

Mongo DB



What is Mongo DB

MongoDB is an open-source document database built on a horizontal scale-out architecture. Founded in 2007, MongoDB has a worldwide following in the developer community.

Instead of storing data in tables of rows or columns like SQL databases, each row in a MongoDB database is a document described in JSON,

- MongoDB = Hum**ong**ous DB
 - open-source
 - document data model
 - high performance
 - automatic scaling
 - C-P on CAP Theorem

NoSQL databases are increasingly used in [big data](#) and [real-time web](#) applications.

NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support [SQL](#)-like query languages, or sit alongside SQL databases in [polyglot persistent](#) architectures

What is Mongo DB

The “documents” in MongoDB are JSON and BSON files.

JSON is powerful for many reasons:

- It is a natural form to store data.
- It is human readable.
- Structured and unstructured information can be stored in the same document.
- You can nest JSON to store complex data objects.
- JSON has a flexible and dynamic schema, so adding fields or leaving a field out is no problem.

What is CAP Theorem

The CAP Theorem is: where C is consistency, A is availability, and P is partition tolerance, you can't have a system that has all three.

Roughly speaking:

- Consistency means that when two users access the system at the same time they should see the same data.
- Availability means up 24/7 and responds in a reasonable time.
- Partition Tolerance means if part of the system fails, it is possible for the system as a whole to continue functioning.

Benefits of Mongo DB

Mongo DB is a highly scalable, NoSQL, schema free data store.

- It stores data as JSON so you can use the same data client-side and server-side. This means you write almost no wiring code, everything just works.
- Flexible Schema – Adaptable to requirements change
- Unstructured data - you can store and retrieve unstructured data easily. Not every document in a collection needs the same fields.
- De-Normalised data - Group related content in a single document.
- Clean and simple API - Mongo is nice to talk to.

When To Use Mongo DB

MongoDB is a great choice if you need to:

- Represent data with natural clusters and variability over time or in its structure. MongoDB is schema free.
- Support rapid iterative development.
- Enable collaboration of a large number of teams
- Scale to high levels of read and write traffic.
- Scale your data repository to a massive size.
- Evolve the type of deployment as the business changes.
- Store, manage, and search data with text, geospatial, or time series dimensions.

MongoDB vs. SQL

MongoDB	SQL
Collection	Table/View
Document	Tuple/Row
PK: _id Field	PK: Any Attribute is not null and unique
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Document	Joins
Shard	Partition
Field	Column
Reference	Foreign Key

Document Example

```
{  
  "_id" :      5f1c151ebc87de769b77b56f  
  "city" :     "ADAMS",  
  "pop" :      2660,  
  "state" :     "TN",  
  "councilman" : {  
    name: "John Smith"  
    address: "13 Scenic Way"  
  }  
}
```

By default, each document contains an 12-byte unique `_id` field.

JSON and BSON

- **JavaScript Object Notation (JSON)**
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
 - name/value pairs
 - Ordered list of values

BSON is the binary encoding of JSON-like documents that MongoDB uses when storing documents in collections. It adds support for data types like Date and Binary that aren't supported in JSON.

In practice, you don't have to know much about BSON when working with MongoDB, you just need to use the native types of your language and the supplied types (e.g. ObjectId) of its driver when constructing documents and they will be mapped into the appropriate BSON type by the driver.

Getting Started with MongoDB

To install mongoDB, go to this link and click on the appropriate OS and architecture: <http://www.mongodb.org/downloads>

First, extract the files (preferably to the C drive).

Finally, create a data directory on C:\ for mongoDB to use
i.e. “md data” followed by “md data\db”

<http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/>

Note: **mongod** is mongo daemon which is a daemon process and
mongo is mongo client(similar to MySQL)

MongoDB Compass Community is GUI based tool. All can be selected to installing MongoDB Server.

Getting Started with MongoDB

Open your **mongodb/bin directory** in CLI and type **mongod** to start the database server.

To establish a connection to the server, open another command prompt window and go to the same directory, type **mongo**. This engages the mongodb shell—it's that easy!

BSON Types

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

The number can be used with the \$type operator to query by type!

```
db.addressBook.insertMany(  
  [  
    { "_id" : 1, address : "2030 Martian Way", zipCode : "90698345" },  
    { "_id" : 2, address: "156 Lunar Place", zipCode : 43339374 },  
    { "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412) },  
    { "_id" : 4, address : "55 Saturn Ring" , zipCode : NumberInt(88602117) },  
    { "_id" : 5, address : "104 Venus Drive", zipCode : ["834847278", "1893289032"]} ]  
)  
  
db.addressBook.find( { "zipCode" : { $type : 2 } } );  
or  
db.addressBook.find( { "zipCode" : { $type : "string" } } );
```

CRUD

Create, Read, Update, Delete

CRUD: Using the Shell

- To check which db you're using : db
- Show all databases : show dbs
- Creating database : use <database>
- (creates database and switches to it)
- Switch database : use <name>
- See what collections exist : show collections
- Drop database : db.dropDatabase()

Note: We do not specify any database name in the above command, because this command deletes the currently selected database

Note: Databases are not actually created until you insert data!

insert()

To insert documents into a collection/make a new collection:

```
db.<collection>.insert(<document>)
```

equivalent to

```
INSERT INTO <table> VALUES(<attribute values>);
```

```
db.books.insert({"isbn": 9780060859749, "title": "After Alice: A Novel",  
"author": "Gregory Maguire", "category": "Fiction", "year": 2016})
```

Note: File name in double quotes is optional

To Insert multiple documents, use an array

```
db.<collection>.insert([<document>,...])
```

or

```
db.<collection>.insertMany([<document>,...])
```

update()

```
db.collection.update(<query>, <update> )
```

```
db.<collection>.update(  
  {<field1>:<value1>},    //all docs in which field = value  
  {$set: {<field2>:<value2>}}, //set field to value  
  {multi:true} )          //update multiple docs
```

if true, creates a new doc when none matches search criteria.

```
UPDATE <table>  
SET <field2> = <value2>  
WHERE <field1> = <value1>;
```


update()

Remove certain fields of a single document that match the query condition

```
db.<collection>.update({<field>:<value>},  
                        { $unset: { <field>: 1}})
```

```
db.books.update({title : "Treasure Island"}, {$unset : {category:""}})
```

Remove certain fields of all documents that match the query condition

```
db.collection.update(<query>, { $unset: { <field>: 1}},  
                    {multi:true} )
```

```
db.books.update({category : "Fiction"}, {$unset : {category:""}},  
                {multi:true})
```

remove()

Remove all documents where field = value

```
db.<collection>.remove({<field>:<value>})
```

equivalent to

```
DELETE FROM <table> WHERE <field> = <value>;
```

As above, but only remove first document

```
db.<collection>.remove({<field>:<value>},justOne:true)
```

db.collection.find()

- Applied on collections.
- Get all docs: **db.<collection>.find()**
 - Returns a cursor, which is iterated over
 - shell to display the documents
 - Add **.limit(<number>)** to limit results
 - SELECT * FROM <table>;
- Get first document of the result set:
 - **db.<collection>.findOne()**
 - **db.<collection>.findOne(<query>)**
 - **db.<collection>.findOne(<query>, <projection>)**
- Get all docs that match the query condition
 - **db.<collection>.find(<query>)**
 - **db.<collection>.find(<query>,<projection>)**
 - **Note:** <query> : Optional. Specifies selection filter using [query operators](#). To return all documents in a collection, omit this parameter or pass an empty document ({}).
 - <projection> : Optional. Specifies the **fields** to return in the documents that match the query filter.

Comparison Query, Logical Query and Evaluation Query Operators

Comparison Query Operators

\$eq

\$gt

\$gte

\$in

\$lt

\$lte

\$ne

\$nin

Logical Query Operators

\$and

\$not

\$nor

\$or

Evaluation Query Operators

\$expr

\$jsonSchema

\$mod

\$regex

\$text

\$where

<https://docs.mongodb.com/manual/reference/operator/query-evaluation/>

Query Examples (\$lt, \$gt, \$or and \$and)

```
db.inventory.find( {  
  $and: [  
    { $or: [ { qty: { $lt : 10 } }, { qty : { $gt: 50 } } ] },  
    { $or: [ { sale: true }, { price : { $lt : 5 } } ] }  
  ]  
})
```

This query will select all documents where:

- the qty field value is less than 10 or greater than 50, **and**
- the sale field value is equal to true **or** the price field value is less than 5.

find() examples

db.collection.find () function returns a pointer to the collection of documents returned which is called a **cursor**.

By default, the cursor will be iterated automatically when the result of the query is returned. But one can also explicitly go through the items returned in the cursor one by one.

```
var myEmployee = db.Employee.find( { Employeeid : { $gt:2 } });

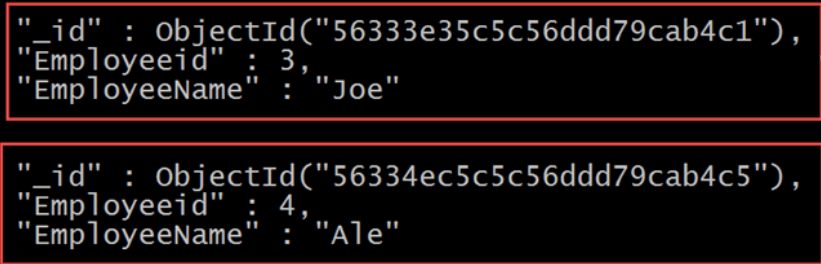
while(myEmployee.hasNext())

{

    print(tojson(myEmployee.next()));

}
```

```
var myEmployee = db.Employee.find({Employeeid:{$gt:2}});
while(myEmployee.hasNext()){ print(tojson(myEmployee.next())); }
```



```
"_id" : ObjectId("56333e35c5c56ddd79cab4c1"),
"Employeeid" : 3,
"EmployeeName" : "Joe"

"_id" : ObjectId("56334ec5c5c56ddd79cab4c5"),
"Employeeid" : 4,
"EmployeeName" : "Ale"
```

Output shows the cursor iterating through documents and printing them in json format.

db.collection.find()

The projection parameter determines which fields are returned in the matching documents. The projection parameter takes a document of the following form:

{ <field1>: <value>, <field2>: <value> ... }

Projection

<field>: <1 or true>

<field>: <0 or false>

Description

Specifies the inclusion of a field.

Specifies the exclusion of a field.

Ex.

```
db.books.find({"title":"Treasure Island"}, {title:true, category:true, _id:false})
```

```
db.books.findOne({}, {_id:0})
```

Applying db.collection.find() on Arrays

- "<field>.\$": <1 or true> With the use of the \$ array projection operator, you can specify the projection to return the first element that match the query condition on the array field; e.g. "arrayField.\$" : 1
- <field>: <array projection> Using the array projection operators **\$elemMatch**, **\$slice**, specifies the array element(s) to include, thereby excluding those elements that do not meet the expressions.

```
db.inventory.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

Query an Array by Array Length

Use the **\$size** operator to query for arrays by number of elements. For example, the following selects documents where the array tags has 3 elements.

```
db.inventory.find( { "tags": { $size: 3 } } )
```


Applying db.collection.find() on Arrays

<https://docs.mongodb.com/manual/tutorial/query-arrays/>

```
db.inventory.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

1. Query for all documents where the field `tags` value is an array with exactly two elements, "red" and "blank", in the specified order

```
db.inventory.find( { tags: ["red", "blank"] } )
```

2. Find an array that contains both the elements "red" and "blank", without regard to order or other elements in the array, use the `$all` operator:

```
db.inventory.find( { tags: { $all: ["red", "blank"] } } )
```

3. Query for all documents where `tags` is an array that contains the string "red" as one of its elements:

```
db.inventory.find( { tags: "red" } )
```

Applying db.collection.find() on Arrays

```
db.inventory.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
  { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
  { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
  { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
  { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

Query for an Array Element that Meets Multiple Criteria

Use [\\$elemMatch](#) operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.

The following example queries for documents where the dim_cm array contains at least one element that is both greater than ([\\$gt](#)) 22 and less than ([\\$lt](#)) 30:

```
db.inventory.find( { dim_cm: { $elemMatch: { $gt: 22, $lt: 30 } } } )
```

Query for an Element by the Array Index Position

Query for all documents where the second element in the array dim_cm is greater than 25:

```
db.inventory.find( { "dim_cm.1": { $gt: 25 } } )
```

MongoDB sort() method

Sample Data

```
db.studentdata.find().pretty()
```

```
{
  "_id" : ObjectId("59bf63380be1d7770c3982af"),
  "student_name" : "Steve",
  "student_id" : 2002,
  "student_age" : 22
}
{
  "_id" : ObjectId("59bf63500be1d7770c3982b0"),
  "student_name" : "Carol",
  "student_id" : 2003,
  "student_age" : 22
}
{
  "_id" : ObjectId("59bf63650be1d7770c3982b1"),
  "student_name" : "Tim",
  "student_id" : 2004,
  "student_age" : 23
}
```

MongoDB sort() method

db.collection_name.find().sort({field_key:1 or -1})

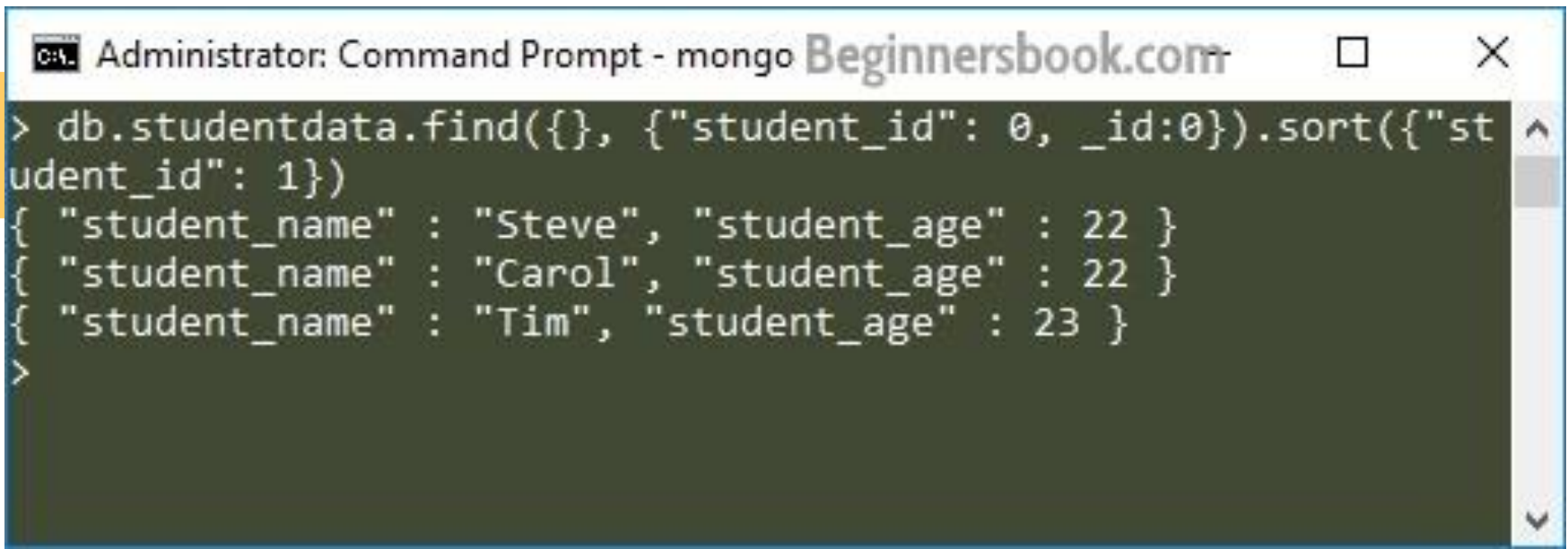
1 is for ascending order and -1 is for descending order. The default value is 1.

db.studentdata.find({}, {"student_id": 1, _id:0}).sort({"student_id": 1})

{ "student_id" : 2002 }

{ "student_id" : 2003 }

{ "student_id" : 2004 }

A screenshot of a Windows Command Prompt window titled "Administrator: Command Prompt - mongo Beginnersbook.com". The window shows a MongoDB command being executed: > db.studentdata.find({}, {"student_id": 0, _id:0}).sort({"student_id": 1}). The output displays three JSON documents sorted by student_id in ascending order: { "student_name" : "Steve", "student_age" : 22 }, { "student_name" : "Carol", "student_age" : 22 }, and { "student_name" : "Tim", "student_age" : 23 }. The prompt is ready for the next command.

```
> db.studentdata.find({}, {"student_id": 0, _id:0}).sort({"student_id": 1})
{ "student_name" : "Steve", "student_age" : 22 }
{ "student_name" : "Carol", "student_age" : 22 }
{ "student_name" : "Tim", "student_age" : 23 }
>
```

MongoDB Relationships (Embedded & Reference)

- Embedded
- Reference

One-to-Many Relationships with embedded documents

With the embedded data model, your application can retrieve the complete information with one query.

```
_id: "joe",
name: "Joe Bookreader",
addresses: [
  {
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
  },
  {
    street: "1 Some Other Street",
    city: "Boston",
    state: "MA",
    zip: "12345"
  }
]
```

One-to-Many Relationships with document references

```
{  
  name: "O'Reilly Media",  
  founded: 1980,  
  location: "CA",  
  books: [123456789, 234567890, ...]  
}
```

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data,

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

```
{  
  _id: 123456789,  
  title: "MongoDB: The Definitive Guide",  
  author: [ "Kristina Chodorow", "Mike Dirolf" ],  
  published_date: ISODate("2010-09-24"),  
  pages: 216,  
  language: "English"  
}  
  
{  
  _id: 234567890,  
  title: "50 Tips and Tricks for MongoDB Developer",  
  author: "Kristina Chodorow",  
  published_date: ISODate("2011-05-06"),  
  pages: 68,  
  language: "English"  
}
```

Indexes in MongoDB



Indexes in MongoDB

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement.

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Fundamentally, indexes in MongoDB are similar to indexes in other database systems.

MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

Indexes in MongoDB

Create an index	<code>db.collection.createIndex({indexField:type})</code> Type 1 means ascending; -1 means descending	<code>db.books.createIndex({title:1})</code>
Create a unique index	<code>db.collection.createIndex({indexField:type}, {unique:true})</code>	<code>db.books.createIndex({isbn:1},{unique:true})</code>
Create an index on multiple fields (compound index)	<code>db.collection.createIndex({indexField1:type1, indexField2:type2, ...})</code>	<code>db.books.createIndex({title:1, author:-1})</code>
Show all indexes in a collection	<code>db.collection.getIndexes()</code>	<code>db.books.getIndexes()</code>
Drop an index	<code>db.collection.dropIndex({indexField:type})</code>	<code>db.books.dropIndex({author:-1})</code>
Show index statistics	<code>db.collection.stats()</code>	<code>db.books.stats()</code>

Replication & Sharding

Replication

A *replica set* in MongoDB is a group of mongod instances that maintain the same data set.

A replica set contains several data bearing nodes and optionally one arbiter node.

Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.

Replica sets provide redundancy and high availability, and are the basis for all production deployments.

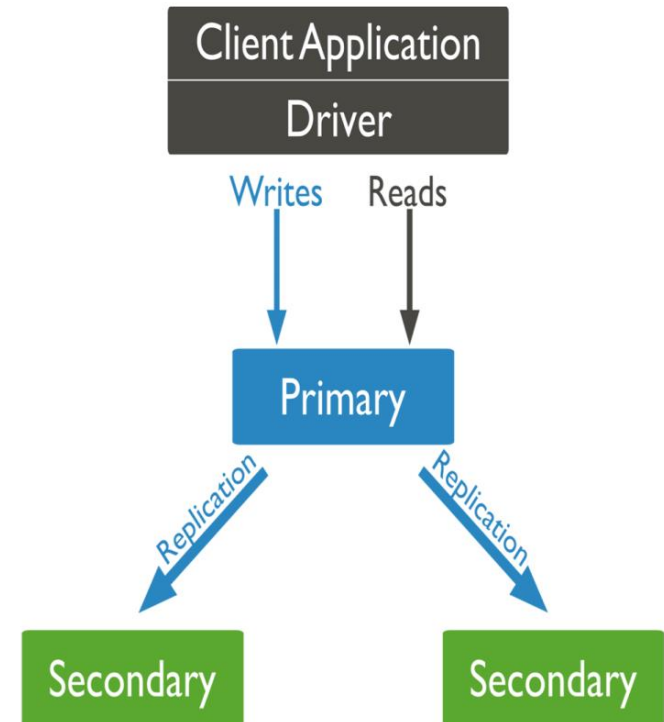


Figure 1: Diagram of default routing of reads and writes to the primary.

Replication in MongoDB

The primary node receives all write operations. A replica set can have only one primary, capable of confirming writes

The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondary's data sets reflect the primary's data set. If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.

Heartbeats. Replica set members send **heartbeats** (pings) to each other every two seconds.

If a **heartbeat** does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

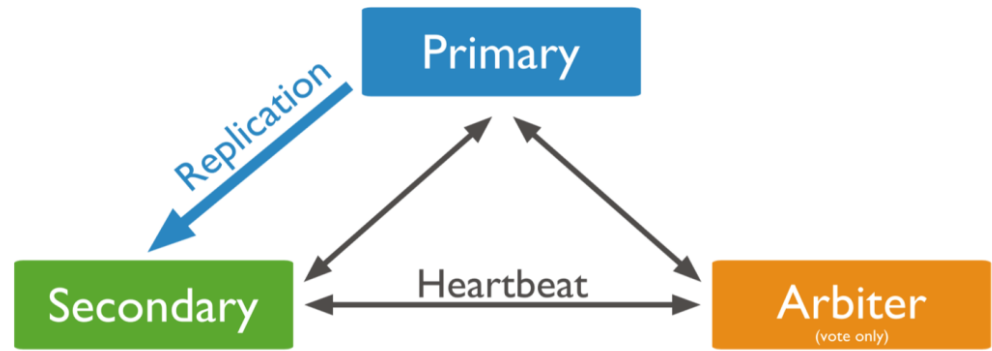
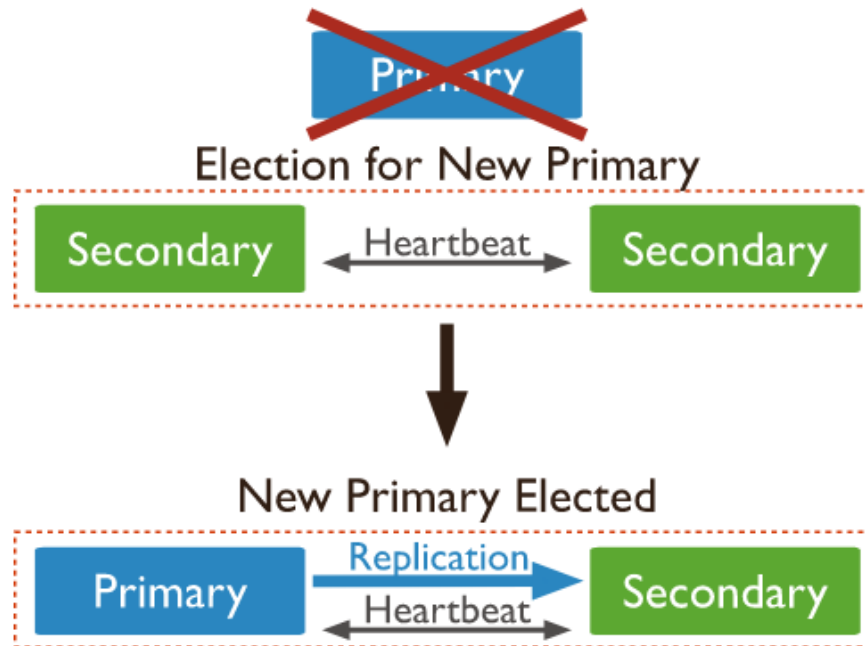


Figure 3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

For example, in the above replica set with a 2 data bearing members (the primary and a secondary), an arbiter allows the set to have an odd number of votes to break a tie:

Note: Arbiters do not maintain a data set. The purpose of an arbiter is to maintain a quorum in a replica set by responding to heartbeat and election requests by other replica set members.

Replication in MongoDB



In the above diagram, the primary node was unavailable for longer than the configured timeout and triggers the automatic failover process. One of the remaining secondary's calls for an election to select a new primary and automatically resume normal operations.

Sharding

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Database systems with large data sets or high throughput applications can challenge the capacity of a single server.

There are two methods for addressing system growth: **vertical and horizontal scaling**.

Vertical Scaling 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.

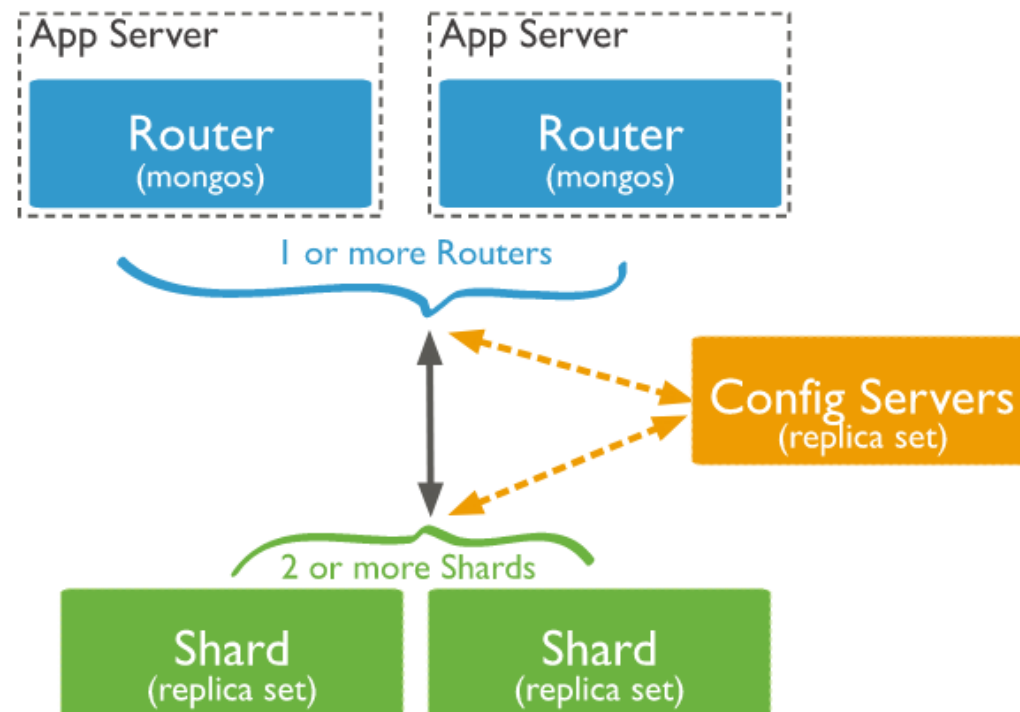
Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required.

MongoDB supports *horizontal scaling* through sharding.

Sharded Cluster

A MongoDB sharded cluster consists of the following components:

- shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.
- Mongos utility: acts as a query router, providing an interface between client applications and the sharded cluster.
- config servers: Config servers store metadata and configuration settings for the cluster.



What is Mongoose



MongooseJS is an [Object Document Mapper](#) (ODM) that makes using MongoDB easier by translating documents in a MongoDB database to objects in the program.

Besides MongooseJS there are several other ODM's that have been developed for MongoDB including [Doctrine](#), [MongoLink](#) , and [Mandango](#).

The main advantage of using Mongoose versus native MongoDB are:

MongooseJS provides an abstraction layer on top of MongoDB that eliminates the need to use named collections.

Thank you