Introduction to NoSQL

NoSQL is a type of database management system (DBMS) that is designed to handle and store large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

The term NoSQL originally referred to "non-SQL" or "non-relational" databases, but the term has since evolved to mean "not only SQL," as NoSQL databases have expanded to include a wide range of different database architectures and data models.

NoSQL databases are generally classified into four main categories:

1. **Document databases:** These databases store data as semi-structured documents, such as JSON or XML, and can be queried using document-oriented query languages.
2. **Key-value stores:** These databases store data as key-value pairs, and are optimized for simple and fast read/write operations.
3. **Column-family stores:** These databases store data as column families, which are sets of columns that are treated as a single entity. They are optimized for fast and efficient querying of large amounts of data.
4. **Graph databases:** These databases store data as nodes and edges, and are designed to handle complex relationships between data.

Key Features of NoSQL:

1. **Dynamic schema:** NoSQL databases do not have a fixed schema and can accommodate changing data structures without the need for migrations or schema alterations.
2. **Horizontal scalability:** NoSQL databases are designed to scale out by adding more nodes to a database cluster, making them well-suited for handling large amounts of data and high levels of traffic.
3. **Document-based:** Some NoSQL databases, such as MongoDB, use a document-based data model, where data is stored in a schema-less semi-structured format, such as JSON or BSON.
4. **Key-value-based:** Other NoSQL databases, such as Redis, use a key-value data model, where data is stored as a collection of key-value pairs.
5. **Column-based:** Some NoSQL databases, such as Cassandra, use a column-based data model, where data is organized into columns instead of rows.
6. **Distributed and high availability:** NoSQL databases are often designed to be highly available and to automatically handle node failures and data replication across multiple nodes in a database cluster.
7. **Flexibility:** NoSQL databases allow developers to store and retrieve data in a flexible and dynamic manner, with support for multiple data types and changing data structures.
8. **Performance:** NoSQL databases are optimized for high performance and can handle a high volume of reads and writes, making them suitable for big data and real-time applications.

Advantages of NoSQL: There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

1. **High scalability:** NoSQL databases use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing

machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra, etc. NoSQL can handle a huge amount of data because of scalability, as the data grows NoSQL scales**The auto** itself to handle that data in an efficient manner.

2. **Flexibility:** NoSQL databases are designed to handle unstructured or semi-structured data, which means that they can accommodate dynamic changes to the data model. This makes NoSQL databases a good fit for applications that need to handle changing data requirements.
3. **High availability:** The auto**,** replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.
4. **Scalability:** NoSQL databases are highly scalable, which means that they can handle large amounts of data and traffic with ease. This makes them a good fit for applications that need to handle large amounts of data or traffic
5. **Performance:** NoSQL databases are designed to handle large amounts of data and traffic, which means that they can offer improved performance compared to traditional relational databases.
6. **Cost-effectiveness:** NoSQL databases are often more cost-effective than traditional relational databases, as they are typically less complex and do not require expensive hardware or software.
7. **Agility:** Ideal for agile development.

Disadvantages of NoSQL: NoSQL has the following disadvantages.

1. **Lack of standardization:** There are many different types of NoSQL databases, each with its own unique strengths and weaknesses. This lack of standardization can make it difficult to choose the right database for a specific application

2. **Lack of ACID compliance:** NoSQL databases are not fully ACID-compliant, which means that they do not guarantee the consistency, integrity, and durability of data. This can be a drawback for applications that require strong data consistency guarantees.
3. **Narrow focus:** NoSQL databases have a very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
4. **Open-source:** NoSQL is an **database**open-source database. There is no reliable standard for NoSQL yet. In other words, two database systems are likely to be unequal.
5. **Lack of support for complex queries:** NoSQL databases are not designed to handle complex queries, which means that they are not a good fit for applications that require complex data analysis or reporting.
6. **Lack of maturity:** NoSQL databases are relatively new and lack the maturity of traditional relational databases. This can make them less reliable and less secure than traditional databases.
7. **Management challenge:** The purpose of big data tools is to make the management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than in a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
8. **GUI is not available:** GUI mode tools to access the database are not flexibly available in the market.
9. **Backup:** Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
10. **Large document size:** Some database systems like MongoDB and CouchDB store data in JSON format. This means that documents are quite large (BigData, network bandwidth, speed),

and having descriptive key names actually hurts since they increase the document size.

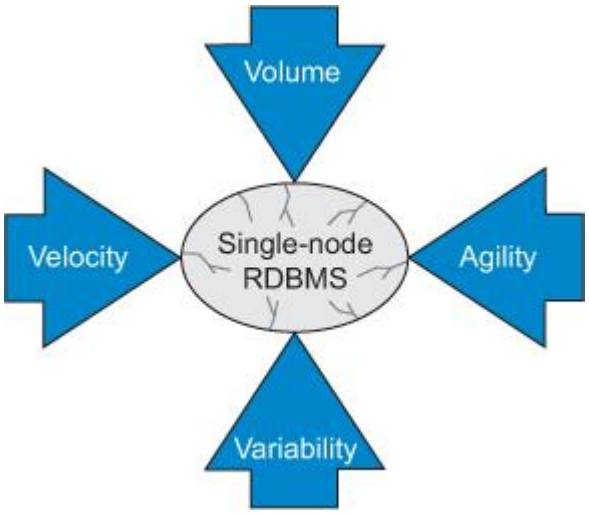
1.2. NoSQL business drivers

The scientist-philosopher Thomas Kuhn coined the term *paradigm shift* to identify a recurring process he observed in science, where innovative ideas came in bursts and impacted the world in nonlinear ways. We'll use Kuhn's concept of the paradigm shift as a way to think about and explain the NoSQL movement and the changes in thought patterns, architectures, and methods emerging today.

Many organizations supporting single-CPU relational systems have come to a crossroads: the needs of their organizations are changing. Businesses have found value in rapidly capturing and analyzing large amounts of variable data, and making immediate changes in their businesses based on the information they receive.

Figure 1.1 shows how the demands of volume, velocity, variability, and agility play a key role in the emergence of NoSQL solutions. As each of these drivers applies pressure to the single-processor relational model, its foundation becomes less stable and in time no longer meets the organization's needs.

Figure 1.1.



1.2.1. Volume

Without a doubt, the key factor pushing organizations to look at alternatives to their current RDBMSs is a need to query big data using clusters of commodity processors. Until around 2005, performance concerns were resolved by purchasing faster processors. In time, the ability to increase processing speed was no longer an option. As chip density increased, heat could no longer dissipate fast enough without chip overheating. This phenomenon, known as the power wall, forced systems designers to shift their focus from increasing speed on a single chip to using more processors working together. The need to scale out (also known as *horizontal scaling*), rather than scale up (faster processors), moved organizations from serial to parallel processing where data problems are split into separate paths and sent to separate processors to divide and conquer the work.

1.2.2. Velocity

Though big data problems are a consideration for many organizations moving away from RDBMSs, the ability of a single processor system to rapidly read and write data is also key. Many single-processor RDBMSs are unable to keep up with the demands of real-time inserts and online queries to the database made by public-facing websites. RDBMSs frequently index many columns of every new row, a process which decreases system performance. When single-processor RDBMSs are used as a back end to a web store front, the random bursts in web traffic slow down response for everyone, and tuning these systems can be costly when both high read and write throughput is desired.

1.2.3. Variability

Companies that want to capture and report on exception data struggle when attempting to use rigid database schema

structures imposed by RDBMSs. For example, if a business unit wants to capture a few custom fields for a particular customer, all customer rows within the database need to store this information even though it doesn't apply. Adding new columns to an RDBMS requires the system be shut down and ALTER TABLE commands to be run. When a database is large, this process can impact system availability, costing time and money.

1.2.4. Agility

The most complex part of building applications using RDBMSs is the process of putting data into and getting data out of the database. If your data has nested and repeated subgroups of data structures, you need to include an object-relational mapping layer. The responsibility of this layer is to generate the correct combination of INSERT, UPDATE, DELETE, and SELECT SQL statements to move object data to and from the RDBMS persistence layer. This process isn't simple and is associated with the largest barrier to rapid change when developing new or modifying existing applications.

1.3. NoSQL case studies

1.3.1. Case study: LiveJournal's Memcache

Background: LiveJournal faced performance issues due to growing traffic and increased demand on their web servers. Each server had its own RAM, leading to inefficient use of memory and repeated SQL queries.
Solution: Engineers at LiveJournal developed Memcache to improve performance. They created a system to cache the results of frequently used database queries in RAM. Each query was assigned a unique "signature" or hash, allowing servers to check if another server had already cached the query result. This reduced the load on the SQL database by avoiding redundant queries.

Impact: Memcache significantly enhanced performance and scalability for LiveJournal. The engineers also open-sourced Memcache and standardized the memcached protocol, enabling others to use and benefit from the technology to manage database load more effectively.

1.3.2. Case study: Google's MapReduce—use commodity hardware to create search indexes

Background: Google's MapReduce system transformed massive amounts of web data into search indexes using low-cost commodity hardware.
Solution: MapReduce consists of two main stages:

1. Map: Extracts, transforms, and filters data.
2. Reduce: Sorts, combines, and summarizes the results.

While the concepts of map and reduce have been around since the 1950s, Google adapted them to process data across thousands of low-cost CPUs efficiently.
Impact: Google's MapReduce demonstrated that functional programming could scale effectively on large datasets with inexpensive hardware. It inspired open-source implementations like Hadoop and renewed interest in functional programming for distributed systems.

1.3.3. Case study: Google's Bigtable—a table with a billion rows and a million columns

Background: Google's Bigtable was designed to handle massive datasets from web crawlers, which were too large for traditional relational databases.
Solution: Bigtable is a distributed storage system that scales easily with data growth without requiring costly hardware. It provides a single, large table for storing structured data and operates across multiple data centers globally.
Impact: Bigtable successfully managed Google's extensive data needs and influenced the development of similar technologies, such as Apache HBase and Apache Cassandra.

1.3.4. Case study: Amazon's Dynamo—accept an order 24 hours a day, 7 days a week

Background: Amazon needed a highly reliable system to support its global web storefront, operating 24/7 without interruptions. Traditional relational databases proved inadequate for their needs.
Problem: Amazon's global customer base shops around the clock, and any downtime could result in significant financial losses. The company required a system that could handle constant transactions with high reliability and scalability.
Solution: Amazon developed Dynamo, a highly available key-value store, to address these challenges. The key-value model allowed for easier data replication and high reliability compared to relational databases. Dynamo's design supported Amazon's continuous operation, ensuring robust and scalable performance.
Impact: Dynamo's introduction marked a significant shift in the NoSQL movement. It demonstrated that key-value stores could offer a reliable, extensible solution for 24/7 online operations, helping Amazon become a leading global retailer.

1.3.5. Case study: MarkLogic

Background: Founded in 2001 by engineers experienced in document search, MarkLogic specializes in managing large XML document collections.
Solution: MarkLogic uses a two-node architecture:

- **Query Nodes:** Handle query requests and coordinate the execution.
- **Document Nodes:** Store XML documents and execute queries locally.

This design, which processes queries where the documents reside, enables linear scalability with petabytes of data.

Impact: MarkLogic became popular among U.S. federal agencies and large publishers for its scalable document storage and search capabilities. It supports ACID transactions, role-based access control, and various programming languages, evolving into a general-purpose document store. It is a commercial product requiring licensing for datasets over 40 GB.