

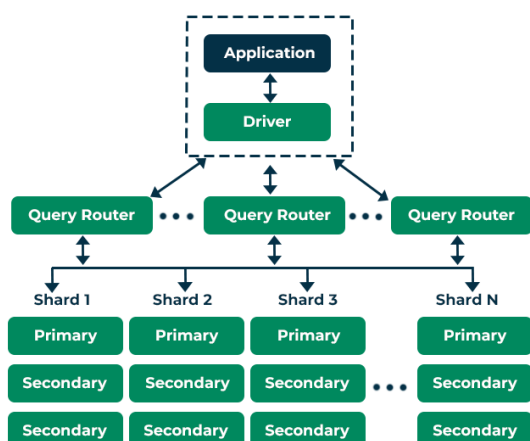
MongoDB

MongoDB is a popular NoSQL database designed to store and manage large amounts of unstructured or semi-structured data. It is document-oriented, meaning it stores data in flexible, JSON-like documents.

SQL vs NoSQL :

SQL	NoSQL
Stands for Structured query language	Stands for Not only SQL
Relational database model	Non relational database model
Fixed Schema	Dynamic Schema
Data is stored in tables like rows and columns	Data is stored in document, key – value, graphs, etc..
Requires vertical scaling to handle large amount of data.	Horizontal scaling is possible for large amount of data
Optimised for complex and transactional operations	Optimised for scalability and performance
Eg; MySQL, Oracle, PostgreSQL	Eg ; MongoDB, Cassandra, Redis

MongoDB Architecture :



Application :

Applications initiate requests to MongoDB by sending queries, inserts, updates, and other operations through the MongoDB drivers.

Drivers :

Drivers are libraries that provide a connection between an application and MongoDB, allowing the application to interact with the database. It act as the intermediary, converting these queries into MongoDB commands that the database can understand.

Query Router :

The query router (mongos) in MongoDB is responsible for distributing client queries across shards in a sharded cluster.

Shards :

A shard is a horizontal partition of data that is spread across multiple servers. MongoDB uses sharding to distribute data across multiple nodes and support horizontal scalability.

Features :

- Collection of json documents.
- All collections are separated
- Dynamic schema
- Horizontal Scaling
- Limited amount of data types
- Used in big data and real time web application.

Shell commands :

show dbs – list all databases in our system

use dbname – to open a particular db

show collections – list all collections in a db

db.collectionName.find() – read all objects inside the collections.

Insert commands

db.collectionName.insertOne({data})

db.collectionName.insertMany([{data1},{data2}])

Complex query commands

\$gt – greater than

\$lt – less than

\$gte – greater than or equal to

\$lte – less than or equal to

\$in – if data is present, return it.

\$nin – return others except the entered data.

\$exists:true – If field exists, return those data.

\$exists:false – If field doesn't exist, return those data.

\$or,\$and,\$not

\$expr – to compare two diff properties of an object.

countDocuments

Update Commands

\$set,\$rename,\$unset,\$push,\$pull,\$replaceOne

db.collectionName.updateOne({key:""},{\$set:{key:""}})

Delete Commands

db.collectionName.deleteOne()

db.collectionName.deleteMany()

Schema Design :

- No algorithm
- No process
- No rules

Design a schema based on the unique needs of your application.

MongoDB supports two primary data modeling approaches: **Embedded** and **References documents**.

Embedded Documents :

Store related data within a single document. Best for data that is frequently accessed together (updating all related data in a single operation).

```
1  {
2    "id": "Country_1",
3    "name": "India",
4    "code": "IND",
5    "states": [
6      {
7        "name": "Kerala",
8        "code": "KL",
9        "cities": [
10         {
11           "name": "Kannur",
12           "code": "CAN"
13         },
14         {
15           "name": "Malappuram",
16           "code": "MLP"
17         }
18       ]
19     }
20 ]
21 }
```

Accessing Kannur city :

```
> db.embedded.aggregate([
  { $unwind: "$states" },
  { $unwind: "$states.cities" },
  { $match: { "states.cities.name": "Kannur" } },
  { $project: { _id: 0, "name": "$states.cities.name", "code": "$states.cities.code" } }
])
< {
  name: 'Kannur',
  code: 'CAN'
}
```

References Documents :

Store related data in separate documents and link them using references (ObjectIDs). Suitable for data that is large, rarely accessed together, or needs to be updated independently.

```

{
  "id": "Country_1",
  "name": "India",
  "code": "IND"
}

{
  "id": "State_1",
  "name": "Kerala",
  "code": "KL",
  "c_id": "Country_1"
}

{
  "id": "Cities_1",
  "name": "Kannur",
  "code": "CAN",
  "s_id": "State_1"
}

```

populate() :

The populate() method in Mongoose is used to automatically replace a field in a document with the actual data from a related document.

```

29
30 State.find()
31   .populate("c_id")
32   .then(states => console.log(states))
33   .catch(error => console.log('Error', error));
34
35

```

Advantages of denormalization in MongoDB :

Denormalization means combining related data into a single document.

Advantages :

- Improved Query Performance
- Reduced Complexity
- Easier Maintenance and Updates
- Improved Read Performance
- Flexibility in Data Structure

Indexes :

Indexes are special data structures that store information about the documents in a way that makes it easier for MongoDB to quickly locate the right data.

Types of indexes:

- **Single field index :**

It means index on a single field of a document. This index is helpful for fetching data in ascending as well as descending order.

```
db.students.createIndex({"<fieldName>" : <1 or -1>});
```

1 – ascending order

-1 – descending order

- **Compound Index :**

An index on multiple fields. Useful when queries frequently involve multiple fields in the filter.

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

- **Multikey index :**

Created when indexing fields that hold arrays. MongoDB automatically indexes each element of the array.

```
db.collection.createIndex({ arrayField: 1 })
```

Creating and using an index:

To create an index on a field in a MongoDB collection, you use the `createIndex()` method. Once an index is created, MongoDB will automatically use it when you run queries that can benefit from the index.

Best practises of indexing :

- Use compound indexes for multiple queries as it can cover multiple fields in a single index, making queries more efficient when filtering by more than one field.
- Choose the right index type
- Eliminate unnecessary indexes as it consumes memory and disk space.
- Use covered queries it occurs when all the fields used in the query (both in the filter and projection) are included in the index.

MongoDB can retrieve the result directly from the index without scanning the collection.

```
db.users.createIndex({ name: 1, age: 1 })
```

```
db.users.find({ name: "John" }, { name: 1, age: 1, _id: 0 })
```

- Use partial indexes as it allow MongoDB to index only the documents that meet a specific condition

Performance impact of indexes :

- Faster Query Performance
- Improved Sorting
- Additional storage
- Slower writes

Aggregation :

It is a database process that allows us to perform complex data transformations and computations on collections of documents or rows. It allows you to perform operations on your data, such as filtering, grouping, sorting, to obtain desired results.

Aggregation pipelines :

The aggregation pipeline consists of a series of stages that process documents in a collection. Each stage transforms the data as it passes through, and the output of one stage can be used as the input for the next.

Stages in Aggregation pipeline :

\$match : Filters documents based on specified criteria

```
1  [
2  {
3    $match: {
4      name: "Jane Austen"
5    }
6  ]
```

\$group : Groups documents by a specified key and allows you to perform aggregate operations (sum, average, count, etc..).

```
Untitled - modified  [SAVE] [CREATE NEW] [EXPORT TO LAN]

1  [
2  {
3    $group: {
4      _id: "$nationality",
5      averageAge: {
6        $avg: "$age"
7      }
8    }
9  }
```

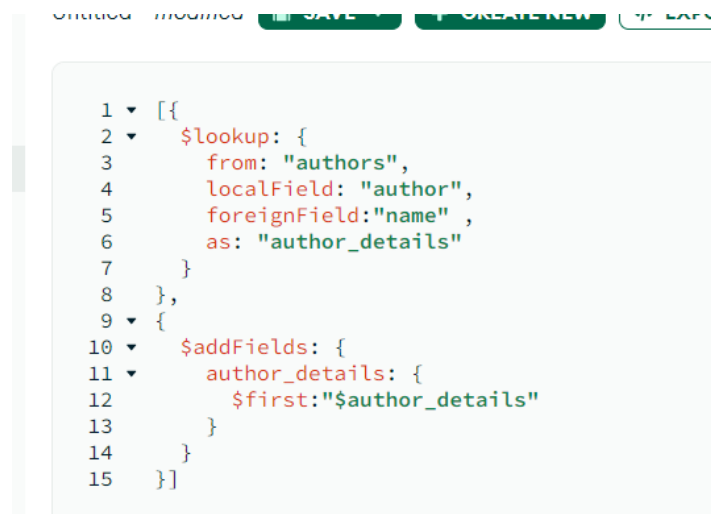
\$sort : Sorts the documents based on specified fields.

```
1  [
2  {
3    $group: {
4      _id: "$nationality",
5      count: {
6        $sum: 1
7      }
8    }
9  },
10 {
11   $sort: {
12     count: 1
13   }
14 }
15 ]
```

\$project: Reshapes documents by including or excluding fields and creating new fields

```
1  [
2  {
3    $match: {
4      genre: "Classic"
5    }
6  },
7  {
8    $project: {
9      title: 1
10     }
11  }
12 ]
```


\$lookup : is used to perform a left outer join between two collections.



```
1  [{
2    $lookup: {
3      from: "authors",
4      localField: "author",
5      foreignField: "name",
6      as: "author_details"
7    }
8  },
9  {
10   $addFields: {
11     author_details: {
12       $first: "$author_details"
13     }
14   }
15 }]
```

Relationships in MongoDB

One to One :

It means that a document in one collection corresponds to exactly one document in another collection.

One to Many :

It means that a document in one collection can be associated with multiple documents in another collection.

Many to one :

Multiple documents in one collection can reference a single document in another collection.

Many to Many :

It means that documents in one collection can be associated with multiple documents in another collection

Embedding vs referencing strategies :

Embedding involves storing related data within a single document. This means that all the data you need is in one place.

Referencing involves creating a relationship between documents by storing the ID of one document within another. This allows for more flexibility and normalization of data.

Schema validation :

- Specify the datatype for each field.
- Define which fields must be present in the document.
- Use regular expressions to enforce a specific format for string fields.
- Validate the structure of arrays, including their lengths and types of elements.
- Validate nested documents using their own schemas.

Using JSON Schema in MongoDB :

A JSON Schema is a JSON document that defines the structure of your data. Using JSON Schema in MongoDB allows you to enforce validation rules on your documents, ensuring that they adhere to a defined structure, including required fields, data types, patterns, and nested structures.

Validation levels :

Strict Validation:

This level enforces all rules and constraints defined in the schema. Any deviation from the specified structure, types, or constraints results in a validation error.

- All required fields must be present.
- All data types must match exactly.
- All constraints must be followed.
- No additional properties are allowed unless explicitly defined.

Moderate Validation :

- No additional properties are allowed unless explicitly defined.
- Type mismatches for certain properties may be tolerated or converted.

- Constraints on arrays and objects may be relaxed (allowing empty arrays).

MongoDB ACID Transactions :

MongoDB supports multi-document ACID (Atomicity, Consistency, Isolation, Durability) transactions, which allow you to execute multiple operations across different documents or collections with guarantees that either all operations succeed or none do.

1. Atomicity :

It states that all the operations of the transaction take place at once and if not then the transaction is aborted.

2. Consistency :

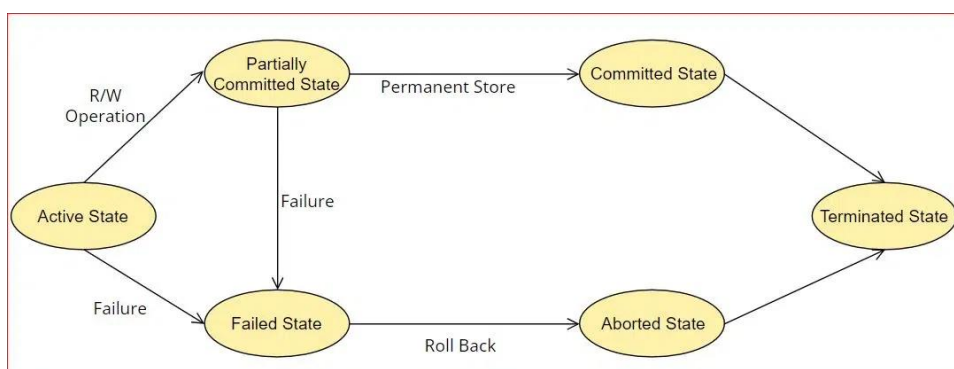
The consistency in transactions ensures that before and after the transaction operations the database remains in a valid state.

3. Isolation :

Transactions are isolated from one another. Changes made in one transaction are not visible to other transactions until they are committed.

4. Durability :

It states that after the transaction completes successfully the changes it has made to the database are permanent even if there are system failures.



When to use transactions :

- Transactional requirements
- Complex data operations
- Data consistency

Best practises :

- Keep Transactions Short
- Always handle errors and abort transactions when necessary to ensure data integrity.
- When executing multiple operations that need to be consistent, make sure to group them under the same session.

Limitations :

- Size Limitations
- Potential for Deadlocks
- Limited Scalability
- Increased latency

MongoDB Replication :

Replication is a process in MongoDB that allows data to be copied across multiple servers, providing redundancy and high availability. Replication ensures that if one server fails, another server can take over and continue serving the data, which is especially useful for fault tolerance and disaster recovery.

Replica set :

It is a group of MongoDB instances (servers) that maintain the same dataset. It consists of multiple **nodes** (servers)

Primary Node :

This is the main server that receives all write operations. It is the only node that can accept writes.

Secondary Node :

These nodes replicate the data from the primary node. They maintain copies of the dataset and can serve read operations.

Failover :

If the primary node fails, an election takes place among the secondary nodes to choose a new primary. This ensures that the system continues to work without any downtime.

Recovery :

Recovery in MongoDB involves a failed primary node rejoining the replica set as a secondary after it comes back online, during which it syncs missed operations from the current primary.

Sharding :

Sharding is a method in MongoDB used to horizontally scale a database by distributing data across multiple servers, known as shards. Each shard holds a subset of the data, and a shard key is used to determine how data is distributed, ensuring efficient query handling and high availability.

Horizontal Scaling :

Horizontal scaling in MongoDB, often called sharding, is a technique where data is distributed across multiple servers or machines.

Sharding Process :

Shard key :

It's a field (or a combination of fields) within a document that MongoDB uses to divide and distribute the data across shards.

Data Distribution :

The data is divided into chunks, which are distributed across the shards. The shard key ensures that documents with similar shard key values are grouped together and stored in the same shard.

Scaling out:

As the data grows, MongoDB can add more shards to distribute the load further. The **config servers** track which shard holds which data, and the **query router** sends queries to the appropriate shards.

Impact of Shard key :

- Data distribution and balancing
- Query efficiency
- Chunk splitting and migration

MongoDB Security:

Data encryption :

Data encryption in MongoDB ensures that your data is protected from unauthorized access, both while stored on disk (encryption at rest) and while transmitted over the network (encryption in transit).

- Encryption at rest :
It protects the data stored on disk from unauthorized access, even if someone gains physical access to the storage media. MongoDB Enterprise and Atlas offer integration with cloud KMS for key management and AES-256 encryption.
- Encryption in transit :
It protects data from being altered while being transmitted between a MongoDB client and server. MongoDB supports **TLS (Transport Layer Security)** and **SSL (Secure Sockets Layer)** to encrypt all network communication between:
 - MongoDB clients and MongoDB servers
 - MongoDB servers in replica sets or sharded clusters

Securing MongoDB deployments :

IP Whitelisting :

It is a security practice that allows access to a system only from specific IP addresses or ranges. In MongoDB, you can use IP whitelisting to limit the network access to your MongoDB deployment. It is especially useful for securing your database when it is hosted in a cloud environment, such as MongoDB Atlas or when accessible over the internet.

- White list : specify the IP addresses or ranges of IPs that are allowed to connect to your MongoDB instance.

- Any IP address not on the whitelist will be denied access.
- It is configured either at the network level or within MongoDB's configuration itself.

explain() :

It allows you to see detailed information about how MongoDB executes a query.

```
db.collection.find({ field: "value" }).explain();
```

Execution stats :

Provides detailed execution statistics, including the number of documents examined and time taken.

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

All plans execution :

Shows stats for all candidate plans considered by the query planner.

```
db.collection.find({ field: "value" }).explain("allPlansExecution");
```

Profiler :

It captures detailed information about the performance of database operations, including queries, updates, and deletes. It can help identify slow queries and other performance bottlenecks.

Index optimization :

- Use compound indexes
- Analyze query patterns
- Limit the no of indexes
- Use covered queries

Query Optimization :

- Use limit() and skip()
- Filter early
- Use projection
- Index sort fields

Memory Management :

Memory management in MongoDB is a critical aspect of its performance and efficiency. It involves how MongoDB allocates, uses, and optimizes RAM for storing data, executing queries, and managing connections.

- MongoDB utilizes available RAM to cache frequently accessed data and indexes, minimizing disk I/O.
- The WiredTiger storage engine is the default for MongoDB and provides advanced memory management features.
- MongoDB allocates memory dynamically based on the workload and available resources. It monitors memory usage and adjusts allocations as needed.

Caching :

Caching is a technique used to temporarily store frequently accessed data in a storage layer that is faster to retrieve than the original source.

- MongoDB caches query results to improve performance. If a query is executed multiple times, the results may be retrieved from the cache rather than re-executing the query against the database.
- MongoDB caches data at both the database and collection levels.
- Lower Load on the Database

Full text search :

It is a powerful feature that allows you to perform searches on string content within your documents.

```
db.collection.find({ $text: { $search: "search term" } });
```

Time series collection :

It is a type of MongoDB collection that is optimized for storing and querying time-stamped data, such as logs, sensor readings, financial transactions, and performance metrics.

```
db.createCollection("sensorData", {  
  timeseries: {
```



```
timeField: "timestamp",  
metaField: "metadata",  
granularity: "seconds"  
}  
});
```

GridFS :

It allows users to manage large datasets effectively by splitting files into smaller chunks and storing them across multiple documents in a collection.

- ***File splitting*** : Large files are divided into chunks, typically 255 KB each. Each chunk is stored as a separate document in a collection.
- ***Metadata storage*** : Alongside file chunks, GridFS also stores metadata for each file in a separate collection (usually fs.files)[filename, content type, upload date, and a reference to the chunks.]