<https://docs.mongodb.com/manual/>

NoSQL, known as Not only SQL database, provides a mechanism for storage and retrieval of data and is the next generation database

Most of the NoSQL are open source and it has a capability of horizontal scalability which means that commodity kind of machines could be added

The capacity of your clusters can be increased. It is schema free and there is no requirement to design the tables and pushing the data to it. NoSQL provides easy replication claiming there are very less manual interventions in this.

The crucial factor about NoSQL is that it can handle huge amount of data and can achieve performance by adding more machines to your clusters and can be implemented on commodity hardware. There are close to 150 NoSQL databases in the market which will make it difficult to choose  to choose the right pick for your system.

There is no specific definition for what NoSQL is, but a set of common observations can describe it:

* Not using the relational model
* Running well on clusters
* Mostly open-source
* Built for the 21st century web estates
* Schema-less

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.

This data came in all shapes and sizes—structured, semistructured, and polymorphic—and defining the schema in advance became nearly impossible. NoSQL databases allow developers to store huge amounts of unstructured data, giving them a lot of flexibility.

NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

**Why NoSQL?**

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

There are different kinds of data that are structured, unstructured and semi-structured and hence RDBMS systems are not designed to manage these types of data in an efficient way. NoSql databases comes in to the picture and are capable to manage it .

In today’s time data is becoming easier to access and capture through third parties such as Facebook, Google+ and others. Personal user information, social graphs, geo location data, user-generated content and machine logging data are just a few examples where the data has been increasing exponentially. To avail the above service properly, it is required to process huge amount of data. Which SQL databases were never designed. The evolution of NoSql databases is to handle these huge data properly.

**RDBMS vs NoSQL**

**RDBMS** - Structured and organized data - Structured query language (SQL) - Data and its relationships are stored in separate tables. - Data Manipulation Language, Data Definition Language - Tight Consistency

**NoSQL** - Stands for Not Only SQL - No declarative query language - No predefined schema - Key-Value pair storage, Column Store, Document Store, Graph databases - Eventual consistency rather ACID property - Unstructured and unpredictable data - CAP Theorem - Prioritises high performance, high availability and scalability - BASE Transaction

**Brief History of NoSQL Databases**

* 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
* 2000- Graph database Neo4j is launched
* 2004- Google BigTable is launched
* 2005- CouchDB is launched
* 2007- The research paper on Amazon Dynamo is released
* 2008- Facebooks open sources the Cassandra project
* 2009- The term NoSQL was reintroduced

**Features of NoSQL**

**Non-relational**

* NoSQL databases never follow the [relational model](https://www.guru99.com/relational-data-model-dbms.html)
* Never provide tables with flat fixed-column records
* Work with self-contained aggregates or BLOBs
* Doesn't require object-relational mapping and data normalization
* No complex features like query languages, query planners,referential integrity joins, ACID

**Schema-free**

* NoSQL databases are either schema-free or have relaxed schemas
* Do not require any sort of definition of the schema of the data
* Offers heterogeneous structures of data in the same domain

**Simple API**

* Offers easy to use interfaces for storage and querying data provided
* APIs allow low-level data manipulation & selection methods
* Text-based protocols mostly used with HTTP REST with JSON
* Mostly used no standard based NoSQL query language
* Web-enabled databases running as internet-facing services

**Distributed**

* Multiple NoSQL databases can be executed in a distributed fashion
* Offers auto-scaling and fail-over capabilities
* Often ACID concept can be sacrificed for scalability and throughput
* Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
* Only providing eventual consistency
* Shared Nothing Architecture. This enables less coordination and higher distribution.

**Types of NoSQL Databases**

* Key-value Pair Based
* Column-oriented Graph
* Graphs based
* Document-oriented

**Key Value Pair Based**

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

* In the key-value storage, database stores data as hash table where each key is unique and the value can be string, JSON, BLOB (Binary Large OBjec) etc.
* A key may be strings, hashes, lists, sets, sorted sets and values are stored against these keys.
* For example a key-value pair might consist of a key like "Name" that is associated with a value like "Robin".

**Column-based**

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, [business intelligence](https://www.guru99.com/business-intelligence-definition-example.html), CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

**Document-Oriented:**

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

* A collection of documents
* Data in this model is stored inside documents.
* A document is a key value collection where the key allows access to its value.
* Documents are not typically forced to have a schema and therefore are flexible and easy to change.
* Documents are stored into collections in order to group different kinds of data.
* Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

Relational Vs. Document

In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

**Graph-Based**

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices.

The following picture presents a labeled graph of 6 vertices and 7 edges.

* Each node represents an entity (such as a student or business) and each edge represents a connection or relationship between two nodes.
* Every node and edge are defined by a unique identifier.
* Each node knows its adjacent nodes.
* As the number of nodes increases, the cost of a local step (or hop) remains the same.
* Index for lookups.

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.

Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

**What is the CAP Theorem?**

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

1. Consistency
2. Availability
3. Partition Tolerance

**Consistency:**

The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

**Availability:**

The database should always be available and responsive. It should not have any downtime.

**Partition Tolerance:**

Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

In theoretically it is impossible to fulfill all 3 requirements. CAP provides the basic requirements for a distributed system to follow 2 of the 3 requirements.

Therefore all the current NoSQL database follow the different combinations of the C, A, P from the CAP theorem.

Here is the brief description of three combinations CA, CP, AP :

**CA -** Single site cluster, therefore all nodes are always in contact. When a partition occurs, the system blocks. **CP -**Some data may not be accessible, but the rest is still consistent/accurate. **AP -** System is still available under partitioning, but some of the data returned may be inaccurate.

**Eventual Consistency**

The term "eventual consistency" means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

BASE: **B**asically **A**vailable, **S**oft state, **E**ventual consistency

**A BASE system gives up on consistency.**

* **B**asically**A**vailable indicates that the system *does* guarantee availability, in terms of the CAP theorem. Basically, available means DB is available all the time as per CAP theorem
* **S**oft state indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
* **E**ventual consistency indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Mongo DB Features

- database -> collections -> documents

- doesn't need to have pre-defined schema, we can create fields on fly

- Sharding

- Map/Reduce Framework

======================

Installation

======================

db.createCollection(name, options)

Options

- capped - True/False - fixed size collection

- size - max size in bytes

- AutoIndexId - automatically create index on ID Field

- max - max documents allowed in capped collection

Time Series Example

db.createCollection("weatherReport", {timeSeries : {timeField : "timestamp", metaField : "metadata", granularity : "hours"}})

db.weatherReport.insert(

{

"metadata" : [{"type" : "temperature"}],

"timestamp" : ISODate("2021-03-8..."),

"temp" : 10

},

{

"metadata" : [{"type" : "temperature"}],

"timestamp" : ISODate("2021-03-8..."),

"temp" : 10

},

{

"metadata" : [{"type" : "temperature"}],

"timestamp" : ISODate("2021-03-8..."),

"temp" : 10

}

)

JSON vs BSON

CRUD Operations

CRUD - Create Read Update Delete

Data Types in Mongo

- Integer

- Double

- Boolean

- String

- Arrays

- Object

- Binary

- Min/Max Keys (min = -1, max = 127)

- RegularExpression - regex

- Symbol

- Javascript

- Javascript code with scope - deprecated

- Date

- TimeStamp

CRUD - Create Read Update Delete

Create

- insert(data, options)

- insertOne(data, options)

- insertMany

Read

- find

- findOne

Update

- updateOne (<filter>, <update>)

- updateMany

- replaceOne

- findOneAndUpdate

Delete

- deleteOne

- deleteMany

- remove

Query Selectors & Projection Operators

Query Selectors

- Comparison

- Array

- Logical

- Element

- Comments

- Geospatial

- Evaluation

Projection Operators

- $

- $elemMatch

- $meta

- $slice

Comparison Operatos

- $eq

- $gt

- $lt

- $gte

- $lte

- $in

- $ne

- $nin

Logical Operators

- $and

- $or

- $not

- $nor

Element Operators

- $exists

- $type

Evaluation Operators

- $expr

- $jsonSchema

- $mod

- $text

- $regex

- $where

Schema Validations

dbpath

logpath

config file

ordered inserts

Write Concern

import data

mongoimport "path\_to\_json" -d db\_name -c collection\_name --jsonArray --drop

mongoimport "F:\\NCU\_MongoDB\\summary.json" -d covid\_db -c summary --jsonArray --drop

mongoimport "F:\\NCU\_MongoDB\\summary.json" -d covid\_db -c summary --drop

Array Projection

db.employees.insertMany([{\_id : "emp\_103", name : "Sam"}, {\_id : "emp\_102", name : "Tom"}, {\_id : "emp\_104", name : "Shawn"}], {ordered : false})

db.employees.insertOne({\_id : "emp\_105", name : "Max"}, {writeConcern : {w : 0}})

db.employees.insertOne({\_id : "emp\_105", name : "Max"}, {writeConcern : {w : 1}})

Graph Data Structure

No Sql DB

- Document DB

- Graph DB

select name from person left join person\_dept on person.id = person\_dept.id left join department on department.id = person\_dept.dept\_id where department.name = "IT"

MATCH (p:person) - [:works\_at] -> (d:dept)

WHERE d.name = "IT"

RETURN p.name

MATCH(n) RETURN(n)

create(person)

create(bank), (employee);

create(person:Ram)

create(person:Shyam:Student:NCU)

create(Mohan:Person{name:"Mohan Sharma", age:30, occupation : "Employee"})

create(Mohan:Person{name:"Mohan Sharma", age:30, occupation : "Employee"}) RETURN Mohan

Create RelationShip

create(Rohan:person{name:"Rohan Sharma", age:30, occupation : "Employee"})

create(HDFC:bank{name:"HDFC"})

CREATE(Rohan)-[r:HAS\_ACCOUNT\_IN]->(HDFC)

RETURN Rohan, HDFC

Create Relationship with Existing Node

MATCH(a:person), (b:bank) WHERE a.name="Manan Sharma" AND b.name="HDFC"

CREATE(a) - [r:HAS\_ACCOUNT\_IN] -> (b)

RETURN a,b

MATCH(a:person), (b:Employee) WHERE a.name="Manan Sharma" AND b.name="BMPL"

CREATE(a) - [r:WORKS\_IN] -> (b)

RETURN a,b

Delete All Node

MATCH(n) DETACH DELETE n

create(Rohan:person{name:"Rohan Sharma", age:30, occupation : "Employee"})

create(Mohan:person{name:"Mohan", age:31, occupation : "Employee"})

create(Aman:person{name:"Aman Tyagi", age:25, occupation : "Employee"})

create(Pooja:person{name:"Pooja Sharma", age:32, occupation : "Employee"})

create(Ekta:person{name:"Ekta Gupta", age:33, occupation : "Employee"})

create(HDFC:bank{name:"HDFC", branch:"Rohini"})

create(ICICI:bank{name:"ICICI", branch:"NSP"})

create(PNB:bank{name:"PNB", branch:"GZB"})

MATCH(a:person), (b:bank) WHERE a.name="Rohan Sharma" AND b.name="ICICI"

CREATE(a) - [r:HAS\_ACCOUNT\_IN] -> (b)

RETURN a,b

MATCH(a:person), (b:company) WHERE a.name="Mohan" AND b.name="Brain Mentors"

CREATE(a) - [r:WORKS\_IN] -> (b)

CREATE(friend:Person {name : "John"})

RETURN friend

CREATE(mac:Person {name : "Mac"})

CREATE(sam:Person {name : "Sam"})

MATCH(sam:Person {name : "Sam"})

MATCH(mac:Person {name : "Mac"})

CREATE(sam)-[rel:IS\_FRIENDS\_WITH]->(mac)

Update Query

MATCH (p:Person {name : "Sam"})

SET p.dob = date('1990-01-10')

RETURN p

Delete Node

MATCH(s:Person {name:"Sam"})

DELETE s

Delete Relationship

MATCH(s:Person {name:"Sam"}) - [r:IS\_FRIENDS\_WITH] -> (m:Person {name:"Mac"})

DELETE r

Delete Node Property

MATCH(s:Person {name:"Sam"})

REMOVE s.dob

Delete Node Property Alternative

MATCH(s:Person {name:"Sam"})

SET s.dob=null

Donot Create Another Mac if exists (Using Merge Instead of Create)

MERGE(mac:Person {name : "Mac"})

RETURN mac

Donot Duplicate Relationship

MATCH(sam:Person {name : "Sam"})

MATCH(mac:Person {name : "Mac"})

MERGE(sam)-[rel:IS\_FRIENDS\_WITH]->(mac)

Filter Data

MATCH(mac:Person {name : "Mac"})

WHERE mac.age = 30

RETURN mac