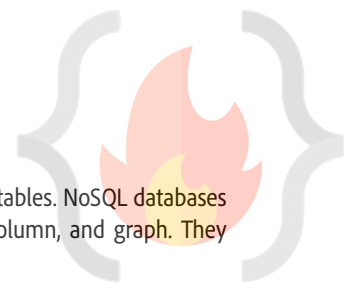


LEC-15: NoSQL



1. **NoSQL databases** (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide **flexible schemas** and **scale** easily with **large amounts of data** and **high user loads**.
 1. They are schema free.
 2. Data structures used are not tabular, they are more flexible, has the ability to adjust dynamically.
 3. Can handle huge amount of data (**big data**).
 4. Most of the NoSQL are open sources and has the capability of horizontal scaling.
 5. **It just stores data in some format other than relational.**
2. **History behind NoSQL**
 1. NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased. Gone were the days of needing to create a complex, difficult-to-manage data model in order to avoid data duplication. Developers (rather than storage) were becoming the primary cost of software development, so NoSQL databases optimised for developer productivity.
 2. Data becoming unstructured more, hence structuring (defining schema in advance) them had becoming costly.
 3. NoSQL databases allow developers to store huge amounts of unstructured data, giving them a lot of flexibility.
 4. Recognising the need to rapidly adapt to changing requirements in a software system. Developers needed the ability to iterate quickly and make changes throughout their software stack — all the way down to the database. NoSQL databases gave them this flexibility.
 5. Cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. They wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up, and to intelligently geo-place their data. Some NoSQL databases like MongoDB provide these capabilities.
3. **NoSQL Databases Advantages**
 - A. **Flexible Schema**
 1. RDBMS has pre-defined schema, which become an issue when we do not have all the data with us or we need to change the schema. It's a huge task to change schema on the go.
 - B. **Horizontal Scaling**
 1. Horizontal scaling, also known as scale-out, refers to bringing on additional nodes to share the load. This is difficult with relational databases due to the difficulty in spreading out related data across nodes. With non-relational databases, this is made simpler since collections are self-contained and not coupled relationally. This allows them to be distributed across nodes more simply, as queries do not have to "join" them together across nodes.
 2. Scaling horizontally is achieved through **Sharding** OR **Replica-sets**.
 - C. **High Availability**
 1. NoSQL databases are highly available due to its auto replication feature i.e. whenever any kind of failure happens data replicates itself to the preceding consistent state.
 2. If a server fails, we can access that data from another server as well, as in NoSQL database data is stored at multiple servers.
 - D. **Easy insert and read operations.**
 1. Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalised, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimised for queries. The rule of thumb when you use MongoDB is data that is accessed together should be stored together. Queries typically do not require joins, so the queries are very fast.
 2. But difficult delete or update operations.
 - E. **Caching** mechanism.
 - F. **NoSQL** use case is more for **Cloud** applications.
4. **When to use NoSQL?**
 1. Fast-paced Agile development
 2. Storage of structured and semi-structured data
 3. Huge volumes of data
 4. Requirements for scale-out architecture
 5. Modern application paradigms like micro-services and real-time streaming.
5. **NoSQL DB Misconceptions**
 1. Relationship data is best suited for relational databases.
 1. A common misconception is that NoSQL databases or non-relational databases don't store relationship data well. NoSQL databases can store relationship data — they just store it differently than relational databases do. In fact, when compared with relational databases, many find modelling relationship data in NoSQL databases to be easier than in relational databases, because related data doesn't have to be split between tables. NoSQL data models allow related data to be nested within a single data structure.
 2. NoSQL databases don't support ACID transactions.

1. Another common misconception is that NoSQL databases don't support ACID transactions. Some NoSQL databases like MongoDB do, in fact, support ACID transactions.

6. Types of NoSQL Data Models

1. Key-Value Stores

1. The simplest type of NoSQL database is a key-value store. Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska").
2. **Use cases** include shopping carts, user preferences, and user profiles.
3. e.g., Oracle NoSQL, Amazon DynamoDB, MongoDB also supports Key-Value store, Redis.
4. A key-value database associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).
5. Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time.
6. There are several use-cases where choosing a key value store approach is an optimal solution:
 - a) Real time random data access, e.g., user session attributes in an online application such as gaming or finance.
 - b) Caching mechanism for frequently accessed data or configuration based on keys.
 - c) Application is designed on simple key-based queries.

2. Column-Oriented / Columnar / C-Store / Wide-Column

1. The data is stored such that each row of a column will be next to other rows from that same column.
2. While a relational database stores data in rows and reads data row by row, a column store is organised as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). **Use cases** include analytics.
3. e.g., Cassandra, RedShift, Snowflake.

3. Document Based Stores

1. This DB store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
2. **Use cases** include e-commerce platforms, trading platforms, and mobile app development across industries.
3. Supports ACID properties hence, suitable for Transactions.
4. e.g., MongoDB, CouchDB.

4. Graph Based Stores

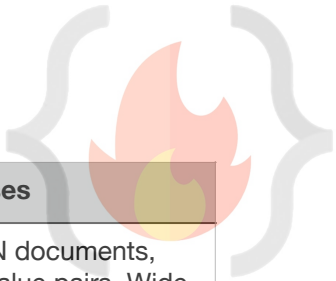
1. A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.
2. A graph database is optimised to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.
3. Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases.
4. **Use cases** include fraud detection, social networks, and knowledge graphs.

7. NoSQL Databases Dis-advantages

1. Data Redundancy

1. Since data models in NoSQL databases are typically optimised for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.
2. Update & Delete operations are **costly**.
3. All type of NoSQL Data model doesn't fulfil all of your application needs
 1. Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analysing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.
4. Doesn't support ACID properties in general.
5. Doesn't support data entry with consistency constraints.

8. SQL vs NoSQL



	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General Purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Fixed	Flexible
Scaling	Vertical (Scale-up)	Horizontal (scale-out across commodity servers)
ACID Properties	Supported	Not Supported, except in DB like MongoDB etc.
JOINS	Typically Required	Typically not required
Data to object mapping	Required object-relational mapping	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

LEC-12: Transaction



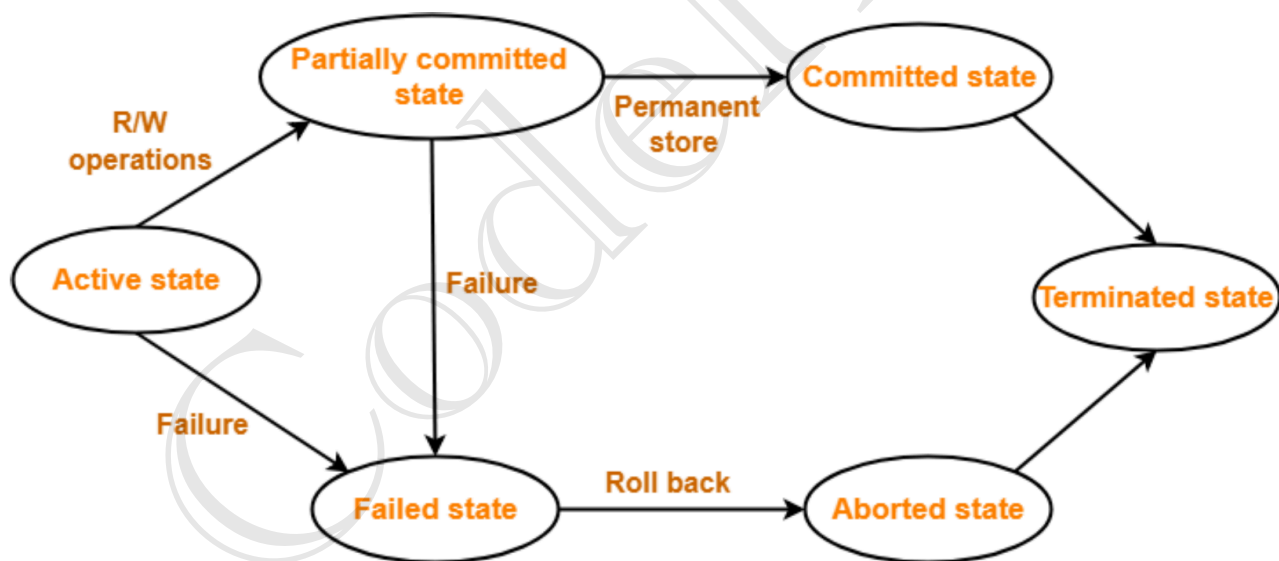
1. Transaction

1. A unit of work done against the DB in a logical sequence.
2. Sequence is very important in transaction.
3. It is a logical unit of work that contains one or more SQL statements. The result of all these statements in a transaction either gets completed successfully (all the changes made to the database are permanent) or if at any point any failure happens it gets rolled back (all the changes being done are undone.)

2. ACID Properties

1. To ensure integrity of the data, we require that the DB system maintain the following properties of the transaction.
2. **Atomicity**
 1. Either all operations of transaction are reflected properly in the DB, or none are.
3. **Consistency**
 1. Integrity constraints must be maintained before and after transaction.
 2. DB must be consistent after transaction happens.
4. **Isolation**
 1. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
 2. Multiple transactions can happen in the system in isolation, without interfering each other.
5. **Durability**
 1. After transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

3. Transaction states



Transaction States in DBMS

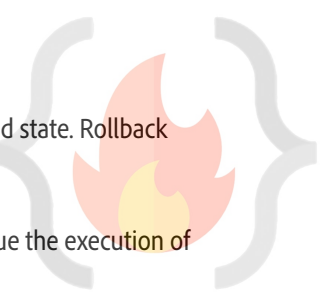
1. Active state

1. The very first state of the life cycle of the transaction, all the read and write operations are being performed. If they execute without any error the T comes to Partially committed state. Although if any error occurs then it leads to a Failed state.

2. Partially committed state

1. After transaction is executed the changes are saved in the buffer in the main memory. If the changes made are permanent on the DB then the state will transfer to the committed state and if there is any failure, the T will go to Failed state.

3. Committed state

- 
1. When updates are made permanent on the DB. Then the T is said to be in the committed state. Rollback can't be done from the committed states. New consistent state is achieved at this stage.
 4. **Failed state**
 1. When T is being executed and some failure occurs. Due to this it is impossible to continue the execution of the T.
 5. **Aborted state**
 1. When T reaches the failed state, all the changes made in the buffer are reversed. After that the T rollback completely. T reaches abort state after rollback. DB's state prior to the T is achieved.
 6. **Terminated state**
 1. A transaction is said to have terminated if has either committed or aborted.

CodeHelp