UNIT III NOSQL DATABASES 9

NoSQL – CAP Theorem – Sharding - Document based – MongoDB Operation: Insert, Update, Delete, Query, Indexing, Application, Replication, Sharding–Cassandra: Data Model, Key Space, Table Operations, CRUD Operations, CQL Types – HIVE: Data types, Database Operations, Partitioning – HiveQL – OrientDB Graph database – OrientDB Features

---

NoSQL

NoSQL, also referred to as "not only SQL" or "non-SQL", is an approach to design database that enables the storage and querying of data outside the traditional structures found in relational databases.

While NoSQL can still store data found within relational database management systems (RDBMS), it just stores it differently compared to an RDBMS.

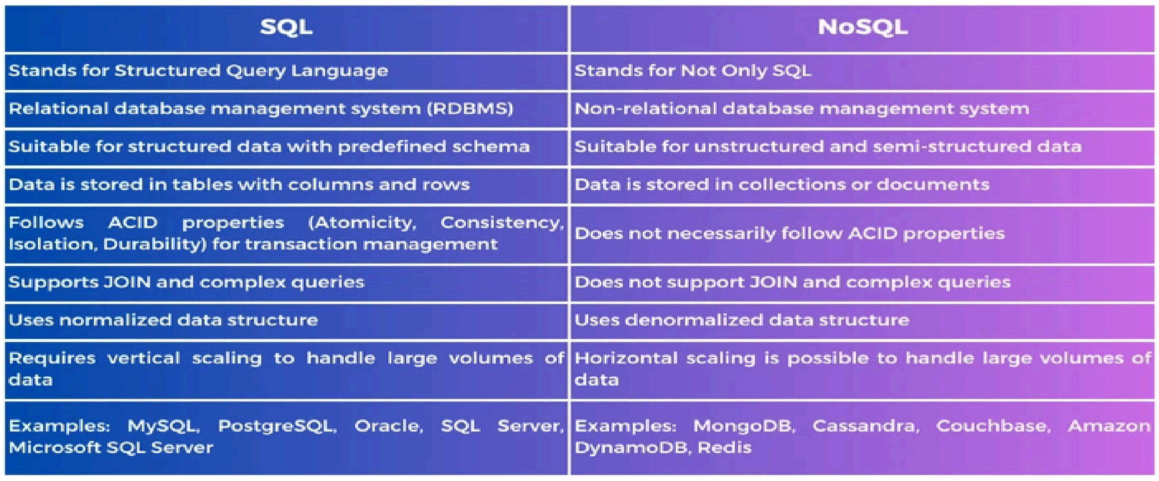
Instead of the typical tabular structure of a relational database, NoSQL databases store data within one data structure. Since this non-relational database design does not require a schema, it offers rapid scalability to manage large and typically unstructured data sets.

NoSQL is also type of distributed database, which means that information is copied and stored on various servers, which can be remote or local. This ensures availability and reliability of data. If some of the data goes offline, the rest of the database can continue to run.

Today, companies need to manage large data volumes at high speeds with the ability to scale up quickly to run modern web applications in nearly every industry.

In this era of growth within cloud, big data, and mobile and web applications, NoSQL databases provide that speed and scalability, making it a popular choice for their performance and ease of use.

NoSQL versus SQL



SQL	NoSQL
Stands for Structured Query Language	Stands for Not Only SQL
Relational database management system (RDBMS)	Non-relational database management system
Suitable for structured data with predefined schema	Suitable for unstructured and semi-structured data
Data is stored in tables with columns and rows	Data is stored in collections or documents
Follows ACID properties (Atomicity, Consistency, Isolation, Durability) for transaction management	Does not necessarily follow ACID properties
Supports JOIN and complex queries	Does not support JOIN and complex queries
Uses normalized data structure	Uses denormalized data structure
Requires vertical scaling to handle large volumes of data	Horizontal scaling is possible to handle large volumes of data
Examples: MySQL, PostgreSQL, Oracle, SQL Server, Microsoft SQL Server	Examples: MongoDB, Cassandra, Couchbase, Amazon DynamoDB, Redis

Types of NoSQL databases

NoSQL provides other options for organizing data in many ways. By offering diverse data structures, NoSQL can be applied to data analytics, managing big data, social networks, and mobile app development.

A NoSQL database manages information using any of these primary data models:

1. Key-value store

- This is typically considered the simplest form of NoSQL databases.
- This schema-less data model is organized into a dictionary of key-value pairs, where each item has a key and a value.
- The key could be like something similar found in a SQL database, like a shopping cart ID, while the value is an array of data, like each individual item in that user's shopping cart.
- It's commonly used for caching and storing user session information, such as shopping carts.
- However, it's not ideal when you need to pull multiple records at a time.

2. Document store

- As suggested by the name, document databases store data as documents.
- They can be helpful in managing semi-structured data, and data are typically stored in JSON, XML, or BSON formats.
- This keeps the data together when it is used in applications, reducing the amount of translation needed to use the data.
- Developers also gain more flexibility since data schemas do not need to match across documents (e.g. name vs. first_name).
- However, this can be problematic for complex transactions, leading to data corruption.
- An example of a document-oriented database is MongoDB.

3. Wide-column store

- These databases store information in columns, enabling users to access only the specific columns they need without allocating additional memory.
- This database tries to solve for the shortcomings of key-value and document stores,
- Apache HBase and Apache Cassandra are examples of open-source, wide-column databases.
- Apache HBase is built on top of Hadoop Distributed Files System that provides a way of storing sparse data sets, which is commonly used in many big data applications.
- Apache Cassandra, on the other hand, has been designed to manage large amounts of data across multiple servers and clustering that spans multiple data centers.

- It’s been used for a variety of use cases, such as social networking websites and real-time data analytics.

4. Graph store

- This type of database typically stores data from a knowledge graph.
- Data elements are stored as nodes, edges and properties.
- Any object, place, or person can be a node.
- An edge defines the relationship between the nodes.
- For example, a node could be a client, like IBM, and an agency like Ogilvy.
- An edge would be to categorize the relationship as a customer relationship between IBM and Ogilvy.
- Graph databases are used for storing and managing a network of connections between elements within the graph.
- Neo4j - a graph-based database service based on Java with an open-source community edition

5. In-memory store

- With this type of database, like IBM solidDB, data resides in the main memory rather than on disk, making data access faster than with conventional, disk-based databases.

Examples of NoSQL databases

Many companies have entered the NoSQL landscape. In addition to those mentioned above, here are some popular NoSQL databases:

- **Apache CouchDB,** an open source, JSON document-based database that uses JavaScript as its query language.
- **Elasticsearch,** a document-based database that includes a full-text search engine.
- **Couchbase,** a key-value and document database that empowers developers to build responsive and flexible applications for cloud, mobile, and edge computing.

Advantages of NoSQL

Each type of NoSQL database has strengths that make it better for specific use cases. However, they all share the following advantages for developers and create the framework to provide better service customers, including:

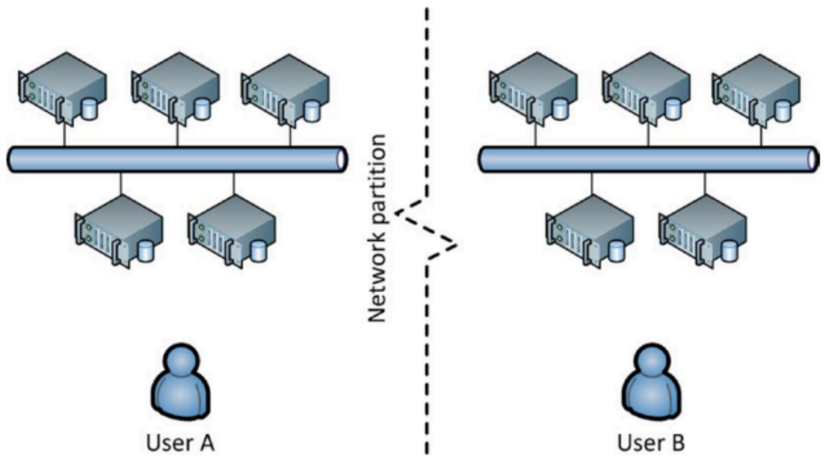
1. **Cost-effectiveness:** It is expensive to maintain high-end, commercial RDBMS. They require the purchase of licenses, trained database managers, and powerful hardware to scale vertically. NoSQL databases allow you to quickly scale horizontally, better allocating resources to minimize costs.
2. **Flexibility**: Horizontal scaling and a flexible data model also mean NoSQL databases can address large volumes of rapidly changing data, making them great for agile development, quick iterations, and frequent code pushes.

3. **Replication**: NoSQL replication functionality copies and stores data across multiple servers. This replication provides data reliability, ensuring access during down time and protecting against data loss if servers go offline.
4. **Speed**: NoSQL enables faster, more agile storage and processing for all users, from developers to sales teams to customers. Speed also makes NoSQL databases generally a better fit for modern, complex web applications, e-commerce sites, or mobile applications.

- In a nutshell, NoSQL databases provide **high performance, availability, and scalability.**

CAP Theorem

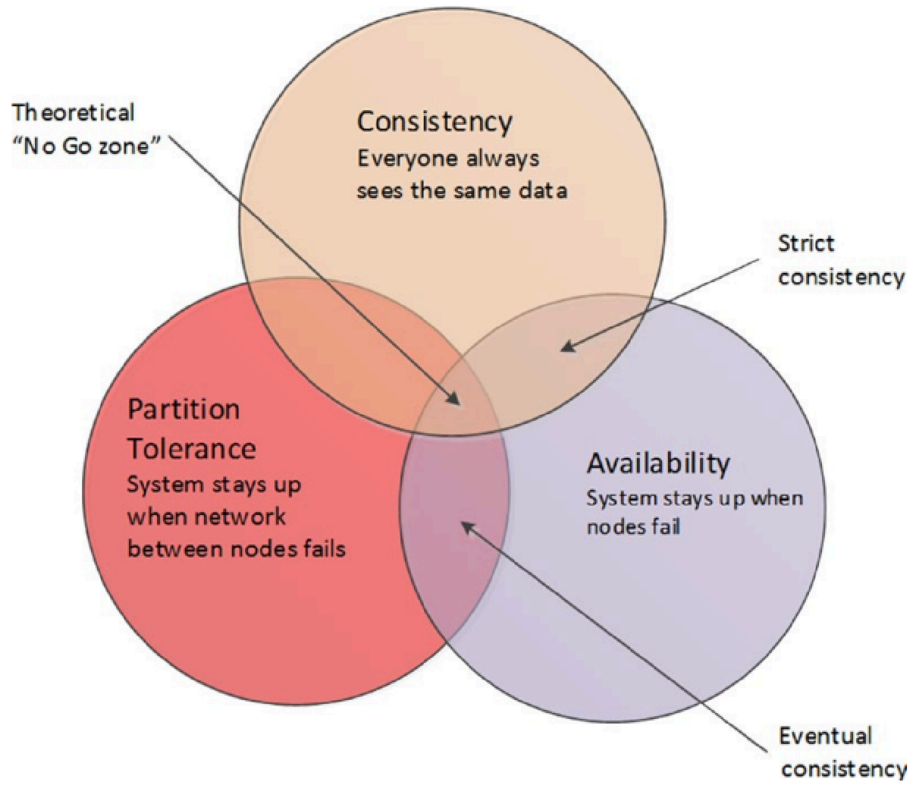
- In 2000, **Eric Brewer** outlined the "**CAP**" conjecture.
- The CAP theorem says that in a distributed database system, can have at most only two of **Consistency, Availability, and Partition tolerance**.
- **Consistency** means that every user of the database has an identical view of the data at any given instant.
- **Availability** means that in the event of a failure, the database remains operational.
- **Partition tolerance** means that the database can maintain operations in the event of the network's failure between two segments of the distributed system.
- The issue of partition tolerance was theoretical in the 2000s.
- Most systems resided in a single data center, and redundant network connectivity within that data center prevented any partition from ever occurring.
- If the data center failed, perhaps a failover data center would be bought online.
- However, there were almost no true multiple data center applications.
- But as web systems became global in scope and aspired to continual availability, partition tolerance became a real issue.



- Consider the distributed application shown in Figure.
- In the event of the network partition shown, the system has two choices:
  - either show each user a different view of the data,
  - or shut down one of the partitions and disconnect one of the users.
- Oracle’s RAC solution, which of course supported the ACID transactional model, would choose consistency. In the event of a network partition—known in Oracle circles as the “split brain” scenario—one of the partitions would choose to shut down. However, in the context of a global social network application, or a worldwide e-commerce system, the desired solution is to maintain availability even if some consistency between users is sacrificed.

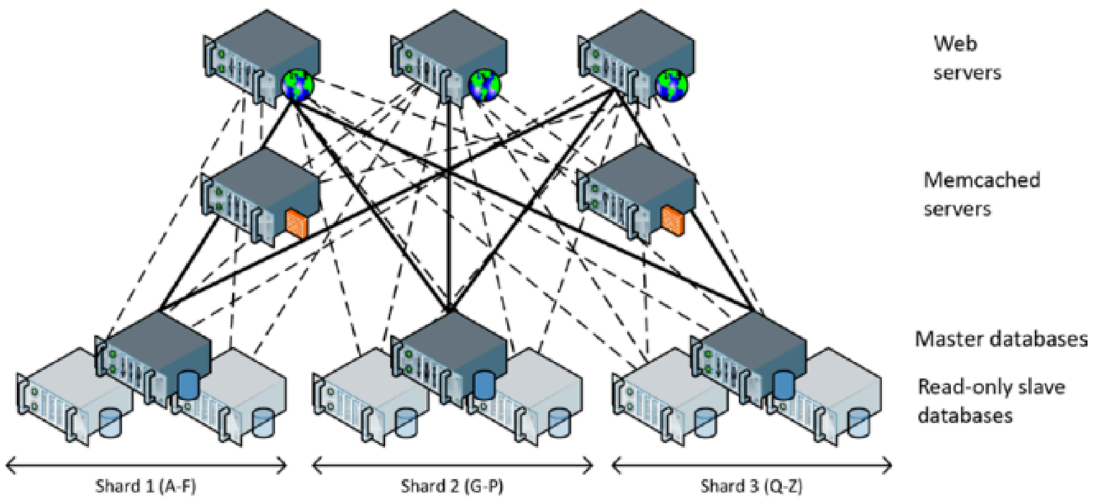
Eventual Consistency

- CAP theorem provides a stark choice:
- If the system wants to be undisturbed by network partitions, it must sacrifice strict consistency between partitions.
- However, even without considerations of CAP theorem, ACID transactions were increasingly untenable in large-scale distributed websites.
- This relates more to performance than to availability.
- In any highly available database system, multiple copies of each data element must be maintained in order to allow the system to continue operating in the event of node failure.
- In a globally distributed system, it becomes increasingly desirable to distribute nodes around the world to reduce latency in various locations.
- To ensure strict consistency, it is necessary to ensure that a database change is propagated to multiple nodes synchronously and immediately.
- For many websites, including social networks and certain e-commerce operations, the worldwide synchronous consistency is unnecessary.
- It doesn’t matter if my friend in Australia can see my tweet a few seconds before my friend in America. Also both friends can see the tweet eventually.
- This concept of eventual consistency has become a key characteristic of many NoSQL databases.
- The concept was most notably outlined by Werner Vogels, CTO of Amazon, and was implemented in Amazon’s Dynamo key-value store.


Theoretical
"No Go zone"
Consistency
Everyone always sees the same data
Strict consistency
Partition Tolerance
System stays up when network between nodes fails
Availability
System stays up when nodes fail
Eventual consistency

Sharding

- Sharding allows a logical database to be **partitioned** across multiple physical servers.
- In a sharded application, the largest tables are partitioned across multiple database servers.
- Each **partition** is referred to as a **shard**.
- This partitioning is based on a Key Value, such as a user ID.
- When operating on a particular record, the application must determine which shard will contain the data and then send the SQL to the appropriate server.
- Sharding is a solution used at the largest websites; Facebook and Twitter are the most well-known examples.
- At both of these websites, data that is specific to an individual user is concentrated in MySQL tables on a specific node.



Scaling with 3 web servers, 2 Memcached servers, and 3 database shards with 2 read-only database slaves per shard

--read-only traffic	--read/write traffic

- The above Figure illustrates the Memcached and replication configuration.
- In this example, there are three shards, and for simplicity's sake, the shards are labeled by the first letter of the primary key.
- Rows with the key GUY are in shard 2, while key BOB would be allocated to shard 1.
- The primary key would be hashed to ensure even distribution of keys to servers.
- The exact number of servers being used at Facebook is constantly changing and not always publicly disclosed, but in around 2011 they did reveal that they were using more than 4,000 shards of MySQL and 9,000 Memcached servers in their configuration.
- This sharded MySQL configuration supported 1.4 billion peak reads per second, 3.5 million row changes per second, and 8.1 million physical IOs per second.
- Sharding involves significant operational complexities and compromises, but it is used for achieving data processing on a massive scale.
- Sharding is simple in concept but incredibly complex in practice.
- The application must contain logic that understands the location of any particular piece of data and the logic to route requests to the correct shard.
- Sharding is usually associated with rapid growth, so this routing needs to be dynamic.
- Requests that can only be satisfied by accessing more than one shard thus need complex coding as well, whereas on a non sharded database a single SQL statement might suffice.
- Sharding—together with caching and replication—is arguably the only way to scale a relational database to massive web use.
- However, the operational costs of sharding are huge.

- The drawbacks of a sharding strategy are:
  1. **Application complexity.**
     - In a statically sharded database, routing SQL would be hard enough
     - Most massive websites are adding shards as they grow, so that a dynamic routing layer must be implemented.
     - This layer is to maintain Memcached object copies and to differentiate between the master database and read-only replicas.
  2. **Crippled SQL.**
     - In a sharded database, it is not possible to issue a SQL statement that operates across shards.
     - SQL statements are limited to row level access.
     - Joins and GROUP BY aggregate operations cannot be implemented in shards.
     - Only programmers can query the database as a whole.
  3. **Loss of transactional integrity.**
     - ACID transactions against multiple shards are not possible.
     - It also creates problems for conflict resolution, can create bottlenecks, has issues for MySQL, and is rarely implemented.
  4. **Operational complexity.**
     - Load balancing across shards becomes extremely problematic.
     - Adding new shards requires a complex rebalancing of data.
     - Changing the database schema also requires a rolling operation across all the shards, resulting in inconsistencies.
     - A sharded database involves a huge amount of operational effort and administrator skill.
