# MODULE 4

# What is NoSQL

- Stands for **N**ot **O**nly **SQL**
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- Used for Big Data and Real time web applications.
- When large amounts of data needs to be stored an retrevied.
- Support of constraints and joins at database level is not needed.
- Data is growing rapidly and the need to scale the database arises regularly.

# Why NoSQL?

- Explosion of Big Data
- Need for scalability & availability
- Flexible, schema-less design
- Built for distributed systems & cloud

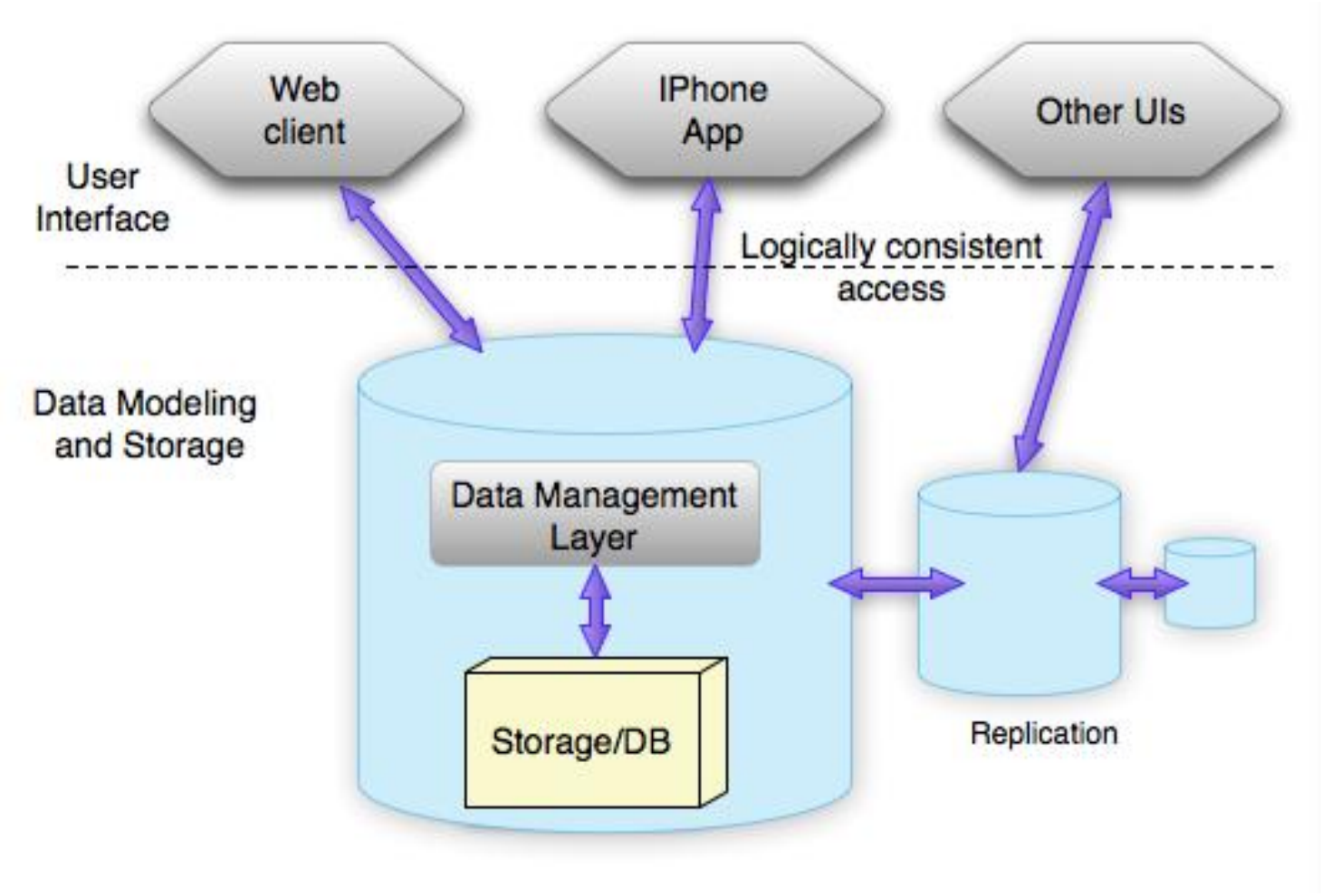
# Characteristics of NoSQL

- Non-relational – Do not follow traditional RDBMS principles
- Flexible schema – Schema-less, can add fields without altering structure
- Alternative query languages – May support queries beyond SQL
- Distributed & horizontally scalable – Designed to run on clusters
- Handles unstructured/less structured data – Suitable for big data
- Open-source driven – Most NoSQL databases are open-source projects
- Consistency model – Follows BASE rather than full ACID transactions
- Supports big data applications – Optimized for large-scale data storage and processing
-

# RDBMS vs NoSQL

- RDBMS → Structured, ACID, centralized
- NoSQL → Flexible, BASE, distributed
- Choose based on requirements

# Architecture of NoSQL



# Types of NoSQL

- **Graph stores** are used to store information about networks of data, such as social connections.
- **Document databases** pair each key with a complex data structure known as a document.
- **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or 'key'), together with its value.
- **Wide-column stores** optimized for queries over large datasets, and store columns of data together, instead of rows.

## Key Value

Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris

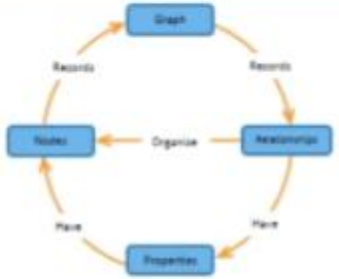
## Document-Based


Example:
MongoDB, CouchDB, OrientDB, RavenDB

## Column-Based

Example:
BigTable, Cassandra, Hbase, Hypertable

## Graph-Based



Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

**Key Value Data Model:**

- In the key value data model, **the client can get the value for a key, put a value for a key, or delete a key from the data store.**

- Here**, the value is a Binary Large Object (BLOB) that is only concerned about data and not what is inside**; it is the responsibility of the application to understand what exactly is stored.

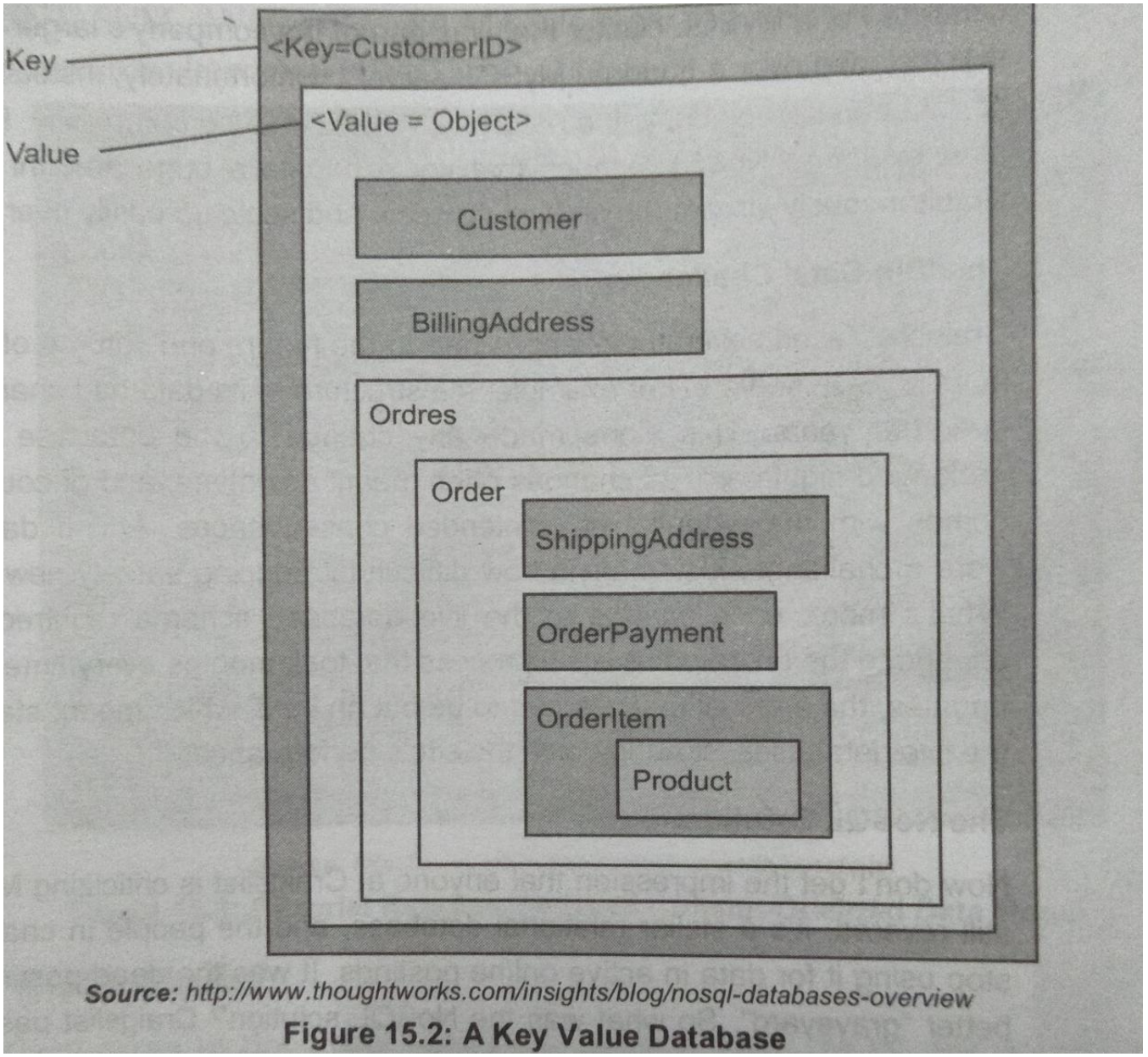
- BLOB is also scalable because it uses primary key access for the different purposes.

- Some examples of popular key/value databases are Riak; Redis; Memcached and its versions berkeley DB: HamsterDB; Amazon DynamoDB; Project Voldemort; and Couchbase.

- All key/value databases are not the same; there are major differences between these products; for exam memcached data is **not persistent**, but Riak data is.

- Therefore, we need to cache user preferences in Memcached data. If we implement user preference Memcached, they would need to be refreshed entirely in case any of the nodes are lost.

- On the other hand, we may not need to worry about losing data if it is stored in the Riak database. However we need to consider how to update stale data. So, it is also important to choose the type of key /value databases as per our requirement




***Source:*** *http://www.thoughtworks.com/insights/blog/nosql-databases-overview*

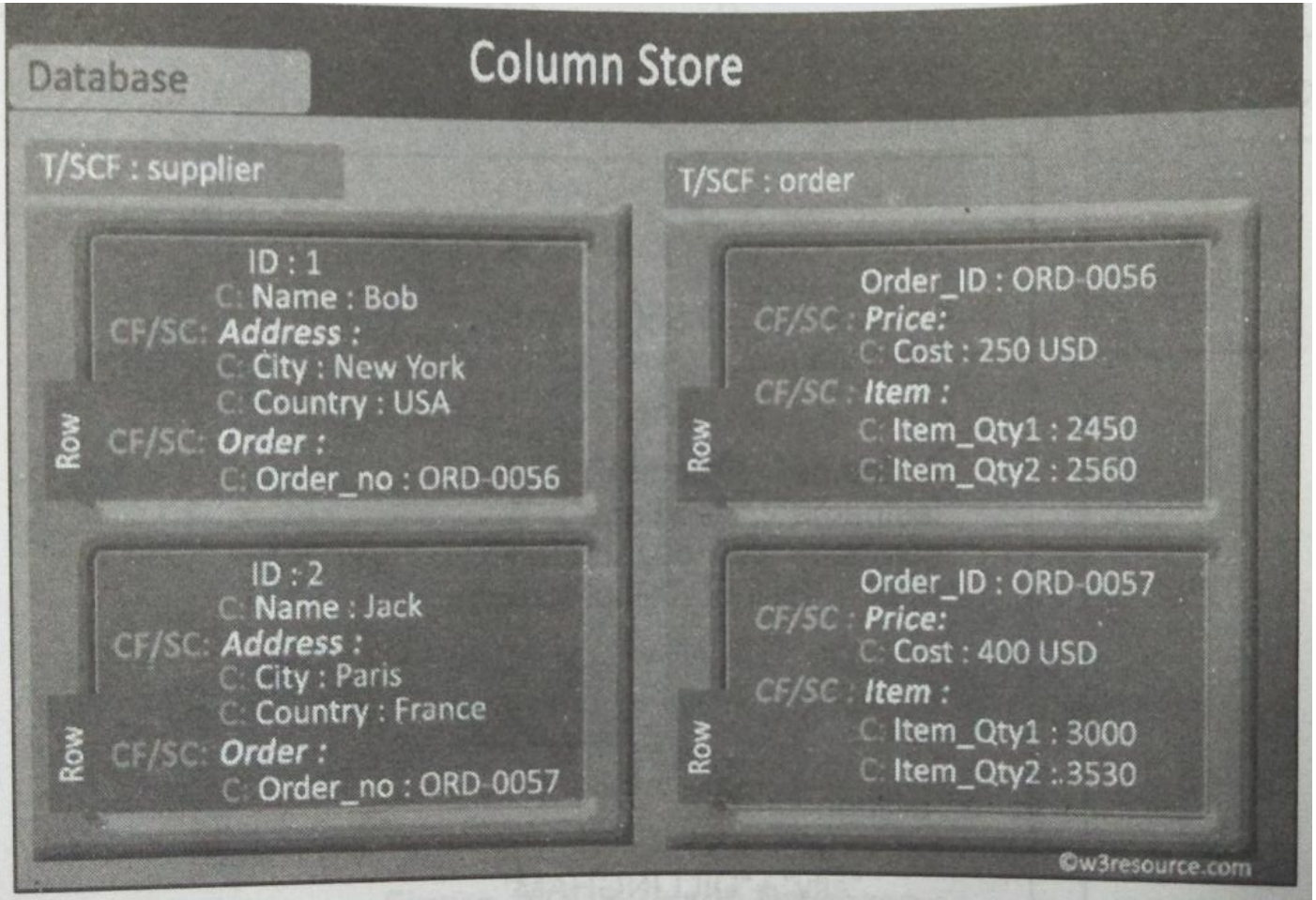
**Figure 15.2: A Key Value Database**

*Column-Oriented Data Model:*

- Column-oriented databases are based on columns and every column is considered individually. The values of a single column are stored contiguously.

- Some examples of column-oriented data models are Cassandra, BigTable, SimpleDB, and HBase. The data maintained by columns are in the form of column-specific files.

- In column-oriented data model the performance on the aggregation queries such as COUNT, SUM, AVG, MIN, and MAX is high.




***Source:*** *http://www.w3resource.com/mongodb/nosql.php*

**Figure 15.3: Pictorial Representation of Column-Oriented Data Model**

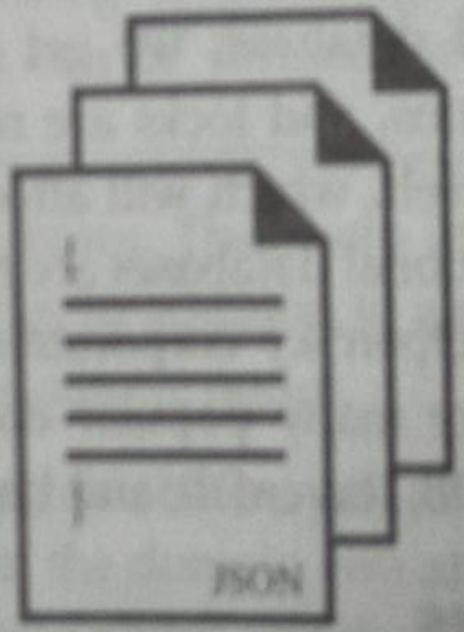
*Document Data Model:*

- There are many types of document databases, such as XML, JSON, BSON, etc.
- These document are self describing hierarchical tree data structures that can contain maps, collections, and scalar values.
- Document databases mainly store documents in the value part of the key/value store think about document databases as key/value stores where the value is examinable.
- For easier transitions from relational databases, document databases provide indexing, and searching, etc.

odel:


Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.




Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Source: http://datanomy.com/sql-vs-nosql-need-know/

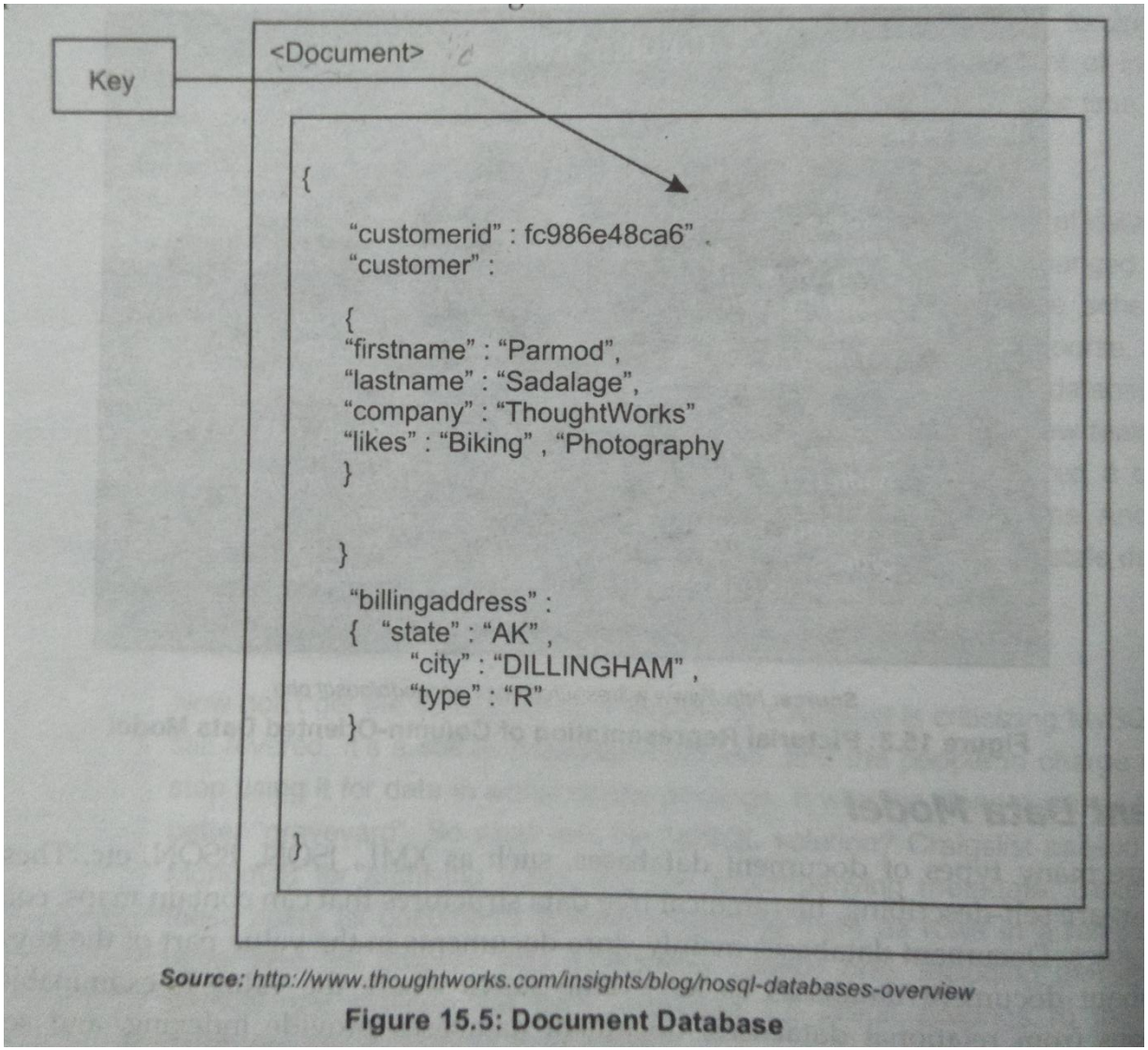
**Figure 15.4: Comparing the Relational Data Model and the Document Data Model**

- Though the document oriented model is rich in performance and scalability, **it doesn't provide ACID and data integrity that a relational data model has by definition.**

- So if we want to use a document oriented database, **we get more performance and scalability, but database-level data integrity, transactions, and locks are no longer there and will need to be separately embed in the application logic itself, which will affect how we write and structure our code.**

- Document-oriented databases and relational databases are no replacements for each other but are useful in various ways.

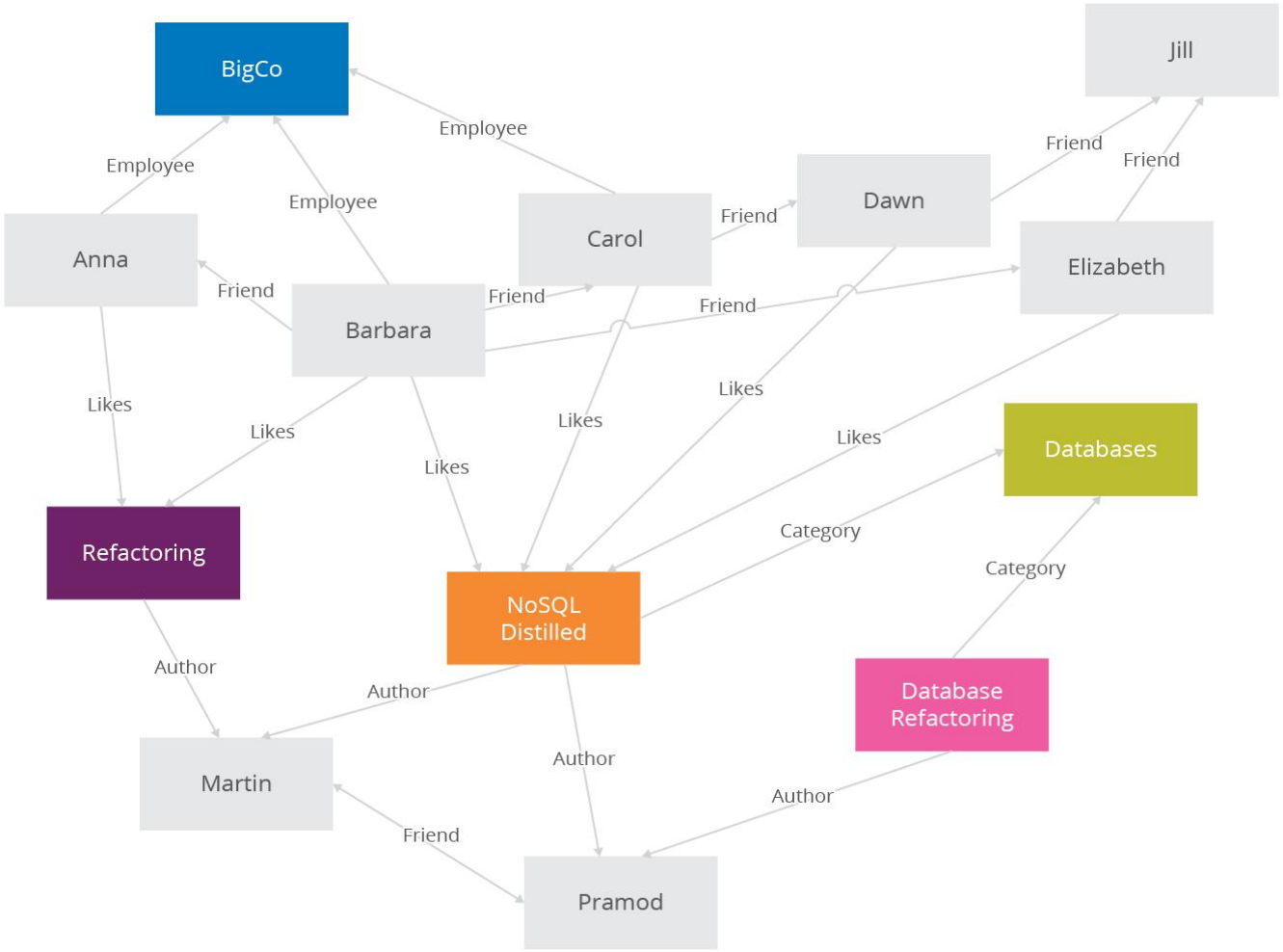
- For example, the OpenSky corporation uses both MySQL (relational) and MongoDB (documentoriented). There are many popular document databases in the market like MongoDB, CouchDB, Terrastore, OrientDB, RavenDB, and Lotus Notes,




**Source:** *http://www.thoughtworks.com/insights/blog/nosql-databases-overview*

**Figure 15.5: Document Database**

- A graph database is a type of database that makes use of graph structures for semantic queries having nodes, edges, and properties to display and store data.
- **Graph databases are mainly used for storing entities and the relationships between these entities.**
- **An entity is nothing but a node with its own properties.**
- **A node** can be perceived as an instance of an **object in an application.**
- **Relations** are also known **as edges**, and **they can have properties as well.**
- **Edges are directions**, and each direction are the graph is organized, and the data in it are stored once but interpreted in different aspects based on the relationships.
- **Eg:** Neo4J, Infinite Graph, OrientDB, or FlockDB


BigCo
Jill
Employee
Employee
Employee
Friend
Friend
Dawn
Anna
Employee
Carol
Friend
Elizabeth
Friend
Friend
Barbara
Friend
Friend
Likes
Likes
Likes
Likes
Likes
Likes
Databases
Refactoring
Category
Category
NoSQL
Distilled
Author
Author
Database
Refactoring
Author
Martin
Author
Friend
Pramod

- Whenever we store a graph-like structure in a **relational database**, it is stored for a **single type of relationship.**

- If we **add another relationship to it**, it means **a lot of schema changes and data movement.** However, this doesn't happen in the case of graph databases.

- Additionally, in relational databases, we model the graph based on the traversal. So, if the traversal changes, the data will have to change accordingly.

- In graph databases, traversing the joins or relationships can be done very quickly. Moreover, the relationship between nodes is not calculated at the time of querying but is actually persisted as a relationship.

- The point is that we can **have nodes with different types of relationships between them.** This allows us to represent relationships between the major domain entities and secondary relationships for category, path, or linked lists for accessing sorted data.

- Moreover there is **no limitation to the number and kind of relationships that can be formed, so relationships can be represented in the same graph database.**

- They help in deriving most of the values in these databases. **Relationships can have properties in addition to a start node and an end node.**

- The most important concept of a database is relations among nodes.

- **Addition of new relationship types is easy.**

- Changing existing nodes and their relationships is similar to migrating data, because the major changes must be done on each node and each relationship in your data.

- Some examples of popular graph databases are Neo4J, Infinite Graph, and FlockDB.

| Data Model | Performance | Flexibility | Scalability | Complexity |
|---|---|---|---|---|
| **Key-value store** | High | High | High | None |
| **Column Store** | High | Moderate | High | Low |
| **Document** | High | High | Variable (High) | Low |
| **Graph Database** | Variable | High | Variable | High |

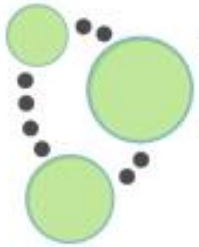
## Document Database

Couchbase

MarkLogic

mongoDB

## Graph Databases



Neo4j

InfiniteGraph

The Distributed Graph Database

## Wide Column Stores

redis

amazon DynamoDB

AEROSPIKE

riak

## Key-Value Databases

accumulo

HYPERTABLE INC

Cassandra

APACHE HBASE

Amazon SimpleDB

# Schema-Less Databases

- As the name suggests, a **database without any schema** is known as schema-less database.
- You need **to define a schema or structure before storing data into relational databases.**
- A well-defined schema of a **database describes the tables, columns, and data types of the values in the columns of the database.**
- Storing data in NoSQL is easier in comparison to data storage in SQL.
- As discussed earlier, in the case of a key/value database**, you can store any type of data under a particular key.** This is also true in the case of document databases, where there is **no restriction on the type of document you want to store.**
- In the case of column-family **databases, you can store any type of data under any column according to your requirement**. Also, in graph databases, **there are no restrictions in adding edges or properties to nodes.**
- You can add edges and simultaneously define node properties as per your convenience or requirement.

- In the case of a schema-defined database, **you need to plan for the collection of data to be stored in the database.**

- However, **no such planning is required for schema-less databases**, and the data type can be changed at run time, if required.

- If you find new data items while storing data, **you can add them easily**. Similarly, you can **discard the items that you feel are not required**, at run time, without worrying about existing related data like in case of schema or relational databases.

- A schema-less database also makes it easier to arrange non-uniform information, where each record has an alternate set of fields.

- You also have various types of information in diverse columns similar to schema databases, which arrange all the rows of a table horizontally.

- You either end up with a set of invalid or null columns (a sparse table), or with unimportant, meaningless columns.

- **A schema-less database permits each record to contain exactly what it needs. This type of database also eliminates many conditions found in fixed-schema databases.**

# Materialized Views

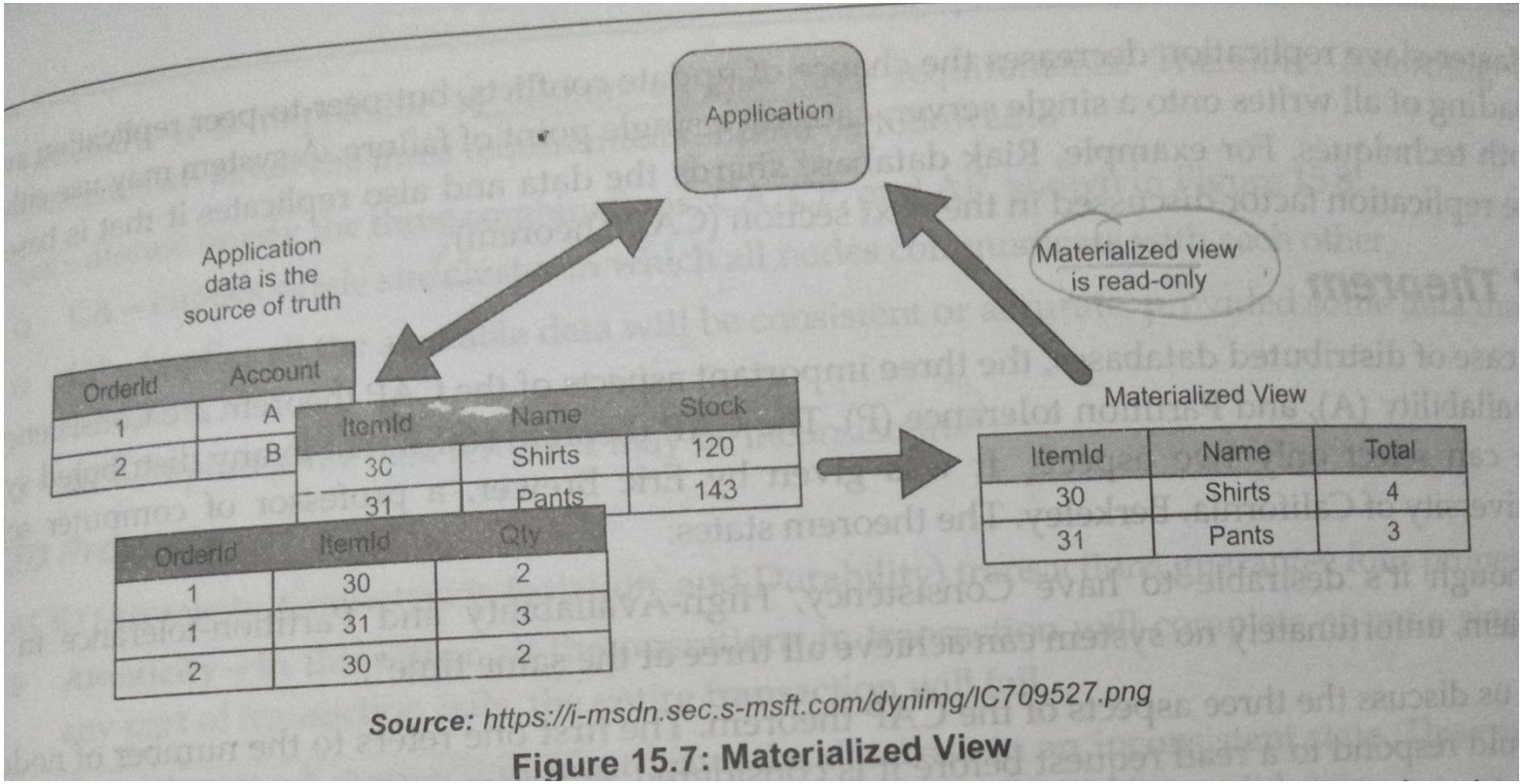
- The priority for developers and data administrators in data storage is often on **how the data is stored, as opposed to how it is read.**

- The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using the document model, data is often represented as a series of aggregates, each of which contains all information about an entity.

- However, this may have a **negative effect on queries.** When a query requires only a subset of the data from some entities, such as a summary of orders of several customers without all the order details, it needs to extract all of the data for the relevant entities in order to obtain the required information.

- **To support efficient querying, a common solution is to generate, in advance, a view that contains the data in a format most suited to generate the required dataset.** This view is known as the materialized view.

- **You can define the materialized view as a pattern that generates pre-populated views of data in environments where the source data is in a format that is not suitable for querying.** In this case, generating a suitable query is also difficult, or its performance is poor due to the nature of the data on the data store.




***Source:*** *https://i-msdn.sec.s-msft.com/dynimg/IC709527.png*

**Figure 15.7: Materialized View**

- These materialized views, **which contain only data required by a query, allow applications to quickly obtain the information they need.**

- **In addition to joining tables or combining data entities, materialized views may include the current values of calculated columns or data items, the results of combining values or executing transformations on the data items, and values specified as part of the query.**

- A materialized view may even be optimized for just a single query.

- An important point to note here is that a materialized view and the data it contains are completely **disposable because they can be entirely rebuilt from the source data stores.**

- **A materialized** view is never up**dated directly by an application, and so it is effectively a specialized cache.**

- When there is **a change in the source data while creating view, the view must be updated to include the new information.**

- **This may occur automatically on an appropriate schedule or when the system detects a change in the original data.** In other cases, it may be necessary to regenerate the view manually.

# *Distribution Models:*

Aggregate-oriented databases allow easy distribution of data. In such databases, the distribution mechanism is just concerned about the movement of aggregate data and not the related data, as all the related data is stored in the aggregate. Distribution of data can be done in two ways:

- **Through sharding** -Sharding is one of the major techniques of **data distribution**. It is used to distribute various types of **data across multiple servers**. Therefore, each server acts as **a single source for a subset of data. .**

- **Through replication**-Replication is one of the major techniques for fault tolerance. The idea is to copy data across multiple servers so that each bit of data can be found in multiple places. Replication occurs in two forms:

- Master-slave replication, which makes one node the authoritative copy that handle writes; while slaves, which are synchronized with the master, handle reads.

- Peer-to-peer replication, which allows writes to any node without requiring authorization. Here, the nodes can coordinate with each other to synchronize their copies of the data.

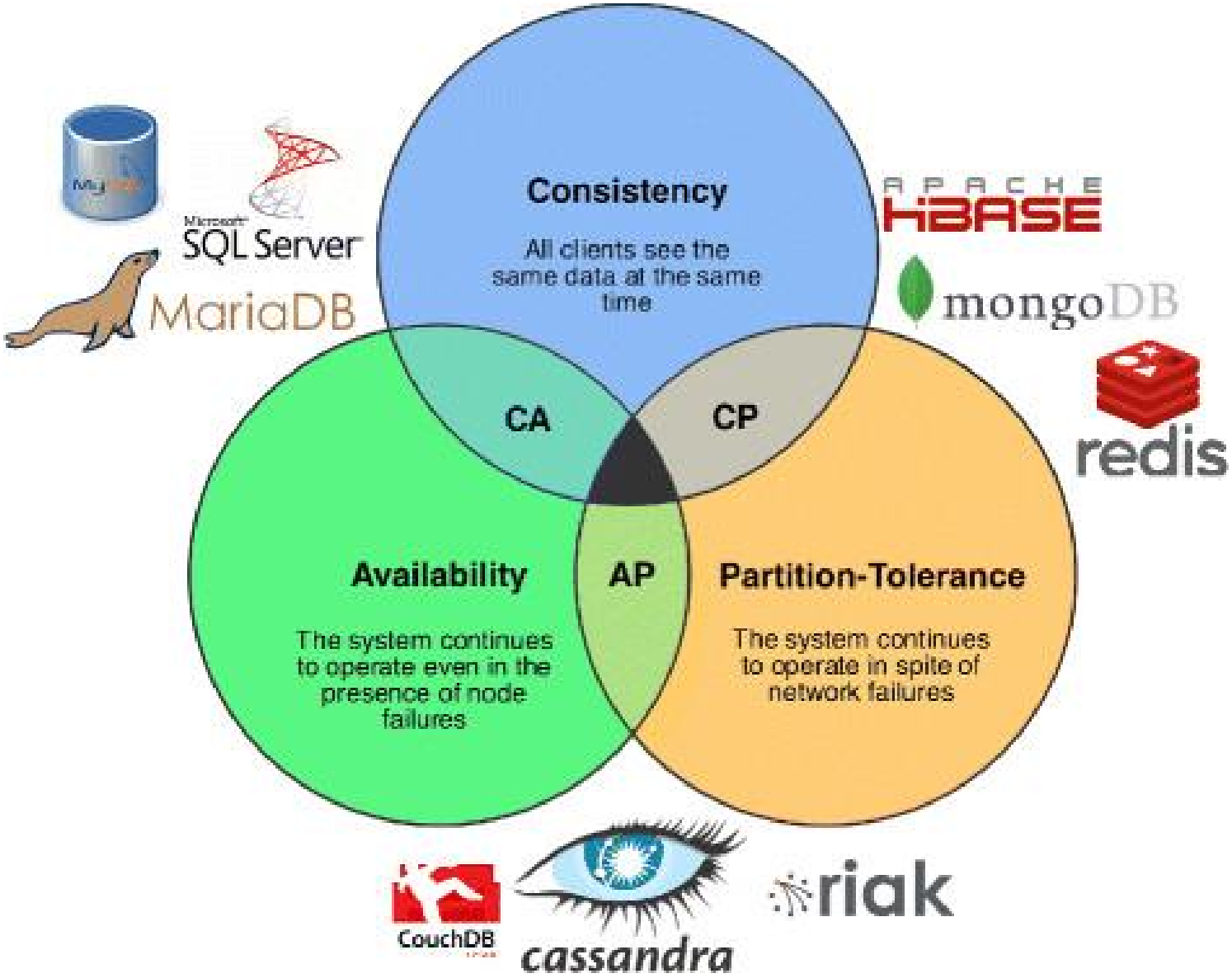
- Master-slave replication decreases the **chance of update conflicts**, but peer-to-peer replication avoid loading of all writes onto a single server, **causing a single point of failure**. A system may use either both techniques.

- For example, Riak database shards the data and also replicates it that is based the replication factor discussed in the next section (CAP Theorem)

# CAP theorem

The limitations of distributed databases can be described in the so called the **CAP theorem**. The *CAP theorem* states that any distributed database with shared data can have at most two of the three desirable properties:

- **C**onsistency: every node always sees the same data at any given instance (i.e., strict consistency)
- **A**vailability: the system continues to operate, even if nodes in a cluster crash, or some hardware or software parts are down due to upgrades
- **P**artition Tolerance: the system continues to operate in the presence of network partitions


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MariaDB
SQL Server
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

- Generally, it is impossible to sustain all the three requirements. Therefore, according to CAP theorem, two out of the three requirements should be followed, Let's discuss briefly the three combinations: **CA, CP, and AP**

- **CA**- Implies single site cluster in which all nodes communicate with each other.

- **CP**-Implies all the available data will be consistent or accurate, provided some data may not be available .

- **AP**-Implies some data returned may be inconsistent.

# ACID Property :

**Atomicity** – All operations in a transaction succeed or none do.

**Example:** Transferring $100 from A to B: either both debit and credit happen, or neither.

**Consistency** – Transaction brings database from one valid state to another.

**Example:** Total bank balance remains the same after a transfer.

**Isolation –** Concurrent transactions do not interfere with each other.

**Example:** Two users updating the same account simultaneously don’t cause errors.

**Durability –** Once a transaction is committed, it remains even after a crash.

**Example:** Money transferred remains in B’s account after a system failure.

# BASE Properties (for NoSQL / distributed databases)

**Basically Available (B):**

The system guarantees availability, meaning it will respond to requests, though the response may be eventually consistent.

**Example:** A NoSQL database always returns a value for a read request, even if some nodes are temporarily out-of-sync.

**Soft state (S):** The state of the system may change over time, even without input, due to eventual consistency mechanisms.

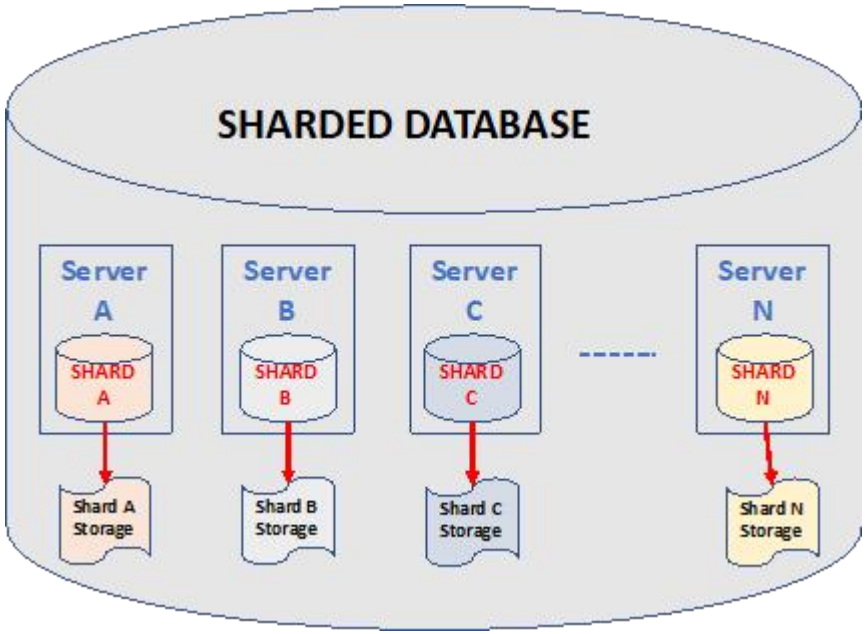
**Example:** Updates propagate asynchronously across nodes; a replica may temporarily have outdated data.

**Eventual consistency (E):** The system guarantees that, given enough time without new updates, all nodes will converge to the same consistent state.

**Example:** After updating a product price in an e-commerce app, all replicas will eventually reflect the new price.

# Sharding in Databases

- Definition:
- Sharding is a database architecture pattern in which large datasets are horizontally partitioned across multiple servers (called shards) to improve performance, scalability, and availability.



# Why Sharding?

- Handles large datasets that cannot fit on a single server.
- Enables horizontal scaling (adding more servers to handle load).
- Improves performance by distributing queries across multiple nodes.
- Enhances availability and fault tolerance (failure of one shard doesn’t bring down the entire system).

# How Sharding Works

• **Horizontal partitioning**: Rows of a table are divided across shards, not columns.

• **Shard key**: A chosen field (e.g., user_id) used to determine which shard stores the data.

• **Shard map / router:** Directs queries to the correct shard.

**Example:**

- Users table with user_id 1–1,000,000
- Shard 1 → users 1–250,000
- Shard 2 → users 250,001–500,000
- Shard 3 → users 500,001–750,000
- Shard 4 → users 750,001–1,000,000

# Types of Sharding

**1. Range-based Sharding**

- Rows assigned to shards based on value ranges.

- Example: user_id 1–1000 → Shard 1, 1001–2000 → Shard 2.

**2. Hash-based Sharding**

- - A hash function determines the shard for each row.
- - Example: hash(user_id) % 4 → shard 0–3.
- **3. Directory-based Sharding**
- - A lookup table maintains mapping from key → shard.
- - More flexible, but requires extra management.

# Key Considerations / Challenges

- Choosing shard key carefully – affects performance and balance.

- Rebalancing shards – moving data when some shards grow faster.
- Cross-shard queries – more complex and slower than single-shard queries. Consistency and replication – ensuring eventual consistency across shards.

# Summary

- NoSQL = flexible, scalable, distributed
- Types: Key-Value, Document, Column, Graph
- Trade-offs: ACID vs BASE, CAP theorem
- Best choice depends on use case