NoSQL, MongoDB with ODM

Reading Material

PW SKILLS

Topics Covered

- Understanding the Concepts
  - Understanding NoSQL, MongoDB, and ODM
  - CAP theorem
  - Partitioning and Sharding
  - Indexes
  - Database and CRUD with Mongodb
  - MongoDB Aggregation
  - Mongoose

Understanding the Concepts

Understanding to NoSQL, MongoDB, and ODM
NoSQL (Not Only SQL)

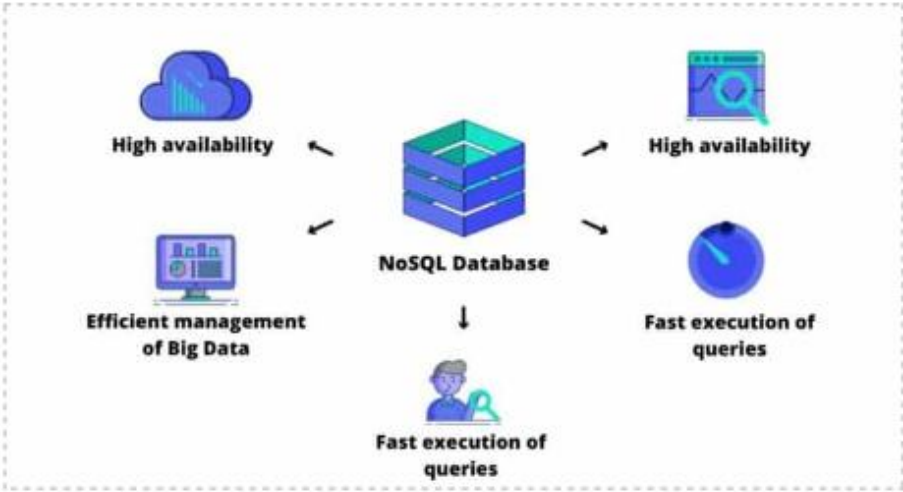


Fig. NoSQL Database

NoSQL databases provide a mechanism for storing, retrieving, and manipulating data in a different approach compared to traditional relational databases (RDBMS).

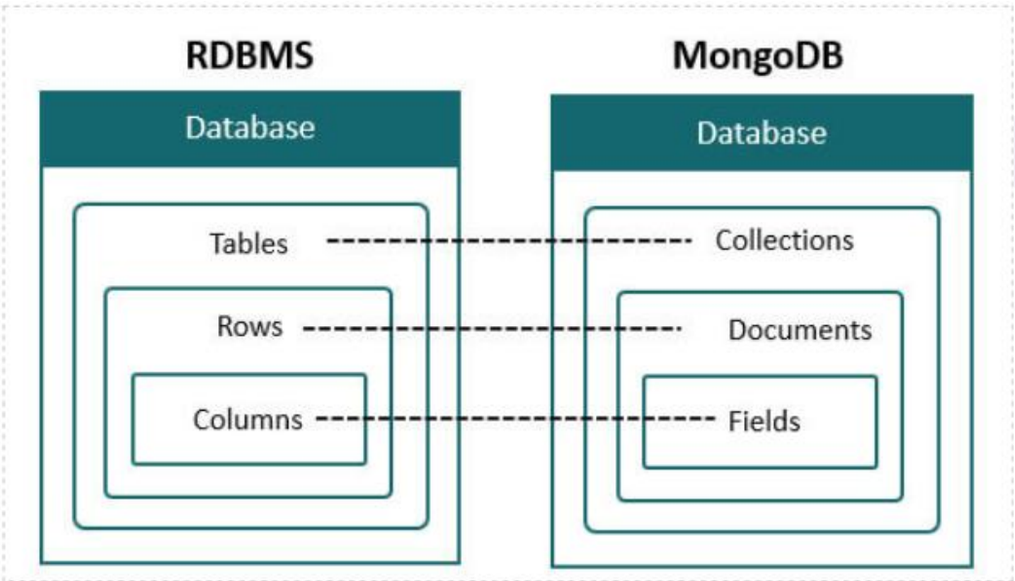
Key features of NoSQL -

- **Scalability** - NoSQL databases like MongoDB offer a flexible schema, allowing data to be stored without a predefined structure, making them suitable for modern applications with evolving data requirements MongoDB
- **Flexibility** - NoSQL databases like MongoDB offer a flexible schema, allowing data to be stored without a predefined structure, making them suitable for modern applications with evolving data requirements
- **Scalability** - NoSQL databases are more scalable than RDBMS, enabling them to handle big data by adding more servers and increasing productivity
- **Tyes can be** - document databases (like MongoDB), key-value stores, column-oriented databases, and graph databases, each catering to different data storage and retrieval needs

MongoDB

MongoDB is a popular NoSQL database that stores data in JSON-like documents with a dynamic schema, allowing different documents in the same collection to have varying structures, it scales horizontally by adding more servers, providing superior performance and flexibility compared to traditional RDBMS

MongoDB offers a flexible and scalable document structure, efficient indexing, and storage techniques, making it faster and more adaptable for modern applications

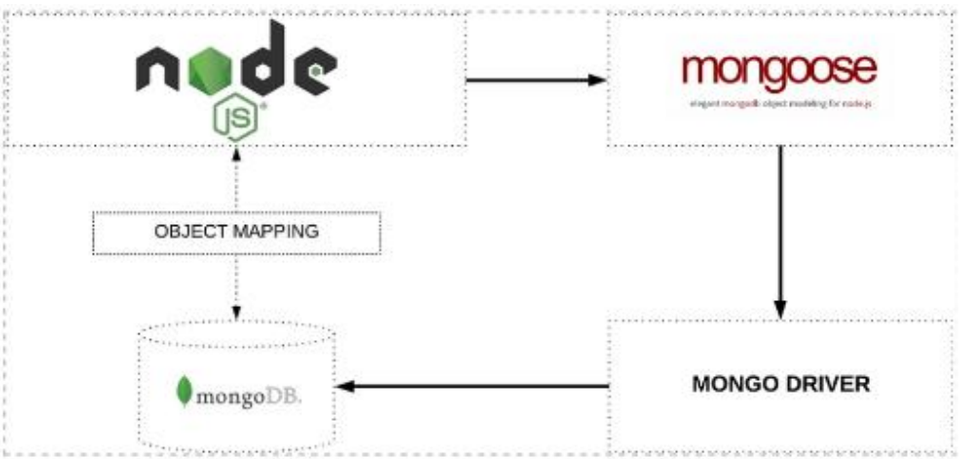


Understanding concepts by comparing MongoDB with RDBMS

The fundamentals/key concepts of MongoDB include -

1. **Collection** - Equivalent to tables in relational databases, collections in MongoDB store multiple JSON documents
2. **Documents** - Equivalent to records or rows in SQL databases, documents in MongoDB hold data in a JSON-like format.
3. **Fields** - Similar to columns in SQL tables, fields in MongoDB represent attributes of a document

ODM

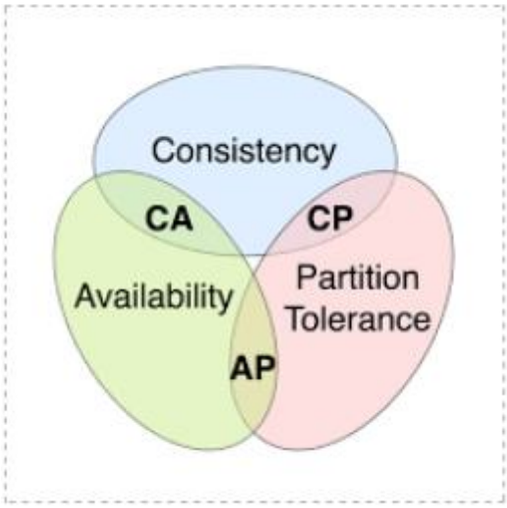


Overview of ODM (mongoose) for MongoDB

ODM which stands for Object-Document Mapping, is a technique that maps objects from an object-oriented programming language to documents in a document-oriented database like MongoDB. It simplifies the interaction between application code and the database, allowing developers to work with objects in code while storing and retrieving data in a document database like MongoDB

Some of the features of ODM include schema validation, data mapping, and query building to streamline the development process and enhance productivity when working with document databases.

CAP theorem



CAP theorem overview

The Cap theorem which is also known as Brewer's theorem, states that in a distributed database system, it is impossible to simultaneously guarantee all three of the following characteristics: Consistency, Availability, and Partition Tolerance.

Here is a breakdown of these characteristics in the context of the CAP theorem -

- **Consistency:** In a consistent system, all nodes see the same data at the same time. This means that any read operation should return the value of the most recent write operation, ensuring that all nodes return the same data.
- **Availability:** Availability ensures that the system remains operational and responsive to requests, even if individual nodes may be down. Every request should receive a non-error response, regardless of the state of a node. However, this does not guarantee that the response contains the most recent write.
- **Partition Tolerance:** Partition tolerance is essential in distributed systems and refers to the system's ability to continue functioning despite network partitions or communication failures between nodes. It involves replicating data across nodes and networks to maintain system operation even in the face of network disruptions.

From the above figure -

- **CA (Consistency and Availability):** Focuses on achieving both data consistency and system availability, with potential performance adjustments to maintain data accuracy.
- **CP (Consistency and Partition Tolerance):** Emphasizes maintaining data consistency even during network partitions, potentially leading to temporary unavailability to uphold data integrity.
- **AP (Availability and Partition Tolerance):** Prioritizes system availability despite network disruptions, accepting temporary data deviations to maintain operational resilience

In distributed systems, balancing between Consistency, Availability, and Partition Tolerance is crucial, and the CAP theorem helps in understanding the trade-offs involved in designing and managing distributed databases and systems.

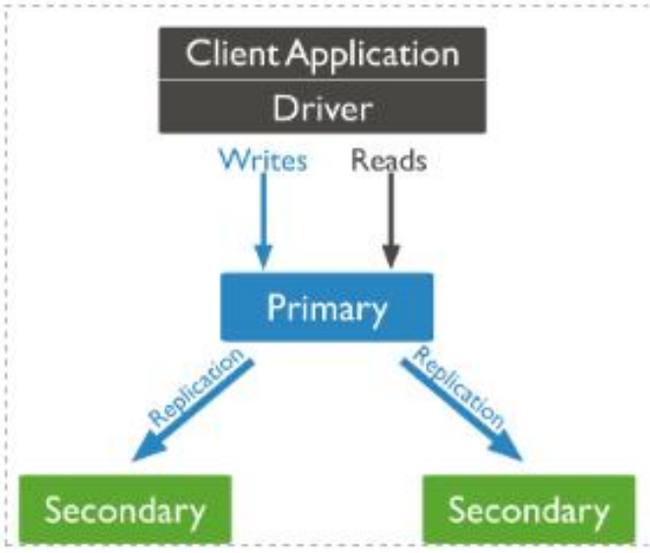
ODM which stands for Object-Document Mapping, is a technique that maps objects from an object-oriented programming language to documents in a document-oriented database like MongoDB. It simplifies the interaction between application code and the database, allowing developers to work with objects in code while storing and retrieving data in a document database like MongoDB

Some of the features of ODM include schema validation, data mapping, and query building to streamline the development process and enhance productivity when working with document databases.

Partitioning and Sharding

Partitioning

Replication in MongoDB is the process of synchronizing data across multiple servers to provide redundancy, increase data availability, and enable high fault tolerance. MongoDB achieves replication using replica sets, which are groups of MongoDB instances that maintain the same data set

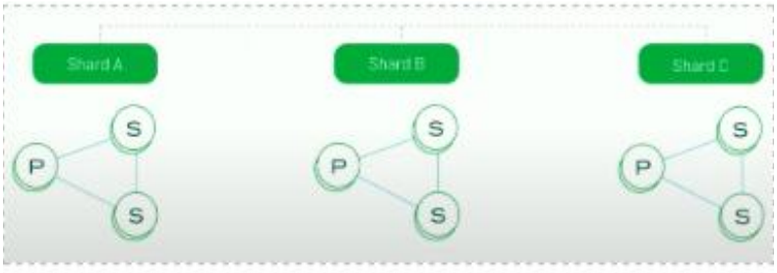


Overview of Replication in MongoDB

Key points about MongoDB replication -

- A replica set consists of multiple nodes, with one primary node that receives all write operations, and secondary nodes that replicate data from the primary
- The primary records all changes to its data sets in an operation log called the oplog. Secondary nodes apply these oplog entries to their data sets to maintain consistency
- If the primary becomes unavailable, an eligible secondary will hold an election to elect itself as the new primary, providing automatic failover
- Both primary and secondary nodes can serve read operations, allowing read scaling and load balancing.
- Replication provides high availability, fault tolerance, read scalability, and data redundancy.

Sharding



Overview of sharding

Sharding in MongoDB is a method for distributing data across multiple machines to support deployments with very large data sets and high throughput operations. It involves splitting the data into smaller chunks and storing them across multiple servers called shards

In system design, there are two methods of addressing system growth/scaling -

1. **Vertically Scaling** - this involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space. Limitations in available technology may restrict a single machine from being sufficiently powerful for a given workload.

2. **Horizontal Scaling** - involves dividing the system dataset and loading over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server

MongoDB supports horizontal scaling through sharding.

The sharding algorithm can be of three types -

1. **Hash-Based Sharding** - uses a hashed single or compound field as the shard key. It provides more even data distribution across the sharded cluster at the cost of reducing targeted operations and instead potentially requiring broadcast operations, particularly for range-based queries.

2. **Range sharding** - allows for efficient queries where reads target documents within a contiguous range as documents with 'close' shard key values are likely to be in the same chunk or shared

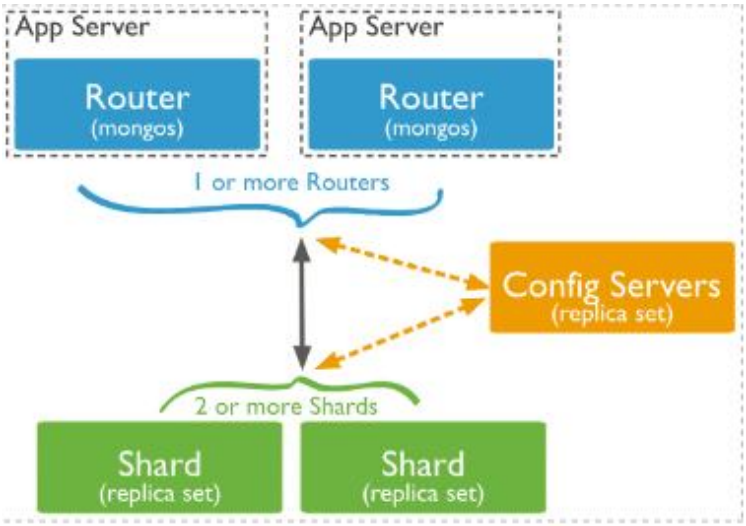
3. **Zone Sharding** - helps improve the locality of data for shared clusters that span multiple data centers.

The key components of a sharded cluster in MongoDB are

1. **Shards** - Each shard contains a subset of the sharded data. Each shard must be deployed as a replica set.

2. **Config Servers** - store metadata and configuration settings for the cluster.

3. **Query Routers (mongos)** - act as a query router, providing an interface between client applications and the shared cluster.



Sharding Architecture

Indexes

Indexes and Query in MongoDB are special data structures that store a small portion of the collection's data set in an easy-to-traverse form. They improve query performance by allowing MongoDB to quickly locate the data without scanning every document in a collection

The use cases can be, for an application that is repeatedly running queries on the same fields, create an index on those fields to improve performance.

For example, if your application frequently queries the product collection to populate data on existing inventory, you can create an index on the item and quantity fields to improve query performance.

To create an index in MongoDB, you can use the createIndex() method i.e
db.collection_name.createIndex({key:1})

The key determines the field on which to create the index, and 1 (or -1) specifies the order (ascending or descending).

Database and CRUD operations with MongoDB

Database operations

MongoDB supports various database operations, including creating, dropping, and listing databases
Some of the database operations commands are as follows -

```
1. show all database
show dbs

2. Show current database
db

3. Create or Switch Database
use exampleDb

4. Drop database
db.dropDatabase()
```

CRUD operations

CRUD operations (Create, Read, Update, Delete) are fundamental in MongoDB for interacting with data stored in collections. These operations enable users to manage data effectively by creating, reading, updating, and deleting documents in MongoDB databases.

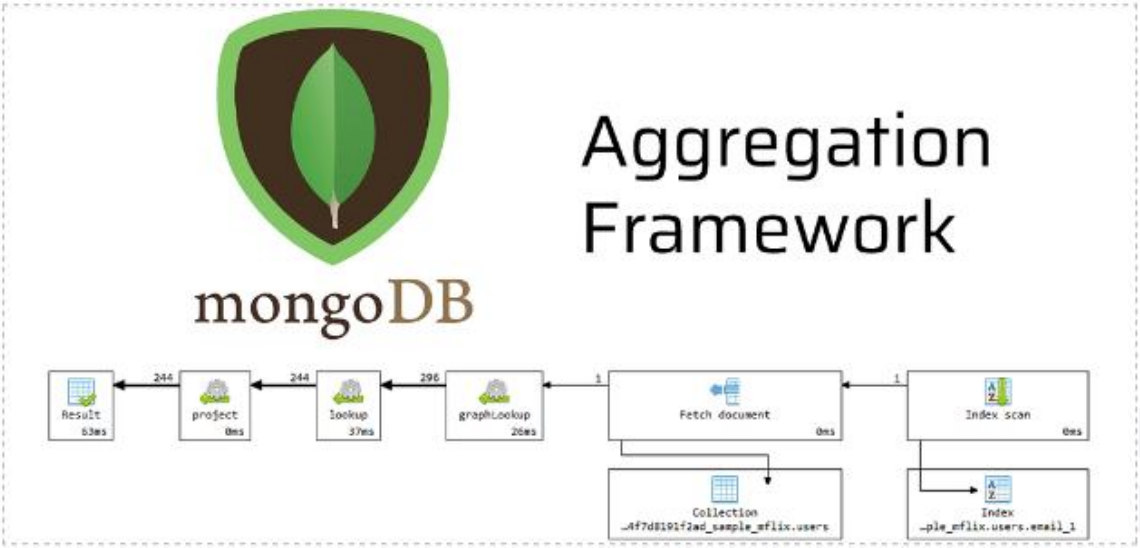
Some of the CRUD operations commands are as follows -

```
1. Create Collection
db.createCollection('myCollects')

2. show collections
show collections
```

```
3. insert one document to the collection ('my-collects')
db.my-collect.insertOne({name: 'PW', lastName: 'Skills'});

4. insert multiple documents to the collection ('my-
collects')
db.my-collects.insertMany([{name: 'Ram', age: 20},{name:
'Raju', age: 18}, {name: 'Chikku', age: 23}])

5. read all docs available in the collection
db.my-collects.find()

6. reall docs available in the collection in format/
structure manner
db.my-collects.find().pretty()

7. find specific doc in the collection
db.my-collects.find({field: value})

8. sort the docs
db.my-collects.find().sort({field: 1}).pretty() // specific
field sort asc
db.my-collects.find().sort({field: -1}).pretty()

9. Count docs
db.my-collects.find().count()
db.my-collects.find({field: value}).count()

10. udpate doc, will update the entire doc object
db.my-collects.update({field1: value1}, {field2:value2})

11. Update doc, will replace only first matching
db.my-collects.updateOne({field1: value1},
{field2:value2,field3:value3})

12. update doc, will replace all matching
db.my-collects.updateMany({field1: value1},
{field2:value2,field3:value3})

13. update doc, specific field
db.my-collects.update({field1: value1}, {$set: {field2:
value2}})

14. delete doc
db.my-collects.remove({field: value})
```

MongoDB Aggregation

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform various operations on the grouped data to return a single result.



MongoDB provides three ways to perform aggregation:

- **the aggregation pipeline** - is a framework for data aggregation modeled on the concept of data processing pipelines
- **Aggregation reference** - provides details on the available stages and expressions in the aggregation pipeline, It covers the syntax and behavior of each stage and expression
- **the map-reduce function** - is an alternative way to perform aggregation in MongoDB. It involves defining a map function that processes each document and emits one or more objects for each input document

Some of the common stages in the Aggregation Pipeline are -

- **$match:** Filters documents based on a condition.

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

- $**group**: Groups documents by a specified key and performs aggregations.

```
Syntax -
{
  $group: {
    _id: <expression>,
    <field1>: { <accumulator1>: <expression1> },
    <field2>: { <accumulator2>: <expression2> },
    ...
  }
}
```

- **$project:** Selects specific fields to include or exclude in the output documents.

```
{ $project: { <field1>: <1 or 0>, <field2>: <1 or 0>, ... }
}
```

- **$sort:** Sorts the documents.

```
Syntax -
{ $sort: { <field1>: <sort order>, <field2>: <sort order>,
... } }
```

- **$limit:** Limits the number of documents passed to the next stage.

```
Syntax -
{ $limit: <number> }
```

- **$lookup:** Performs a join with another collection.

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

- **$unwind:** Deconstructs an array field to output a document for each element of the array.

```
Syntax -
{ $unwind: <path> }
```

Some of the other available aggregation operations are -

```
1. sum i.e $sum
2. Average i.e $avg
3. Minimum i.e $minimum
4. Maximum i.e $max
5. push and addToSet i.e $push and $addToSe
6. skip i.e $skip
```

Some other important mongo operations can be of -

```
1. limit docs
db.my-collects.find().limit(2).pretty()

2. find one doc
db.my-collects.findOne({field: value })

3. chaining
db.my-collects.find().limit(2).sort({ field: 1 }).pretty()

4. increment field
db.my-collects.update({ field1: 'value1' },{$inc: { field2:
value-with-number }})

5. rename field
db.my-collects.update({field: value},
{
  $rename: {
   oldFieldName: newFieldName
  }
})

// additional exmple query -
6. add sub docs exmple
db.posts.update({ title: 'Post One' },
{
  $set: {
    comments: [
      {
        body: 'Comment One',
        user: 'Mary Williams',
        date: Date()
      },
      {
        body: 'Comment Two',
        user: 'Harry White',
        date: Date()
      }
    ]
  }

})

7. find by element in Array($elemMatch)
db.posts.find({
  comments: {
     $elemMatch: {
       user: 'Mary Williams'
       }
    }
  }
)

8. find docs with text search
db.posts.find({
  $text: {
    $search: "\"Post O\""
    }
})

9. find docs example of greater and less than
db.posts.find({ views: { $gt: 2 } }) // greater than
db.posts.find({ views: { $gte: 7 } }) // greater than equal
db.posts.find({ views: { $lt: 7 } }) // less than
db.posts.find({ views: { $lte: 7 } }) // less than equal
```

Mongoose

Mongoose is a popular ODM library for MongoDB and Node.js that simplifies data modeling and interaction with MongoDB databases. It simplifies data modeling and interaction with MongoDB databases by providing a structured approach to -

- defining schemas,
- enforcing validation rules, and
- facilitating the translation between objects in code and MongoDB documents.

Walk through of Mongoose

```
1. Installing mongoose
npm install mongoose --save

2. connecting to mongodb database
const mongoose = require('mongoose');
const uri = process.env.MONGO_URI ||
'mongodb://127.0.0.1:27017/test';
mongoose.connect(uri, function(err, res) {
  ...
});

3. Defining a schema
const schemaName = new mongoose.Schema({
  name: {
    first: String,
    last: { type: String, trim: true }
  },
  age: { type: Number, min: 0 },
  posts: [ { title: String, url: String, date: Date } ],
  updated: { type: Date, default: Date.now }
});

4. Valid SchemaTypes include - String, Number, Date,
Buffer, Boolean, Mixed, ObjectId, Array,
Decimal128,Map,Schema, UUID and BigInt

5. creating instance of the model
const User = mongoose.model('User', userSchema);
var u = new User({
  name: {
    first: 'Tony',
    last: 'Pujals'
  },
  age: 99
});
```

```
6. Create or insert document
syntax - Model.create(documents, [options], [callback])
7. Find all document
syntax - Model.find(conditions, [projection], [options],
[callback])

8. Find one document
syntax - Model.findOne(conditions, [projection], [options],
[callback])

9. Find document by ID
syntax - Model.findById(id, [projection], [options],
[callback])

10. update one document
syntax - Model.updateOne(filter, update, [options],
[callback])

11. Update document by ID
syntax - Model.findByIdAndUpdate(id, update, [options],
[callback])

12. Delete one document
syntax - Model.deleteOne(conditions, [options], [callback])

13. Delete document by ID
Model.findByIdAndDelete(id, [options], [callback])
```