What are NoSQL databases ?

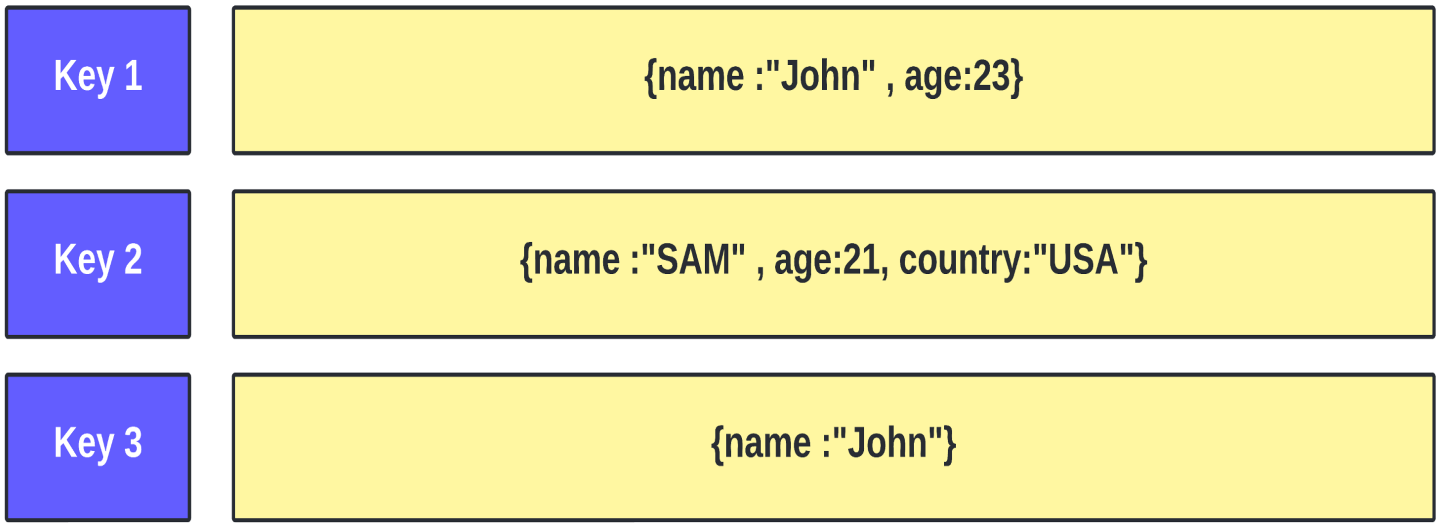
NoSQL refers to **Non-relational SQL** or **Not-Only SQL**. It allows users to store and retrieve data that is modeled in means other than the tabular relations.

Since the database schema is flexible there are different types of NoSQL databases. A few of them are:

Key-Value NoSQL database:

It stores data as a collection of key-value pairs in which the key serves as a unique identifier. Each value can have a different schema. It can range from simple objects to complex objects.

Redis, Amazon Dynamo DB, and Riak are a few popular Key-Value NoSQL databases.



Each value can have different schema

Document NoSQL database:

In document NoSQL database, instead of storing data in the tabular form, it stores the object and all its metadata in a document. Each document stores the data in **field-value** pairs. Each document can

have a different schema as per the need.

Documents can be stored in formats like JSON, XML, etc.

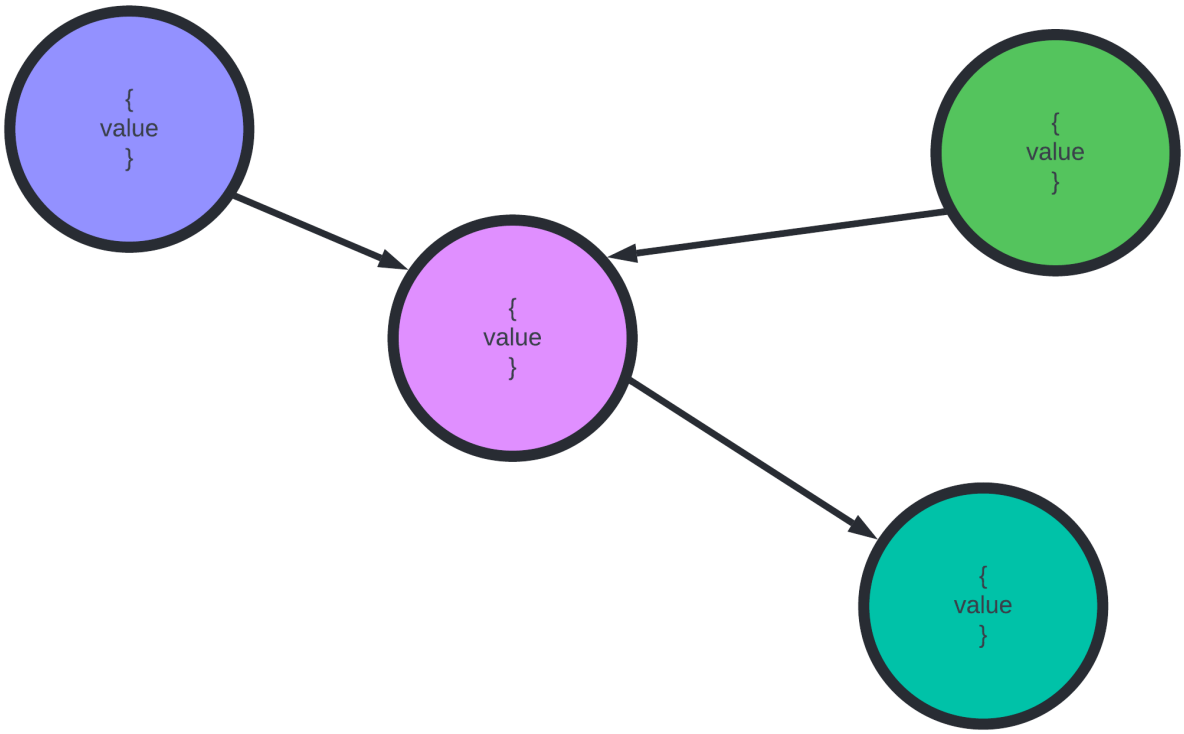
MongoDB, Azure CosmosDB, Apache CouchDB are few Document NoSQL databases.

```
"document1" : {
    name : "John",
    age:26,
    address:{
      country:"USA",
      code:123
    },
    metadata: {...}
},
"document2" : {
    name : "Sam",
    age:26,
    metadata:{...}
}
```

Graph NoSQL database:

In a graph database, nodes are used to store data, and edges are used to define the edges between those entities.

Neo4j, Nebula Graph are a few Graph NoSQL databases.



Advantages of NoSQL databases over SQL databases.

Insertions and retrievals

In a NoSQL database, the entire object and its metadata are stored in a single document. So if we want all the relevant data then we can simply get that document. However, in SQL databases, the pointer has to sequentially read all the values (for that row). Also, it needs if another relation references the current relation then we might need a join (which is expensive). So **Data retrieval is faster in the case of NoSQL database**.

For insertions, we usually collect all metadata for an object and then insert it (Just like we collect values of all attributes and then insert the row). Since a document contains all the data for an object we just need to add a new document. So **Insertions are faster in case of NoSQL database.**

E.g., When we are adding a new user to the database we don't store the name and age and leave the rest of the details for later. We try to collect as much data as possible. Similarly, if we need to display a user profile we need all relevant data.

Flexibility

NoSQL databases provide flexible schema. We can add certain fields to some documents while ignoring the rest.

E.g., We can have a field called **middle_name** for users who have a middle name and omit it for users who have only first and last name.

SQL databases do not provide such flexibility. We can either add a new field for all users (by the field we mean an attribute) or not add any new field.

Considering the above example, we have to add a new attribute called **middle_name**. Users who have first and last names only will have this attribute set to NULL.

Also, if we are adding a field that is not needed only for new insertions we can easily do it in NoSQL databases. However, it is a very expensive operation in the case of SQL databases (because we need locks for data consistency).

Scalability and Availability

NoSQL databases usually relax some of the ACID properties of relational databases for scaling horizontally. NoSQL databases also have inbuilt horizontal partitioning. NoSQL databases also tradeoff consistency to provide availability.

So **NoSQL databases usually provide more scalability and availability compared to SQL databases**

NoSQL databases are built for data aggregation/analytics

Advantages of SQL over NoSQL databases.

Updates in data

Updates are not inherently supported in NoSQL databases. If we want to update a field we have to delete the document or key-value pair and add the new document with the updated field.

Updating values is much simpler in the case of SQL databases.

So, **NoSQL databases are not suitable when we have frequent updates.**

Transactions

As we have discussed earlier, NoSQL databases relax some ACID properties for scalability, which means we cannot have transactions in NoSQL databases (which is the reason why financial systems do not use NoSQL databases).

Joins

Joining is much more difficult in NoSQL databases. To join two NoSQL tables we have to run through every block of data, find the relevant fields that are needed for joining, and then merge both the tables. So we can say, **joins are manual in the case of NoSQL databases**.

Compared to that SQL databases are built for joins.

Relations are not implicit in the case of NoSQL databases

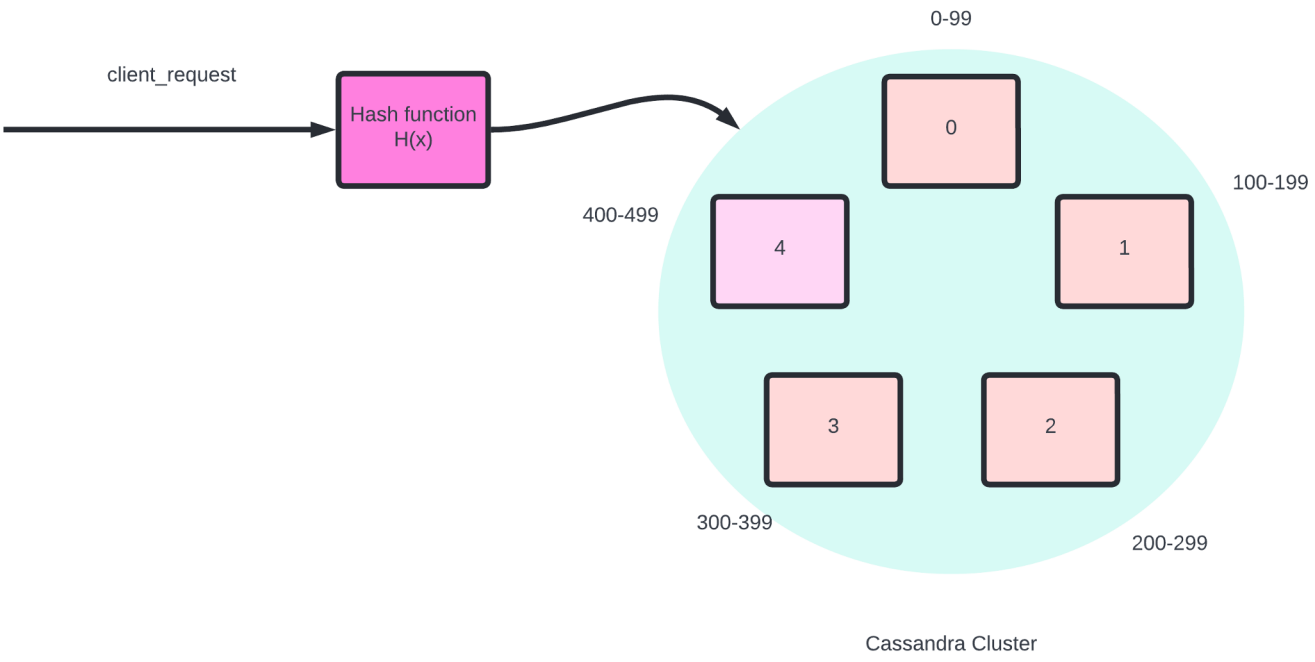
Now that we know are advantages of SQL and NoSQL databases let us discuss the working of a popular **NoSQL** database called **Cassandra**.

What is Cassandra?

Cassandra is a free and open-source, distributed, NoSQL database system.

How does Cassandra work?

- Let's say we have a distributed Cassandra cluster with 5 nodes.
- We have a hash function that generates uniformly random values between 0 and 499. (The resultant value is called a key)
- Requests from 0 to 99 are routed to Node 0.
  Requests from 100 to 199 are routed to Node 1 and so on.



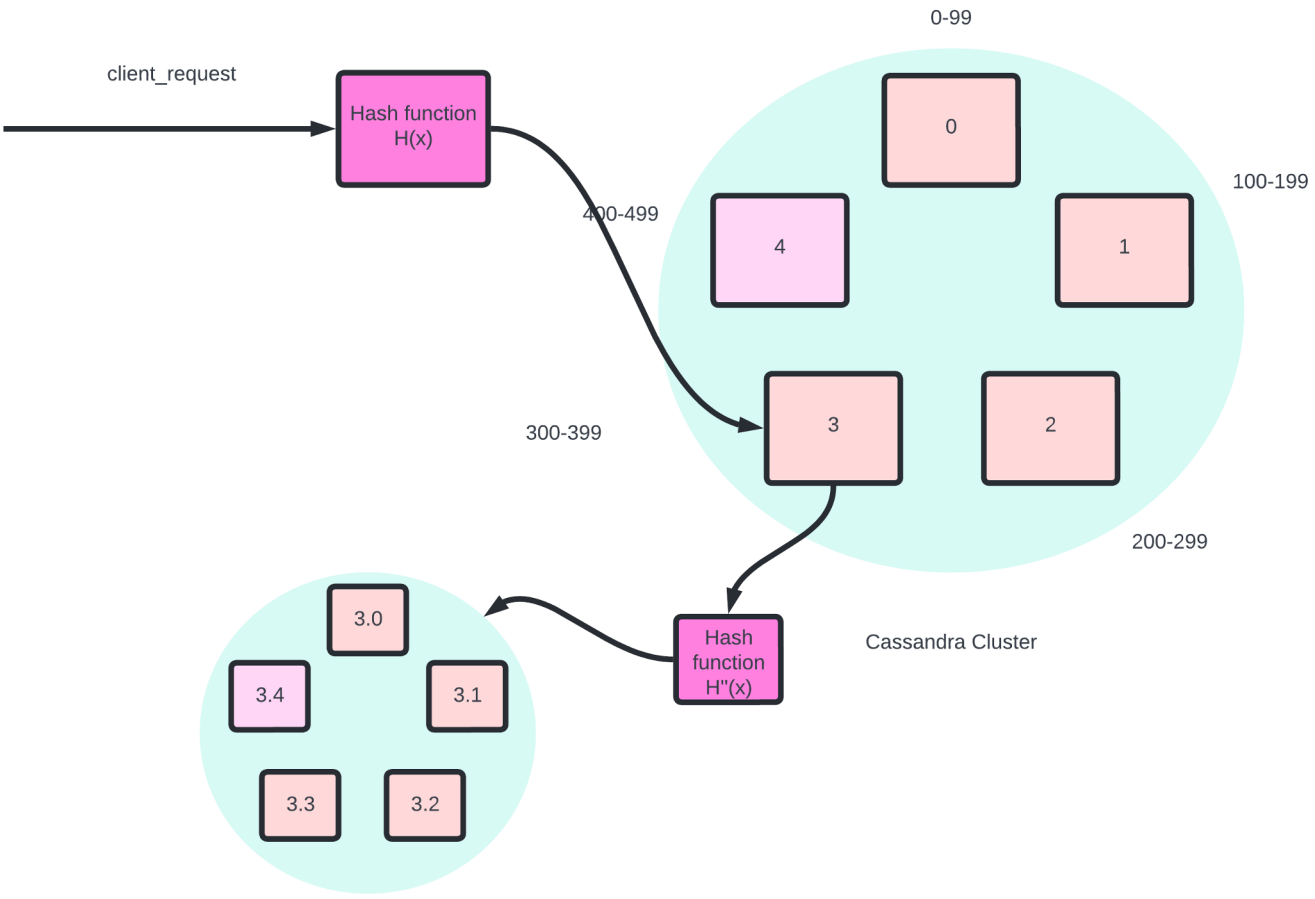
Distributing load across nodes

Since there are 5 nodes and our hash function generates uniformly random keys then it is safe to assume that load is distributed uniformly.

Now let's assume that the load on a particular node (let's say 3) increases (maybe due to a bad hash function or we get a lot of requests from the same ID etc.)

To solve this issue we can add another layer of clusters.

In our case, we will add another cluster to node 3. Now when 3 gets a request, rather than storing the request in its database it sends the request to its cluster. We can use another hash function to distribute the load.



Depending upon the needs we can have multiple levels of clusters.

E.g., if we have a hash function that uses a country name to generate keys then certain nodes will have more load than others (India will have more users and all the users from India will have the same keys so they will be routed to the same node). So we can go for multi-level sharing to prevent database overload.

Avoiding loss of data

Once we have persisted data in a particular node we do not want to lose it. Even if that node crashes. To avoid data loss we can store the copy of data on another node (Select the next nodes to store the data).

Now when we have a query we check for the data in the required node and its next nodes.
If we have 2 replicas then we need to only check the next node. If we have 3 replicas we check the next 2 nodes.

It also provides us with the following features:

- **Load balancing**
- **Redundancy for fast read queries**
- **Replication for data guarantee**

Quorum

One piece of data is stored in multiple nodes. So we need some sort of mechanism so that nodes agree on a particular return value.

Need for quorum

Let's take an example,

- Suppose we replicate the data on 3 nodes. Node 0,1 and 2.
- We made a written request to node 0. Data is written on Node 0 and Node 1 and 2 are waiting for the data.
- Meanwhile, we have a read request for the newly inserted data but Node 0 fails.
- Since Node 0 fails it checks in Node 1 and 2. But the replicas are still not inserted in these nodes.
- So our database sends a **data not found** error which is false.
- Instead, the database should have sent **database error**.

To avoid such issues we need a **distributed consensus** and one way to implement that is through Quorum.

How does quorum work?

In quorum, if a **particular number of nodes (which is called the Quorum factor) agree on a particular value that value is sent back.**

If we have data on multiple nodes then we take the data with the latest timestamp (Timestamp is just one factor. You can choose any factor as per your use case). So even if one of our nodes fails users can still read the data.

Does quorum guarantee that the return value will always be correct? **NO.**

Considering the above example only. Suppose we have a quorum factor of 2. There are 3 participating nodes and 2 nodes agree that data does not exist (which is false).

If we have a Quorum factor of 3 then the query will fail.

So, **low quorum factor provides more availability at the cost of consistency** whereas **high quorum factor provides more consistency but makes the system less available.**