NoSQL: SQL, NoSQL, Definition, A Brief History of NoSQL, ACID vs. BASE, CAP Theorem (Brewer's Theorem), The BASE, NoSQL Advantages and Disadvantages, Advantages of NoSQL, Disadvantages of NoSQL, SQL vs. NoSQL Databases, Categories of NoSQL Databases

Introducing MongoDB: History, MongoDB Design Philosophy, Speed, Scalability, and Agility, Non-Relational Approach, JSON-Based Document Store, Performance vs. Features, Running the Database Anywhere, SQL Comparison

---

PYQ : (Previous Year Mumbai University Question)

Nov – 18

1. Compare ACID Vs BASE
2. With the help of neat diagram explain , the CAP theorem
3. What are the advantages and Disadvantage of NOSQL Database
4. What are the different categories of NOSQL Database ? Explain each with an example

Apr – 19

1. Briefly explain the CAP Theorem
2. Difference between SQL and NOSQL database

Nov – 19

1. Write a short note on CAP Theorem
2. Discuss the various categories of NOSQL database

Nov – 22

1. Explain Brewer's Theorem along with a neat diagram
2. How consistency can be implemented at both read and write operations levels explain
3. List the Categories of NOSQL databases. Also explain the ways in which MongoDB is different from SQL
4. What is NOSQL? Explain the advantages of NOSQL databases

Dec – 23

1. Compare and Contrast ACID vs BASE

2. State and Explain the advantages and disadvantages of NOSQL databases
3. Describe the categories of NOSQL database

Nov – 24

1. What are distinguishing characteristics between SQL and NOSQL technologies
2. Explain the CAP Theorem
3. How is consistency implemented at both read and write operation levels in NOSQL databases

---

"NoSQL is a new way of designing Internet-scale database solutions. It is not a product or technology but a term that defines a set of database technologies that are not based on the traditional RDBMS principles."

SQL :

Question 1: Describe the ACID properties in SQL databases with examples.

The idea of RDBMS was borne from E.F. Codd's 1970 whitepaper titled "A relational model of data for large shared data banks." The language used to query RDBMS systems is SQL (Sequel Query Language). RDBMS systems are well suited for structured data held in columns and rows, which can be queried using SQL. The RDBMS systems are based on the concept of ACID transactions. ACID stands for Atomic, Consistent, Isolated, and Durable.

ACID properties are crucial for ensuring reliable transactions in SQL databases. The four properties are:

1. **Atomicity**
2. **Consistency**
3. **Isolation**
4. **Durability**

Let’s look at each property in detail with examples:

Atomicity

Explanation :

Atomicity ensures that all operations within a transaction are completed successfully. If any operation fails, the entire transaction is rolled back, leaving the database unchanged.

Example:

Consider a bank transfer transaction where $100 is transferred from Account A to Account B. The transaction involves two operations:

1. Deduct $100 from Account A.
2. Add $100 to Account B.

If either operation fails (e.g., due to insufficient funds), the transaction is rolled back, and neither account is affected.

Consistency

Explanation :

Consistency ensures that a transaction takes the database from one valid state to another, maintaining database rules and constraints.

Example:

In a database that enforces the rule that account balances cannot be negative, any transaction that violates this rule will be rolled back to maintain consistency.

Isolation

Explanation:

Isolation ensures that transactions are executed independently of one another. Changes made by one transaction are not visible to other transactions until they are committed.

Example:

If two transactions are occurring simultaneously, such as one transaction transferring money from Account A to Account B and another querying the balance of Account B, isolation ensures that the second transaction does not see the intermediate state of the first transaction.

Durability

Explanation :

Durability ensures that once a transaction is committed, its changes are permanent, even in the case of a system crash.

Example:

After committing a bank transfer transaction, the changes (balance updates) remain in the database even if the system crashes immediately afterward.

NOSQL :

A Brief History of NoSQL :

In 1998, Carlo Strozzi coined the term NoSQL . He used this term to identify his database because the database didn't have a SQL interface. The term resurfaced in early 2009 when Eric Evans (a Rackspace employee) used this

term in an event on open source distributed databases to refer to distributed databases that were non-relational and did not follow the ACID features of relational databases.

Categories of NOSQL :

Question 1: Explain NoSQL databases with their types and provide examples.

Answer:

NoSQL databases are a type of database designed to handle large volumes of diverse data and are optimized for horizontal scaling and flexible data models. Unlike traditional SQL databases that use tables and fixed schemas, NoSQL databases can store data in various formats.

Types of NoSQL Databases:

1. **Key-Value Stores:**
   Description: Store data as key-value pairs.
   Example: Redis, Amazon DynamoDB.
   Use Case: Session management, caching.

2. **Document Stores:**
   Description: Store data as documents, usually in JSON or BSON format.
   Example: MongoDB, CouchDB.
   Use Case: Content management systems, catalogs.

3. **Column-Family Stores:**

Description: Store data in columns rather than rows, allowing for efficient read and write operations.

Example: Apache Cassandra, HBase.

Use Case: Real-time analytics, data warehousing.

4. **Graph Databases:**

Description: Store data in graph structures with nodes, edges, and properties.

Example: Neo4j, Amazon Neptune.

Use Case: Social networks, recommendation engines.

Advantages of NoSQL Databases:

1. **Scalability:** Easily scale out by adding more servers.
2. **Flexibility:** Handle unstructured, semi-structured, and structured data.
3. **Performance:** Optimized for high performance and large-scale data processing.

Example Scenario:

E-commerce Website:

Key-Value Store: Use Redis for session management to store user session data for quick access.

Document Store: Use MongoDB to store product information, including descriptions, prices, and inventory status.

Column-Family Store: Use Cassandra to store and analyze user clickstream data for personalized recommendations.

Graph Database: Use Neo4j to manage and analyze relationships in user social interactions for friend recommendations.

These NoSQL databases allow the e-commerce platform to manage various types of data efficiently and provide a seamless user experience.

Table 2-4. *Feature Comparison*

Feature	Column-Oriented	Document Store	Key-Value Store	Graph
Table-like schema support (columns)	Yes	No	No	Yes
Complete update/fetch	Yes	Yes	Yes	Yes
Partial update/fetch	Yes	Yes	Yes	No
Query/filter on value	Yes	Yes	No	Yes
Aggregate across rows	Yes	No	No	No
Relationship between entities	No	No	No	Yes
Cross-entity view support	No	Yes	No	No
Batch fetch	Yes	Yes	Yes	Yes
Batch update	Yes	Yes	Yes	No

Note The term was used initially to mean "do not use SQL if you want to scale." Later this was redefined to "not only SQL," which means that in addition to SQL other complimentary database solutions exist.

BASE :

Question: Explain BASE in Big Data Technology

BASE is an acronym that stands for **Basically Available, Soft state, Eventual consistency**. It is a consistency model used in the design of modern distributed databases, contrasting the traditional ACID properties of relational databases. Let's break down each component:

1. **Basically Available**:

- This means the system is guaranteed to be available most of the time.

- Example: Even during network partitions or failures, the system responds to queries, though the responses might not always be the most recent data.

2. **Soft state**:

- The state of the system may change over time, even without input.

- Example: Due to replication and eventual consistency, data might be in an intermediate state and not immediately synchronized across all nodes.

3. **Eventual consistency****(Eventually – After some time)** :

- The system will become consistent over time, given that no new updates are made.

- Example: In a distributed database, updates to a record might propagate to all nodes eventually, ensuring that all copies of the data will converge to the same value.

Why BASE is Important in Big Data:

- **Scalability**: BASE properties allow systems to scale out across multiple nodes, handling large volumes of data and high traffic efficiently.

- **Flexibility**: It supports systems where uptime is crucial and data can tolerate temporary inconsistency.

- **High Availability**: Ensures that the system remains operational and available even under failures, which is critical for real-time applications.

Example of BASE in Action:

Consider a NoSQL database like Cassandra:

- **Basically Available**: Cassandra is designed to ensure that the system is available for reads and writes even if some of the nodes are down.

- **Soft state**: Data in Cassandra is replicated across multiple nodes, and the state of the data might change over time as the replicas synchronize.

- **Eventual consistency**: Writes to a node in Cassandra are propagated to other nodes eventually, ensuring that all replicas have the same data over time.

ACID vs. BASE:

Question: Compare and contrast ACID and BASE properties in the context of database management systems.

Answer:

1. Consistency vs. Availability:

- **ACID:** Prioritizes consistency over availability. Every transaction ensures the database moves from one consistent state to another.
- **BASE:** Prioritizes availability over consistency. The system is always available, even if the data is not always consistent.

2. Transaction Handling:

- **ACID:** Transactions are handled in an all-or-nothing manner. Either all operations of a transaction are completed, or none are.
- **BASE:** Transactions can be partially completed and eventually consistent. Operations may be executed in a way that eventual consistency is achieved.

3. Use Case Suitability:

- **ACID:** Suitable for applications requiring strong consistency and reliability, such as banking systems.
- **BASE:** Suitable for distributed systems that can tolerate temporary inconsistencies, like social media platforms or online retail.

4. Atomicity:

- **ACID:** Ensures atomicity, meaning all parts of a transaction must succeed together.
- **BASE:** Does not enforce atomicity strictly. Partial updates are possible until consistency is achieved.

5. Consistency:

- **ACID:** Guarantees consistency after every transaction.
- **BASE:** Consistency is eventual, meaning data will become consistent over time.

6. Isolation:

- **ACID:** Ensures transactions are isolated from each other, preventing concurrent transactions from interfering with each other.
- **BASE:** Allows for some level of concurrency, and transactions may not be completely isolated.

7. Durability:

- **ACID:** Ensures that once a transaction is committed, it remains so even in case of a system failure.
- **BASE:** Durability is more flexible, and the system may be designed to handle failures in a way that does not guarantee immediate durability.

8. Complexity:

- **ACID:** Typically simpler to reason about due to strict rules and guarantees.
- **BASE:** More complex to manage due to eventual consistency and the need to handle temporary inconsistencies.

9. Performance:

- **ACID:** May have lower performance due to the overhead of maintaining strict consistency and isolation.
- **BASE:** Typically higher performance due to relaxed constraints, allowing for more parallel operations and higher availability.

10. System Design:

- **ACID:** Often implemented in traditional relational database management systems (RDBMS) like Oracle, MySQL.
- **BASE:** Commonly used in NoSQL databases and distributed systems like Cassandra, Dynamo DB.

11. Error Handling:

- **ACID:** Errors in a transaction typically result in a rollback, ensuring no partial updates.
- **BASE:** Errors may be handled with retries or partial updates, with eventual consistency being the goal.

12. Data Model:

- **ACID:** Often used with structured data models where schema and data integrity are critical.
- **BASE:** More flexible data models, often used with unstructured or semi-structured data.

Examples:

- **ACID Example:** A bank transfer where the withdrawal from one account and the deposit into another must both succeed or fail together.

- **BASE Example:** A product catalog in an e-commerce site where updates to the inventory are propagated across multiple servers, and temporary inconsistencies are acceptable.

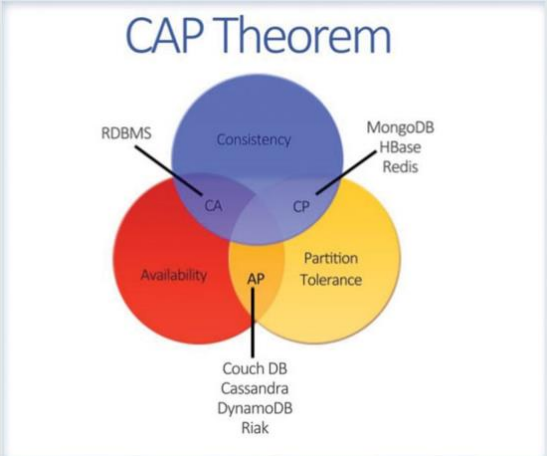
In summary, ACID properties ensure strict transaction integrity and consistency, making them ideal for critical systems. In contrast, BASE properties offer flexibility and high availability, suitable for large-scale, distributed systems that can handle temporary inconsistencies.

CAP Theorem (Brewer's Theorem) :

Question 1. **Explain the CAP Theorem with an example.**

Question 2. **What are the implications of the CAP Theorem in distributed systems?**

Question 3. **How does the CAP Theorem affect the design of distributed databases?**



Answer:

The CAP Theorem, also known as Brewer's theorem, is a principle that states that a distributed database system can only guarantee two out of the following three properties at the same time:

1. **Consistency (C):** Every read receives the most recent write or an error. This means that all nodes in the distributed system have the same data at the same time.

2. **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write. This ensures that the system is operational and responsive.

3. **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes. This means the system can handle network failures that split the network into parts.

Example:

Consider an online banking system that needs to manage user transactions across different branches:

- **Consistency:** Ensures that if you check your account balance after a deposit, you will see the updated balance immediately across all branches.

- **Availability:** Ensures that you can always access your account and perform transactions, even if some branches are temporarily unreachable.

- **Partition Tolerance:** Ensures that the system continues to function even if the network connection between some branches is lost.

Scenario:

Imagine a scenario where a network partition occurs, splitting the branches into two groups that cannot communicate with each other.

- If the system chooses **Consistency and Partition Tolerance (CP)**, it will ensure that the account balances remain consistent across all branches. However, during the partition, some branches might be unavailable to process transactions.

- If the system chooses **Availability and Partition Tolerance (AP)**, it will ensure that transactions can still be processed at all branches, but the account balances might not be immediately consistent across all branches.

- If the system chooses **Consistency and Availability (CA)**, it ensures consistent and available data, but it cannot tolerate network partitions, leading to possible system failures during network issues.

In practice, achieving all three properties simultaneously is impossible. Systems must make trade-offs based on their requirements. For instance, many NoSQL databases (like Cassandra) prioritize availability and partition tolerance (AP) over strict consistency.

This understanding of the CAP Theorem helps in designing systems that are robust and suitable for specific needs, balancing consistency, availability, and partition tolerance based on the use case.

Conclusion:

The CAP Theorem is crucial in the design and management of distributed systems, guiding architects to make informed decisions about which properties to prioritize depending on the system's requirements and expected behavior during network failures.

Question 1 : How consistency can be implemented at both read and write operations levels explain

Question 2 : How is eventual consistency implemented in NoSQL databases using the NRW notation?

Answer:

Eventual Consistency in NoSQL Databases:

Eventual consistency means that if you make a change to a piece of data, eventually, all copies of that data will be the same, even if there are delays or network issues.

NRW Notation:

NRW helps decide how many copies of the data need to agree before a change is considered successful.

- **N (Number of Replicas):** This is how many copies of the data exist.

- **R (Read Quorum):** How many copies need to agree when reading data.

- **W (Write Quorum):** How many copies need to agree when writing data.

Example:

Let's say you have a NoSQL database with 5 copies of your data (N=5). You decide that for reading (R), at least 2 copies need to agree, and for writing (W), at least 3 copies need to agree.

- If you want to read some data, you'll read from at least 2 copies.

- If you want to write some data, you'll write to at least 3 copies.

Benefits of NRW:

- It allows flexibility in choosing how many copies need to agree, based on your needs.

- It helps the system scale by adding more copies without losing consistency.

Challenges:

- Resolving conflicts that can arise when different copies of the data are changed at the same time.

- Sometimes, reading or writing data might take longer due to the need to coordinate between copies.

Conclusion:

NRW notation is a way to manage eventual consistency in NoSQL databases, ensuring that even in distributed systems, data eventually becomes consistent across all copies, balancing between consistency, availability, and fault tolerance.

Advantages and Disadvantages of NOSQL:

Question 1: What are the advantages and Disadvantage of NOSQL Database
Question 2: What is NOSQL? Explain the advantages of NOSQL databases

Advantages of NoSQL :

• High scalability : This scaling up approach fails when the transaction rates and fast response requirements increase. In contrast to this, the new generation of NoSQL databases is designed to scale out (i.e. to expand horizontally using low-end commodity servers).

• Manageability and administration : NoSQL databases are designed to mostly work with automated repairs, distributed data, and simpler data models, leading to low manageability and administration.

• Low cost : NoSQL databases are typically designed to work with a cluster of cheap commodity servers, enabling the users to store and process more data at a low cost.

• Flexible data models : NoSQL databases have a very flexible data model, enabling them to work with any type of data; they don't comply with the rigid RDBMS data models. As a result, any application changes that involve updating the database schema can be easily implemented

Disadvantages of NoSQL :

In addition to the above mentioned advantages, there are many impediments that you need to be aware of before you start developing applications using these platforms

• Maturity : Most NoSQL databases are pre-production versions with key features that are still to be implemented. Thus, when deciding on a NoSQL database, you should analyze the product properly to ensure the features are fully implemented and not still on the To-do list . • Support : Support is one limitation that you need to consider. Most NoSQL databases are from start-ups which were open sourced. As a result, support is very minimal as compared to the enterprise software companies and may not have global reach or support resources.

• Limited Query Capabilities : Since NoSQL databases are generally developed to meet the scaling requirement of the web-scale applications, they provide limited querying capabilities. A simple querying requirement may involve significant programming expertise.

• Administration : Although NoSQL is designed to provide a no-admin solution, it still requires skill and effort for installing and maintaining the solution.

• Expertise : Since NoSQL is an evolving area, expertise on the technology is limited in the developer and administrator community. Although NoSQL is becoming an important part of the database landscape, you need to be aware of the limitations and advantages of the products to make the correct choice of the NoSQL database platform

---

SQL Vs. NOSQL

Question 1: Difference between SQL and NOSQL database
Question 2: What are distinguishing characteristics between SQL and NOSQL technologies?

Table 2-2. *SQL vs. NoSQL*

	SQL Databases	NoSQL Databases
Types	All types support SQL standard.	Multiple types exists, such as document stores, key value stores, column databases, etc.
Development History	Developed in 1970.	Developed in 2000s.
Examples	SQL Server, Oracle, MySQL.	MongoDB, HBase, Cassandra.
Data Storage Model	Data is stored in rows and columns in a table, where each column is of a specific type. The tables generally are created on principles of normalization. Joins are used to retrieve data from multiple tables.	The data model depends on the database type. Say data is stored as a key-value pair for key-value stores. In document-based databases, the data is stored as documents. The data model is flexible, in contrast to the rigid table model of the RDBMS.
Schemas	Fixed structure and schema, so any change to schema involves altering the database.	Dynamic schema, new data types, or structures can be accommodated by expanding or altering the current schema. New fields can be added dynamically.
Scalability	Scale up approach is used; this means as the load increases, bigger, expensive servers are bought to accommodate the data.	Scale out approach is used; this means distributing the data load across inexpensive commodity servers.
Supports Transactions	Supports ACID and transactions.	Supports partitioning and availability, and compromises on transactions. Transactions exist at certain level, such as the database level or document level.
Consistency	Strong consistency.	Dependent on the product. Few chose to provide strong consistency whereas few provide eventual consistency.
Support	High level of enterprise support is provided.	Open source model. Support through third parties or companies building the open source products.
Maturity	Have been around for a long time.	Some of them are mature; others are evolving.
Querying Capabilities	Available through easy-to-use GUI interfaces.	Querying may require programming expertise and knowledge. Rather than an UI, focus is on functionality and programming interfaces.
Expertise	Large community of developers who have been leveraging the SQL language and RDBMS concepts to architect and develop applications.	Small community of developers working on these open source tools.

==========