MongoDB

A study material for the students of GLS University

Prepared by : Ms. Harshita Maheshwari

Introduction

If we who wants to develop a good application then we should have the knowledge three major components.
They are (3-Tier Architecture).

UI (Presentation Tier)
Middle Layer (Application Tier)
Database (Database Tier)

So as a programmer:
a person should understand and be able to interact with a database is must, so than the data which is collected from the UI is processed and stored permanently.
(eg. Any management systems, set top box, washing machine, mobile application etc.).

Why do we need databases (Use Case)?

We need databases because they organize data in a manner which allows us to store, query, sort, and manipulate data in various ways. Databases allow us to do all this things.

Many companies collects data from different resource (like Weather data, Geographical data, Finance data, Scientific data, Transport data, Cultural data, etc.)

NoSQL

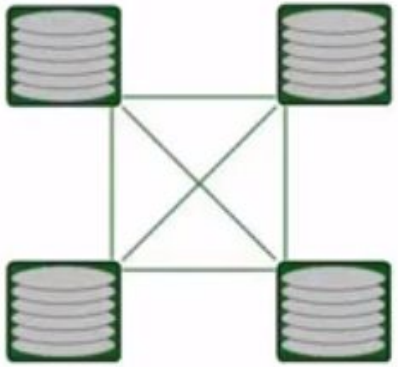
NoSQL database are primarily called as non-SQL or non-relational database. It provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.

Why NoSQL

Next Generation Databases

Not Only SQL

Non – Relational



Distributed Architecture

Open Source

Horizontally Scalable

Vertical Scaling (Scaling Up)

Vertical scaling involves increasing the resources of a single machine or server to handle more load.

Add more CPU, RAM, storage, or upgrade to a more powerful machine.

The application continues to run on a single machine with enhanced capabilities.

Horizontal Scaling (Scaling Out)

Horizontal scaling involves adding more machines or servers to distribute the load.

Add more servers to a system, often as part of a distributed architecture.

Data is divided across machines (e.g., sharding) or replicated for redundancy.

Key Differences

Aspect	Vertical Scaling	Horizontal Scaling
Method	Add resources to a single server	Add more servers to the system
Scalability Limit	Limited by hardware	Practically unlimited
Cost	Expensive hardware upgrades	Commodity hardware, cost-effective
Complexity	Simple to implement	Complex distributed architecture
Downtime	Possible during upgrades	Minimal, depending on architecture
Redundancy	No inherent redundancy	Built-in redundancy

RDBMS vs. NoSQL

SQL	NoSQL
Relational Database Management System (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have dynamic schema
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
E-commerce Platform, HR Management etc	CMS, Social Media Platforms, Gaming etc
Examples: MySQL, PostgreSQL, Oracle.	Examples: MongoDB, Cassandra, Redis.

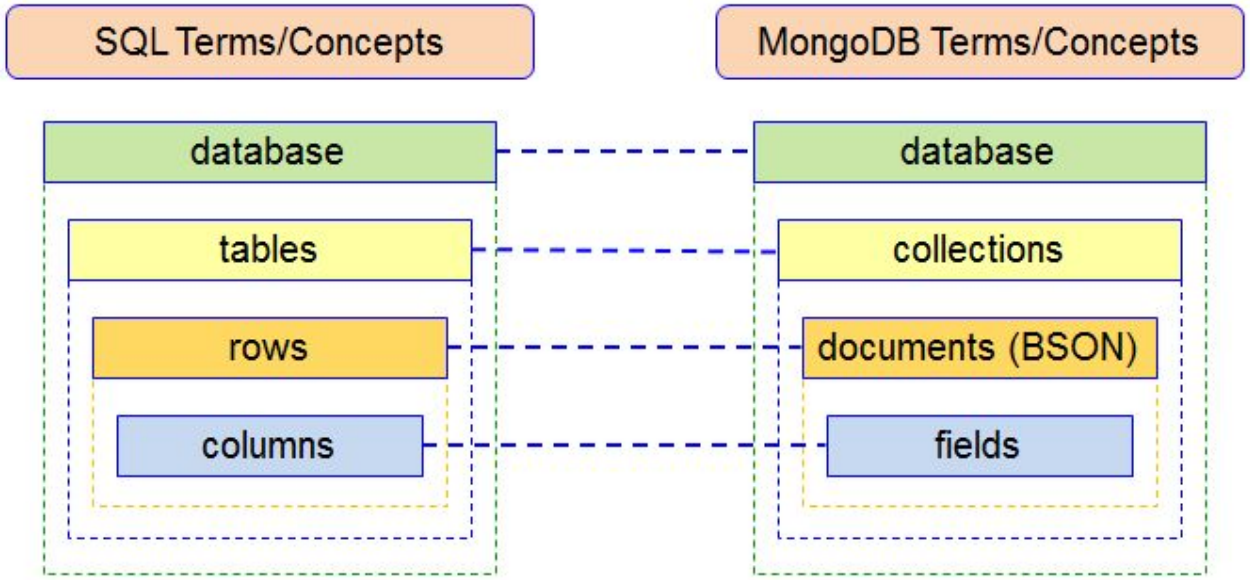
NoSQL Types

Key-value stores	Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. e.g. • **Redis, Cassandra**
Column-oriented	Wide-column stores organize data tables as columns instead of as rows. e.g. • **hBase**
Document oriented	Document databases, also called document stores, store semi-structured data and descriptions of that data in document format. e.g. • **MongoDB, CouchDB**
Graph	Graph data stores organize data as nodes. e.g. • **Neo4j**

MongoDB

What is MongoDB

MongoDB is an open source, document-oriented database designed with both scalability and developer agility in mind.
Instead of storing your data in tables and rows as you would with a relational database, in MongoDB you store JSON-like documents with dynamic schemas(schema-free, schema less).



Why use MongoDB

SQL was invented in the 70's to store data.

MongoDB stores documents (or) objects.

Now-a-days, everyone works with objects(Python/Ruby/Java/etc.)

And we need Databases to persist our objects. Then why not store objects directly ?

Embedded documents and arrays reduce need for joins. No Joins and No-multi document transactions.

Features

- Flexible Schema Design
- Scalability and Performance
- Document- Oriented Storage
- Dynamic Queries
- Aggregation Framework
- Open Source and Community

What is MongoDB great for –

RDBMS replacement for Web Applications.

Semi-structured Content Management.

Real-time Analytics & High-Speed Logging.

Caching and High Scalability

What is MongoDB not great for –

Highly Transactional Applications.

Problems requiring SQL.

MongoDB is easy to use

Relational

Person:

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rom

no relation

Car:

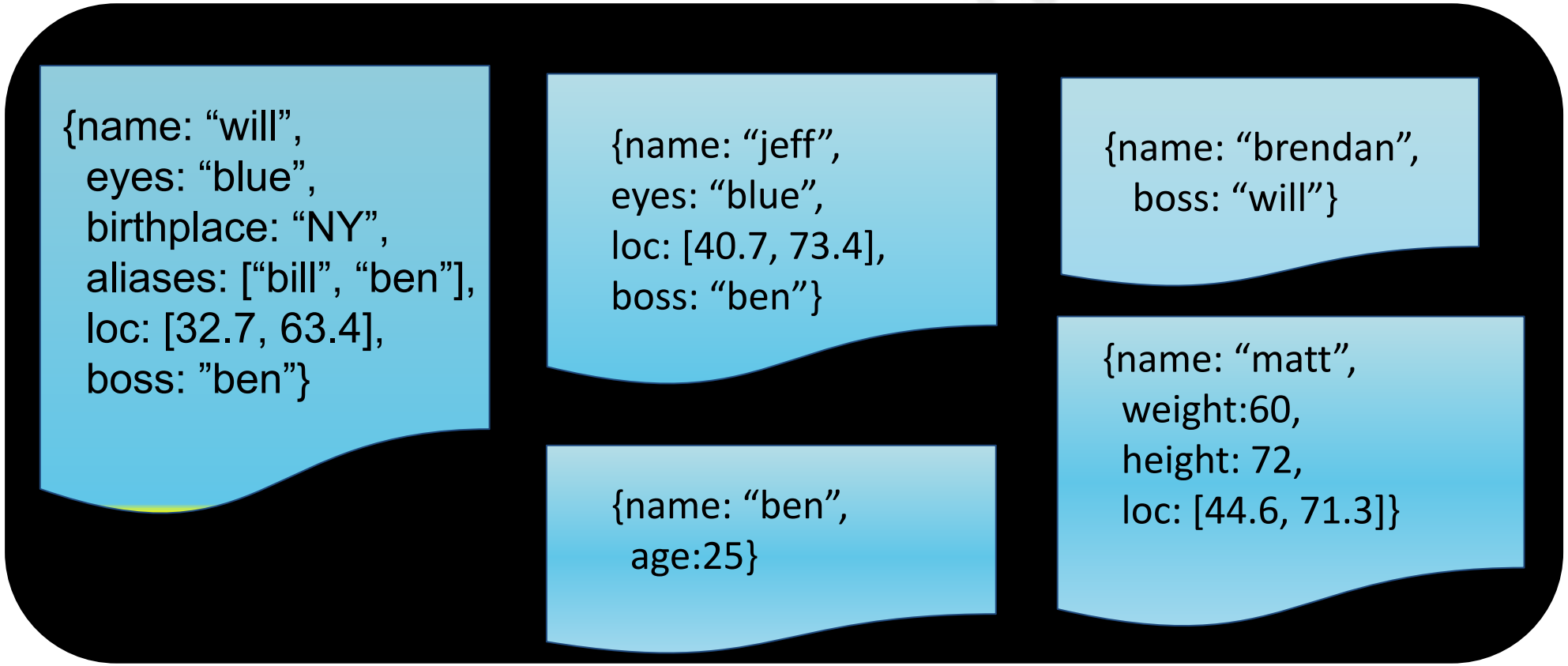
Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB Document

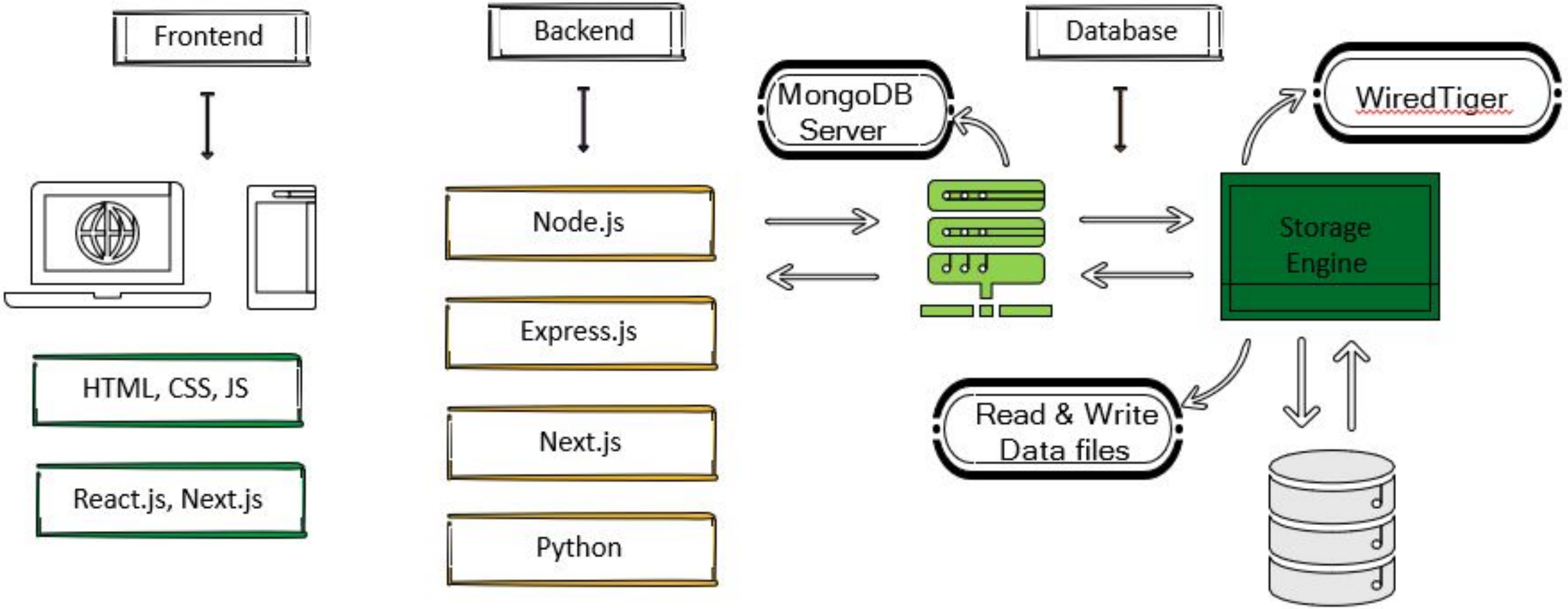
```
{
  first_name: 'Paul',
  surname: 'Miller'
  city: 'London',
  location: [45.123,47.232],
  cars: [
   { model: 'Bentley',
     year: 1973,
     value: 100000, … },
   { model: 'Rolls Royce',
     year: 1965,
     value: 330000, … }
  ]
}
```

Schema Free

MongoDB does not need any pre-defined data schema
Every document could have different data!



How MongoDB works??



JSON vs BSON

In MongoDB, we write in JSON format only but behind the scene data is stored in BSON (Binary JSON) format, a binary representation of JSON.

By utilizing BSON, MongoDB can achieve higher read and write speeds, reduced storage requirements, and improved data manipulation capabilities, making it well-suited for handling large and complex datasets while maintaining performance efficiency.

BSON

- Binary JSON Format: BSON, Binary JSON, is used in MongoDB for data storage and transmission.
- Efficient Storage: Designed for efficient data storage and transmission in MongoDB.
- Diverse Data Types: Supports a wider range of data types, including Binary, Date, and Regular Expression.
- Compact & Fast: BSON's binary format is more compact, leading to smaller storage and faster processing.
- Native to MongoDB: MongoDB stores data in BSON format, ensuring seamless integration.
- Performance Boost: Faster serialization improves data access and manipulation speed.

Install mongoDB

https://www.mongodb.com/try/download/community

https://www.mongodb.com/try/download/shell

https://www.mongodb.com/try/download/database-tools

The _id Field

By default, each document contains an _id field. This field has a number of special characteristics:

- Value serves as primary key for collection.
- Value is unique, immutable, and may be any non-array type.
- Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.” Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

MongoDB uses ObjectIds as the default value of _id field of each document, which is auto generated while the creation of any document.
Size -12 bytes

ObjectId()

CRUD

Create, Read, Update, Delete

Database

Database is a collection of data.

Database can also be described as a physical container for collections.

A database can have any number of collections

Each database gets it’s own set of files on the file system.

A MongoDB server can hosts multiple databases inside it.

Command	Description
show dbs or show databases	Returns list of all database name
db or db.getName()	Returns the current database name
use <databasename>	Switch to a database. / create a new database
db.dropDatabase()	Removes the current database, deleting the associated data files.

In the mongo shell, db is the variable that references the current database.

Collection

Group of MongoDB Documents.
Can relate a collection as a “table”
Unlike tables , collection does not have any schema definition.

Commands	Description
db.getCollectionNames() or show collections	Returns the collections in the current database.
db.createCollection(name, options)	used to create collection
db.COLLECTION_NAME.drop()	used to drop a collection from the database
db.collection.renameCollection(target, dropTarget)	Rename the collection dropTarget : If true, mongod drops the target of renameCollection prior to renaming the collection. The default value is false.

Document

MongoDB documents are composed of field-and-value pairs.

Every document has a unique value via Key “ _id”.

Documents have flexible and dynamic schema.

Document storage in BSON (Binary form of JSON)

Document

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

string	true
number	false
object	null
array	

```
{
  name: "sue",                          ← field: value
  age: 26,                              ← field: value
  status: "A",                          ← field: value
  groups: [ "news", "sports" ]          ← field: value
}
```

createCollection() Method

db.createCollection(name, options)

Data Types

- String
- Integer
- Boolean
- Double
- Min/Max keys
- Arrays
- Timestamp
- Object
- Null
- Symbol
- Date
- Code
- ObjectId
- Binary Data
- Regular Expression

Insert operation in mongoDB

db.collection.insertOne() : -

insertOne() inserts a document into a collection.

Syntax:- db.collection.insertOne({<document>})

Example:- db.emp.insertOne({ ename: 'x', job: 'pqr', salary: 2000 })

db.collection.insertMany() :-

Inserts multiple documents into a collection.

Syntax:- db.collection.insertMany([{<document 1>} , {<document 2>}, ...])

Example:- db.emp.insertMany([{ ename: 'x', salary: 2000}, { ename : 'y', job: 'hr' }])

Order and unordered inserts

Ordered inserts -

When executing bulk write operations, "ordered" and "unordered" determine the batch behavior.

Ordered Inserts

Default behavior is ordered, where MongoDB stops on the first error.

db.<collection-name>.insertMany([doc1, doc2, ...]);

Unordered Inserts

When executing bulk write operations with unordered flag, MongoDB continues processing after encountering an error.

db.<collection-name>.insertMany([doc1, doc2, ...], { ordered: false });

Read Operation in mongoDB

Read Data from Collection

Display all documents in non-structured way:-

```
db['collection'].find()
db.collection.find()
db.getCollection('name').find()
```

Find first document in collection:- `db.collection.findOne()`

Importing JSON in MongoDB

```
mongoimport jsonfile.json -d database_name -c collection_name

mongoimport products.json -d shop -c products

mongoimport products.json -d shop -c products --jsonArray
```

Here, --jsonArray accepts the import of data expressed with multiple MongoDB documents within a single JSON array.

Limited to imports of 16 MB or smaller.

comparison operator

$eq	**{<key>:{$eq:<value>}}**	Matches values that are equal to a specified value.
$gt	**{<key>:{$gt:<value>}}**	Matches values that are greater than a specified value.
$gte	**{<key>:{$gte:<value>}}**	Matches values that are greater than or equal to a specified value.
$lt	**{<key>:{$lt:<value>}}**	Matches values that are less than a specified value.
$lte	**{<key>:{$lte:<value>}}**	Matches values that are less than or equal to a specified value.
$ne	**{<key>:{$ne:<value>}}**	Matches all values that are not equal to a specified value.
$in	**{<key>:{$in:[<value1>, <value2>,……<valueN>]}}**	Matches any of the values specified in an array.
$nin	**{<key>:{$nin:[<value1>, <value2>,……<valueN>]}}**	Matches none of the values specified in an array.

Logical Operators

$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do not match the query expression.

$or

```
{ $or: [ { <expr1> }, { <expr2> }, ... , { <exprN> } ] }
```

```
db.emp.find({$or: [{job: 'manager'}, {job: 'salesman'}]})
```

$and

```
{ $and: [ { <expr1> }, { <expr2> }, ... , { <exprN> } ] }
```

```
db.emp.find({$and: [{job:'manager'}, {sal:3400}]})
```

$not

```
{ field: { $not: { <operator-expression> } } }
```

```
db.emp.find({ job: {$not: {$eq: 'MANAGER'}}})
```

Projection

Set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.
You cannot include and exclude fields simultaneously in the same query projection.

db.collection.find({},{KEY:1})

Examples -
db.emp.find({}, {ename:1, job: true});
db.emp.find({job:'manager'}, {ename:true, job:true})
db.emp.find({job:'manager'}, {_id:false, ename:true, job:true})

Introduction to Cursors

Cursors in MongoDB are used to efficiently retrieve large result sets from queries, providing control over the data retrieval process.

MongoDB retrieves query results in batches, not all at once.
Default batch size is usually 101 documents.
This improves memory efficiency and network usage

Cursor Methods :-

- count()
- limit()
- skip()
- sort()

db.collection.find().sort({ })

Specifies the order in which the query returns matching documents.

1 : Ascending

-1 : Descending

Syntax:-

db.collection.find(query, projection).sort({ field: value })

Example –

```
db.emp.find({}, {ename:true}).sort({ename: 1});

db.emp.find({}, {ename:true}).sort({ename: -1});
```

db.collection.find().limit()

The method accepts one number type argument, which is the number of documents that you want to be displayed.
A limit() value of 0 (i.e. .limit(0)) is equivalent to setting no limit.

Syntax :-
db.collection.find(query, projection).limit(<number>)

Example :-

```
db.emp.find({},{ename:true}).limit(0);    // all documents
db.emp.find({},{ename:true}).limit(2);
```

db.collection.find().skip()

Accepts number type argument and is used to skip the number of documents.
the default value in skip() method is 0.

Syntax:-
db.collection.find(query, projection).skip(<offset_number>)

Example :-
db.emp.find().skip(4);
db.emp.find({},{”ename":1,_id:0}).limit(1).skip(1)

db.collection.find().count()

Counts the number of documents referenced by a cursor. Append the count() method to a find() query to return the number of matching documents. The operation does not perform the query but instead counts the results that would be returned by the query.

Syntax:- db.collection.find(<query>).count()

Example :-

```
db.emp.find().count();
db.emp.find({job: 'manager'}).count();
```

db.collection.distinct()

Finds the distinct values for a specified field across a single collection or view and returns the results in an array.

Syntax: - db.collection.distinct(field, query, options)

Example :- db.emp.distinct("job")

Elements Operator

$exists :
Matches documents that have the specified field.
{ field: { $exists: <boolean>} }

$type :
Selects documents if a field is of the specified type.
{ field: { $type: "<bson-data-type>" } }

$size

MongoDB $size operator is used to find the total number of elements in an array.

It is used in aggregation pipeline stages and is important to find the size of the array within documents.

This operator is valuable for various data analysis tasks, allowing users to efficiently count the elements within arrays and perform operations based on the array size.

```
{ field: { $size: <array-length> } }
```

Embedded Documents

Embedded document or nested documents are those types of documents which contain a document inside another document.

In MongoDB, you can only nest document up to 100 levels.
The overall document size must not exceed 16 MB.

Query documents inside embedded documents using dot notation.
db.collection.find({ “parent.child”: value })

$all vs $elemMatch

The $all operator selects the documents where the value of a field is an array that contains all the specified elements.

{ <field>: { $all: [<value1> , <value2> ...] } }

The $elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria.

{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }

Update Operation in mongoDB

db.collection.updateOne()

updates a single document within the collection based on the filter.

Syntax: - db.collection.updateOne({filter}, {$set:{update} }, {options})

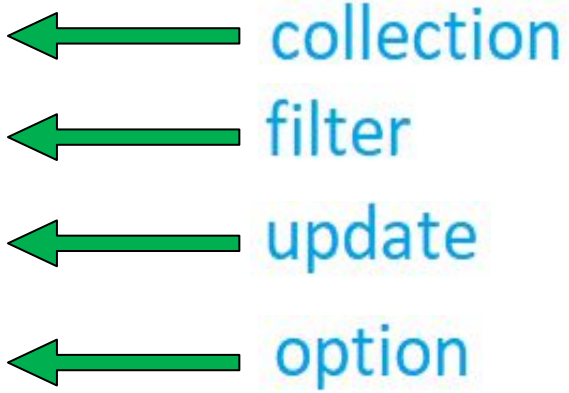
```
db.emp.updateOne (
    { ename : 'saleel' },
    { $set : { job:'Manager' } },
    { upsert : true }
)
```

db.collection.updateMany()

updates multiple documents within the collection based on the filter.

Syntax: - db.collection.updateMany({filter}, {$set:{update} }, {options})

```
db.emp.updateMany(
   { sal : { $gt : 2000 } },
   { $set: { color : ['red',  'blue'] } },
   { upsert : true }
)
```



Removing and Renaming Fields

```
db.collectionName.updateOne( { filter }, { $unset: { fieldName: 1 } }
);

db.collectionName.updateOne(
{ filter },
{ $rename: { oldFieldName: "newFieldName" } }
);
```

Updating arrays and Embedded Documents

```
db.collectionName.updateOne( { filter },
 { $push: { arrayField: "new element" } }
);

db.collectionName.updateOne({ filter },
 { $pop: { arrayField: value } }
);

db.collectionName.updateOne({ filter },
 { $set: { "arrayField.$.text": "Updated text" } }
);
```

db.collection.replaceOne()

replaces a single document within the collection based on the filter.

Syntax:- db.collection.replaceOne(filter, replacement, options)

Example:-

```
db.emp.replaceOne({ename: 'saleel'}, {x: 500, y: 500 })
```

Delete Operation in mongoDB

Prepared by : Ms. Harshita Maheshwari

db.collection.deleteOne()

deleteOne() removes a single document from a collection. Specify an empty document { } to delete the first document returned in the collection.

Syntax:-
db.collection.deleteOne({ <filter> })

Example:-
db.emp.deleteOne({})
db.emp.deleteOne({job: 'manager'})

db.collection.deleteMany()

deleteMany() removes all documents that match the filter from a collection.

Syntax:- db.collection.deleteMany({<filter>})

Examples:-
db.emp.deleteMany({});
db.emp.deleteMany({job: 'manager'})

Indexes

Indexes

Indexes are specialized data structures that optimize data retrieval speed in MongoDB.

Indexes store a fraction of data in a more searchable format.

They enable MongoDB to locate data faster during queries. Indexes are separate from collections and multiple indexes can exist per collection.

Benefits of Indexes

Faster Querying: Indexes drastically accelerate data retrieval, particularly for large collections.

Efficient Sorting: Indexes facilitate rapid sorting based on specific fields.

Improved Aggregation: Aggregation operations become more efficient with optimized indexes.

Indexing on Multiple Fields: Complex queries can be executed efficiently by utilizing multiple fields in indexes.

Managing Indexes

db.products.createIndex({ field: 1 });
(1) for storing indexes in ascending order.
(-1) for storing indexes in descending order.

db.collection.getIndexes();
_id is a default index.

db.collection.dropIndex({ field: 1 });

db.collection.dropIndex("index_name");

db.collection.createIndex({ field: 1 }, { unique: true });

explain()

Use explain() method to understand query execution in detail.

db.products.find({ field: value }).explain();

db.products.find({ field: value }).explain("executionStats");

Use it to measure the time taken to execute a query.

Aggregation

Aggregation

Aggregation is the process of performing transformations on documents and combining them to produce computed results.

Pipeline Stages: Aggregations consist of multiple pipeline stages, each performing a specific operation on the input data.

Benefits:

- Aggregating Data: Complex calculations and operations are possible.
- Advanced Transformations: Data can be combined, reshaped, and computed for insights.
- Efficient Processing: Aggregation handles large datasets efficiently.

aggregation

Stages –

$match WHERE clause	$project SELECT clause	$unwind PIVOT an array	$group GROUP BY clause	$sort ORDER BY clause	$limit TOP clause

Syntax:-

db.collection.aggregate([{ <stage1> }, { <stage2> }, ... , { <stageN> }])

Example:- db.emp.aggregate([])

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

Syntax:- { $match: { <query> } }

Examples:-

db.emp.aggregate ([{$match: {job: 'manager'} }])

db.emp.aggregate ([{$match: {comm: {$eq: null} } }])

db.emp.aggregate ([{$match: {sal: {$gt: 4000} }}, {$group: {_id: '$job', count: {$sum: '$sal'} } }])

db.emp.aggregate([{$match: {favouriteFruit: {$size: 1} } }])

db.emp.aggregate([{$match: {'favouriteFruit.0': 'Orange'} }, {$project: {favouriteFruit: true} }])

$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an _id field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group's _id field.

Syntax:- { $group: { _id: '$<expression>', <field1>: { <accumulator1> : <expression1> }, ... } }

Example:-

```
db.emp.aggregate([ {$group: {_id: null, count: {$sum: 1} } } ])

db.emp.aggregate([ {$group: {_id: null, total: {$sum: "$sal"} } } ])

db.emp.aggregate([ {$group: {_id: "$job", count: {$sum: 1} } } ])
```

$group on multiple fields

Syntax:-

{ $group: { _id: { <field1>: '$<expression>', ... }, <field1>: { <accumulator1> : '$<expression1'> }, ... } }

Example:-

db.emp.aggregate([{ $group: {_id: { job: "$job", deptno: "$deptno“ }, count : { $sum: 1 } } }])

$project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

Syntax:- { $project: { <specification(s)> } }

Examples:- db.emp.aggregate ([{$project: { ename: true } }])
db.emp.aggregate ([{$project: {_id: false, sal: true, comm: true } }])

db.emp.aggregate ([{$project: {sal: true, sm: {$sum: '$sal'} } }])

db.emp.aggregate ([{$project: {_id: false, sal: true, comm: true, xx: {$max: ['$sal', '$comm'] } } }])

db.emp.aggregate([{$project :{_id: false, indexID: true, favouriteFruit: {$size: '$favouriteFruit'} } }])

$push

The $push stage adds elements to an array field within documents.

{ $push: <expression> }

```
db.products.aggregate([
{ $group: { _id: { company: "$company" }, products: { $push:
"$name" } } }
]);
```

$unwind

The $unwind stage deconstructs an array field and produces multiple documents.

{ $unwind: <array> }

```
db.products.aggregate([
{ $unwind: "$colors" },
{ $group: { _id: { company: "$company" }, products: { $push: "$colors" } } }
]);
```

Deconstructs the colors array field, groups products by company, and creates an array of colors for each company.

$avg

Returns the average value of the numeric values. $avg ignores non-numeric values.

```
db.sales.aggregate([
    {
      $group:
        {
          _id: "$item",
          avgAmount: { $avg: { $multiply: [ "$price", "$quantity" ] } },
          avgQuantity: { $avg: "$quantity" }
        }
    }
  ]
)
```

$sum

Calculates and returns the collective sum of numeric values. $sum ignores non-numeric values.

```
db.sales.aggregate([
    {
      $group:
        {
          _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
          totalAmount: { $sum: { $multiply: [ "$price", "$quantity" ] } },
          count: { $sum: 1 }
        }
    }
  ]
)
```

arithmetic expression operators

Arithmetic expressions	
$abs	x: { $abs: '$<number>' }
$add	x: { $add: ['$<expression1>', '$<expression2>', ...] }
$subtract	x: { $subtract: ['$<expression1>', '$<expression2>'] }
$multiply	x: { $multiply: ['$<expression1>', '$<expression2>', ...] }
$divide	x: { $divide: ['$<expression1>', '$<expression2>'] }
$mod	x: { $mod: ['$<expression1>', '$<expression2>'] }
$trunc	x: { $trunc: '$<number>' }

Examples :-

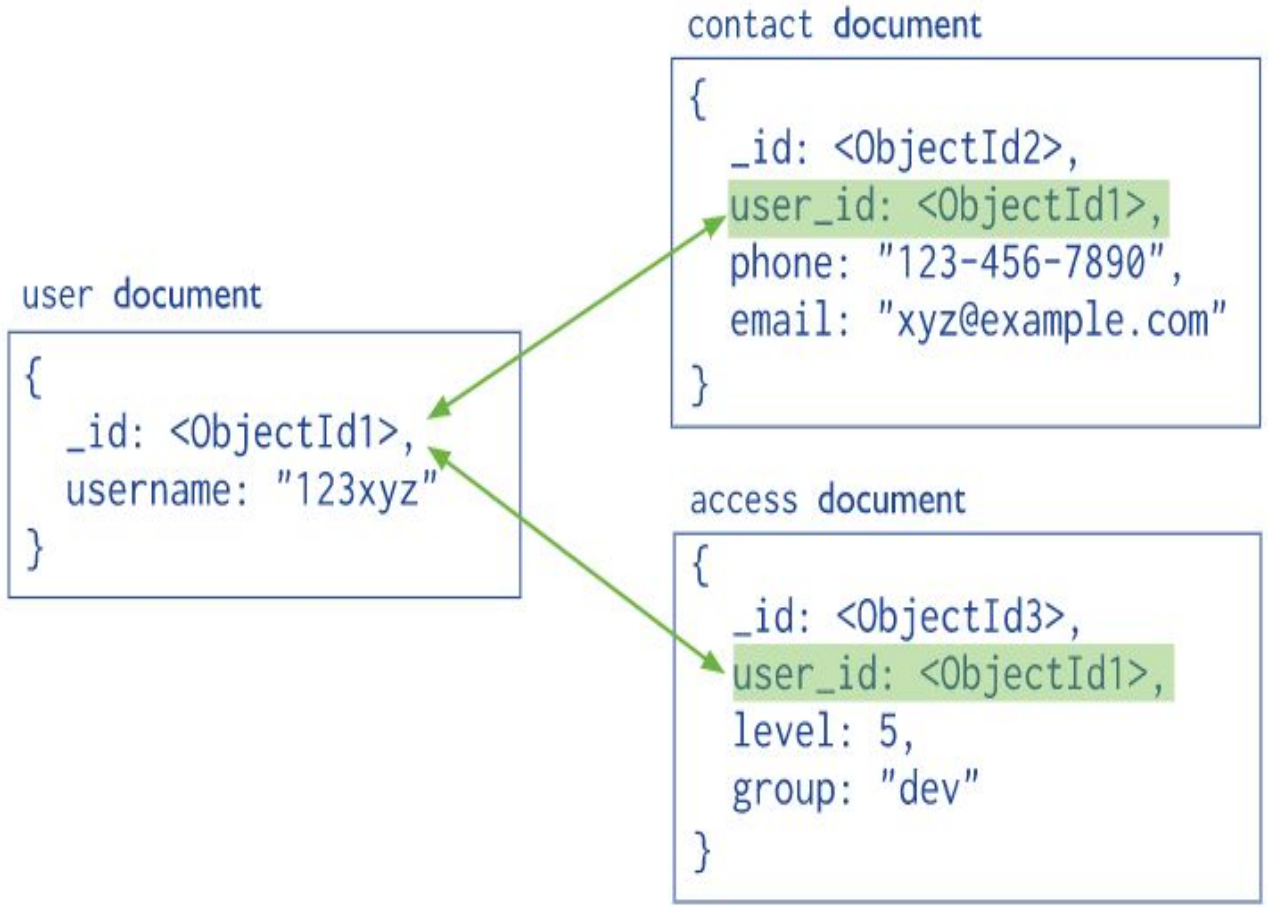
```
db.emp.aggregate ([ {$project : { op: { $trunc: "$sal" } } } ])

db.emp.aggregate([ {$project: { sal: true,  op : { $add: ['$sal', 1000] } } } ])
```

$lookup

References store the relationships between data by including links or references from one document to another.

In order to retrieve all the data in the referenced documents, a minimum of two queries or $lookup required to retrieve all the information.

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the
"from" collection>,
       as: <output array field>
     }
}
```



MongoDB Atlas

MongoDB Atlas

MongoDB Atlas is MongoDB's fully managed cloud database service.

It offers an easy way to deploy, manage, and scale MongoDB databases in the cloud.
Atlas eliminates the need for manual setup and maintenance, allowing developers to focus on their applications.

It provides automated scaling options to accommodate growing workloads.
Atlas supports global clusters, enabling databases to be deployed across multiple regions for better data availability and reduced latency.

Working with mongoose

mongoose

It's an Object Data Modeling (ODM) library for MongoDB and Node.js.

It makes MongoDB interaction more straightforward and organized.

It provides a structured, schema-based data modeling approach.

Why Mongoose instead of official driver?

1. Structured Schemas
2. Validation
3. Relationships
4. Middleware
5. Complex Queries

Thank you !!