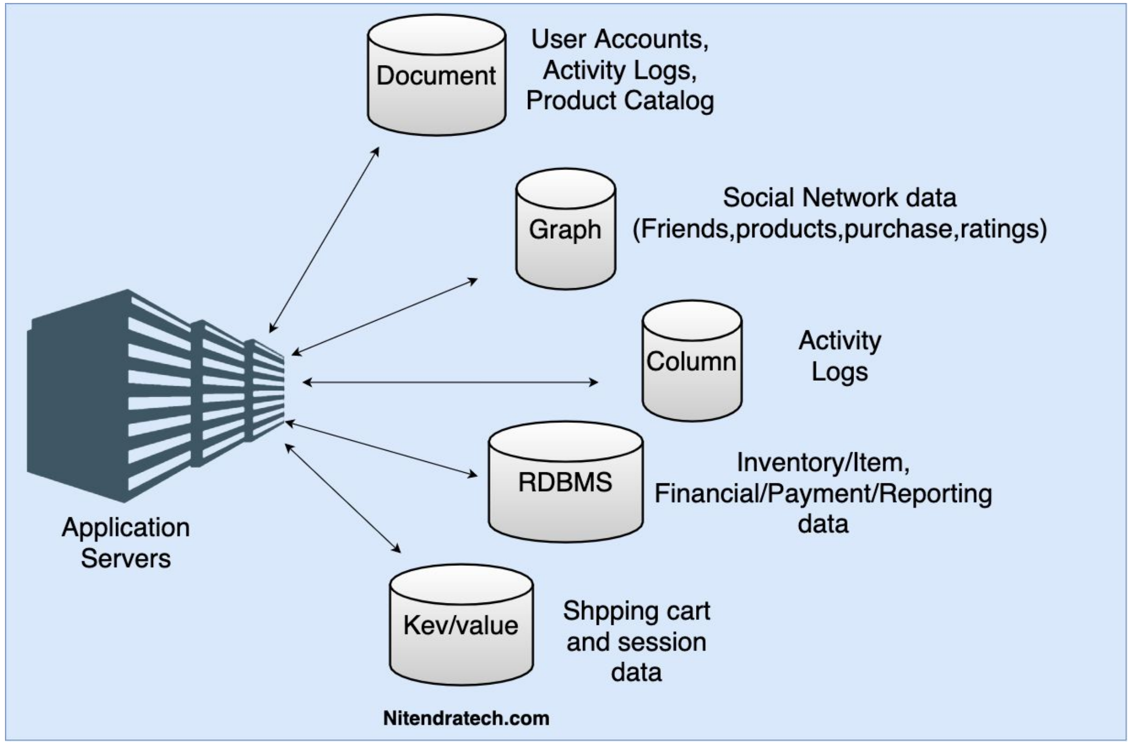
Source: Link

Benefits of NoSQL

- Scalability
  - NoSQL supports Horizontal Scaling (add more commodity servers or cloud instances)
  - SQL does not support horizontal scaling (Why?)
  - Vertical scaling requires significant additional engineering (like making joins faster)
  - Examples:
    - Games like Pokemon Go, Clash of Clans etc. stores data of millions of users
    - IoT devices:
      - More than billion IoT devices are connected to the Internet
      - This data is semi-structured and continuous

Benefits of NoSQL

- Auto-Sharding
  - NoSQL databases often comes with built in auto-sharding features
  - This is essential for horizontal scaling
- Polyglot Persistence
  - Means when when storing data use multiple data storage technologies, chosen based on the way data is used
  - Similar to Polyglot Programming



NoSQL Tradeoffs

Now, the question is what are we losing ?

- No Relationship among data ⇒ No Joins
- However, we are losing something more ⇒ consistency (What !!)
- CAP Theorem
  - *Consistency:* Once data is written, all future read requests will contain that data
  - *Availability:* The database is always available and responsive
  - *Partition Tolerance:* One part of the database can go down without affecting others
- This theorem says that in a distributed we can choose only 2 out C, A and P.
- NoSQL ensures *availability* and *partition-tolerance*
- However it ensures *eventual consistency*

Outline

- What is NoSQL?
- How is it different from SQL?
- Why do we need NoSQL?
- **NoSQL Database types**
- How to choose between SQL and NoSQL?
- Case Study: Amazon DynamoDB

NoSQL Database Types

4 types of NoSQL DB:

- Document Based
  - Uses collections and documents rather than tables and rows
  - Usual formats: XML, JSON, BSON
  - Use cases: CMS, blogging platforms, real-time analytics, ecommerce-applications
  - Examples: MongoDB, CouchDB, Amazon DocumentDB etc.
- Graph Based
  - Used to store information about networks of data, such as social connections
  - Examples: Neo4j, Giraph etc.

NoSQL Database Types (contd.)

- Key-Value Pairs
  - Similar to hash tables with a unique key and pointer to a data (usually BLOBs)
  - Use Case: maintaining session info, user profiles, preferences, shopping cart etc.
  - Examples: Redis, Amazon DynamoDB, Facebook's Memcached etc.
  - Note: Avoid using K-V pairs if you want to query by data
- Column based
  - Data is arranged as columns instead of rows, with keys pointing to multiple columns
  - Supports efficient representation of sparse data
  - Designed to store and process large amounts of data distributed over many machines
  - Examples: Apache Cassandra, HBase etc.

How to choose between SQL and NoSQL?

Criteria	Use Case	SQL	NoSQL
ACID Compliance	Banking Systems, Inventory Management Systems	Suitable: SQL ensures ACID compliance	Not Suitable (No ACID compliance)
Complex Queries	Reporting, analytics, and data manipulation	Suitable: Supports complex queries with JOINs	Not Suitable: Best for simple queries and fast lookups
Scalability	Handling large amounts of data	Not Suitable (Vertical Scaling)	Suitable (Horizontal Scaling)
Data Relationships	E-commerce System: Managing products, categories, and customers.	Suitable	Not Suitable
Data Variety	Structures (ERP), Unstructured (Big Data Applications)	Suitable for Structured data	Suitable for Unstructured Data

Outline

- What is NoSQL?
- How is it different from SQL?
- Why do we need NoSQL?
- NoSQL Database types
- How to choose between SQL and NoSQL?
- **Case Study: Amazon DynamoDB**

DynamoDB

Amazon DynamoDB

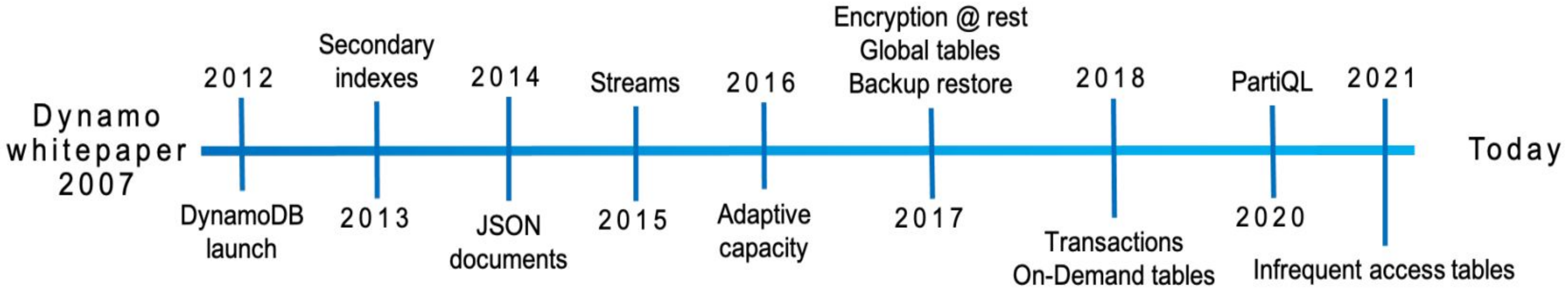
Let's begin with a story:

In 2021, there was a 66-hour Amazon Prime Day shopping event

- The event generated some staggering stats:
- Trillions of API calls were made to the database by Amazon applications
- The peak load to the database reached 89 million requests per second
- The DB provided single-digit ms performance while maintaining high availability

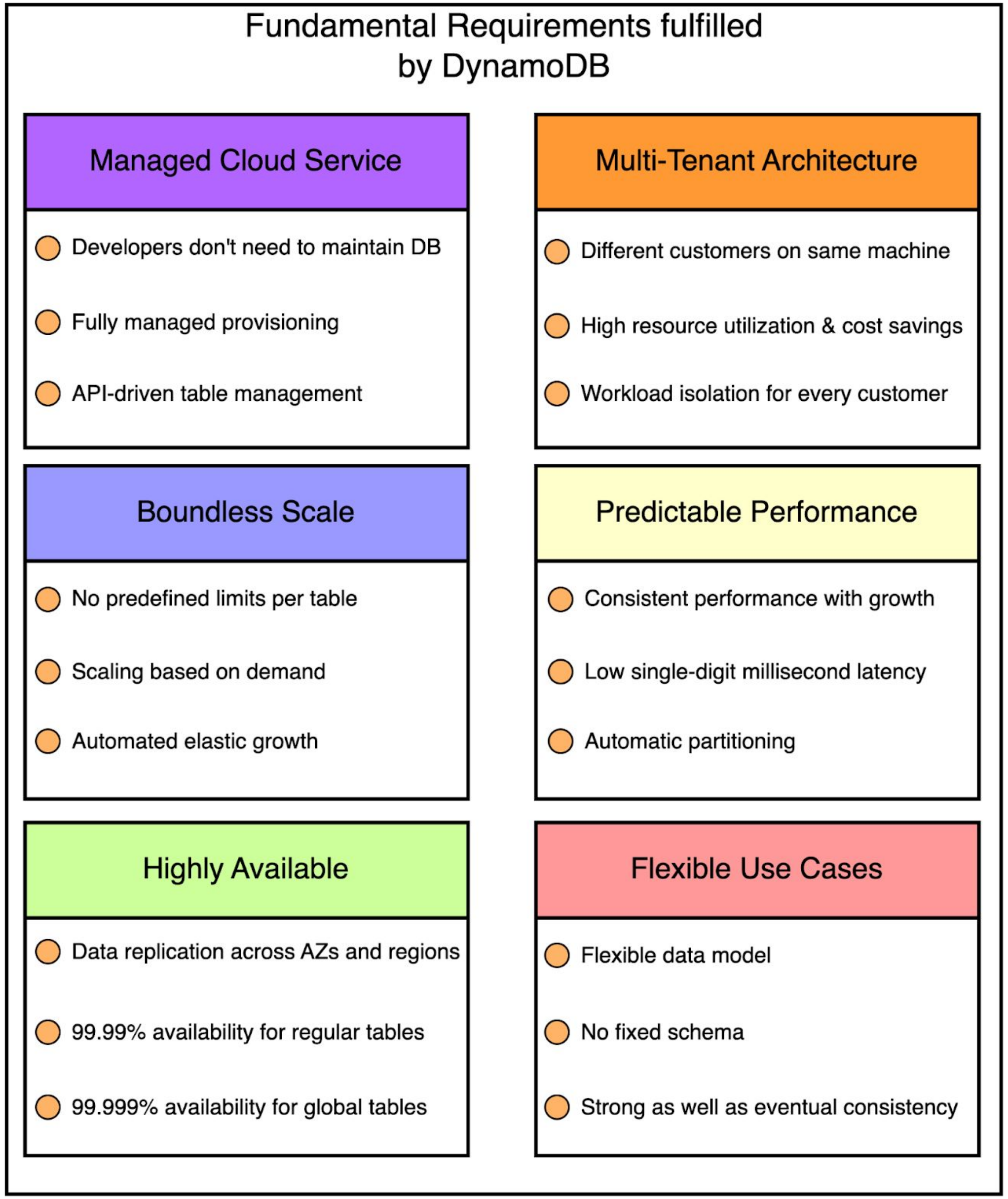
All of this was made possible by DynamoDB

Amazon DynamoDB (contd.)



What is DynamoDB?

- Fully managed NoSQL database
- Multi-Tenant
- Flexible Schema
- Predictable Performance
- Highly Available
- Boundless Scale



source: Link

DynamoDB Architecture

DynamoDB Tables

- Consists of items which is in turn a collection of attributes
- Items uniquely identified by primary key
- Schema of primary key specified at table creation
- The primary key can be a simple partition key or a composite key, or a combination of both partition and sort keys
- Partition key determines the physical storage location of the item
- DynamoDB also supports secondary indexes to query data using alternate keys

DynamoDB Architecture (contd.)

Interface

DynamoDB Interface

Operation	Functionality
PutItem	Insert a new item or replace an existing item with a new item
UpdateItem	Updates an existing item or adds a new item to the table if it doesn't exist
DeleteItem	Delete a single item from the table based on the primary key
GetItem	Returns a set of attributes for the item for the given primary key

DynamoDB Architecture (contd.)

Partitioning and Replication

A DynamoDB table is divided into multiple partitions. This provides two benefits:

- Handling more throughput as requests increase
- Store more data as the table grows

But what about the availability guarantees of these partitions?

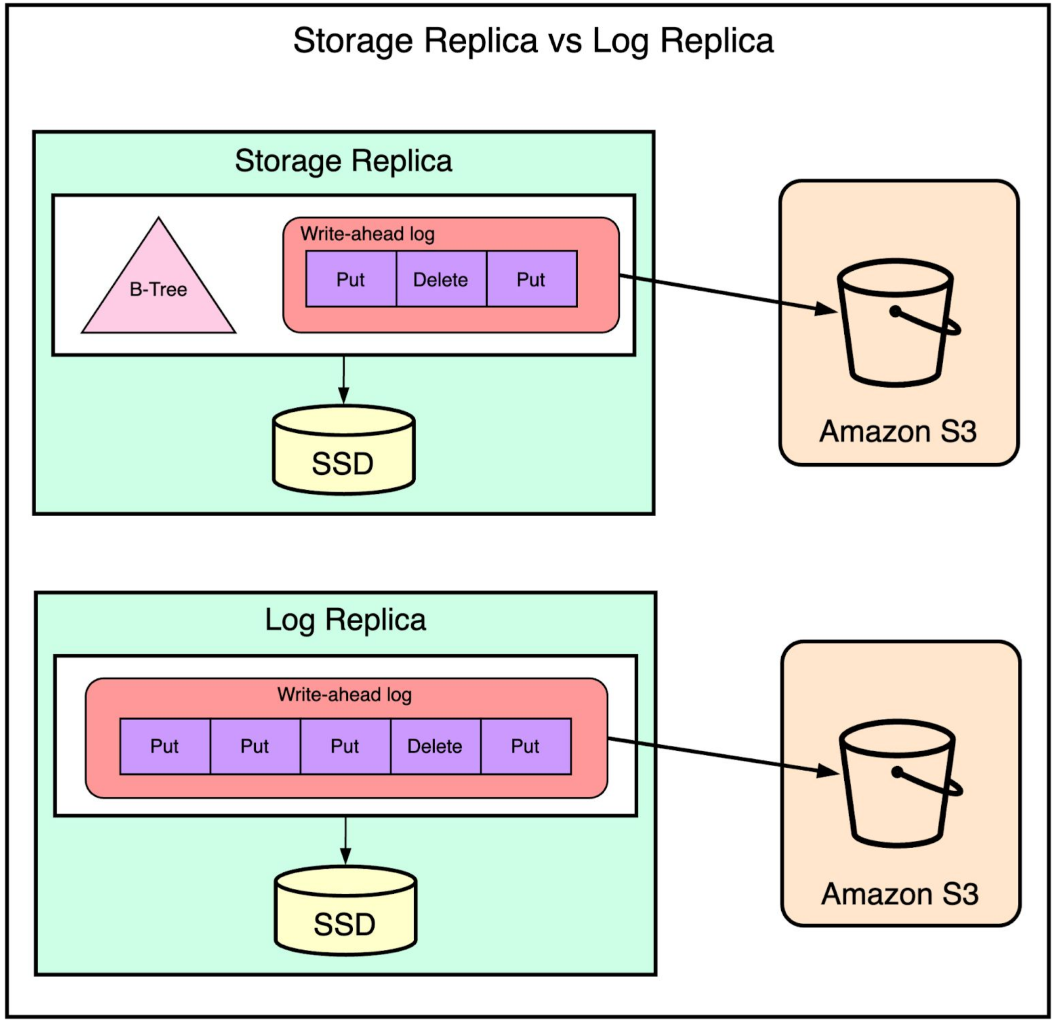
- Each partition has multiple replicas distributed across availability zones
- Together, these replicas form a replication group and improve the partition's availability and durability

What are these ?

DynamoDB Architecture (contd.)

More on Replication Groups

- They consist of storage replicas containing:
  - Write-Ahead Logs (WALs)
  - B-tree that stores the key value data
- They can also contain just the WAL entries
- They are known as log replicas



DynamoDB Architecture (contd.)

An issue in Partitioning and Replication

While replicating data across multiple nodes, guaranteeing a consensus becomes a big issue. What if each partition has a different value for a particular key?

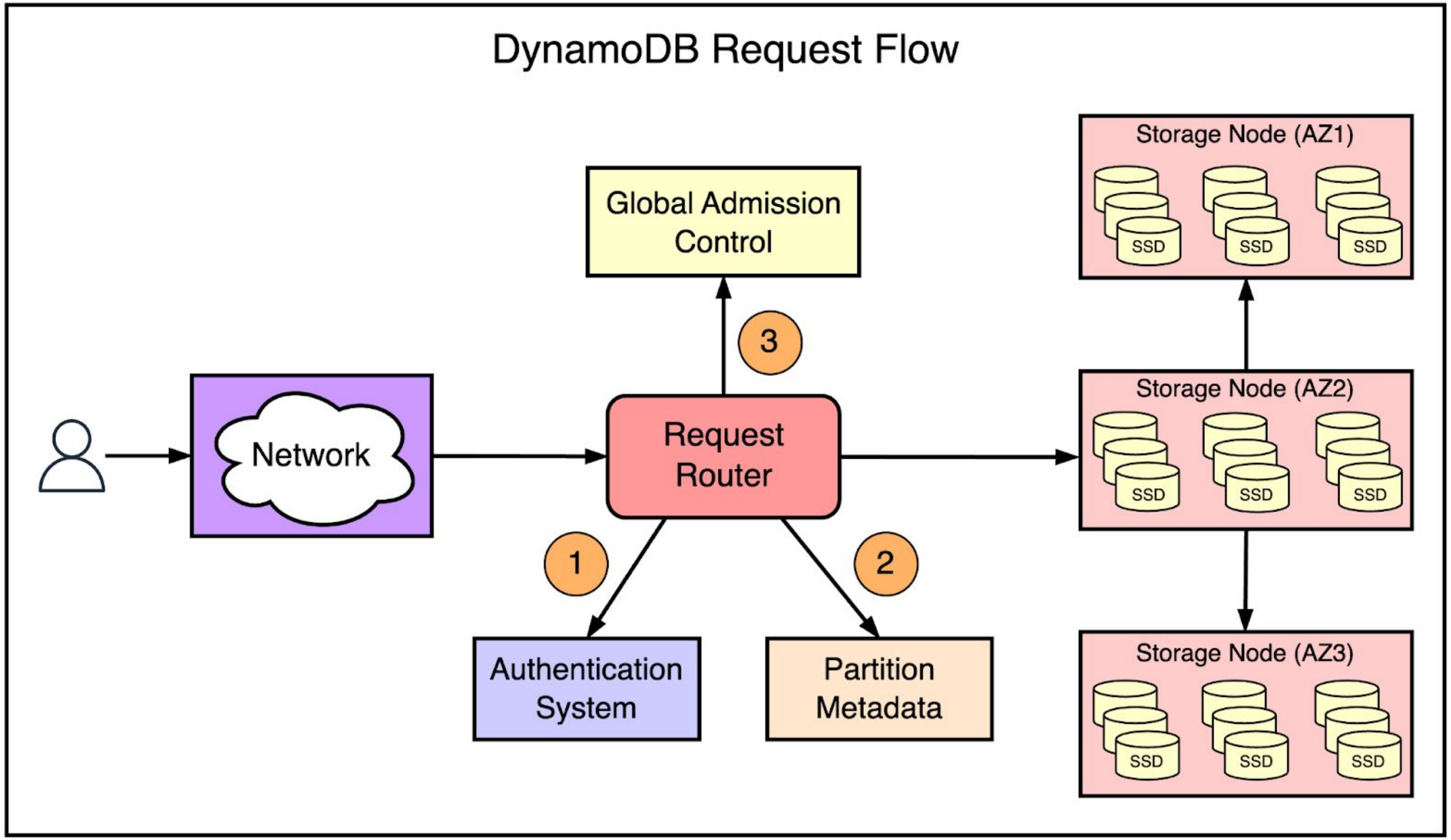
⇒ DynamoDB solves it using Multi-Paxos

Key idea is as follows:

- The leader processes all write requests by generating a WAL record and sending it to the replicas. A write is acknowledged to the application once a quorum of replicas stores the log record to their local write-ahead logs.
- The leader also serves strongly consistent read requests. On the other hand, any other replica can serve eventually consistent reads.

DynamoDB Architecture (contd.)

DynamoDB Request Flow



DynamoDB Architecture (contd.)

DynamoDB Request Flow

- Requests arrive at the request router service. This service is responsible for routing each request to the appropriate storage node
- The request router first checks whether the request is valid by calling the authentication service (AWS IAM)
- Next, the request router fetches the routing information from the metadata service. The metadata service stores routing information about the tables, indexes, and replication groups for keys of a given table or index
- The request router also checks the global admission control to make sure that the request doesn’t exceed the resource limit for the table

Hot Partitions and Throughput dilation

- In the initial release, DynamoDB allowed customers to explicitly specify the throughput requirements for a table in terms of read capacity units (RCUs) and write capacity units (WCUs).
- As the demand from a table changed (based on size and load), it could be split into partitions.

For eg:

- Let’s say a partition has a maximum throughput of 1000 WCUs.
- Table Capacity 3200 WCUs ⇒ 4 partitions, each of 800 WCU
- Now, if Table Capacity increases to 6000 WCUs ⇒ 8 partitions, each of 750 WCU

Hot Partitions and Throughput dilation (contd.)

- All of this was controlled by the admission control system to make sure that storage nodes don’t become overloaded.
- However, this approach assumed a <u>uniform distribution of throughput</u> across all partitions, resulting in some problems.
- Two direct consequences of this approach:
    - **Hot Partitions:** More traffic consistently went to a few items on the tables rather than an even distribution
    - **Throughput dilution:** Splitting a partition reduces per-partition throughput, as it is equally divided among the child partitions(in earlier example: 1000 WCU → 800 WCU → 750 WCU)

Hot Partitions and Throughput dilation (contd.)

Well… then how did the Amazon Engineers solved it ?

They introduced 2 main ideas to solve it:

- Bursting:
  - The idea behind bursting was to let applications tap into this unused capacity at a partition level to absorb short-lived spikes for up to 300 seconds.
  - It's the same as storing money in the bank from your salary each month to buy a new car with all those savings.
- Adaptive Capacity:
  - monitors the provisioned and consumed capacity of all the tables
  - If a table experiences throttling while staying within its table-level throughput, it automatically boosts the allocated throughput of its partitions and vice-versa

References

- Introduction to NoSQL: Link
- NoSQL Databases ~ Couchbase: Link
- DynamoDB paper: Link (Usenix ATC 2022), Link (annotated by Arpit Bhayani)
- A deep dive into DynamoDB ~ ByteByteGo: Link
- Amazon AWS DynamoDB page: Link
- Apache Cassandra: Link
- Memcached: Link
- ScyllaDB: Link