



Introduction to MongoDB



22 march 2022 by
Bechir Brahem



Course outline:

- I. Database concepts
- II. Introducing Mongoddb
- III. CRUD operations
- IV. Replication & Sharding
- V. MongoDB with python

I. Database concepts




1. Relational databases

They are databases that follow the relational model to represent data.

The relational model proposed in 1970 stores the data in tables for example student table. And each table has columns for example the student table has name, age, student id... as its columns

LOGICAL TABLE STRUCTURE

MATERIAL	CATEGORY	REVENUE (EUR)
GLOVE	SPORT	500
CAP	SPORT	200
CHAIR	HOUSING	450
TABLE	HOUSING	100
PROTEIN	SPORT	600



To query over a relational database we use SQL “structured query language” which is a standardized programming language used to to:

- Read
- Write
- Update
- Delete
- Add users
- Import and export data
- ...

```
CREATE TABLE Country
(
  Pk_Country_Id INT IDENTITY PRIMARY KEY,
  Name VARCHAR(100),
  Officiallang VARCHAR(100),
  Size INT(11),
);

CREATE TABLE UNrepresentative
(
  Pk_UNrepresentative_Id INT PRIMARY KEY,
  Name VARCHAR(100),
  Gender VARCHAR(100),
  Fk_Country_Id INT UNIQUE FOREIGN KEY REFERENCES Country(Pk_Country_Id)
);

INSERT INTO Country ('Name','Officiallang','Size')
VALUES ('Nigeria','English',923,768);

INSERT INTO Country ('Name','Officiallang','Size')
VALUES ('Ghana','English',238,535);

INSERT INTO Country ('Name','Officiallang','Size')
VALUES ('South Africa','English',1,219,912);

INSERT INTO UNrepresentative
('Pk_Unrepresentative_Id','Name','Gender','Fk_Country_Id')
VALUES (51,'Abubakar Ahmad','Male',1);

INSERT INTO UNrepresentative
('Pk_Unrepresentative_Id','Name','Gender','Fk_Country_Id')
VALUES (52,'Joseph Nkrumah','Male',2);

INSERT INTO UNrepresentative
('Pk_Unrepresentative_Id','Name','Gender','Fk_Country_Id')
VALUES (53,'Lauren Zuma','Female',3);

SELECT * FROM Country
SELECT * FROM UNrepresentative;
```



We have customer table and orders table

- A customer can have many orders
- An order only has one customer

This relationship is called “**one-to-many**”

To model it we store the customers ID in each order

Customers table

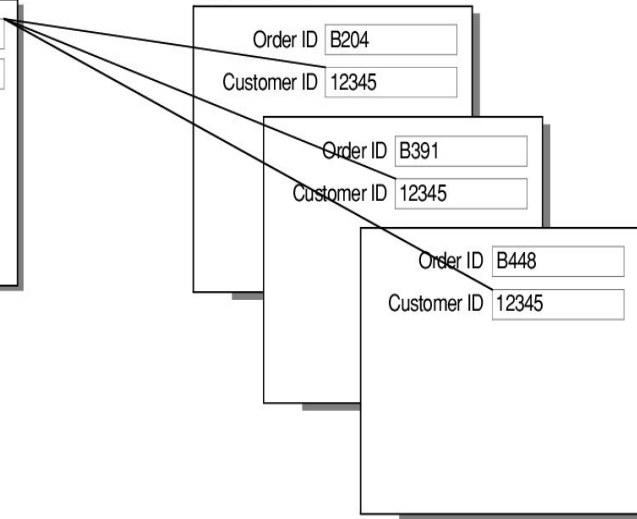
Customer ID	12345
Name	Tang

Orders table

Order ID	B204
Customer ID	12345

Order ID	B391
Customer ID	12345

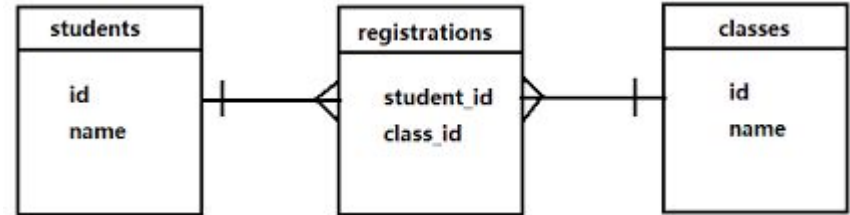
Order ID	B448
Customer ID	12345



Many to many relationship

- A student can have many classes
- A class can have many students

To represent this type of relationship we must create a third table. We call this table the join table. This table contains the `student_id` and `class_id`.



```
MariaDB [esgitech]> select * from students  
-> ;
```

id	name
1	Sara
2	Ahmed

2 rows in set (0.000 sec)

```
MariaDB [esgitech]> select * from classes;
```

id	name
15	math
20	anglais
30	mongodb
35	mysql

```
MariaDB [esgitech]> select * from registrations;
```

student_id	class_id
1	15
1	35
2	30
2	20
2	15

```
MariaDB [esgitech]> SELECT name FROM classes
```

```
-> JOIN registrations ON registrations.class_id=classes.id  
-> WHERE student_id=2;
```

name
mongodb
anglais
math

3 rows in set (0.001 sec)

Examples of RDBMS

RDBMS: relational database management system

MySQL, PostgreSQL, MsSQL (microsoft sql server), Oracle database server





Problems with RDBMS

- **RDBMS Are Not Designed To Handle Change**
- Querying data with JOIN can be very slow
- Scaling is very difficult
- Database design is very limited

Benefits of RDBMS

- Have been around for decades: have a big community and are very stable
- It is cheaper to provision RDBMS databases on the cloud
- High consistency
- High security



Database properties

ACID principles

Atomicity means that you guarantee that either all of the transaction succeeds or none of it does. You don't get part of it succeeding and part of it not. If one part of the transaction fails, the whole transaction fails. With atomicity, it's either "all or nothing".

Consistency This ensures that you guarantee that all data will be consistent. All data will be valid according to all defined rules, including any constraints, cascades, and triggers that have been applied on the database.

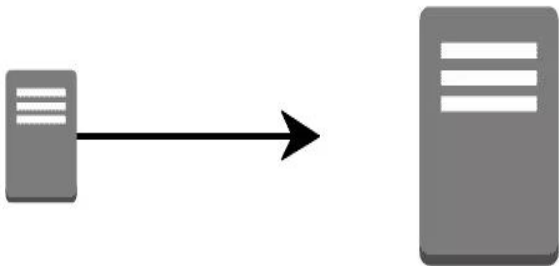
Isolation Guarantees that all transactions will occur in isolation. No transaction will be affected by any other transaction. So a transaction cannot read data from any other transaction that has not yet completed.

Durability means that, once a transaction is committed, it will remain in the system – even if there's a system crash immediately following the transaction. Any changes from the transaction must be stored permanently. If the system tells the user that the transaction has succeeded, the transaction must have, in fact, succeeded.



Vertical Scalability


Scaling vertically (scaling up) means adding more resources such as CPU, RAM... to the instance



Horizontal Scalability

Scaling horizontally (or scaling out) means adding more computer instances and distributing tasks across these different instances





Scaling vertically is easier to do and to maintain. Because it does not add any complexity to the existing architecture. But at some point we can no longer scale vertically because there is no existing cpu that can handle that workload. Or it becomes very expensive to buy the latest cpu,ram,ssd components.

Example: on average

- Buying 8GB GPU is around 200\$ to 500\$
- Buying 16GB GPU is around 1200\$ to 1700\$

Not only price is the limit but sometimes there doesn't exist 64 GB GPU

The solution is to distribute the work across many different computers: **scaling horizontally**.



What is NoSQL?

NoSQL are databases that don't use SQL and are not relational databases. These databases are used generally to focus on speed, scalability, and availability and dropping strict consistency rules.

These databases are very popular and are used heavily in big data applications, web application, social networks, financial services... and they have very varied specifications



Types of databases

Must use SQL

- RDBMS: mysql, postgresql...

NoSQL

- Key value store: redis, dynamoDB(AWS)...
- Columnar database: cassandra, hbase...
- Document database: mongodb...
- Time Series database: influxdb...

II. Introducing MongoDB






What is mongoDB

MongoDB is an open-source document database built for horizontal scalability that uses a flexible schema for storing data. Founded in 2007, MongoDB has a worldwide following in the developer community.

mongodB is very prominent for its ease of use, performance, and especially scalability.



Instead of storing data in tables of rows or columns like SQL databases, each record in a MongoDB database is stored as a document in a collection.

A document is represented in JSON format.

A document can contain arrays, subdocuments, arrays of subdocuments...

```
{
  "_id": 1,
  "name": {
    "first": "Ada",
    "last": "Lovelace"
  },
  "title": "The First Programmer",
  "interests": ["mathematics", "programming"]
}
```



Reasons to choose MongoDB

1. Flexibility: mongodb is very flexible. It does not have a strict schema. Documents have subdocuments and arrays. This makes developing projects in mongodb easier as it requires less planning and shortens the time to market
2. Highly scalable
3. High availability
4. Ease of use
5. Huge community and support



Setup mongoDB

On windows:

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

On Mac:

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>

On linux:

Figure it out yourself



Connect to mongodb

With mongosh

```
mongosh "mongodb+srv://cluster0.bfp8u.mongodb.net" --username xx|
```

With mongo

```
mongo "mongodb+srv://cluster0.bfp8u.mongodb.net" --username xx
```



This is the mongo shell where we enter our commands and the mongo server will return responses

```
Atlas atlas-wd3gqf-shard-0 [primary] test>
```

Mongodb can have many databases that are separated from each other.


Each database contains many collections. And every collection has many documents.

In RDBMS, a collection is the equivalent of a table. And a document is the equivalent of a record in the table



“show dbs” command returns a list of all available databases created in the server

```
Atlas atlas-wd3gqf-shard-0 [primary] test> show dbs
sample_airbnb      54.6 MB
sample_analytics   9.57 MB
sample_geospatial 1.43 MB
sample_guides      41 kB
sample_mflix       49.6 MB
sample_restaurants 7.28 MB
sample_supplies    1.17 MB
sample_training    54.6 MB
sample_weatherdata 3.03 MB
admin              307 kB
local              1.47 GB
Atlas atlas-wd3gqf-shard-0 [primary] test>
```

- 
- “Use <dbname>” switches to that db
 - “show collections” prints all available collections in <dbname>

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_airbnb> use sample_training
switched to db sample_training
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> show collections
companies
grades
inspections
posts
routes
trips
zips
Atlas atlas-wd3gqf-shard-0 [primary] sample_training>
```




To view one document in the collections we use:

“db.<collection name>.findOne()”

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.findOne()  
{  
  _id: ObjectId("5c8eccc1caa187d17ca6ed16"),  
  city: 'ALPINE',  
  zip: '35014',  
  loc: { y: 33.331165, x: 86.208934 },  
  pop: 3062,  
  state: 'AL'  
}
```



To represent the data, internally, mongodb uses a format called BSON (binary JSON).

Parsing JSON text is slow so for this reason mongodb converts the JSON data to binary for more performance gains. BSON is not humanly readable. Also BSON offers the possibility to add more types such as dates, geospatial data...

We can import and export mongodb data in json or in bson:

For BSON we use: mongodump and mongorestore

For JSON we use: mongoexport and mongoimport

III. CRUD operations



CRUD operations

Create, Read, Update, Delete

`“db.<collectionName>.find()”`

Returns an array of all available documents.

In the shell it displays only a part of the result (20 documents) we can see more using the command
“it”

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find()
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed16"),
    city: 'ALPINE',
    zip: '35014',
    loc: { y: 33.331165, x: 86.208934 },
    pop: 3062,
    state: 'AL'
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed17"),
    city: 'BESSEMER',
    zip: '35020',
    loc: { y: 33.409002, x: 86.947547 },
    pop: 40549,
    state: 'AL'
  },
  {
    id: ObjectId("5c8eccc1caa187d17ca6ed18"),
```

Read



To search using a condition on a specific field we use this syntax

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find({state:"AL"})
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed16"),
    city: 'ALPINE',
    zip: '35014',
    loc: { y: 33.331165, x: 86.208934 },
    pop: 3062,
    state: 'AL'
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed17"),
    city: 'BESSEMER',
    zip: '35020',
    loc: { y: 33.409002, x: 86.947547 },
    pop: 40549,
    state: 'AL'
  },
  {

```

Read



We can have multiple conditions

This query searches for zips that have: state="AL" and pop=2369

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find({
... state:"AL",
... pop:2369
... })
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed29"),
    city: 'CLEVELAND',
    zip: '35049',
    loc: { y: 33.992106, x: 86.559355 },
    pop: 2369,
    state: 'AL'
  }
]
```

Read



We can also search using comparison operators:

\$gt for greater than, \$lt for less than, \$gte for greater than or equal, \$lte for less than or equal, \$ne for not equal

```
Atlas atlas-wd3ggf-shard-0 [primary] sample_training> db.zips.find({ pop: { "$gt": 110000 } })
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca7044d"),
    city: 'CHICAGO',
    zip: '60623',
    loc: { y: 41.849015, x: 87.7157 },
    pop: 112047,
    state: 'IL'
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca7307f"),
    city: 'BROOKLYN',
    zip: '11226',
    loc: { y: 40.646694, x: 73.956985 },
    pop: 111396,
    state: 'NY'
  }
]
```

Read



We can also use the “.count()” to know how many documents adhere to that condition.

```
Atlas atlas-wd3ggf-shard-0 [primary] sample_training> db.zips.find({ pop: 0 }).count()  
67
```

```
Atlas atlas-wd3ggf-shard-0 [primary] sample_training> db.zips.find(  
... pop:{"$gt":100000}  
... }).count()  
4
```


Read



We can also use logic operators like:
and, or, not, nor

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find({
...   "$or":[
...     {state:"NY", pop: {"$gt":100000}},
...     {state:"IL", pop: 112047}
...   ]
... })
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca7044d"),
    city: 'CHICAGO',
    zip: '60623',
    loc: { y: 41.849015, x: 87.7157 },
    pop: 112047,
    state: 'IL'
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca72fa0"),
    city: 'NEW YORK',
    zip: '10021',
    loc: { y: 40.768476, x: 73.958805 },
    pop: 106564,
    state: 'NY',
    'capital?': false
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca72fa5"),
    city: 'NEW YORK',
    zip: '10025',
    loc: { y: 40.797466, x: 73.968312 },
    pop: 100027,
    state: 'NY',
    'capital?': false
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca7307f"),
    city: 'BROOKLYN',
    zip: '11226',
    loc: { y: 40.646694, x: 73.956985 },
    pop: 111396,
    state: 'NY'
  }
]
```

Create



To insert a document in a collection we use:
“db.<collectionName>.insert(<document>)

The document schema is not important. We can have any field name we want.

A document can have subdocuments like grades in this example

If the collection name does not exist. A collection with that name will be created and the document will be inserted in it

```
Atlas atlas-wd3ggf-shard-0 [primary] esgitechDB> db.students.insert({
... name:"fatima",
... student_id:99,
... grades:{
.... english:11,
.... french:12,
.... math:7,
.... physics:0,
.... }
... }
... )
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("62386636074da445e63e6a78") }
}
```

Create



Insert operation automatically create a field name `_id` that is a unique reference to that document. The `_id` field has a type of `ObjectId`

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.find({
... student_id:99})
[
  {
    _id: ObjectId("62386636074da445e63e6a78"),
    name: 'fatima',
    student_id: 99,
    grades: { english: 11, french: 12, math: 7, physics: 0 }
  }
]
```

Update

Update query has the following structure:

```
db.<collectionName>.updateMany({documents to be updated},{ "$set":{<field to update>:value}})
```

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.updateMany({name:"fatima"},{"$set":{"grades.english":20}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.find({student_id:99})
[
  {
    _id: ObjectId("62386636074da445e63e6a78"),
    name: 'fatima',
    student_id: 99,
    grades: { english: 20, french: 12, math: 7, physics: 0 }
  }
]
```

Update



There are various operators that are used to update documents:

- `$set`: specifies the exact value
- `$inc`: increases the existing field by a value
- `$mul`: multiplies
- `$min`: updates the field to value specified if it is less than this value
- `$max`: same as min but for greater values



Complex example for update

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.updateMany(
... {"$or":[
...   {"grades.programming":{"$lte":10}},
...   {"grades.programming":null}
... ]},
... {"$set":{"passing:false}}
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 3,
  upsertedCount: 0
}
```

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.find()  
[  
  {  
    _id: ObjectId("62386580074da445e63e6a75"),  
    name: 'sara',  
    student_id: 15,  
    grades: { english: 20, french: 20, programming: 19, databases: 14 }  
  },  
  {  
    _id: ObjectId("623865b7074da445e63e6a76"),  
    name: 'james',  
    student_id: 42,  
    grades: { english: 12, french: 2, programming: 9, databases: 140 },  
    passing: false  
  },  
  {  
    _id: ObjectId("623865de074da445e63e6a77"),  
    name: 'ahmed',  
    student_id: 420,  
    grades: { english: 18, french: 12, programming: 9, databases: 7 },  
    passing: false  
  },  
  {  
    _id: ObjectId("62386636074da445e63e6a78"),  
    name: 'fatima',  
    student_id: 99,  
    grades: { english: 20, french: 12, math: 7, physics: 0 },  
    passing: false  
  }  
]
```

Delete



db.<collectionName>.delete({conditions of documents to delete})

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.findOne({name:"ahmed"})
{
  _id: ObjectId("623865de074da445e63e6a77"),
  name: 'ahmed',
  student_id: 420,
  grades: { english: 18, french: 12, programming: 9, databases: 7 },
  passing: false
}
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.deleteMany({name:"ahmed"})
{ acknowledged: true, deletedCount: 1 }
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db.students.findOne({name:"ahmed"})
null
```




In the next part we will see how to query over arrays and subdocuments:

To reference a field in a subdocument we use the “.” (dot) for example to find the grade english of a student we write “grades.english”

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find({"loc.y":33.992106})
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed29"),
    city: 'CLEVELAND',
    zip: '35049',
    loc: { y: 33.992106, x: 86.559355 },
    pop: 2369,
    state: 'AL'
  }
]
```

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.grades.find()  
[  
  {  
    _id: ObjectId("56d5f7eb604eb380b0d8d8ce"),  
    student_id: 0,  
    scores: [  
      { type: 'exam', score: 78.40446309504266 },  
      { type: 'quiz', score: 73.36224783231339 },  
      { type: 'homework', score: 46.980982486720535 },  
      { type: 'homework', score: 76.67556138656222 }  
    ],  
    class_id: 339  
  },  
]
```


```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.grades.find({"scores.0.score":{"$gt":99.998}})  
[  
  {  
    _id: ObjectId("56d5f7ed604eb380b0d92fce"),  
    student_id: 2227,  
    scores: [  
      { type: 'exam', score: 99.99859239806152 },  
      { type: 'quiz', score: 25.01425209123588 },  
      { type: 'homework', score: 80.09012914970444 },  
      { type: 'homework', score: 96.97989358475645 }  
    ],  
    class_id: 408  
  },  
]
```

To add an element in an array we use the \$push operator

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.grades.update(
...  {"scores.0.score":{"$gt":99.998}},
...  {"$push":{"scores":{"type":"tp", score:100}}})
DeprecationWarning: Collection.update() is deprecated. Use updateOne, updateMany, or bulkWrite.
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.grades.find({"scores.0.score":{"$gt":99.998}})
[
  {
    _id: ObjectId("56d5f7ed604eb380b0d92fce"),
    student_id: 2227,
    scores: [
      { type: 'exam', score: 99.99859239806152 },
      { type: 'quiz', score: 25.01425209123588 },
      { type: 'homework', score: 80.09012914970444 },
      { type: 'homework', score: 96.97989358475645 },
      { type: 'tp', score: 100 }
    ],
    class_id: 408
  }
]
```

Consider we have this collections about courses and their prerequisites

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db. formations.find()  
[  
  {  
    _id: ObjectId("62388ab6b57ce2746e4d2798"),  
    name: 'iot',  
    prerequisites: [ 'math', 'electronique', 'C', 'C++', 'python', 'networking' ]  
  },  
  {  
    _id: ObjectId("62388ae9b57ce2746e4d2799"),  
    name: 'deep learning',  
    prerequisites: [ 'math', 'machine learning', 'programming', 'algorithms' ]  
  },  
  {  
    _id: ObjectId("62388afdb57ce2746e4d279a"),  
    name: 'python',  
    prerequisites: []  
  },  
  {  
    _id: ObjectId("62388b1cb57ce2746e4d279b"),  
    name: 'MongoDB',  
    prerequisites: []  
  },  
  {  
    _id: ObjectId("62388e42b57ce2746e4d279d"),  
    name: 'blockchain',  
    prerequisites: [ 'programming', 'linux', 'networking', 'math' ]  
  }  
]
```



To find all documents that contain both “networking” and “math” in their array prerequisites we use the following query

```
Atlas atlas-wd3gqf-shard-0 [primary] esgitechDB> db. formations.find({prerequisites:{"$all":["networking","math"]}})
[
  {
    _id: ObjectId("62388ab6b57ce2746e4d2798"),
    name: 'iot',
    prerequisites: [ 'math', 'electronique', 'C', 'C++', 'python', 'networking' ]
  },
  {
    _id: ObjectId("62388e42b57ce2746e4d279d"),
    name: 'blockchain',
    prerequisites: [ 'programming', 'linux', 'networking', 'math' ]
  }
]
```

Projection

We can select the fields that we want to receive when we use the find command

The `_id` field is included by default.
We can add "`_id:0`" to hide it

We can also choose fields that we want to hide by attributing 0 to them

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find(
... {state:"NY"},
... {city:1,pop:1})
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca72f89"),
    city: 'FISHERS ISLAND',
    pop: 329
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca72f8a"),
    city: 'NEW YORK',
    pop: 18913
  },
  {

```

LIMIT

To only show a specified number of documents we use the .limit

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find().limit(2)
[
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed16"),
    city: 'ALPINE',
    zip: '35014',
    loc: { y: 33.331165, x: 86.208934 },
    pop: 3062,
    state: 'AL'
  },
  {
    _id: ObjectId("5c8eccc1caa187d17ca6ed17"),
    city: 'BESSEMER',
    zip: '35020',
    loc: { y: 33.409002, x: 86.947547 },
    pop: 40549,
    state: 'AL'
  }
]
```


SORT



To sort documents we use the
“.sort()” command

Inside the sort() we specify what
are the fields we are going to sort
on

1 means ascending sort
0 means descending sort

```
Atlas atlas-wd3gqf-shard-0 [primary] sample_training> db.zips.find({},
... {city:1,pop:1,_id:0}
... ).sort({pop:-1})
[
  { city: 'CHICAGO', pop: 112047 },
  { city: 'BROOKLYN', pop: 111396 },
  { city: 'NEW YORK', pop: 106564 },
  { city: 'NEW YORK', pop: 100027 },
  { city: 'BELL GARDENS', pop: 99568 },
  { city: 'CHICAGO', pop: 98612 },
  { city: 'LOS ANGELES', pop: 96074 },
  { city: 'CHICAGO', pop: 95971 },
  { city: 'CHICAGO', pop: 94317 },
  { city: 'NORWALK', pop: 94188 },
  { city: 'CHICAGO', pop: 92005 },
  { city: 'CHICAGO', pop: 91814 },
  { city: 'CHICAGO', pop: 89762 },
  { city: 'CHICAGO', pop: 88377 },
  { city: 'JACKSON HEIGHTS', pop: 88241 },
  { city: 'ARLETA', pop: 88114 },
  { city: 'BROOKLYN', pop: 87079 },
  { city: 'SOUTH GATE', pop: 87026 },
  { city: 'RIDGEWOOD', pop: 85732 },
```

IV. Replication & Sharding

REPLICATION



to achieve high availability we replicate the primary server's data on other servers that acts as secondary nodes to it.

for a cluster of three nodes that are in the same replicaset one node is primary which receives write operations then it transmits them to the other replicas.

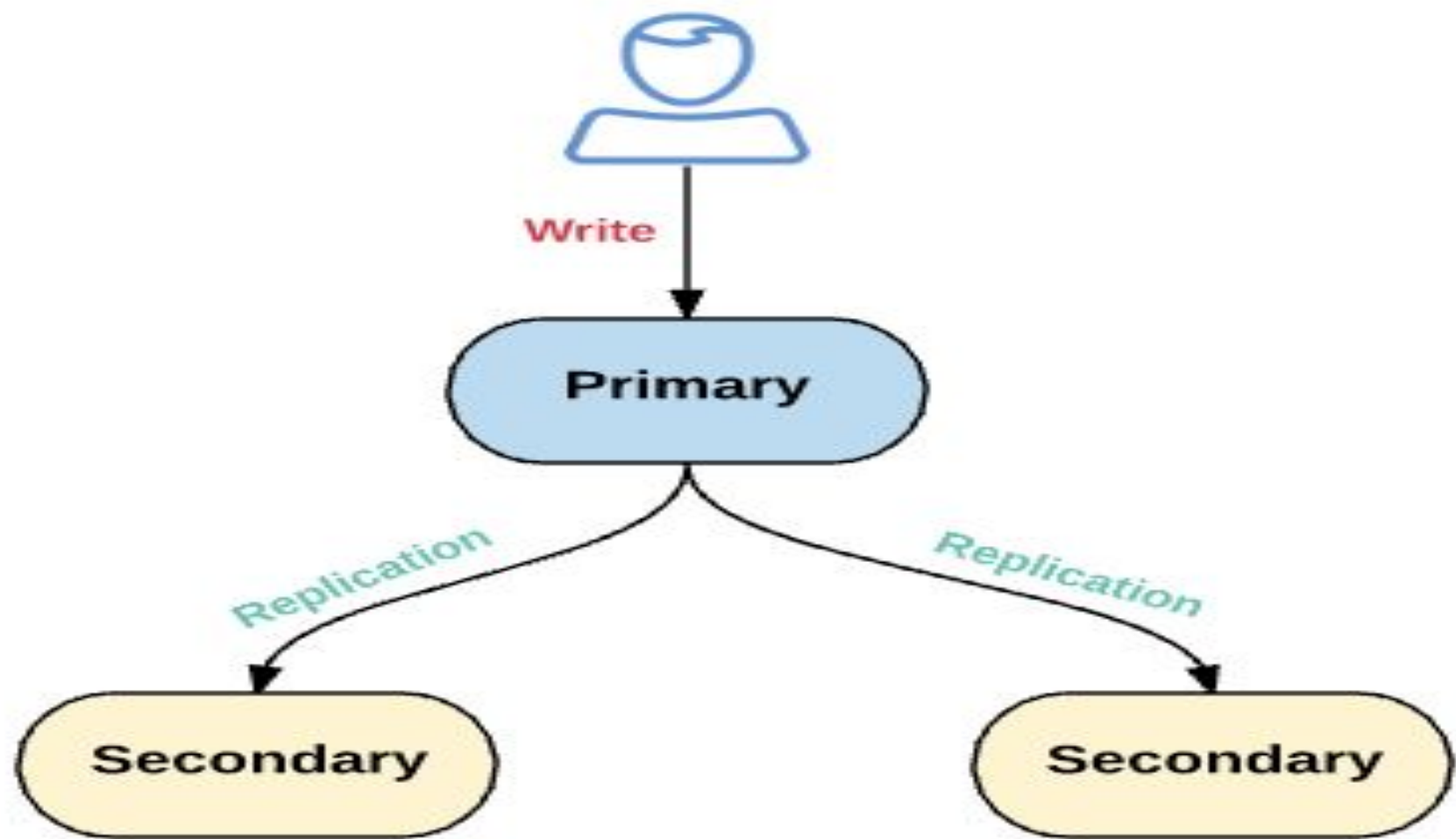
when a server goes down the secondary servers perform an election to see who will become the new primary server.

write operations will not be executed until the election is completed

for each server we can set:

priority: which represent how likely this server will become primary

votes: number of votes it has (0 will make it an arbiter)



Sharding

when the workload on the database grows we can only scale it vertically up to a certain limit. scaling vertically is very expensive so another alternative is to scale horizontally.

sharding means splitting the data into many chunks according to some index for example

$0 \leq \text{price} < 5$ and $5 \leq \text{price} < 10$ and $10 \leq \text{price} < 15$. after this we deploy each chunk on a different server called a shard. finally when we receive read query that has $\text{price} = 3$ we redirect it to the first chunk and so on.

sharding can save a lot of workload on a single node server and allows for better scalability

for mongodb the sharding is implemented with three components:

- the router called mongos which redirects queries to the specific server.
- the config servers which contains information about the keys and users and other metadata
- the shards which are mongod processes
and every server (except mongos) will be replicated for high availability in its own replicaset

