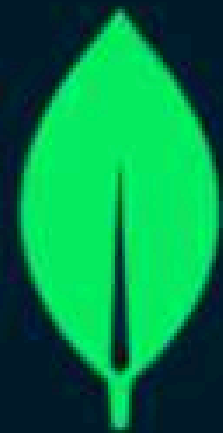
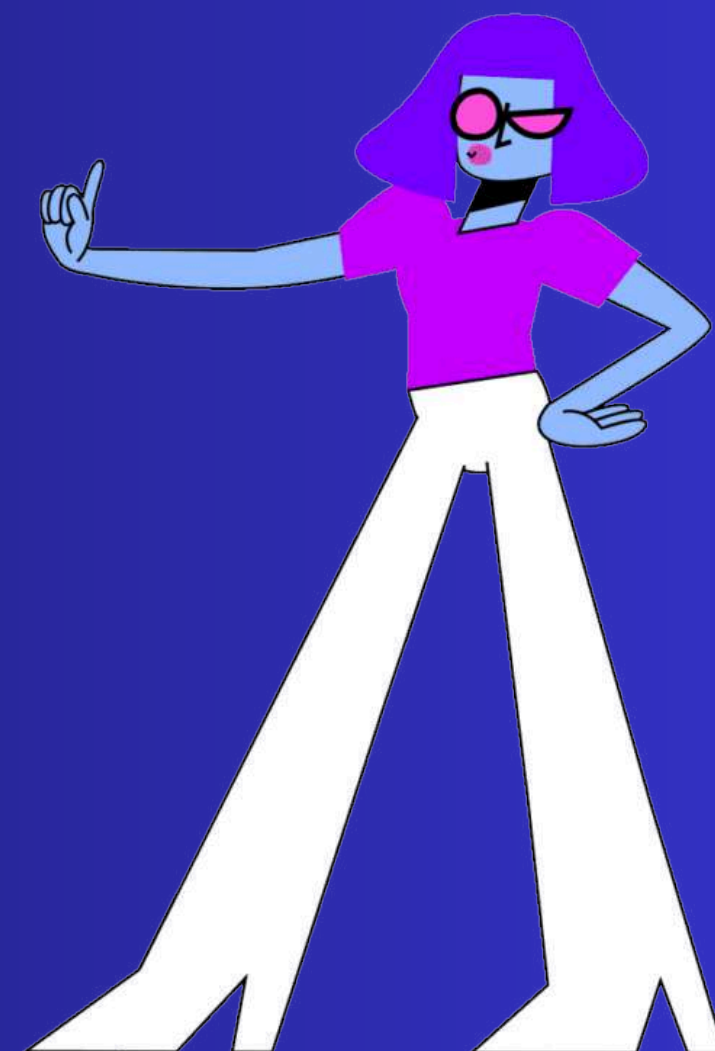


MPRASHANT



MongoDB®



Topics

- What is mongodb?
- Installation (MAC & Windows)
- Mongodb ATLAS
- CRUD Operations
- Operators (conditions, logical, relations)
- Aggregate Framework
- Data Modeling (Relationship)
- Schema Validations
- Indexes
- Python + Mongodb
- Interview questions

What is a Database?



What is a Database?

An organised collection of data.

A method to manipulate and access the data.

Unorganized

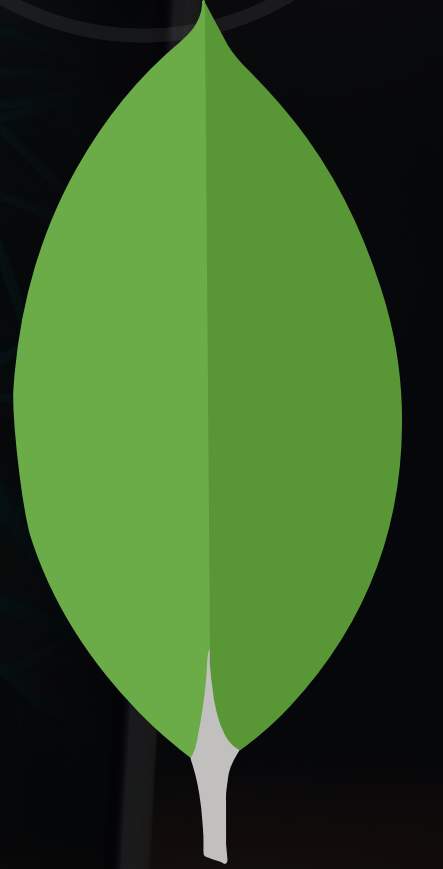


Organized



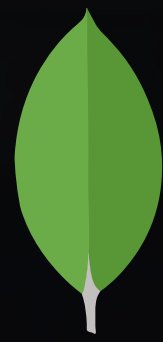
MPRASHANT

What is MongoDB?





MongoDB is a **NoSQL database** that stores data in flexible, JSON-like documents, making it easy to scale and handle large volumes of unstructured data.



MongoDB Structure

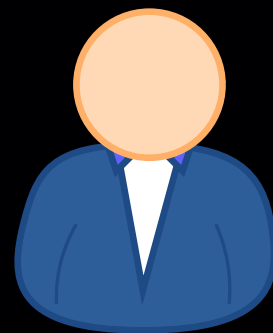
MongoDB stores data in **collections** and **documents**.

A collection is a group of MongoDB documents

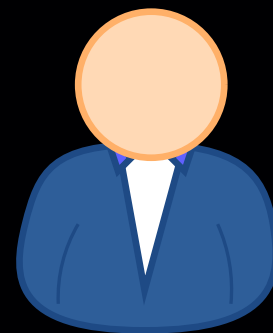
A document is a set of **key-value** pairs.



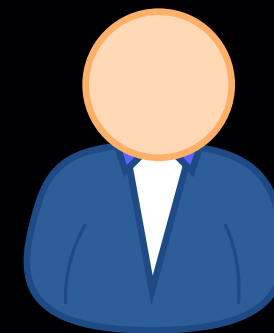
Users



Raju



Sham



Baburao



collection: users



name: Raju
age: 25



name: Sham
age: 28



name: Baburao
age: 45

fields

documents



```
{  
  "_id": 1,  
  "name": "Raju",  
  "email": "raju@example.com",  
  "age": 35,  
  "address": {  
    "street": "123 Bollywood Blvd",  
    "city": "Mumbai",  
    "zip": "400001"  
  },  
  "hobbies": ["acting", "reading"]  
}
```




MongoDB:

- **Collection:** A container for documents (like a table).
- **Document:** A JSON-like record within a collection (like a row).
- **Field:** A key-value pair within a document (like a column).

Brief history and background.

MondoDB was first released in 2009. It was developed by a company called 10gen, which later rebranded as MongoDB, Inc.



MongoDb Features

- **Scalable:** Easy to distribute data across multiple machines as data grows.
- **Flexible:** Each document can have different number and types of fields.
- **Performance:** NoSQL databases are optimized for high-speed read and write operations, making them ideal for real-time data processing and analytics.



SQL vs NoSQL



Scalability

- **NoSQL**: Designed to scale horizontally by adding more servers, distributing the load across multiple nodes.
- **SQL**: Typically scales vertically by adding more resources to a single server, which has limits and can be costly. Horizontal scaling (sharding) is more complex and less efficient.



Schema Flexibility

- **NoSQL**: Schema-less design allows for easy storage of diverse and rapidly changing data without needing to alter the schema.
- **SQL**: Requires a predefined schema. Altering the schema (e.g., adding new columns) can be complex and time-consuming, especially with large datasets.



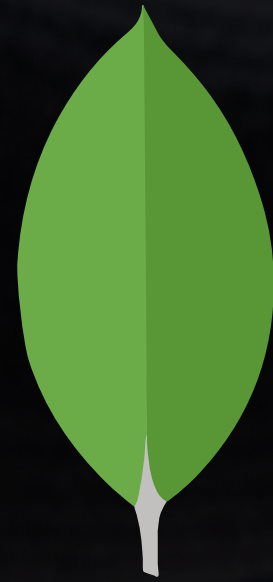
Performance

- **NoSQL**: Optimized for high-speed read and write operations, making it ideal for real-time data processing and analytics.
- **SQL**: Though performant, it can suffer from slower read/write operations when dealing with very large volumes of data due to ACID transaction overhead and complex joins.

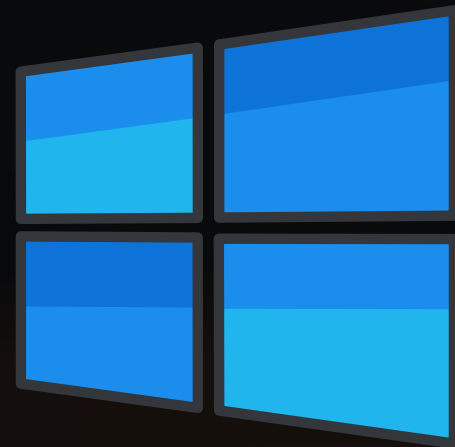


Handling Unstructured Data

- **NoSQL**: Efficiently handles unstructured or semi-structured data, which is common in real-time analytics (e.g., varied user activity types).
- **SQL**: Best suited for structured data. Handling unstructured data requires complex transformations or additional systems (e.g., storing JSON data in columns).



Installation



Default port of mongod is 27017

**To listen on different port, we can use
mongod --port 27019**

- **Database**
 - **Creating, connect, listing, dropping**
- **CRUD**
 - **Create - New collection**
 - **Inserting data**
 - **Read - How to read data**
 - **Update data**
 - **Delete data**

MPRASHANT



Create Database and Collection

- **use database_name**
- **db.createCollection("myCollection")**
- **show dbs**
- **show collections**



DROP

Database and Collection

- **collection_name.drop()**
- **db.dropDatabase()**



CRUD Operations

CRUD

- **CREATE**
- **READ**
- **UPDATE**
- **DELETE**





CREATE

`insertOne()`

`insertMany()`

READ

`find()`

`findOne()`

UPDATE

`updateOne()`

`updateMany()`

DELETE

`deleteOne()`

`deleteMany()`



- **maker**
- **model**
- **fuel type**
- **transmission**
- **engine**
 - **cc**
 - **torque**
- **features**
 - **sunroof**
 - **airbags**





UPDATE

updateOne()
updateMany()



```
db.cars.updateOne(  
  { model: "Nexon" },  
  { $set: { color: "Red" } }  
);
```

```
db.cars.updateOne(  
  { model: "Nexon" }, // Filter to find the specific document  
  { $push: { features: "Heated Seats" } } // Add the new feature  
);
```




```
db.cars.updateMany(  
  { model: "Nexon" }, // Filter to find all documents where  
  { $set: { fuel_type: "Electric" } } // Update the fuel_t  
);
```

```
db.cars.updateOne(  
  { model: "Nexon" }, // Filter to find the specific document  
  {  
    $set: {  
      "engine.cc": 1300, // Update the engine's cc to 1300  
      "engine.torque": "180 Nm" // Update the engine's torque to 180 Nm  
    }  
  }  
);
```




Updating Array

```
db.cars.updateOne(  
  { model: "Nexon" }, // Filter to find the specific document  
  { $push: { features: "Heated Steering Wheel" } }  
);
```

```
db.cars.updateOne(  
  { model: "Nexon" }, // Filter to find the specific document  
  { $pull: { features: "Bluetooth Connectivity" } } //  
);
```



Updating Multiple Values Array

```
db.cars.updateOne(  
  { model: "Nexon" }, // Filter to find the specific document  
  { $push: { features: { $each: ["Wireless Charging", "Voice Control"] } } }  
);
```

```
db.collection.update(  
  { _id: 1 },  
  { $unset: { "address": "" } }  
)
```

Unset Field

upsert is a combination of the operations
"update" and "insert."

If no document matches the query criteria, MongoDB
will insert a new document into the collection.

```
db.students.updateMany(  
  { name: "Jane Doe" },  
  { $set: { age: 22, course: "Mathematics" } },  
  { upsert: true }  
)
```




Update Operators

Use update operators to modify MongoDB documents. You can set field values, manipulate arrays, and more.





Data Types



- MongoDB stores data in **BSON (Binary JSON) format.**
- BSON includes all JSON datatypes and adds more.
- Choosing the correct datatype is essential for efficient storage and querying.



```
_id: ObjectId("507f1f77bcf86cd799439011"), // ObjectId
name: "John Doe", // String
age: 30, // Integer
price: 19.99, // Double
isActive: true, // Boolean
tags: ["mongodb", "database", "NoSQL"], // Array
address: { // Object/Embedded Document
  street: "123 Main St",
  city: "New York",
  state: "NY",
  zip: 10001
},
```



```
createdAt: ISODate("2023-08-21T14:23:00Z"), // Date  
middleName: null, // Null
```

```
ts: Timestamp(1638306013, 1), // Timestamp  
salary: Decimal128("12345.67"), // Decimal128
```



BSON Types

MongoDB uses BSON (Binary JSON) field types to store and serialize documents.

MPRASHANT



Operators



**Case: Find the cars with engine
more than 1400cc**

Relational Operators

Operators	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to



Comparison Operators

`$eq` `$lt` `$gt` `$lte` `$gte` `$ne`
= < > <= >= !=

Example: `{age:{$eq:25}}`



**Case: Find the cars with engine
having 1498 and 2179cc**

Here we are trying search based on multiple values



`$in $nin`

`{age:`

`{ $in:[20,30]}`

`}`



Case: I want a car which is Diesel, Sunroof & Turbocharged Engine

Car with all these features

Logical Operators

Diesel

AND

Turbocharged

I want a car with both Diesel and Turbocharged

Automatic

OR

Sunroof

**I want a car with either
Automatic or Sunroof
or both will work.**

Automatic

NOR

Sunroof

I don't want a car with either Automatic or Sunroof



Logical Operators



```
db.collection.find({  
  $and: [  
    { condition1 },  
    { condition2 },  
    // additional conditions as needed  
  ]  
})
```



\$and \$or \$nor \$not

```
{ $or: [  
  { age: { $eq: 28 } },  
  { name: "raju" }  
] }
```

Element Operators



- **\$exists**

- **{age:{\$exists:true}}**

- **\$type**

- Here we can filter the content based on BSON type like string, bool etc
- This can be useful to find field with null values
- **{name:{\$type:"string"}}**

Array Operators



- **\$size**

- Return all documents that match specified array size
- **db.collection.find({hobbies:{\$size:4}})**

- **\$all**

- Return all documents that match the pattern
- (all user with hobbies of play and read)
- **db.collection.find({hobbies:{\$all:["play","read"]}})**

UseCase:

**How to find no. of
records/documents?**

Cursor Methods

- **Count**
 - **find().count()**
- **Sort**
 - **find().sort({"name":1}) -1 is for descending order**
- **Limit**
 - **find().limit(2)**
- **Skip**
 - **find().skip(3)**

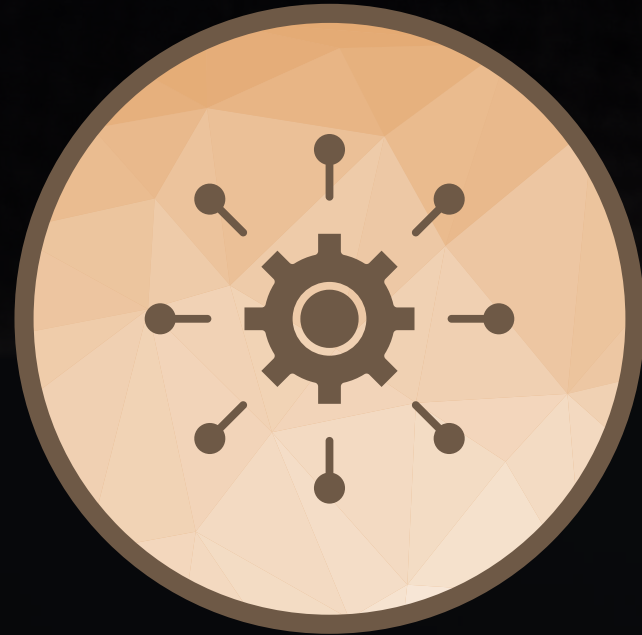


Aggregation Operators

Contains a list of aggregation operators to use in aggregation stages.



MPRASHANT



Aggregate Framework



**Aggregation is a powerful framework for
Complex operations**

**like filtering, grouping, sorting, reshaping, and
summarizing data in a flexible way via pipeline.**



stage 1



stage 2



stage 3



```
db.collection.aggregate(  
  [  
    {stage1},  
    {stage2}...  
  ], {option}  
)
```



```
db.orders.aggregate( [  
  
    // Stage 1: Filter pizza order documents by pizza size  
    {  
        $match: { size: "medium" }  
    },  
  
    // Stage 2: Group remaining documents by pizza name and calculate total quantity  
    {  
        $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }  
    }  
  
] )
```


Most commonly used stages in MongoDB aggregation:

- **\$match**
- **\$group**
- **\$project**
- **\$sort**
- **\$limit**
- **\$unwind**
- **\$lookup**
- **\$addFields**
- **\$count**
- **\$skip**

GROUPING

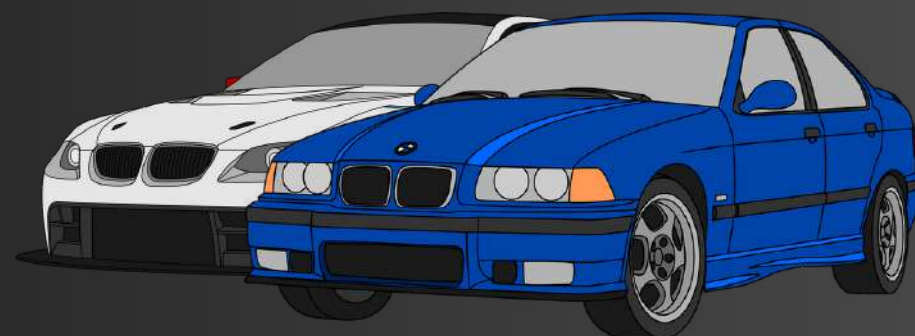
No. of cars of each Brand (maker)

```
[
  { _id: 'Maruti Suzuki', TotalCars: 3 },
  { _id: 'Honda', TotalCars: 3 },
  { _id: 'Hyundai', TotalCars: 4 },
  { _id: 'Tata', TotalCars: 3 },
  { _id: 'Mahindra', TotalCars: 1 }
]
```





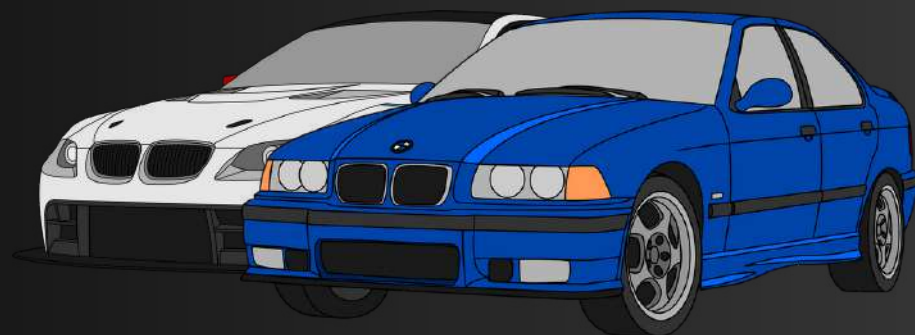

Hyundai



Tata



Toyota



Suzuki



Honda

GROUP

Syntax of 'group'

```
db.collection.aggregate([
{
  $group: {
    _id: "$category",
    totalAmount: { $sum: "$amount" },
    averageAmount: { $avg: "$amount" },
    minAmount: { $min: "$amount" },
    maxAmount: { $max: "$amount" },
    amountsList: { $push: "$amount" },
    uniqueAmounts: { $addToSet: "$amount" }
  }
}
])
```



Print all the car brands..

```
db.cars.aggregate([  
  {$group:  
    {_id:"$maker"}  
  }])
```



Find no. of cars of each brand (we have to group)

```
db.cars.aggregate([  
  { $group:  
    { _id: "$maker",  
      TotalCars: { $sum: 1 }  
    }  
  }  
])
```

The \$sum: 1 operation counts the number of documents in each group.



Find no. of cars of different fuel_type

```
db.cars.aggregate([  
  { $group:  
    { _id: "$fuel_type",  
      TotalCars: { $sum: 1 }  
    }  
  } ] )
```

MATCH



Hyundai cars having engine of more than 1200cc

```
db.cars.aggregate(  
  [{$match:  
    {maker:"Hyundai",  
      "engine.cc":{$gt:1000}  
    }  
  ]  
})
```


COUNT



Print Total no. of Hyundai cars

```
db.cars.aggregate(  
  [{$match:  
    {maker: "Hyundai"}},  
   {$count: "Total_cars"}  
])
```



Count no. of diesel and petrol cars of Hyundai brand

```
db.cars.aggregate(  
  [ {$match:{maker:"Hyundai"}},  
    {$group:  
      {_id:"$fuel_type",  
        Totalcars:{$sum:1}}  
    }  
  ]  
)
```


Project



Find all the Hyundai cars and only show Maker, Model and Fuel_type details

```
db.cars.aggregate(  
  [{ $match: { maker: "Hyundai" }},  
    { $project:  
      { maker: 1, model: 1, fuel_type: 1, _id: 0 } }  
  ]  
)
```

sort



For the previous output, sort the data based on Model

```
db.cars.aggregate(  
  [{ $match: { maker: "Hyundai" }},  
   { $project:  
     { maker: 1, model: 1, fuel_type: 1, _id: 0 }},  
   { $sort: {model:1}}  
  ])
```

Similarly we can use { \$limit: 10 }, {\$skip:5}

sortByCount



Group the cars by Maker and then sort based on count(no. of cars)

```
db.cars.aggregate(  
    {$sortByCount:"$maker"}  
)
```


Unwind



We do have multiple owners for each car right (owners are list of documents), now if you want to work on each owner then we can use unwind

```
db.cars.aggregate([  
  { $unwind: "$owners" }  
])
```

String Operators

**\$concat \$toUpper \$toLower \$regexMatch
\$ltrim \$split**


```
db.collection.aggregate([
  {
    $project: {
      fullName: { $concat: ["$firstName", " ", "$lastName"] }
      // Concatenates firstName and lastName with a space
    }
  }
])
```



List down all the Hyundai cars and print the name as
Maker + Model i.e. **CarName Hyundai Creta**

```
db.cars.aggregate(  
  [ { $match: { maker: "Hyundai" } },  
    { $project: { _id: 0,  
      CarName: { $concat: ["$maker", " ",  
        "$model"] } } } ] )
```

\$regexMatch

```
{  
  $regexMatch: {  
    input: <string_expression>,  
    regex: <regex_pattern>,  
    options: "<options>" // Optional, e.g., "i" for case-insensitive  
  }  
}
```

Performs a regular expression (regex) pattern matching and returns true or false.



Add a flag `is_diesel = true/false` for each car

```
db.cars.aggregate([
  { $project: { model: 1, _id: 0,
    is_diesel:
      { $regexMatch:
        { input: "$fuel_type",
          regex: "Die" }
      } } }])
```

OUT



After aggregating, store the result in an another collection 'hyundai_cars'

```
db.cars.aggregate(  
  [ { $match: { maker: "Hyundai" } },  
    { $project: { _id: 0,  
                  CarName: { $concat: ["$maker", " ",  
                                         "$model"] } } },  
    { $out: "hyundai_cars" } ] )
```


Airthmetic Operators

**\$add \$subtract \$divide \$multiply \$round
\$abs \$ceil**

```
db.collection.aggregate([
  {
    $project: {
      sum: { $add: [2, 3] } // 2 + 3
    }
  }
])
```



Print all the cars model and price with hike of 55000
(similarly we can use \$subtract too)

```
db.cars.aggregate(  
  {$project:  
    {model:1,_id:0,  
     price:{$add:["$price",50000]}}} )
```


AddFields / Set



Print details of cars with price in lakhs (15 lakhs)

```
db.cars.aggregate([  
  {$project:{model:1,_id:0,price:1}},  
  {$addFields:  
    {price_in_lakhs:  
      {$divide: ["$price", 100000]}  
    }}])
```



Calculate Total service cost of each Hyundai Car.

```
db.cars.aggregate([  
  { $match: { maker: "Hyundai" } },  
  { $set: { total_service_cost: { $sum: "$service_history.cost" } } },  
  {$project: {model:1, maker:1, _id:0, total_service_cost:1}}])
```


Conditional Operators

\$cond \$ifNull \$switch

The \$cond operator in MongoDB is a ternary conditional operator

{ \$cond: [<condition>, <true-case>, <false-case>] }

Suppose we want to check if a car's fuel_type is "Petrol" and categorize the cars into 'Petrol Car' & 'Non-Petrol'

```
db.cars.aggregate([
  { $project: {
    _id: 0, maker: 1, model: 1,
    fuelCategory: {
      $cond: {
        if: { $eq: ["$fuel_type", "Petrol"] },
        then: "Petrol Car",
        else: "Non-Petrol Car"
      }
    }
  }
}])
```


\$switch

```
{
  $switch: {
    branches: [
      { case: <condition1>, then: <result1> },
      { case: <condition2>, then: <result2> },
      ...
    ],
    default: <defaultResult>
  }
}
```

Suppose we want to categorize the price of the car into three categories: "Budget", "Midrange", and "Premium"

```
db.cars.aggregate([
  {$project: {
    _id: 0, maker: 1, model: 1,
    priceCategory: {
      $switch: {
        branches: [
          { case: { $lt: ["$price", 500000] },
            then: "Budget" },
          { case: { $and: [{ $gte: ["$price", 500000] }, { $lt: ["$price", 1000000] } ] },
            then: "Midrange" },
          { case: { $gte: ["$price", 1000000] },
            then: "Premium" }
        ],
        default: "Unknown"
      }
    }
  }
]);
```

DATE Operators

**\$dateAdd \$dateDiff \$month \$year \$hour
\$dateOfMonth \$dayOfYear**


```
db.collection.aggregate([
{
  $project: {
    newDate: {
      $dateAdd: {
        startDate: new Date("2024-08-29"), // Starting date
        unit: "day",                        // Unit to add (e.g., "day", "month", "year")
        amount: 7                          // Amount to add
      }
    }
  }
}
])
```



Aggregation Operations

Aggregation operations process multiple documents and return computed results.



Variables

System Generated Variables

Document: <https://www.mongodb.com/docs/manual/reference/aggregation-variables/#system-variables>

```
db.cars.aggregate(  
    {$project:{_id:0,model:1,date:"$$NOW"}})
```


User Defined Variables

These variables allow you to store values and reuse them within the same pipeline, making the pipeline more readable and efficient in certain scenarios.

```
{  
  _id: ObjectId('66bfcd969a4c39150c228fc0'),  
  maker: 'Hyundai',  
  model: 'i20',  
  total_service_cost: 9500,  
  cost_status: 'Low'  
},
```

```

car_dealership> db.cars.aggregate([
...     {
...         $match: { maker:"Hyundai" }
...     },
...     {
...         $set: {
...             total_service_cost: {
...                 $sum: "$service_history.cost"
...             }
...         }
...     },
...     {
...         $project: {
...             maker: 1,
...             model: 1,
...             total_service_cost: 1,
...             cost_status: {
...                 $let: {
...                     vars: { totalCost: "$total_service_cost" },
...                     in: {
...                         $cond: {
...                             if: { $gte: ["$$totalCost", 10000] },
...                             then: "High",
...                             else: "Low"
...                         }
...                     }
...                 }
...             }
...         }
...     }
... ]);

```



Data Modeling



Relations



MongoDB is a NoSQL database, it doesn't enforce strict schema relationships like foreign keys in relational databases.



We can still model relationships between documents in MongoDB using a few approaches.

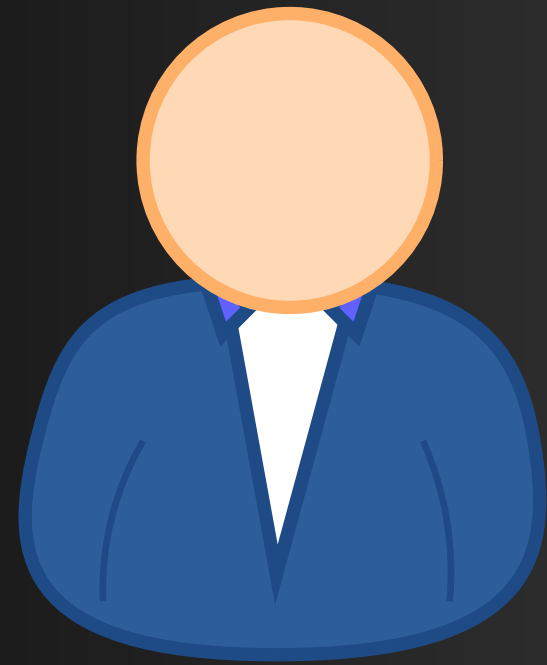
The two main types of relationships are:

- 1. Embedded Documents (Denormalization)**
- 2. Referenced Documents (Normalization)**

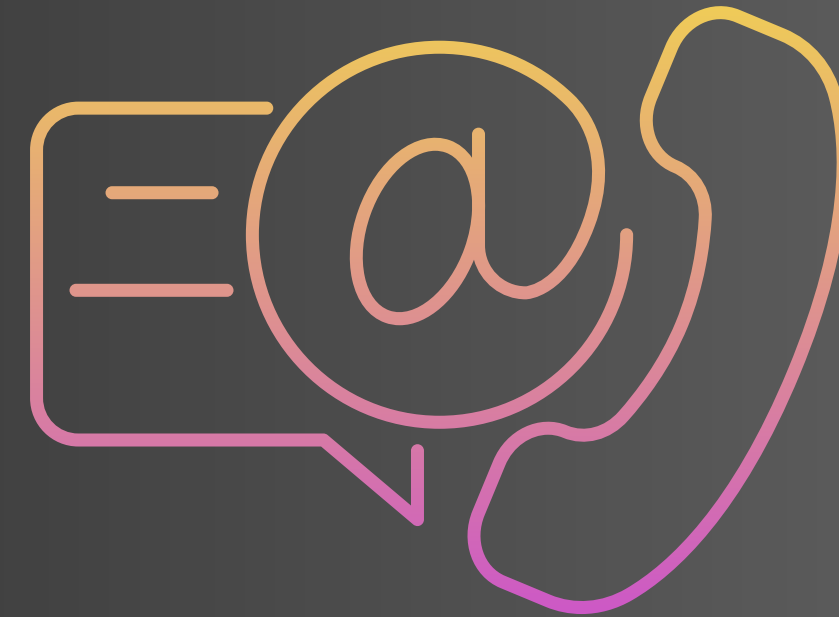
Types of Relationship

- One to One
- One to Many
- Many to Many

1:1

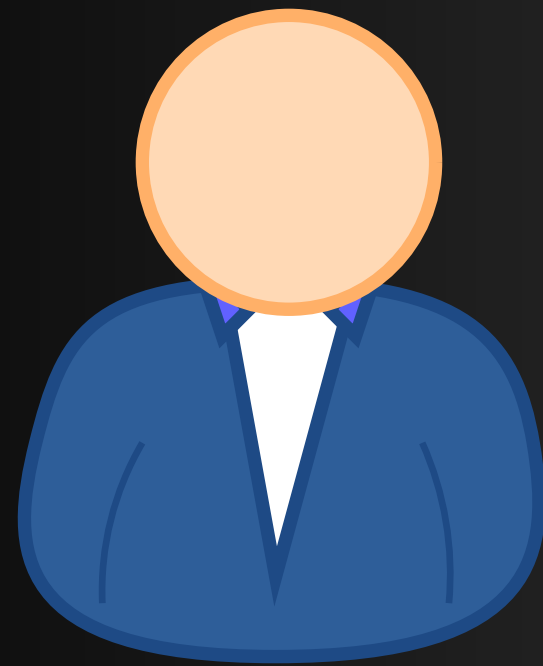


User

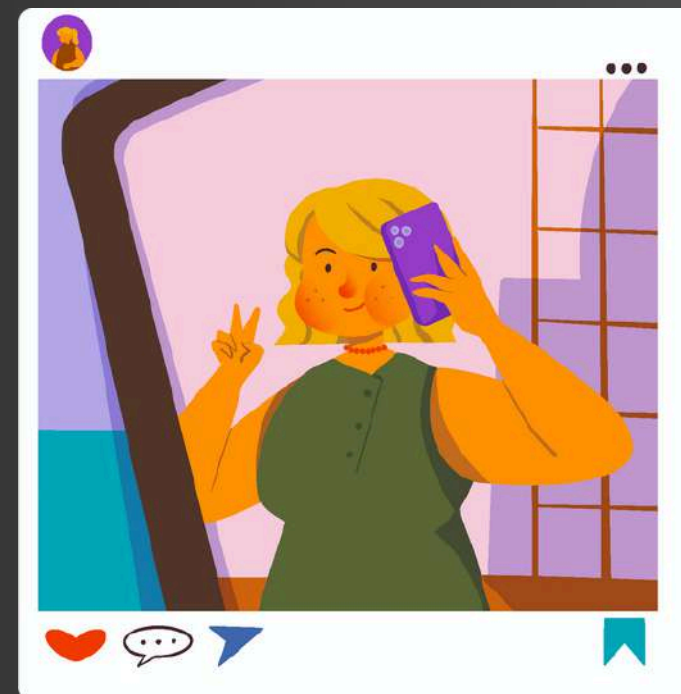
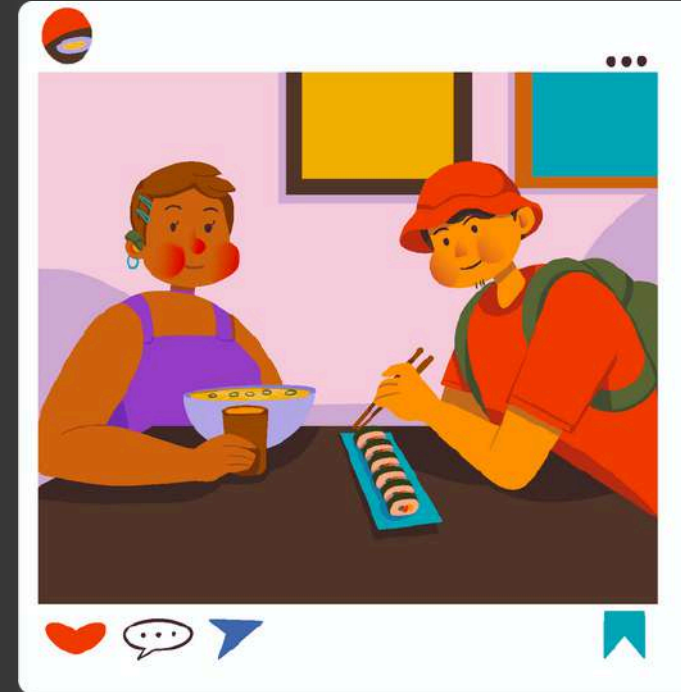


Contact

1 : MANY



User



Posts

Many : Many



Courses



Students



Java



Raju



Sham



Math



Raju



Sham



Python



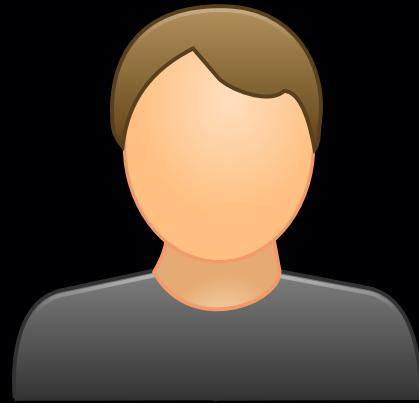
SQL



Mongodb

**How can we maintain
relationship in MongoDB?**

users



orders



Embedded Documents

- userA
 - orderA
 - orderB

```
{
  "_id": "user1",
  "name": "Amit Sharma",
  "email": "amit.sharma@example.com",
  "orders": [
    {
      "_id": "order1",
      "product": "Laptop",
      "amount": 50000,
      "order_date": "2024-08-01"
    },
    {
      "_id": "order2",
      "product": "Mobile Phone",
      "amount": 15000,
      "order_date": "2024-08-05"
    }
  ]
}
```

Referenced Documents

```
{
  "_id": "user1",
  "name": "Amit Sharma",
  "email": "amit.sharma@example.com",
  "phone": "+91-987654210",
  "address": "MG Road, Mumbai, Maharashtra"
},
{
  "_id": "user2",
  "name": "Priya Verma",
  "email": "priya.verma@example.com",
  "phone": "+91-987654211",
  "address": "Nehru Place, New Delhi, Delhi"
}
```

users

```
{
  "_id": "order1",
  "user_id": "user1",
  "product": "Laptop",
  "amount": 50000,
  "order_date": "2024-08-01"
},
{
  "_id": "order2",
  "user_id": "user2",
  "product": "Mobile Phone",
  "amount": 15000,
  "order_date": "2024-08-05"
}
```

orders



JOIN with \$lookup



```
db.users.aggregate([
```

```
{
```

```
  "$lookup": {
```

```
    "from": "orders",
```

```
    "localField": "_id",
```

```
    "foreignField": "user_id",
```

```
    "as": "orders"
```

```
  }
```

```
}
```

```
1)
```

```
// The target collection to join with  
// The field from the 'users' collection  
// The field from the 'orders' collection  
// The name of the new array field to add to the 'users'
```

Embedded Documents

1. Document Size Limit

MongoDB imposes a maximum document size limit of **16 MB**. This means that the entire document, including all embedded documents, cannot exceed 16 MB in size. This limit is generally large enough for most use cases, but it can become a concern if you have a lot of embedded data.

2. Nesting Depth Limit

MongoDB allows documents to be nested up to **100 levels deep**. However, deeply nested documents can become difficult to manage and may lead to performance issues, so it is generally advisable to keep the nesting as shallow as possible.

MPRASHANT



Schema Validation

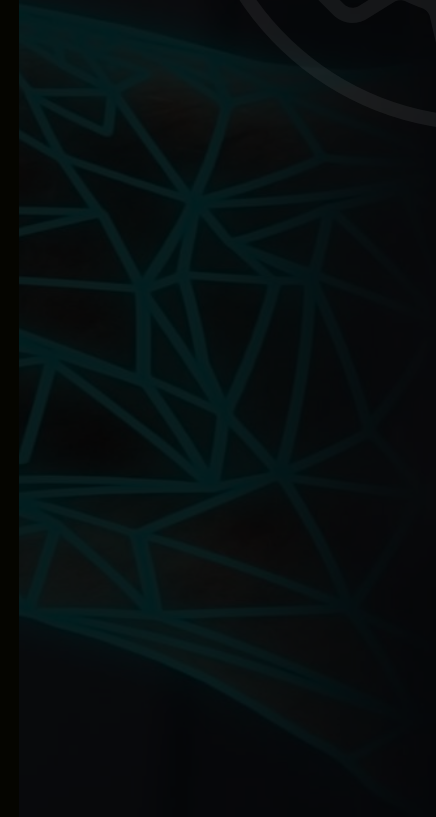


MongoDB uses a JSON Schema format to define the validation rules.

Allows you to specify various constraints and rules for your documents, such as required fields, field types, and value ranges.

**We can add validation while
creating collection...**

```
db.createCollection("users", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        age: {
          bsonType: "int",
          minimum: 18,
          description: "must be an integer and is required"
        }
      }
    }
  },
  validationLevel: "strict",
  validationAction: "error"
})
```



**we can update the existing
collection to add validation**

```
db.runCommand({
  collMod: "users",
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        age: {
          bsonType: "int",
          minimum: 18,
          description: "must be an integer and is required"
        }
      }
    }
  },
  validationLevel: "moderate", // Changes validation level to moderate
  validationAction: "warn"    // Changes validation action to warn
})
```




Validation Levels:

- **strict:** The document must fully comply with the schema validation rules. If a document does not comply, it will not be inserted or updated in the collection.
- **moderate:** Only new documents and modified fields in existing documents are validated against the schema. This allows for partial validation and can be useful for legacy systems or gradual schema enforcement.



Validation Actions:

- **error**: If a document does not meet the schema validation criteria, MongoDB will throw an error and reject the insert or update operation.
- **warn**: MongoDB logs a warning message when a document does not meet the schema validation criteria but still allows the insert or update operation.



Schema Validation

Use schema validation to ensure there are no unintended schema changes or improper data types.



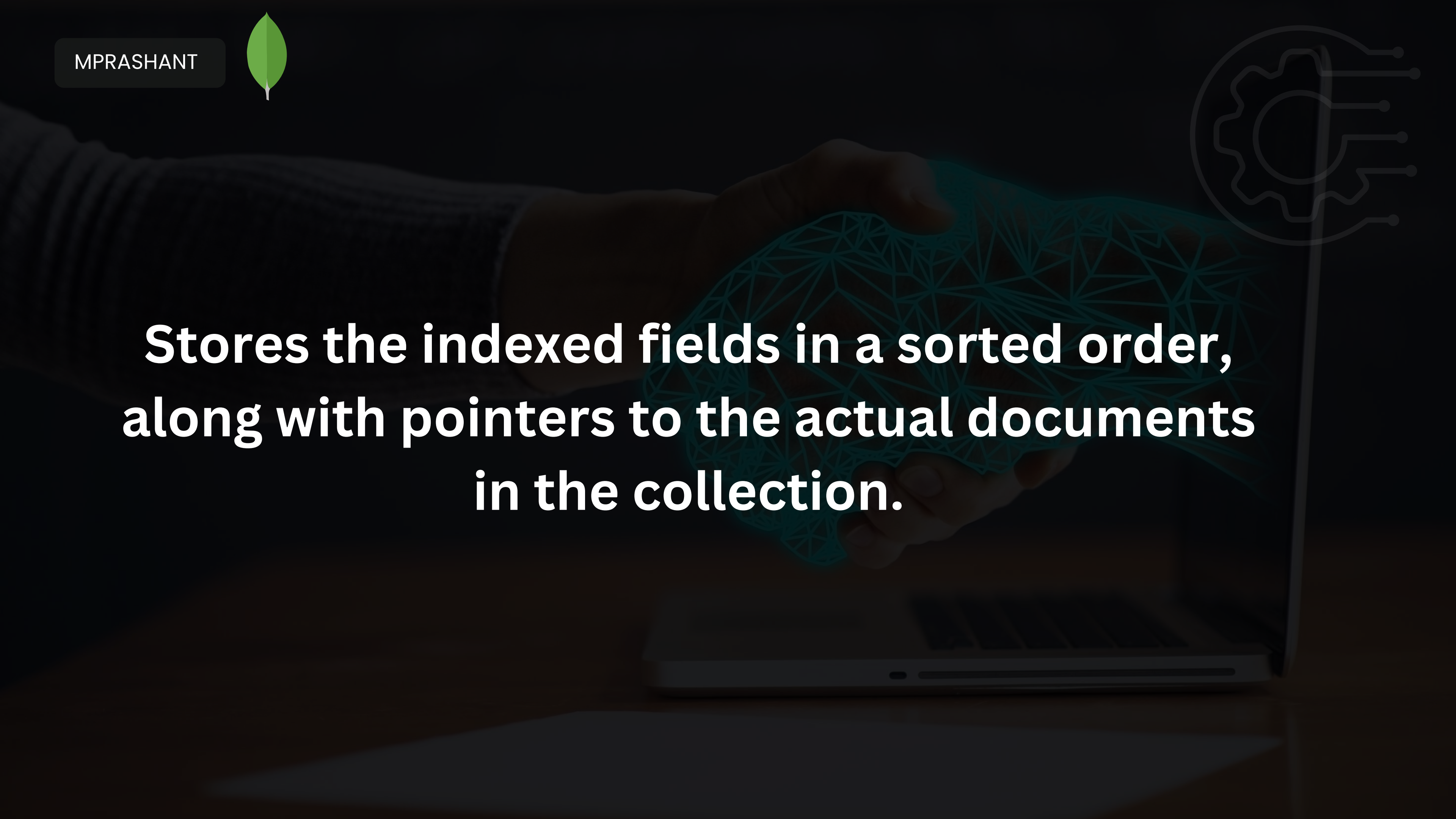
MPRASHANT



INDEXES



An INDEX is a data structure that improves the speed of query operations by allowing the database to quickly locate and access the required data without scanning every document in a collection.

The background of the slide features a dark, low-key photograph of a person's hand resting on a laptop. Overlaid on this image is a complex, glowing blue network of interconnected lines and nodes, resembling a data mesh or a neural network. In the top right corner, there is a faint, light gray icon of a gear with circuit lines extending from it.

**Stores the indexed fields in a sorted order,
along with pointers to the actual documents
in the collection.**



Index based on name

Baburao

Raju

Sham

name: Raju
age: 25

name: Sham
age: 28

name: Baburao
age: 45



```
db.collection.createIndex(  
  { <field1>: <type1>,  
    <field2>: <type2>, ... }, {<options> })
```




```
db.cars.createIndex({ maker: 1 })  
db.cars.createIndex({ model: 1 }, { unique: true })
```

```
db.cars.dropIndex("maker")  
db.cars.getIndexes()
```



Types of Indexes:

- **Single Field Index**
- **Compound Index:** Involves multiple fields.
- **Unique Index:** Index that ensures no two documents have the same value for the indexed field.
- **TTL Index:** TTL (Time to Live) indexes that are used to automatically remove documents after a certain period.



Performance Considerations:

- **Impact on Write Operations:** While indexes speed up reads, they can slow down insertions, updates, and deletions because the indexes need to be maintained.
- **Indexing Large Collections:** Learn about the considerations when indexing large collections, such as index size and the impact on RAM.

MPRASHANT



Transaction

A transaction in MongoDB is a sequence of operations that are executed as a single unit, ensuring that all operations either complete successfully or are fully rolled back, maintaining ACID properties across multiple documents and collections.

Example of a Multi-Document Transaction:

You want to transfer money from one account to another, which requires updating two documents within a single transaction.



-1000



+1000

MPRASHANT



Replications

Replication is a group of MongoDB servers that maintain identical copies of data to ensure high availability, redundancy, and data durability, with one primary node handling writes and multiple secondary nodes replicating the data.

MPRASHANT



Sharding

Sharding is a method of distributing data across multiple servers (shards) to enable horizontal scaling, allowing the database to handle large datasets and high-throughput operations efficiently.



Interview Questions



What is MongoDB, and how is it different from a relational database?

Answer: MongoDB is a NoSQL database that stores data in flexible, JSON-like documents, allowing for schema-less data storage. Unlike relational databases, which use tables and rows, MongoDB uses collections and documents. This allows for more flexible and scalable data models.



Explain the structure of a MongoDB document. What are its key components?

Answer: A MongoDB document is a BSON (Binary JSON) object that consists of field-value pairs. The key components include fields (which are similar to columns in relational databases) and values (which can be of various data types, such as strings, numbers, arrays, or even nested documents).



What are the advantages of using MongoDB over traditional RDBMS systems?

Answer: Advantages include:

- **Flexibility in schema design.**
- **Horizontal scalability via sharding.**
- **High availability through replica sets.**
- **Easier to manage unstructured or semi-structured data.**
- **Rich querying and aggregation capabilities.**



How does MongoDB store data? Explain the concept of collections and documents.

Answer: MongoDB stores data in BSON format within documents. These documents are grouped into collections, which are analogous to tables in relational databases. Each document within a collection can have a different structure, allowing for flexible data modeling.



What is a replica set in MongoDB? How does it help in ensuring high availability?

Answer: A replica set in MongoDB is a group of mongod instances that maintain the same data set. One node acts as the primary node, and others are secondary nodes. The replica set ensures high availability by automatically failing over to a secondary node if the primary node goes down.



What is a sharded cluster in MongoDB, and why is it used?

Answer: A sharded cluster in MongoDB is a method for distributing data across multiple servers. Sharding is used to handle large datasets and high-throughput operations by distributing data and load across multiple machines.



Explain the differences between a find() query and an aggregate() query in MongoDB.

Answer: The find() query is used to retrieve documents from a collection based on certain criteria. The aggregate() query, on the other hand, is used for more complex data processing, allowing for data transformation and computation through an aggregation pipeline.



How does MongoDB handle indexing, and what are the types of indexes available?

Answer: MongoDB supports various types of indexes to improve query performance, including:

- **Single field index**
- **Compound index (multiple fields)**
- **Multikey index (for arrays)**
- **Text index (for searching text)**
- **Geospatial index (for querying geographical data)**
- **TTL index (for automatic deletion of documents after a certain period)**



What is the role of the `_id` field in MongoDB? Can it be modified or removed?

Answer: The `_id` field is a unique identifier for each document in a MongoDB collection. It is mandatory and cannot be removed. However, it can be modified if necessary, though this is not recommended as it may lead to data inconsistency.



What is the difference between insert() and insertMany() operations in MongoDB?

Answer: insert() is used to insert a single document into a collection, while insertMany() allows for the insertion of multiple documents in a single operation. insertMany() is more efficient for bulk inserts.



Explain how MongoDB's aggregation framework works. What are some common stages used in an aggregation pipeline?

Answer: The aggregation framework in MongoDB allows for processing data in a pipeline, where each stage transforms the data. Common stages include:

- **\$match:** Filters documents based on a condition.
- **\$group:** Groups documents by a specified key and performs aggregation operations.
- **\$sort:** Sorts documents.
- **\$project:** Reshapes the documents by including/excluding fields.
- **\$lookup:** Performs a left join with another collection.



How does MongoDB handle transactions, and what is the significance of multi-document transactions?

Answer: MongoDB supports multi-document transactions, which allow for ACID-compliant operations across multiple documents or collections. This ensures that either all operations within a transaction are applied, or none are, providing consistency and reliability in complex operations.



What are the pros and cons of embedding documents versus using references in MongoDB?

Embedding (Pros):

- Better performance for read operations.
- Simpler queries.

Embedding (Cons):

- Can lead to large documents and duplication of data.

References (Pros):

- More normalized data structure.
- Reduces duplication.

References (Cons):

- Requires additional queries (joins) which can be slower.



What is the difference between `updateOne()`, `updateMany()`, and `replaceOne()` in MongoDB?

Answer:

- **`updateOne()`:** Updates the first document that matches the query criteria.
- **`updateMany()`:** Updates all documents that match the query criteria.
- **`replaceOne()`:** Replaces the entire document with a new one, based on the query criteria.



How does MongoDB handle concurrency and ensure data consistency?

Answer: MongoDB handles concurrency using a combination of locks (e.g., collection-level locks) and journaling. It uses the WiredTiger storage engine, which provides document-level locking, allowing for higher concurrency. Consistency is maintained through replica sets and transactions.



What are the different types of data modeling strategies in MongoDB?

Answer: Common data modeling strategies include:

- **Embedding documents for data that is frequently accessed together.**
- **Referencing documents to normalize data and avoid duplication.**
- **Hybrid models that combine embedding and referencing for different use cases.**



Explain the concept of TTL (Time to Live) indexes in MongoDB and when they might be useful.

Answer: TTL indexes automatically delete documents from a collection after a specified period. This is useful for data that becomes irrelevant after a certain time, such as session logs, temporary data, or caching.



How would you optimize a MongoDB query for better performance?

Answer: To optimize a MongoDB query:

- **Use indexes effectively.**
- **Avoid full collection scans by using selective queries.**
- **Limit the amount of data returned by using projections.**
- **Use the explain() method to analyze query performance.**
- **Consider denormalization to reduce the number of joins.**



Explain how to perform a backup and restore operation in a MongoDB database.

Answer:

Backup: Use mongodump to create a binary backup of the database.

Restore: Use mongorestore to restore the data from a backup.

Additionally, for cloud deployments, MongoDB Atlas provides automated backups.



Describe the process of migrating data from a relational database to MongoDB. What challenges might you face?

Answer: The migration process involves:

- **Analyzing the relational schema.**
- **Designing a MongoDB schema, often using denormalization and embedding.**
- **Exporting data from the relational database (e.g., using SQL queries).**
- **Importing data into MongoDB (e.g., using mongoimport or custom scripts).**
- **Challenges: Schema design differences, handling joins, data type conversion, and ensuring data consistency during migration.**



What are MongoDB's limitations, and how can you work around them?

Answer: Limitations include:

- **No built-in support for joins (use aggregation or manual joins).**
- **Limited support for multi-document ACID transactions (introduced in later versions).**
- **Large documents can impact performance (use references to mitigate).**
- **Working around limitations often involves thoughtful schema design, indexing, and understanding MongoDB's strengths.**



How does MongoDB Atlas differ from running MongoDB on-premises?

Answer: MongoDB Atlas is a fully managed cloud database service that automates deployment, scaling, and backups. It offers built-in security features and integration with other cloud services. Running MongoDB on-premises requires manual management of hardware, scaling, backups, and security.



Explain the role of journaling in MongoDB. How does it help in ensuring durability?

Answer: Journaling in MongoDB is a mechanism that logs write operations to a journal file before applying them to the database. In case of a crash, MongoDB can use the journal to recover to a consistent state, ensuring durability of write operations.



How would you handle schema versioning in a MongoDB application?

Answer: Schema versioning can be handled by:

- **Embedding a version field in each document.**
- **Using a migration script to update existing documents to the new schema version.**
- **Designing the application to handle multiple schema versions during the transition period.**

MPRASHANT



MPRASHANT

