SQL databases 🡪 Relational Databases (RDBMS);

NoSQL database 🡪 non-relational or distributed database.

SQL databases 🡪 table based

NoSQL databases 🡪 document based, key-value pairs, graph databases or wide-column stores.

SQL databases 🡪 predefined schema

NoSQL databases 🡪 dynamic schema for unstructured data.

SQL databases 🡪 vertically scalable

NoSQL databases🡪 horizontally scalable

**SPEED:**

SQL databases: slow as compared to NoSQL

NoSQL databases: fast

**PERFORMANCE:**

SQL databases: poor compared to NoSQL

NoSQL databases: good

**For complex queries:**

SQL databases --> good

NoSQL databases --> not good

**For the type of data to be stored:**

SQL databases --> not best fit for hierarchical data storage.

NoSQL database --> better for the hierarchical data storage

**For properties:**

SQL databases emphasizes on **ACID** properties **( Atomicity, Consistency, Isolation and Durability)**

NoSQL database follows the **Brewers CAP theorem** **( Consistency, Availability and Partition tolerance )**

* In NoSQL data is stored in many ways: it can be column-oriented, document-oriented, graph-based or organized as a KeyValue store. This flexibility means that:
  + - You can create documents without having to first define their structure.
    - Each document can have its own unique structure.
    - The syntax can vary from database to database.
    - You can add fields as you go.
* NoSQL database examples: MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb
* NoSQL database are highly preferred for large data set (i.e for big data). Hbase is an example for this purpose.
* **Schema-less data storage is useful in the following scenarios:**
  + - Fast access to the data and simplicity of access than reliable transactions or consistency.
    - Storing a large volume of data, and don’t want to lock into a schema, as changing the schema later could be slow and painful.
    - Unstructured data from one or more sources and want to keep the data in its original form for maximum flexibility.
    - Want to store data in a hierarchical structure, but you want those hierarchies to be described by the data itself, not an external schema.
  + **Shared-nothing architecture**
    - In a shared-nothing design, each server node in the cluster operates independently of every other node. The system doesn’t have to get consensus from every single node to return a piece of data to a client. Queries are fast because they can be returned from whichever node is closest or most convenient.
    - Another advantage of shared-nothing is **resiliency** and **scale-out**. Scaling out the cluster is as easy as spinning up new nodes in the cluster and waiting for them to sync with the others. If a NoSQL node goes down, the other servers in the cluster will continue to chug along. All the data remains available, even if fewer nodes are available to serve requests.
  + **EXAMPLES:**
    - MySQL is a strong choice for any business that will benefit from its pre-defined structure and set schemas. For example, applications that require multi-row transactions - like accounting systems or systems that monitor inventory.
    - MongoDB, on the other hand, is a good choice for businesses that have rapid growth or databases with no clear schema definitions. More specifically, if your schema continues to change - as is often the case with mobile apps, real-time analytics, content management systems, etc.- MongoDB can be a strong choice for you.
  + **NoSQL limitations:**
    - If NoSQL provides so much freedom and flexibility, why not abandon SQL entirely? The simple answer: Many applications still call for the kinds of constraints, consistency, and safeguards that SQL databases provide. In those cases, some “advantages” of NoSQL may turn to disadvantages.
    - **Eventual consistency**

Instead of immediate consistency across the cluster, you have eventual consistency, due to the time needed to copy updates to other nodes in the cluster. Data inserted into the cluster is eventually available everywhere, but you can’t guarantee when.

Transaction semantics, which in a SQL system guarantee that all steps in a transaction (e.g. executing a sale and reducing inventory) are either completed or rolled back, aren’t typically available in NoSQL. For any system where there needs to be a “single source of truth,” such as a bank, the NoSQL approach won’t work well.

* + - **NoSQL lock-in**

Most NoSQL systems are conceptually similar, but are implemented very differently.

**MongoDB Commands**

* show dbs (it shows the avaliable databases).
* show collections (shows all collections in current database)
* show users (shows users in current database).
* use <database\_name>. (we can also use name of the database that doesn’t exist, in that case it will be created)
* db (shows current database)
* db.createCollection('<collection\_name>');
* db.createUser({ (It will create a new user)

user:"ankur",

pwd:"1234",

roles: ["readWrite","dbAdmin"]

});

//==================================

**DATABASE COMMANDS**

* db.copyDatabase(‘from’,’to’)
* db.dropDatabase() (drop the current database)
* db.shutdownServer() (shuts down the server only if database is admin)

**COLLECTION COMMANDS**

* db.<collection\_name>.copyTo('<new\_collection\_name>') (copies collection\_name to new\_collection\_name)
* db.<collection\_name>.count() (counts number of documents in collection)
* db.<collection\_name>.drop() (remove <collection\_name> from databse)
* db.<collection\_name>.renameCollection('<new\_name>')

**QUERY COMMAND (db.<collection\_name>)**

* db<collection\_name>find().pretty() (It will display all the data )
* db.<collection\_name>.find({},{name:true,\_id:false}) (only either true or false except with \_id)
* db.file1.findOne() (find any arbitrary document)
* db.file1.findOne({name:'Steve watson'}) (find one document by attribute)
* db.<collection\_name>.insert({first\_name:"Ankur",last\_name:"Khandelwal"});
* db.inventory.insertMany([

{ item: "notebook", qty: 50, status: "A",

size: { h: 8.5, w: 11, uom: "in" }, tags: [ "red", "blank" ] },

{ item: "paper", qty: 100, status: "D",

size: { h: 8.5, w: 11, uom: "in" }, tags: [ "red", "blank", "plain" ] },])

* db.<collection\_name>.update({<attribute of document to be replaced>},{<attribute to be replaced>}) (it will replace the entire document)
* db.<collection\_name>.update({<to be replaced>},{$set:{<add\_new\_data>}})
* db.<collection\_name>.update({<to be replaced>},{$inc:{<add\_new\_data>}})
* db.<collection\_name>.update({<to be replaced>},{$unset:{<field\_name>}});

//---------------------------------

db.<package\_name>.remove({<key:value>})

db.<package\_name>.remove({<key:value>},{justone:true}) (only first match will be deleted)

db.<package\_name>.update({<to be replace>},{$rename:{"old\_key":"new\_key"}}

//=========================================

db.<package\_name>.find({key:value})

db.<package\_name>.find({age:{$lt:40}})

[gt-->greater then || lt-->less than || lte-->less than equal to || gte ]

db.customers.find().forEach(function(docs){print("Customer Name:"+doc.first\_name)})