What is NoSQL?

Source: https://www.mongodb.com/nosql-explained

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

- Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.
- Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
- Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
- Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

NoSQL Database Types

- **Document databases** pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- **Graph stores** are used to store information about networks of data, such as social connections. Graph stores include Neo4J and Giraph.
- **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. Examples

of key-value stores are Riak and Berkeley DB. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.
- **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Agile sprints, quick schema iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Geographically distributed scale-out architecture instead of expensive, monolithic architecture

Dynamic Schemas

Relational databases require that schemas be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed. Developers have typically had to add application-side code to enforce data quality controls, such as mandating the presence of specific fields, data types or permissible values. More sophisticated NoSQL databases allow validation rules to be applied within the database, allowing users to enforce governance across data, while maintaining the agility benefits of a dynamic schema.

Auto-sharding

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure acceptable performance for cross-table joins and transactions. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution to support rapidly growing applications is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

"Sharding" a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary infrastructure administration tasks. Developers no longer need to construct complex, expensive platforms to support their applications, and can concentrate on writing application code. Commodity servers can provide the same processing and storage capabilities as a single high-end server for a fraction of the price.

Replication

Most NoSQL databases also support automatic database replication to maintain availability in the event of outages or planned maintenance events. More sophisticated NoSQL databases are fully self-healing, offering automated failover and recovery, as well as the ability to distribute the database across multiple geographic regions to withstand regional failures and enable data localization. Unlike relational databases, NoSQL databases generate have no requirement for separate applications or expensive add-ons to implement replication.

Integrated Caching

A number of products provide a caching tier for SQL database systems. These systems can improve read performance substantially, but they do not improve write performance, and they add operational complexity to system deployments. If your application is dominated by reads then a distributed cache could be considered, but if your application has just a modest write volume, then a distributed cache may not improve the overall experience of your end users, and will add complexity in managing cache invalidation.

Many NoSQL database technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer. Some NoSQL databases also offer fully managed, integrated in-memory database management layer for workloads demanding the highest throughput and lowest latency.

NoSQL vs. SQL Summary

	SQL Databases	NOSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in late 2000s to deal with limitations of SQL databases, especially scalability, multi-structured data, geo-distribution and agile development sprints
Examples	MySQL, Postgres, Microsoft SQL Server, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Related data is stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary.	Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored as BLOBs within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.

Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic, with some enforcing data validation rules. Applications can add new fields on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required, and core relational features such as JOINs, referential integrity and transactions are typically lost.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source
Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB, with tunable

consistency for reads) whereas others offer eventual consistency (e.g., Cassandra).