NOSQL

NoSQL

NoSQL is a non-relational database management systems, different from traditional relational database management systems in some significant ways.

It is designed for distributed data stores where very large scale of data storing needs.

For example Google or Facebook which collects terabits of data every day for their users.

These type of data storing may not require fixed schema, avoid join operations and typically scale horizontally.

RDBMS

- Structured and organized data
- Structured query language (SQL)
- Data and its relationships are stored in separate tables.
- Data Manipulation Language, Data Definition Language
- Tight Consistency

Impedance Mismatch

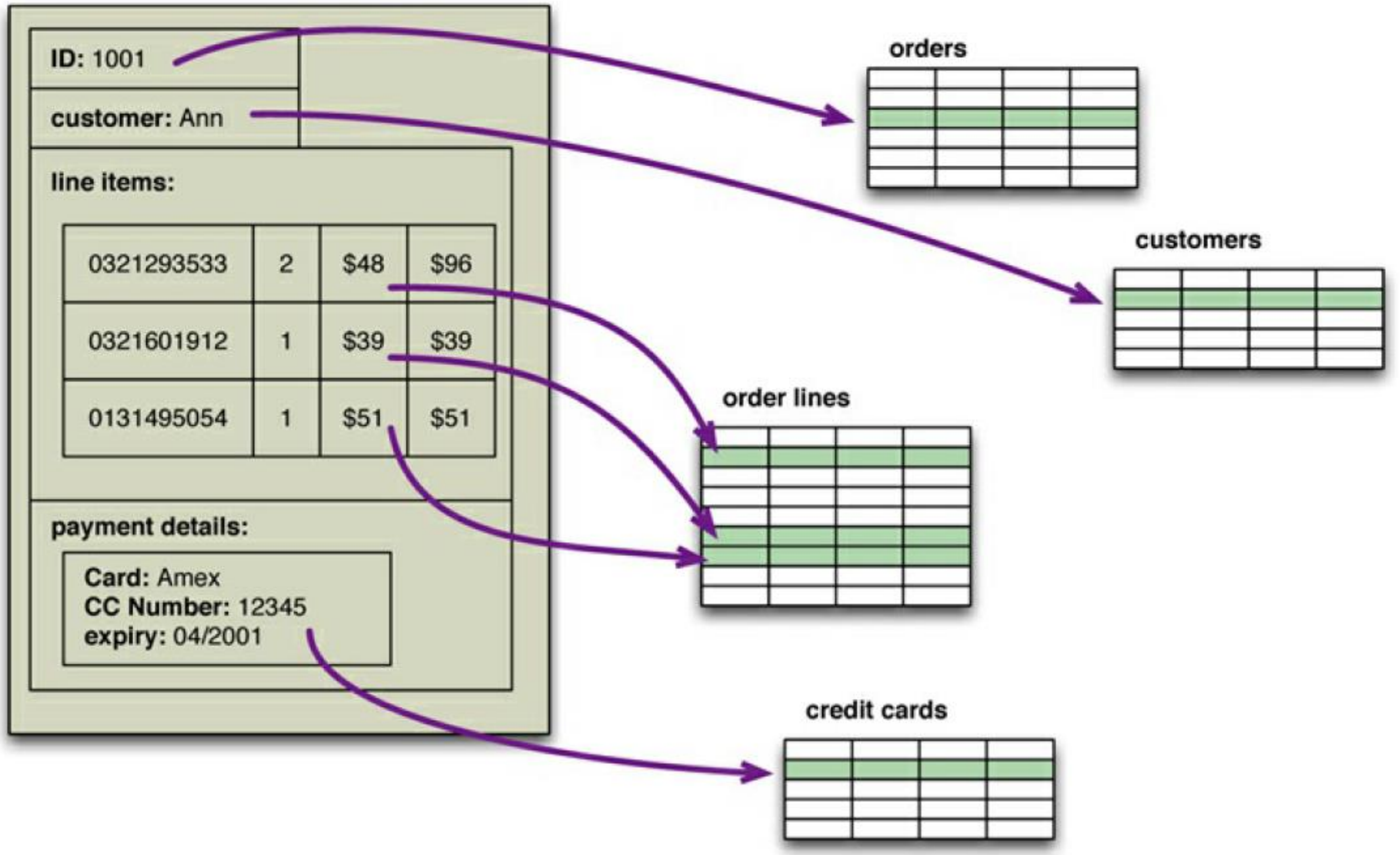


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

The emergence of NoSQL

- Stands for Not Only SQL
- No predefined schema
- Key-Value pair storage, Column Store, Document Store, Graph databases
- Eventual consistency rather ACID property
- Unstructured and unpredictable data
- CAP Theorem

SQL vs NoSQL

1. SQL databases are relational, NoSQL are non-relational.
2. SQL databases use structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data.
3. SQL databases are vertically scalable, NoSQL databases are horizontally scalable.
4. SQL databases are table based, while NoSQL databases are document, key-value, graph or wide-column stores.
5. SQL databases are better for multi-row transactions, NoSQL are better for unstructured data like documents or JSON.

NoSQL Categories

- There are four general types (most common categories) of NoSQL databases.
- Each of these categories has its own specific attributes and limitations.
- There is not a single solutions which is better than all the others, however there are some databases that are better to solve specific problems.
    - Key-value stores
    - Column-oriented
    - Document oriented
    - Graph database

Key-value stores

- Key-value stores are most basic types of NoSQL databases.
- Designed to handle huge amounts of data (Based on Amazon’s Dynamo paper).
- Key value stores allow developer to store schema-less data.
- In the key-value storage, database stores data as hash table where each key is unique and the value can be string, JSON etc.
- For example a key-value pair might consist of a key like "Name" that is associated with a value like "Robin".
- Key-Values stores would work well for shopping cart contents.
- Example of Key-value store DataBase : Redis, Dynamo, Riak. etc.

Column-oriented databases

- Most databases have a row as a unit of storage which, in particular, helps write performance.
- However, there are many scenarios where writes are rare, but we often need to read a few columns of many rows at once.
- In this situation, it’s better to store groups of columns for all rows as the basic storage unit—which is why these databases are called column stores.
- Example of Column-oriented databases : BigTable, Hbase, Cassandra etc.

To get a particular customer’s name from Figure 2.5 we could do something like get('1234', 'name').

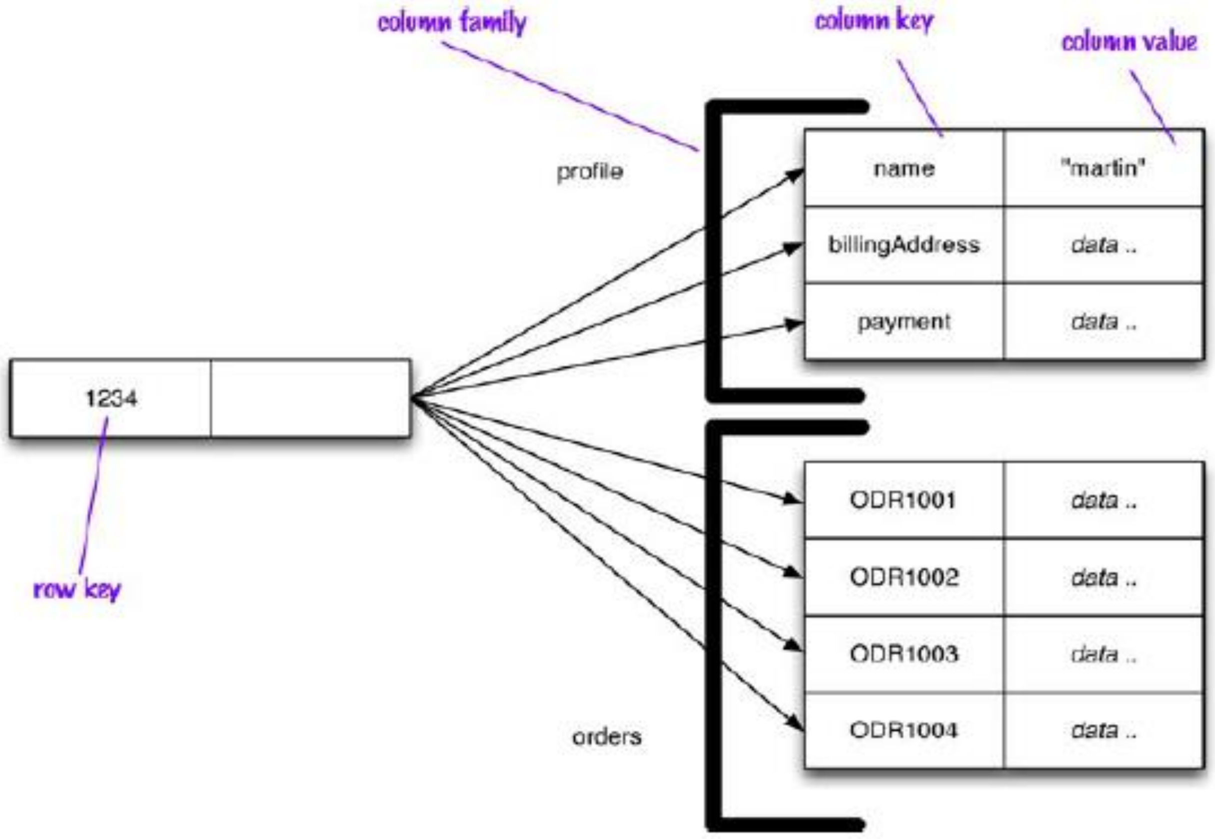


Figure 2.5. Representing customer information in a column-family structure

Document Oriented databases

- A collection of documents
- Data in this model is stored inside documents.
- A document is a key value collection where the key allows access to its value.
- Documents are not typically forced to have a schema and therefore are flexible and easy to change.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- Example of Document Oriented databases : MongoDB, CouchDB etc.

Aggregate data models

- A data model is the model through which we perceive and manipulate our data.

- For people using a database, the data model describe how we interact with the data in the database.

- Data model : the model by which the database organizes data.

- The dominant data model of the last couple of decades is the relational data model, which is best visualized as a set of tables.

- Each table has rows, with each row representing some entity of interest.

- We describe this entity through columns, each having a single value.

- A column may refer to another row in the different table, which constitutes a relationship between those entities.

Aggregates

- The relational model takes the information that we want to store and divides it into tuples (rows).
- A tuple is a limited data structure: It captures a set of values, so we cannot nest one tuple within another to get nested records, nor can we put a list of values or tuples within another.
- Aggregate orientation takes a different approach.
- We often want to operate on data in units that have a more complex structure than a set of tuples.
- key-value, document, and column-family databases all make use of this more complex record

- However, there is no common term for this complex record; here (we) use the term “ **aggregate**.”

- **aggregate is a collection of related objects that we wish** to treat as a unit.

- In particular, it is a unit for data manipulation and management of consistency.

- Aggregates are also often easier for application programmers to work with, since they often manipulate data through aggregate structures.

Example of Relations and Aggregates

- Consider an example of building an e-commerce website;
- we are going to be selling items directly to customers over the web, and we will have to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data.
- We can use this scenario to model the data using a relation data store as well as NoSQL data stores and talk about their pros and cons.

For a relational database, we might start with a data model shown in Figure 2.1.

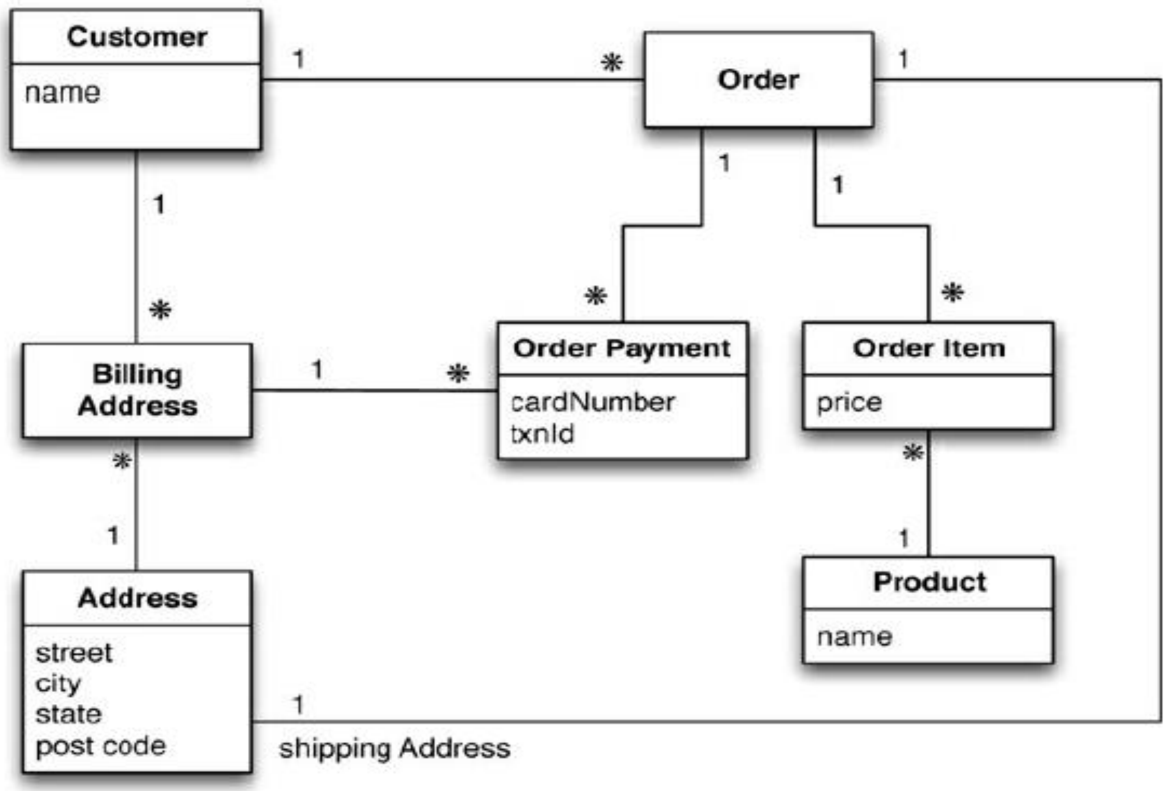
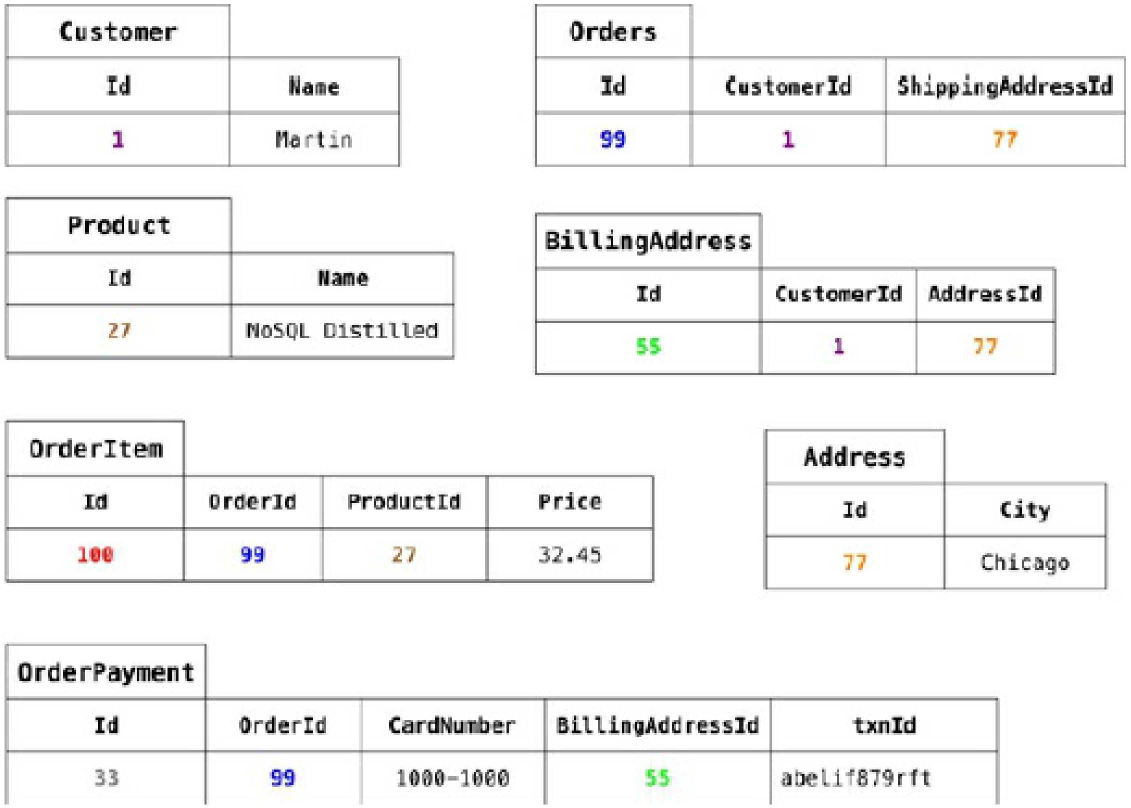


Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

Figure 2.2 presents some sample data for this model



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figure 2.2. Typical data using RDBMS data model

The model might look when we think in more aggregate oriented terms (Figure 2.3).

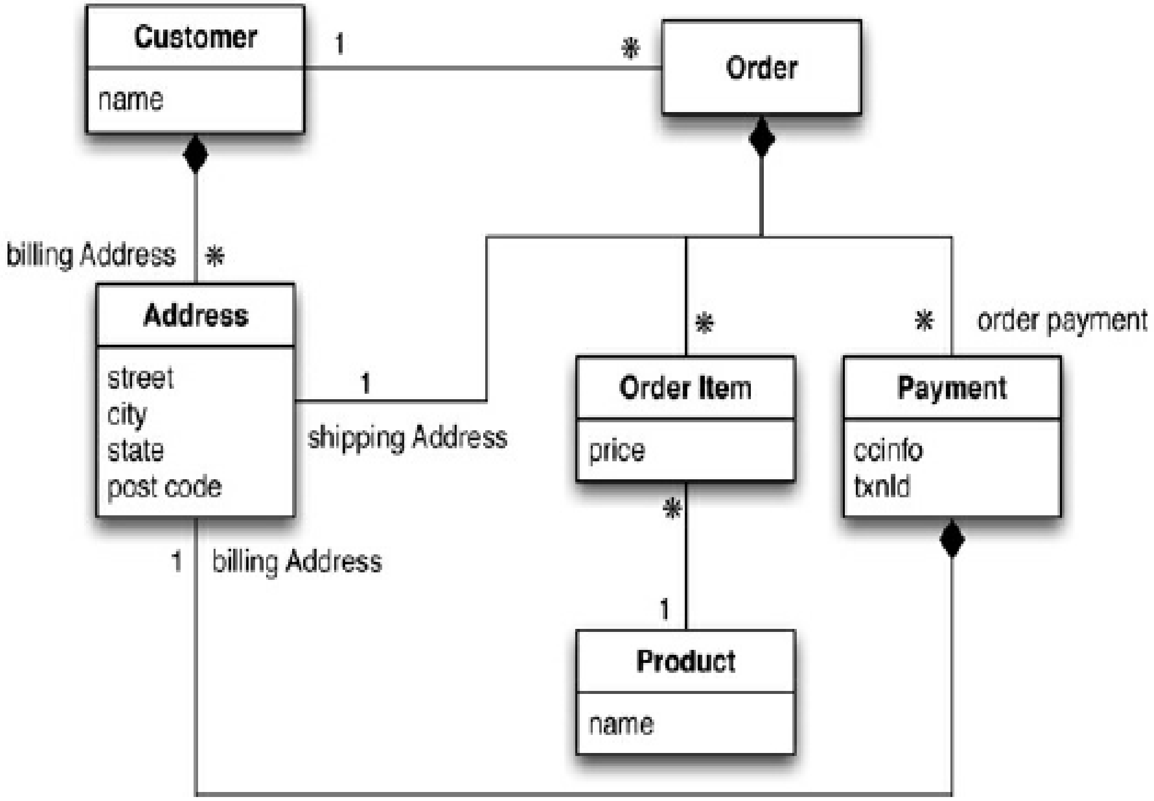


Figure 2.3. An aggregate data model

- sample data, which is shown in JSON format as a common representation for data in NoSQL land.

```
// in customers
{
        "id":1,
        "name":"Martin",
        "billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
      "id":99,
      "customerId":1,
      "orderItems":[
          {     "productId":27,
                "price": 32.45,
                "productName": "NoSQL Distilled"
          }
      ],
      "shippingAddress":[{"city":"Chicago"}]
      "orderPayment":[
          {     "ccinfo":"1000-1000-1000-1000",
               "txnId":"abelif879rft",
               "billingAddress": {"city": "Chicago"}
          }
    ],
}
```

- In this model, we have two main aggregates: customer and order.

- The black-diamond composition marker in UML to show how data fits into the aggregation structure.

- The customer contains a list of billing addresses; the order contains a list of order items, a shipping address, and payments.

- A single logical address record appears three times in the example data, but instead of using IDs it’s treated as a value and copied each time.

- With aggregates, we can copy the whole address structure into the aggregate as we need to.

- Indeed we could draw our aggregate boundaries differently, putting all the orders for a customer into the customer aggregate (Figure 2.4).

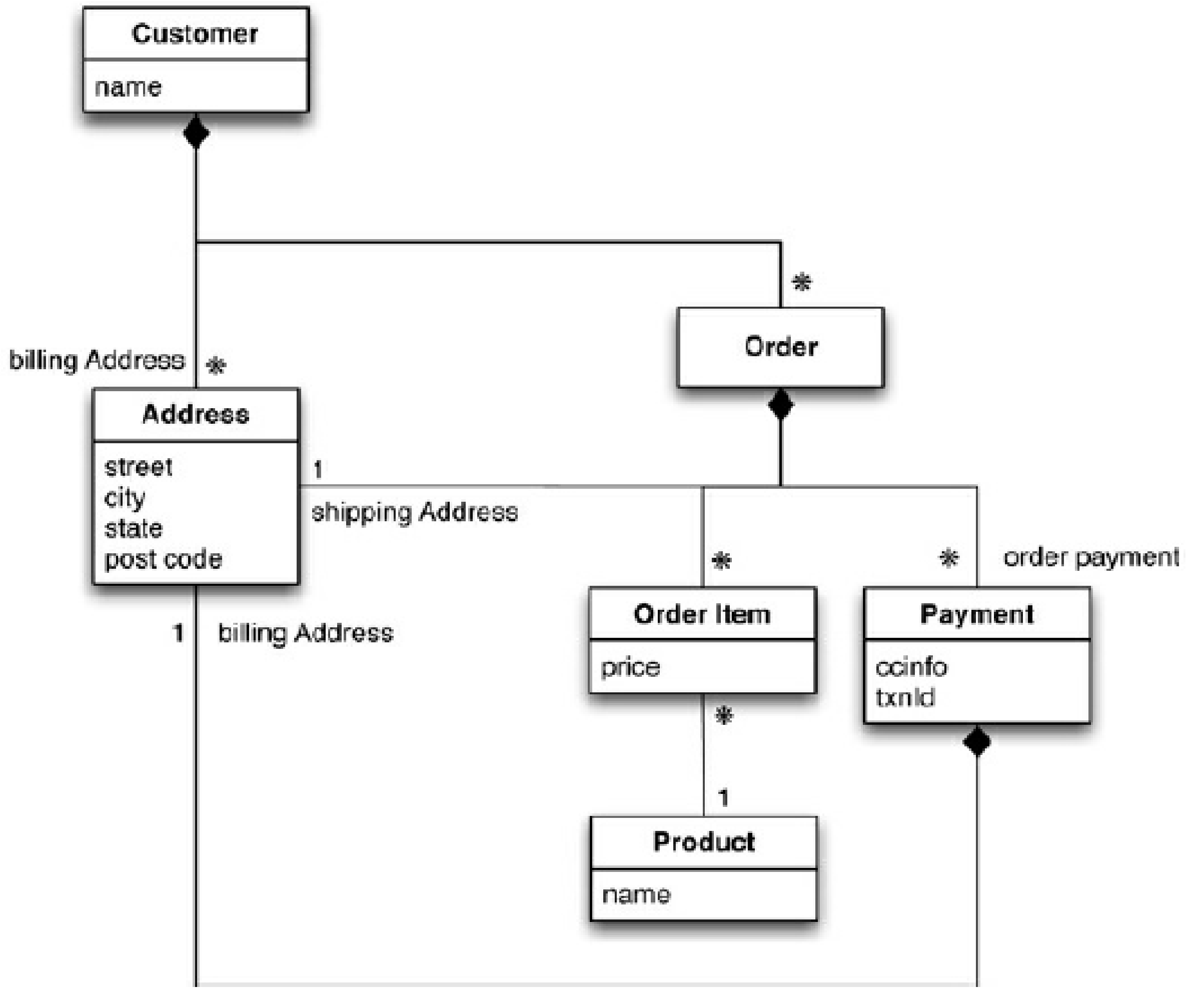
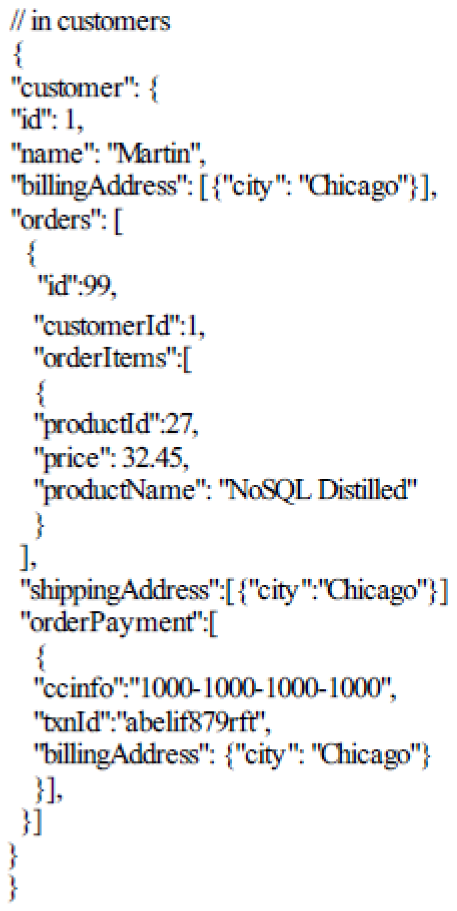


Figure 2.4. Embed all the objects for customer and the customer's orders

- Using the above data model, an example Customer and Order would look like this:



```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
   "customerId":1,
   "orderItems":[
   {
   "productId":27,
   "price": 32.45,
   "productName": "NoSQL Distilled"
   }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
   {
   "ccinfo":"1000-1000-1000-1000",
   "txnId":"abelif879rft",
   "billingAddress": {"city": "Chicago"}
   }],
  }]
}
}
```

- Like most things in modeling, there’s no universal answer for how to draw our aggregate boundaries.
- It depends entirely on how we tend to manipulate our data.
- If we tend to access a customer together with all of that customer’s orders at once, then we would prefer a single aggregate.
- However, if we tend to focus on accessing a single order at a time, then we should prefer having separate aggregates for each order.

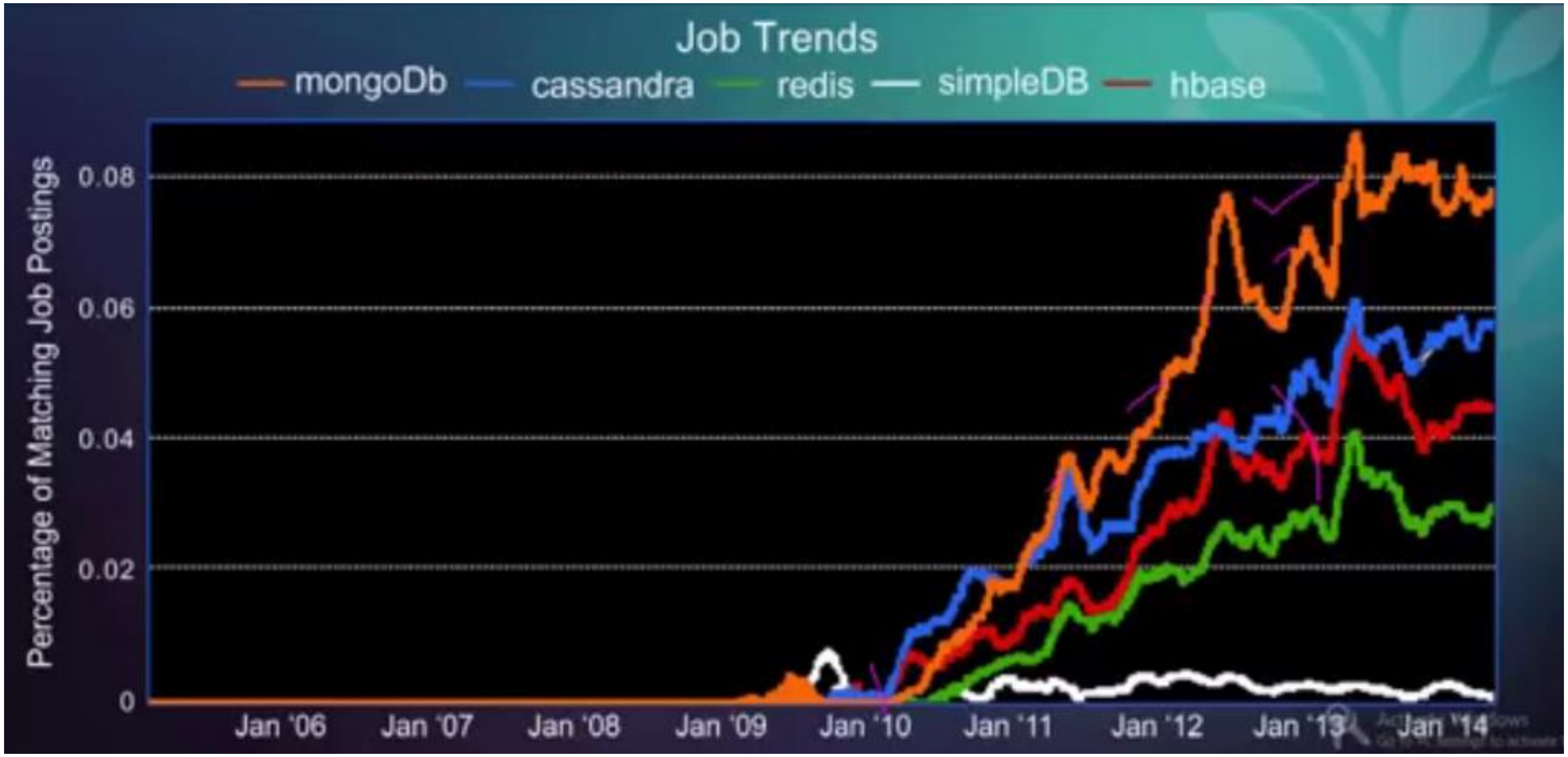
mongoDB

- MongoDB is an open-source document database and leading NoSQL database.

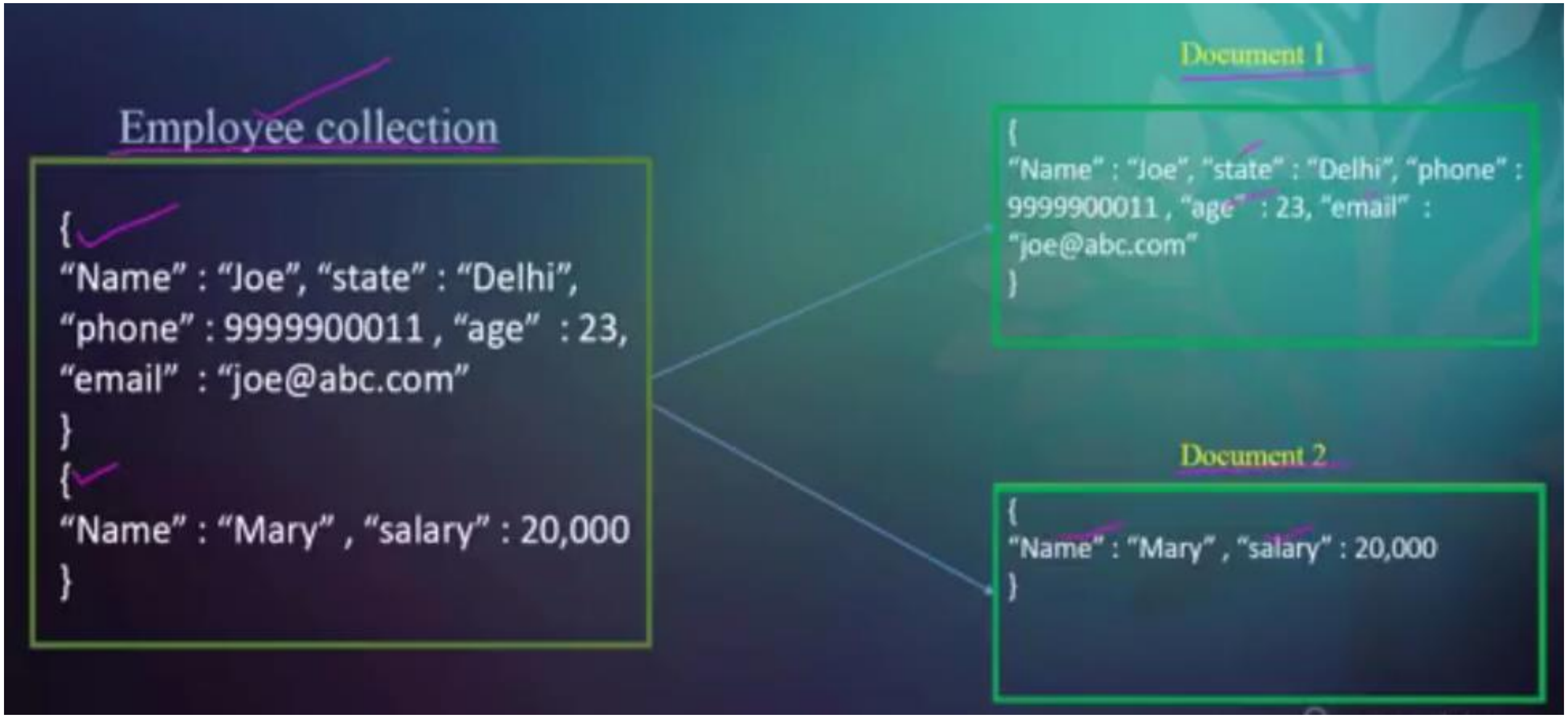
- MongoDB is written in C++

- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability.

- MongoDB works on concept of collection and document.


Job Trends
mongoDb
cassandra
redis
simpleDB
hbase
Percentage of Matching Job Postings
0.08
0.06
0.04
0.02
0
Jan '06
Jan '07
Jan '08
Jan '09
Jan '10
Jan '11
Jan '12
Jan '13
Jan '14

Schema-less

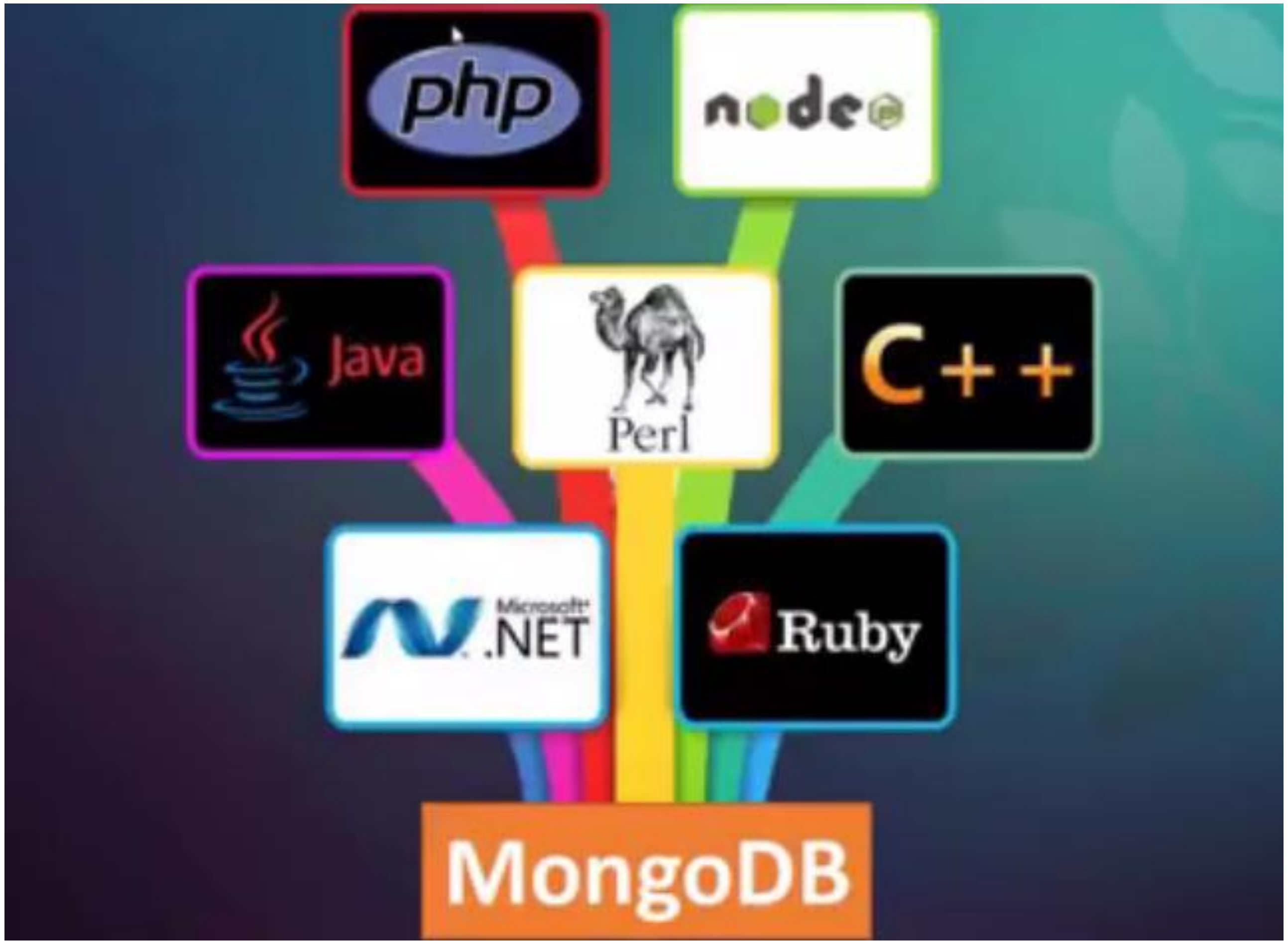


```
{
"Name" : "Joe", "state" : "Delhi",
"phone" : 9999900011 , "age"  : 23,
"email"  : "joe@abc.com"
}
{
"Name" : "Mary" , "salary" : 20,000
}
```

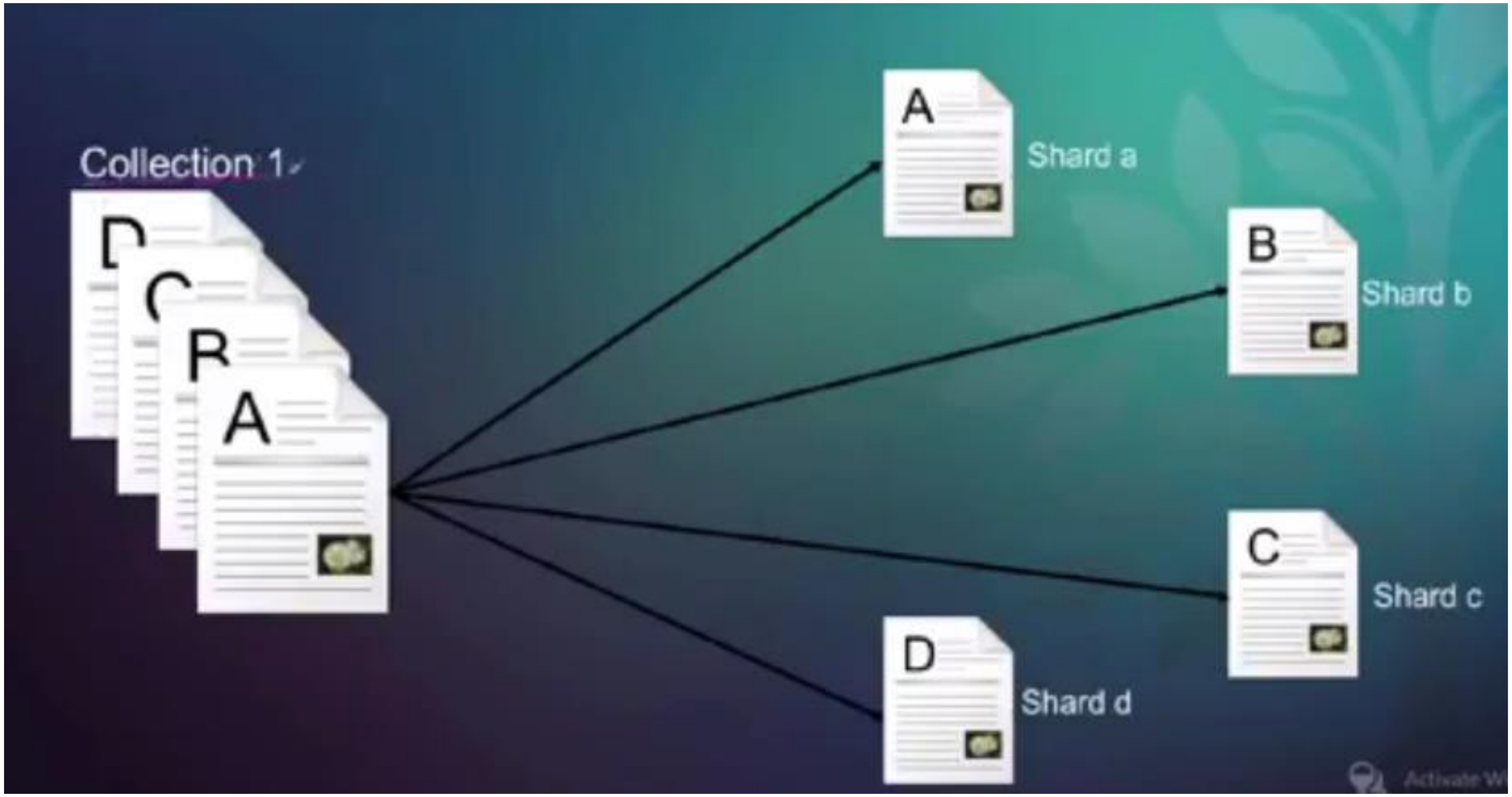
```
{
"Name" : "Joe", "state" : "Delhi", "phone" :
9999900011 , "age"  : 23, "email"  :
"joe@abc.com"
}
```

```
{
"Name" : "Mary" , "salary" : 20,000
}
```

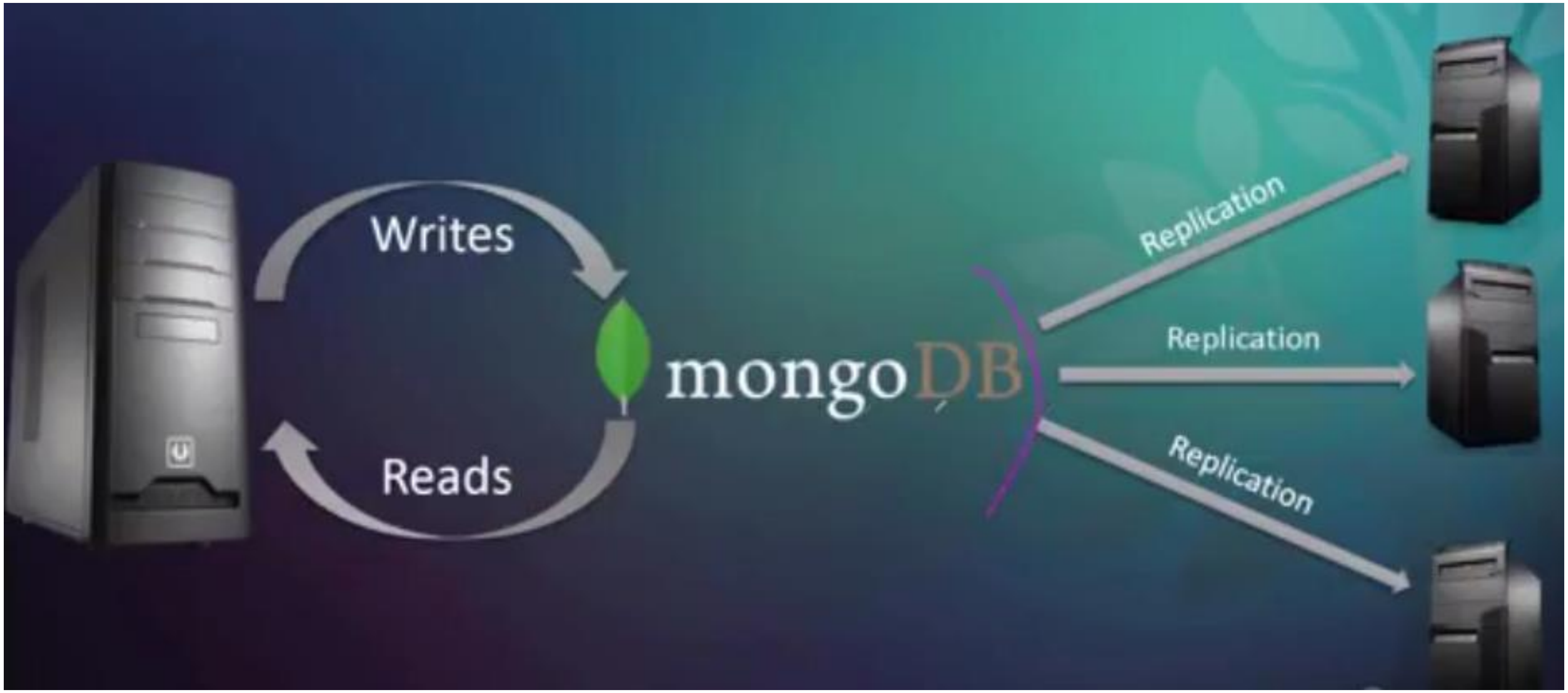
Extensive driver support



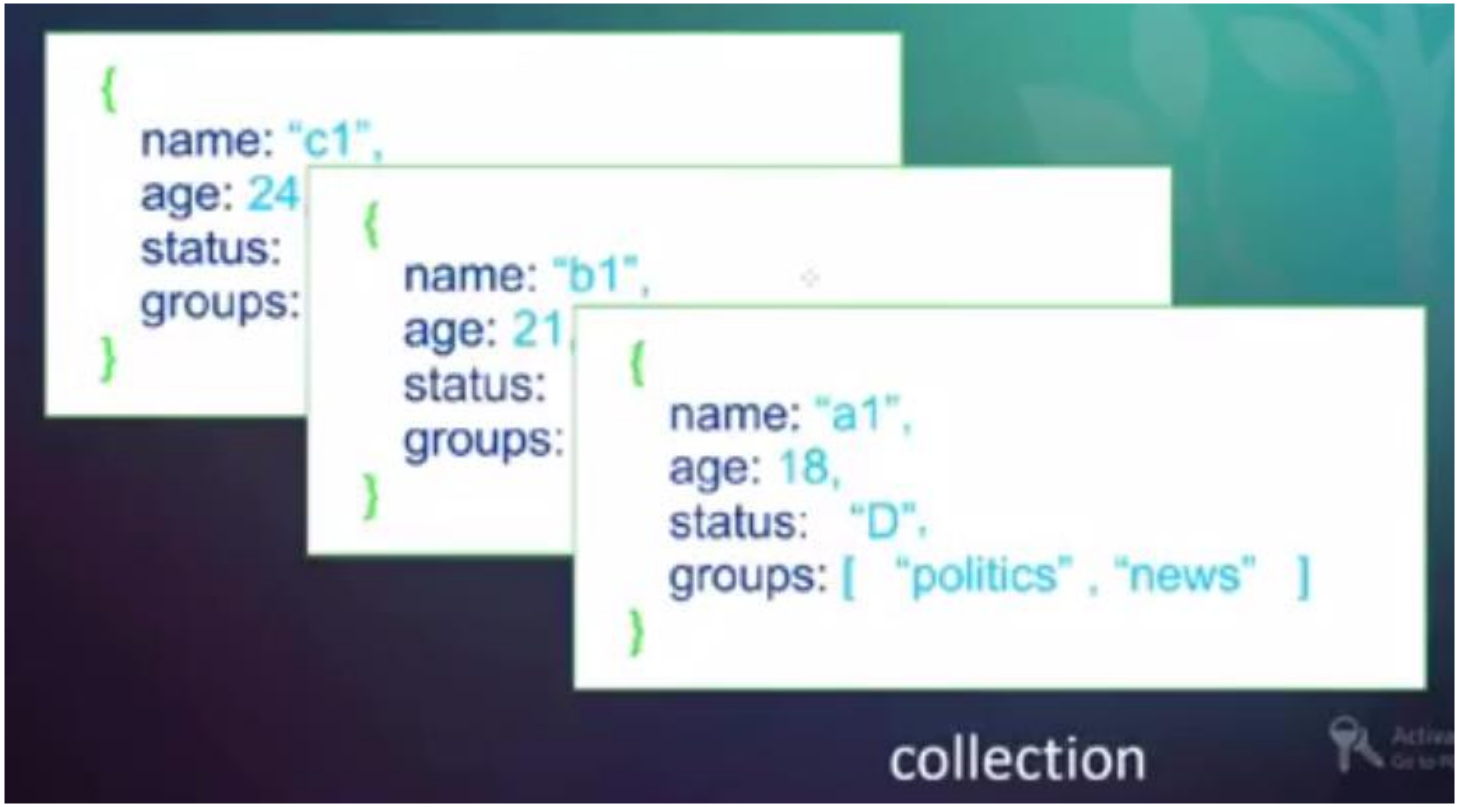
Auto-sharding



Replication and High availability



Document oriented storage



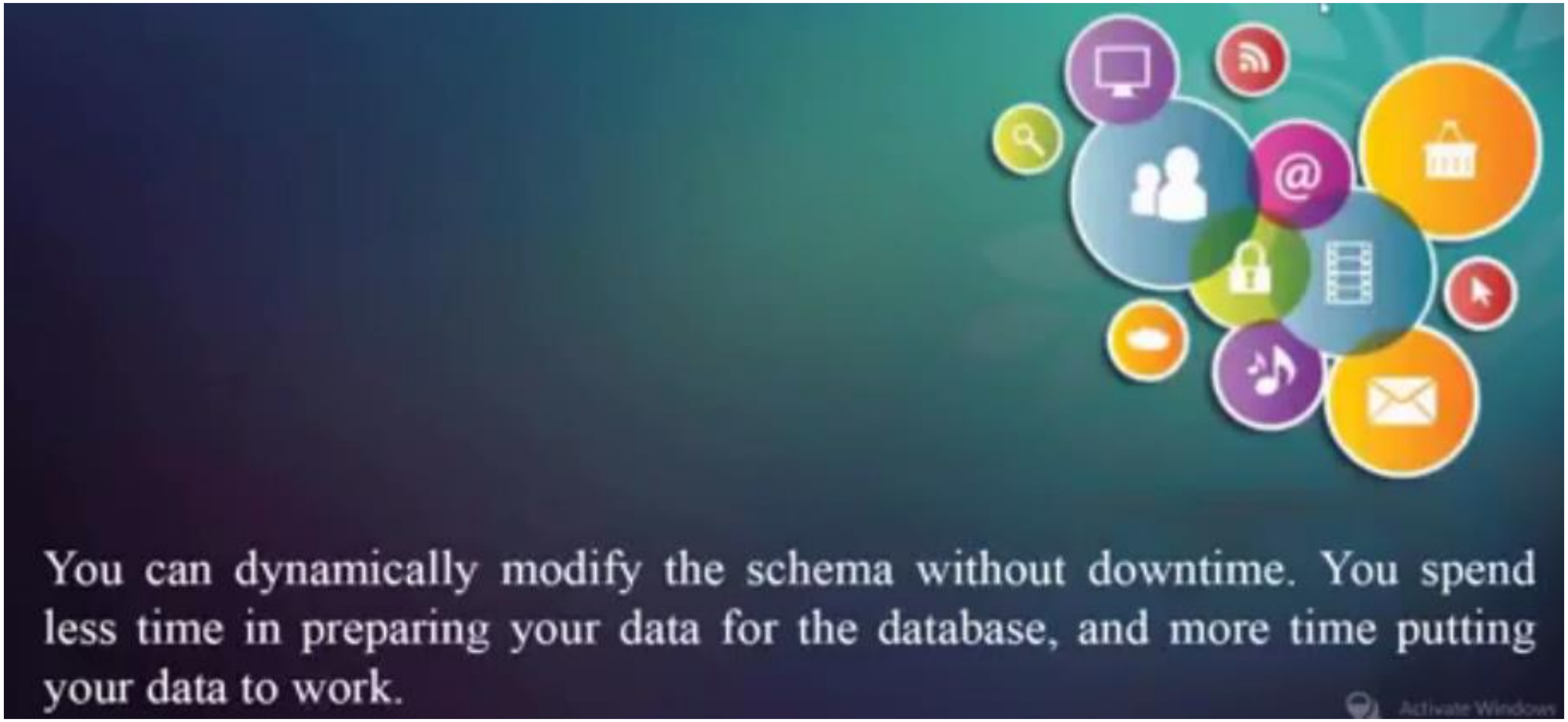
```
{
    name: "c1",
    age: 24
    status:
    groups:
}
```

```
{
    name: "b1",
    age: 21
    status:
    groups:
}
```

```
{
    name: "a1",
    age: 18,
    status:   "D",
    groups: [  "politics" , "news"  ]
}
```

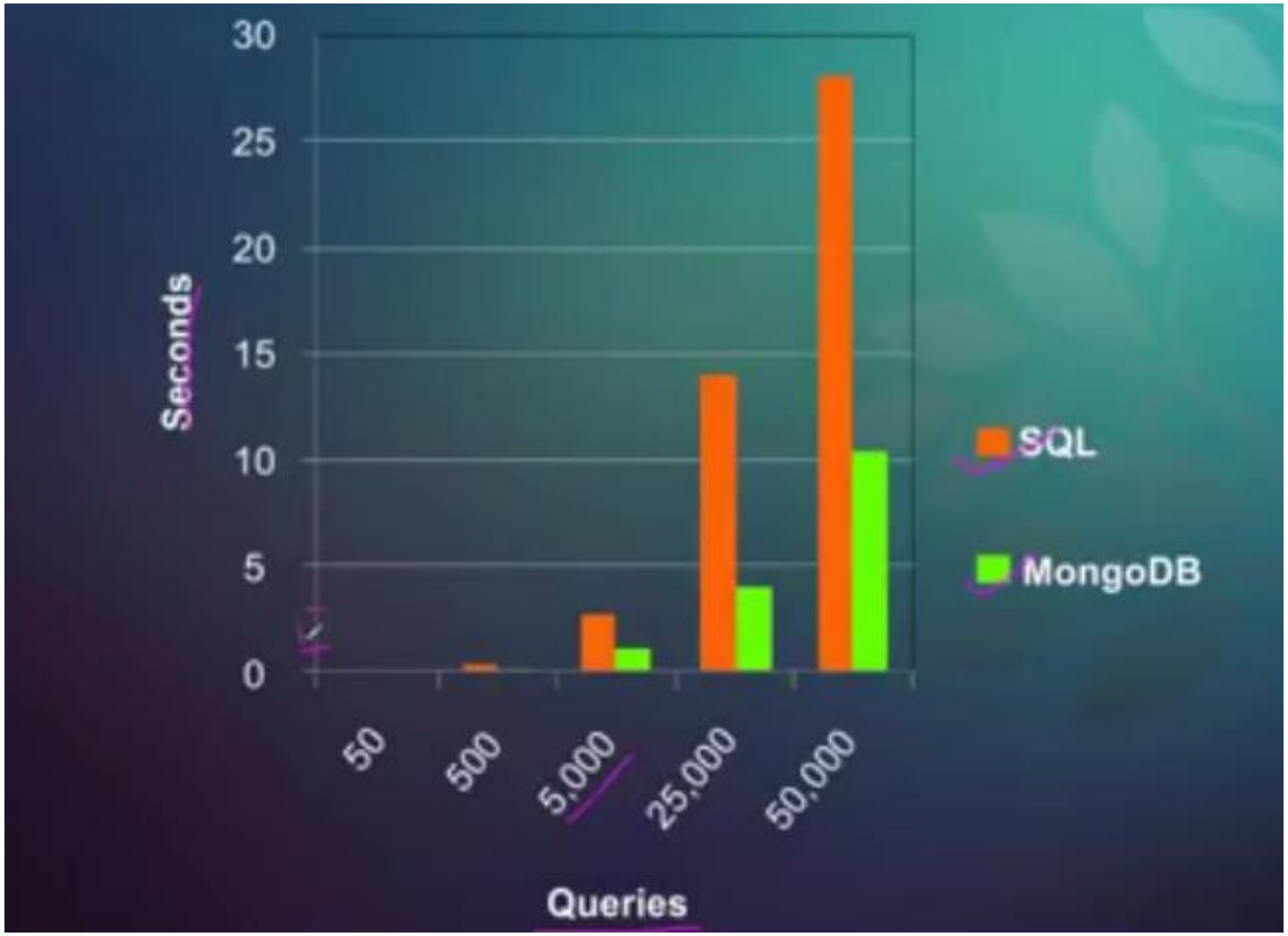
Advantages of MongoDB

Flexibility

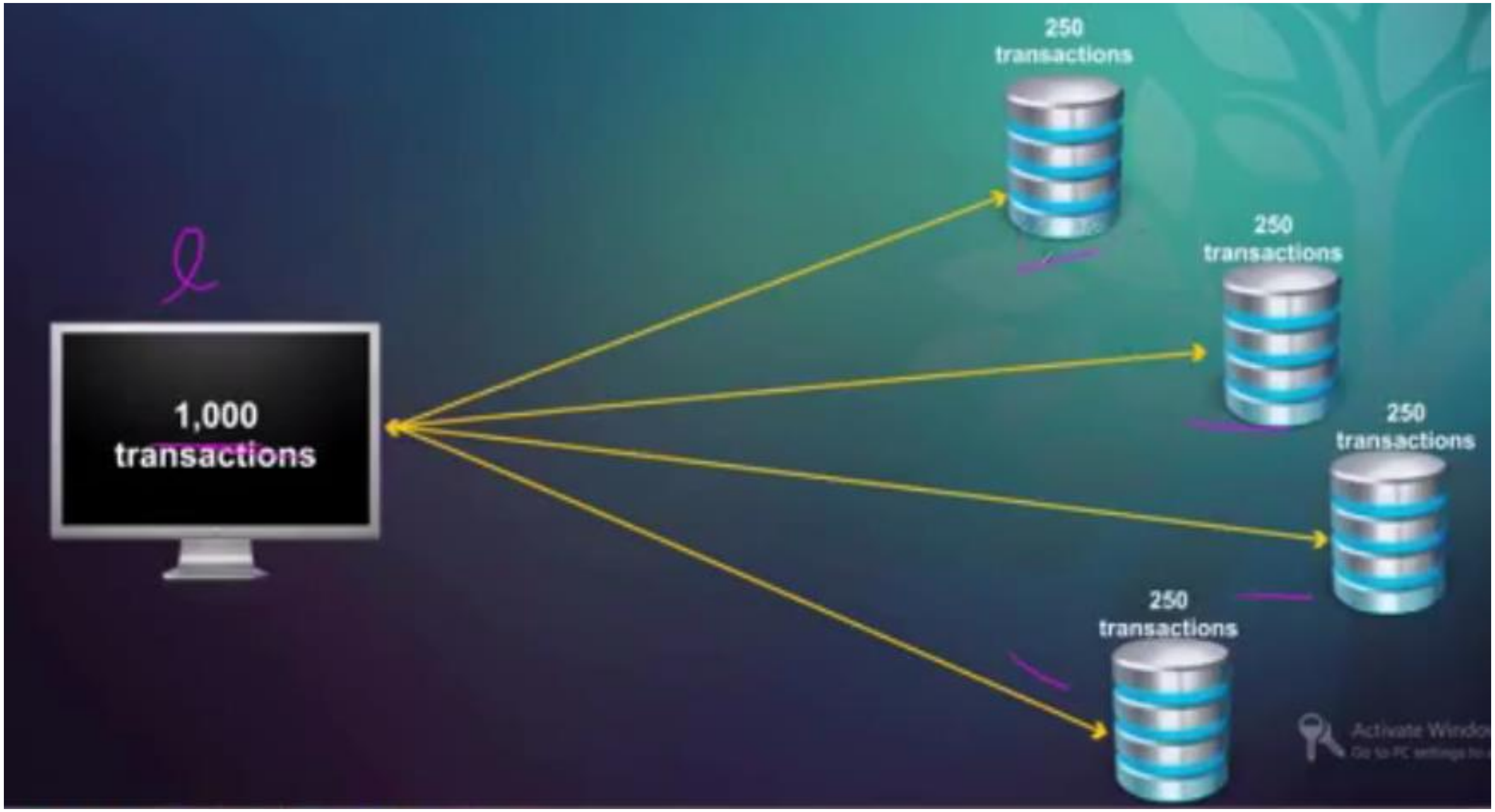


You can dynamically modify the schema without downtime. You spend less time in preparing your data for the database, and more time putting your data to work.

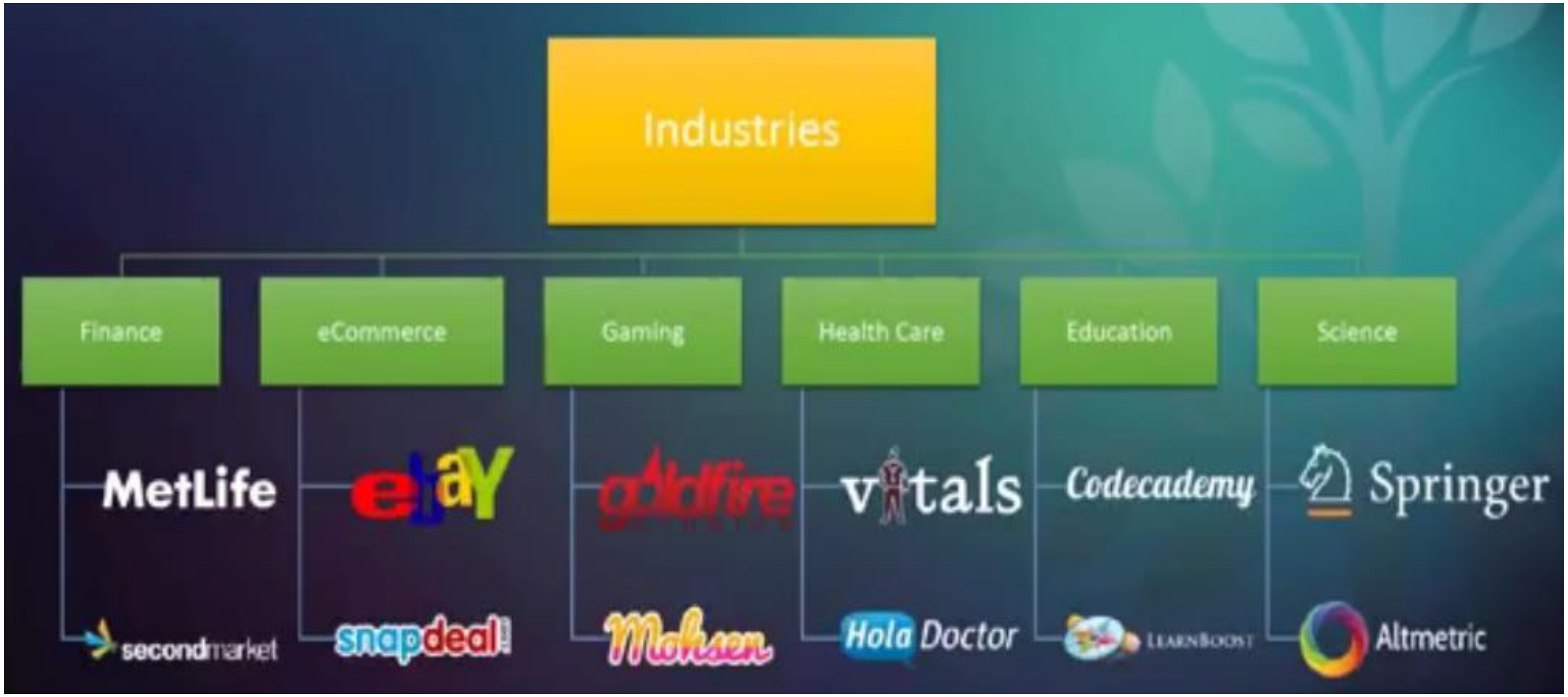
Performance



Scalability



Industries Using MongoDB


Industries
Finance
eCommerce
Gaming
Health Care
Education
Science
MetLife
secondmarket
ebay
snapdeal
goldfire
Mohsen
vitals
Hola Doctor
Codecademy
LearnBoost
Springer
Altmetric

Database

Database is a physical container for collections.

Each database gets its own set of files on the file system.

A single MongoDB server typically has multiple databases.

Collection

- Collection is a group of MongoDB documents.
- It is the equivalent of an RDBMS table.
- A collection exists within a single database.
- Collections do not enforce a schema. Documents within a collection can have different fields.
- Typically, all documents in a collection are of similar or related purpose.

Document

- A document is a set of key-value pairs.
- Documents have dynamic schema.
- Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Install MongoDB community server

- https://www.mongodb.com/try/download/community (windows)
- Create directory /data/db
- Check the environment variable
- Check if MongoDB Compass is installed
- Install mongo shell

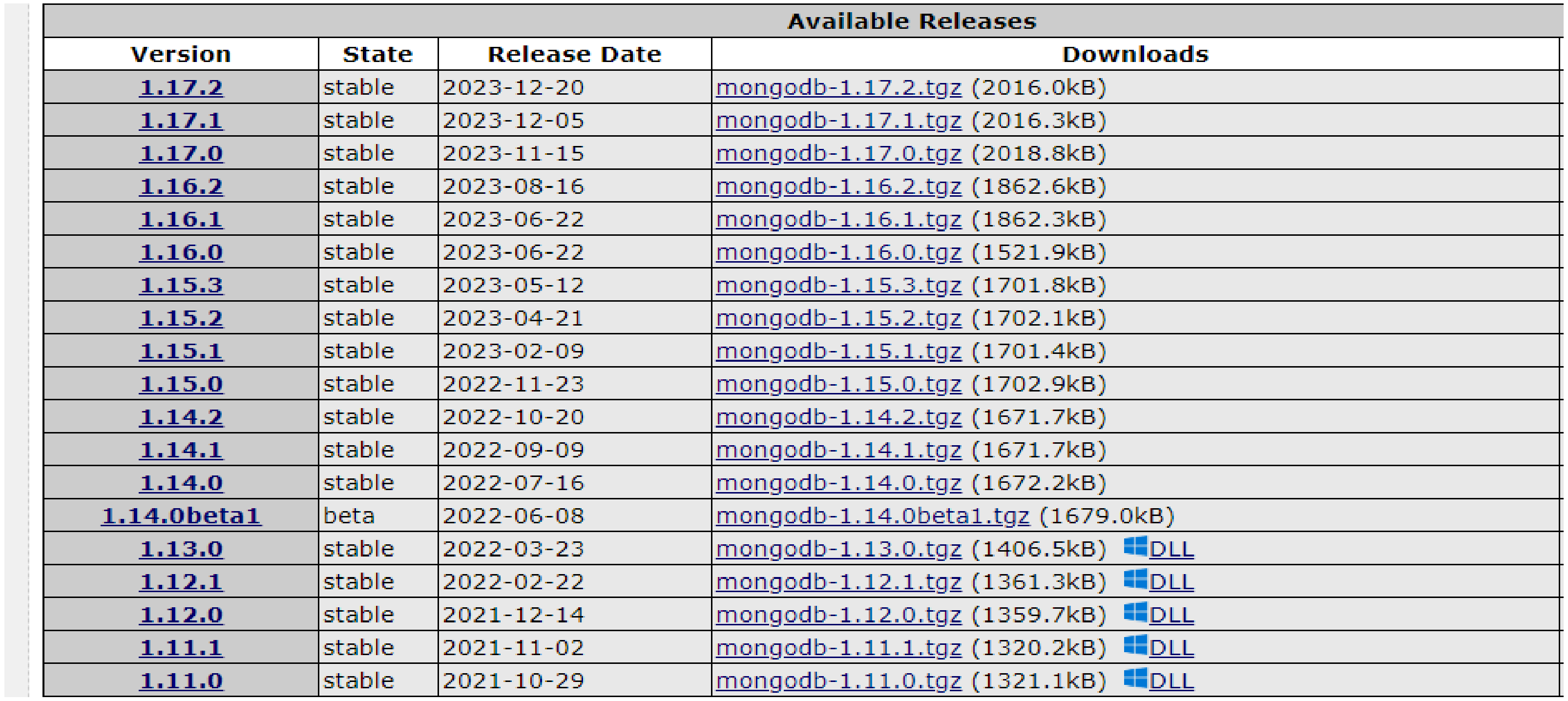
Install the MongoDB PHP Library

- Installing the Extension
- Installing the Library
  - Using Composer

Install the MongoDB PHP Library..

Installing the Extension

https://pecl.php.net/package/mongodb



Available Releases

Version	State	Release Date	Downloads
1.17.2	stable	2023-12-20	mongodb-1.17.2.tgz (2016.0kB)
1.17.1	stable	2023-12-05	mongodb-1.17.1.tgz (2016.3kB)
1.17.0	stable	2023-11-15	mongodb-1.17.0.tgz (2018.8kB)
1.16.2	stable	2023-08-16	mongodb-1.16.2.tgz (1862.6kB)
1.16.1	stable	2023-06-22	mongodb-1.16.1.tgz (1862.3kB)
1.16.0	stable	2023-06-22	mongodb-1.16.0.tgz (1521.9kB)
1.15.3	stable	2023-05-12	mongodb-1.15.3.tgz (1701.8kB)
1.15.2	stable	2023-04-21	mongodb-1.15.2.tgz (1702.1kB)
1.15.1	stable	2023-02-09	mongodb-1.15.1.tgz (1701.4kB)
1.15.0	stable	2022-11-23	mongodb-1.15.0.tgz (1702.9kB)
1.14.2	stable	2022-10-20	mongodb-1.14.2.tgz (1671.7kB)
1.14.1	stable	2022-09-09	mongodb-1.14.1.tgz (1671.7kB)
1.14.0	stable	2022-07-16	mongodb-1.14.0.tgz (1672.2kB)
1.14.0beta1	beta	2022-06-08	mongodb-1.14.0beta1.tgz (1679.0kB)
1.13.0	stable	2022-03-23	mongodb-1.13.0.tgz (1406.5kB) DLL
1.12.1	stable	2022-02-22	mongodb-1.12.1.tgz (1361.3kB) DLL
1.12.0	stable	2021-12-14	mongodb-1.12.0.tgz (1359.7kB) DLL
1.11.1	stable	2021-11-02	mongodb-1.11.1.tgz (1320.2kB) DLL
1.11.0	stable	2021-10-29	mongodb-1.11.0.tgz (1321.1kB) DLL

Install the MongoDB PHP Library..

Installing the Extension

https://pecl.php.net/package/mongodb

After downloading the appropriate archive for your PHP environment, extract the php_mongodb.dll file to PHP's extension directory (XAMPP can be used) and add the following line to your php.ini file:

```
extension=php_mongodb.dll
```

Installing the Library

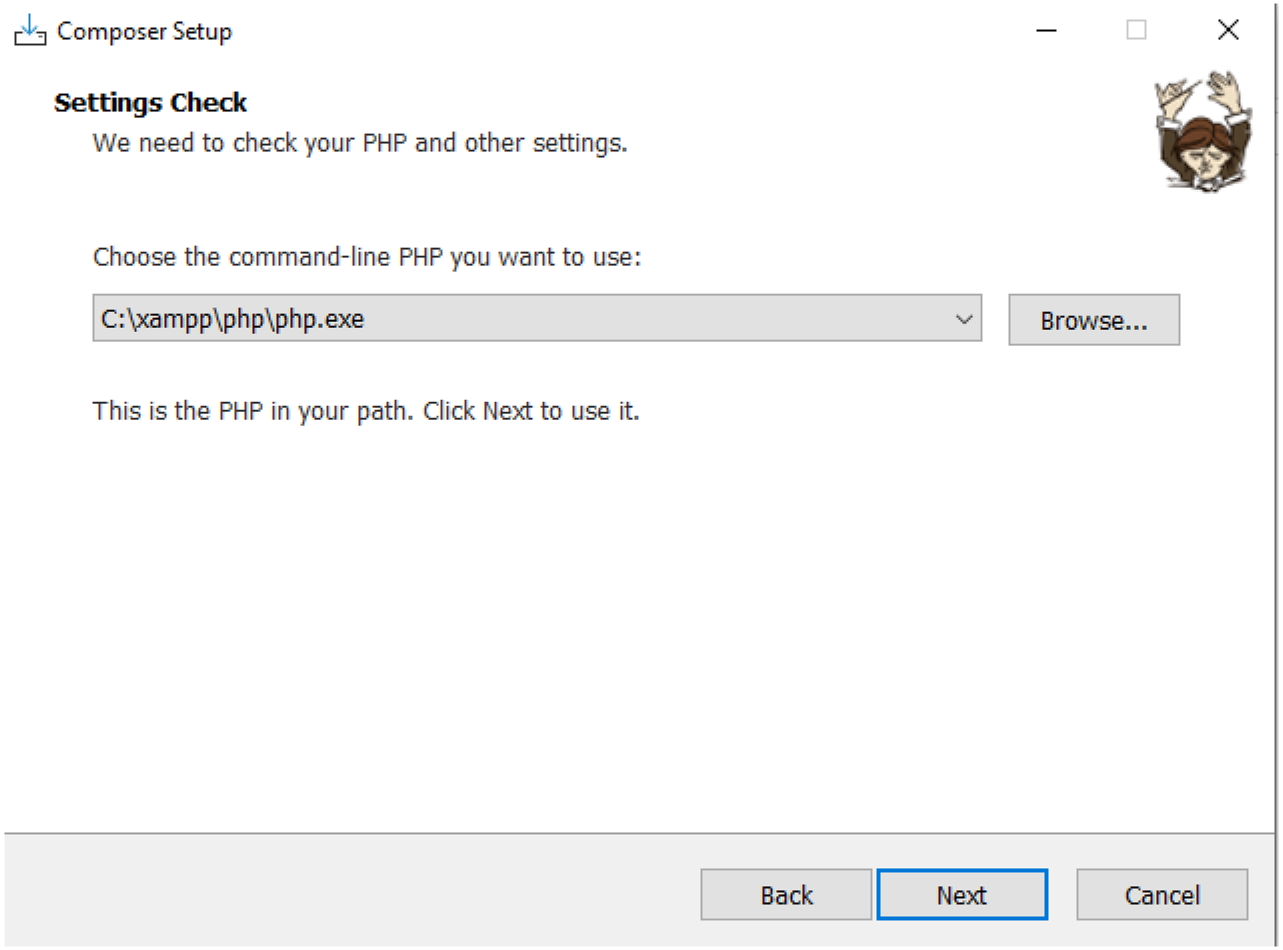
Using Composer

The preferred method of installing the MongoDB PHP Library is with Composer by running the following command from your project root:

https://getcomposer.org/

Installing the Library

Using Composer



Composer Setup

Settings Check

We need to check your PHP and other settings.

Choose the command-line PHP you want to use:

C:\xampp\php\php.exe

This is the PHP in your path. Click Next to use it.

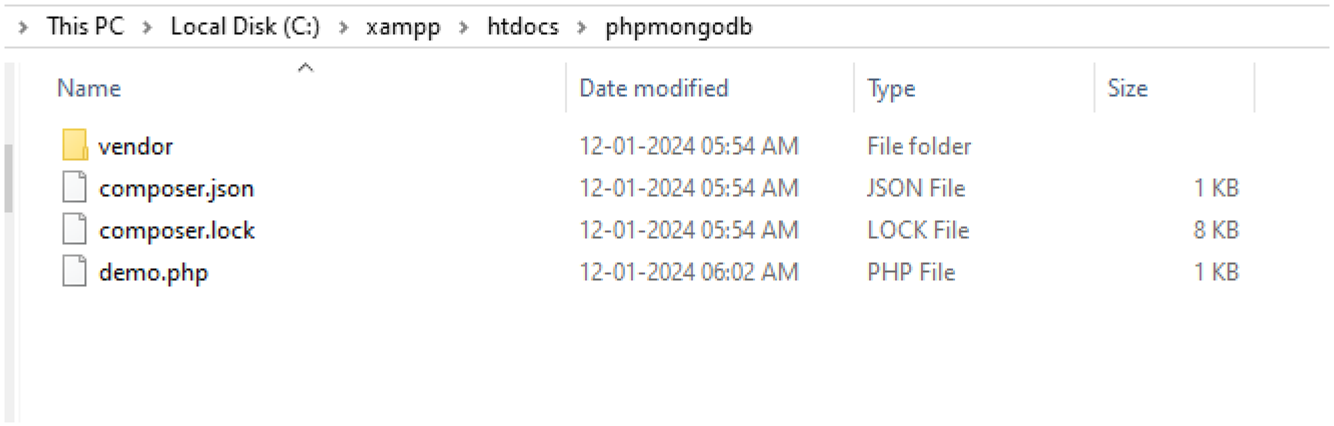
Installing the Library

Using Composer

The preferred method of installing the MongoDB PHP Library is with Composer by running the following command from your project root:

composer require mongodb/mongodb

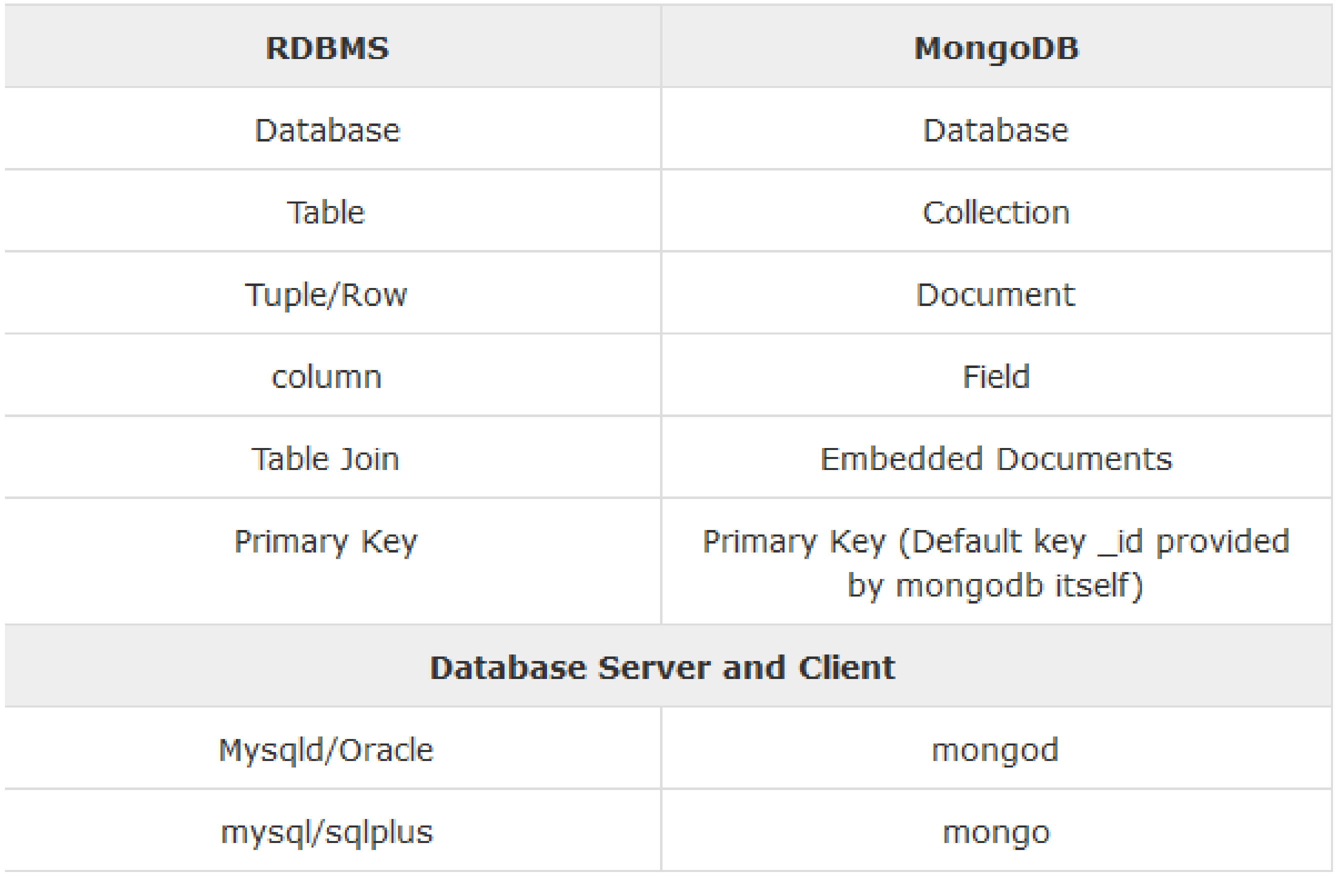
(run on cmd)



This PC › Local Disk (C:) › xampp › htdocs › phpmongodb

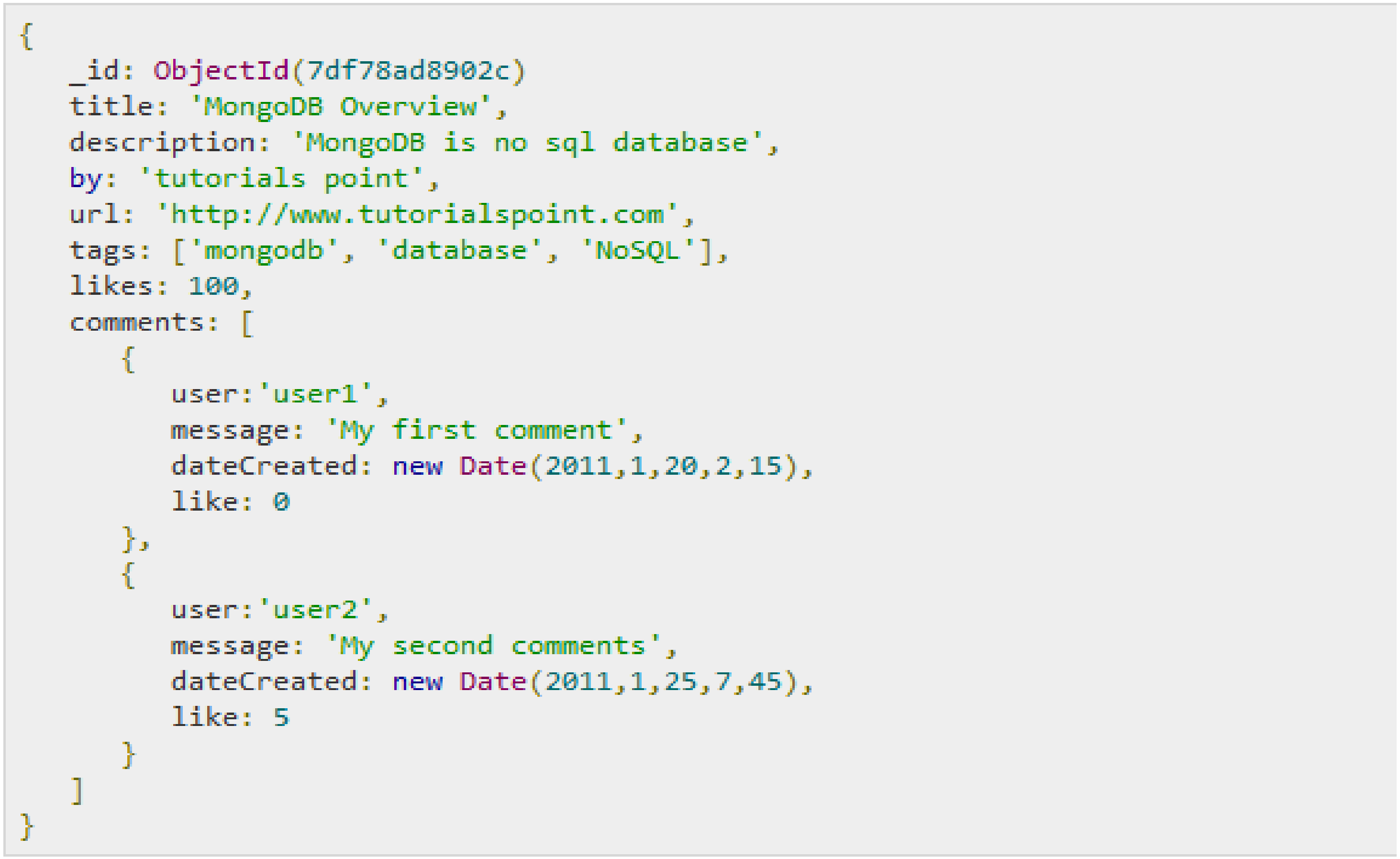
Name	Date modified	Type	Size
vendor	12-01-2024 05:54 AM	File folder	
composer.json	12-01-2024 05:54 AM	JSON File	1 KB
composer.lock	12-01-2024 05:54 AM	LOCK File	8 KB
demo.php	12-01-2024 06:02 AM	PHP File	1 KB

```
<?php
require 'vendor/autoload.php';
$client=new MongoDB\Client;
$companydb=$client->companydb;
$result1=$companydb-
>createCollection('empcollection');
var_dump($result1);
?>
```



RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

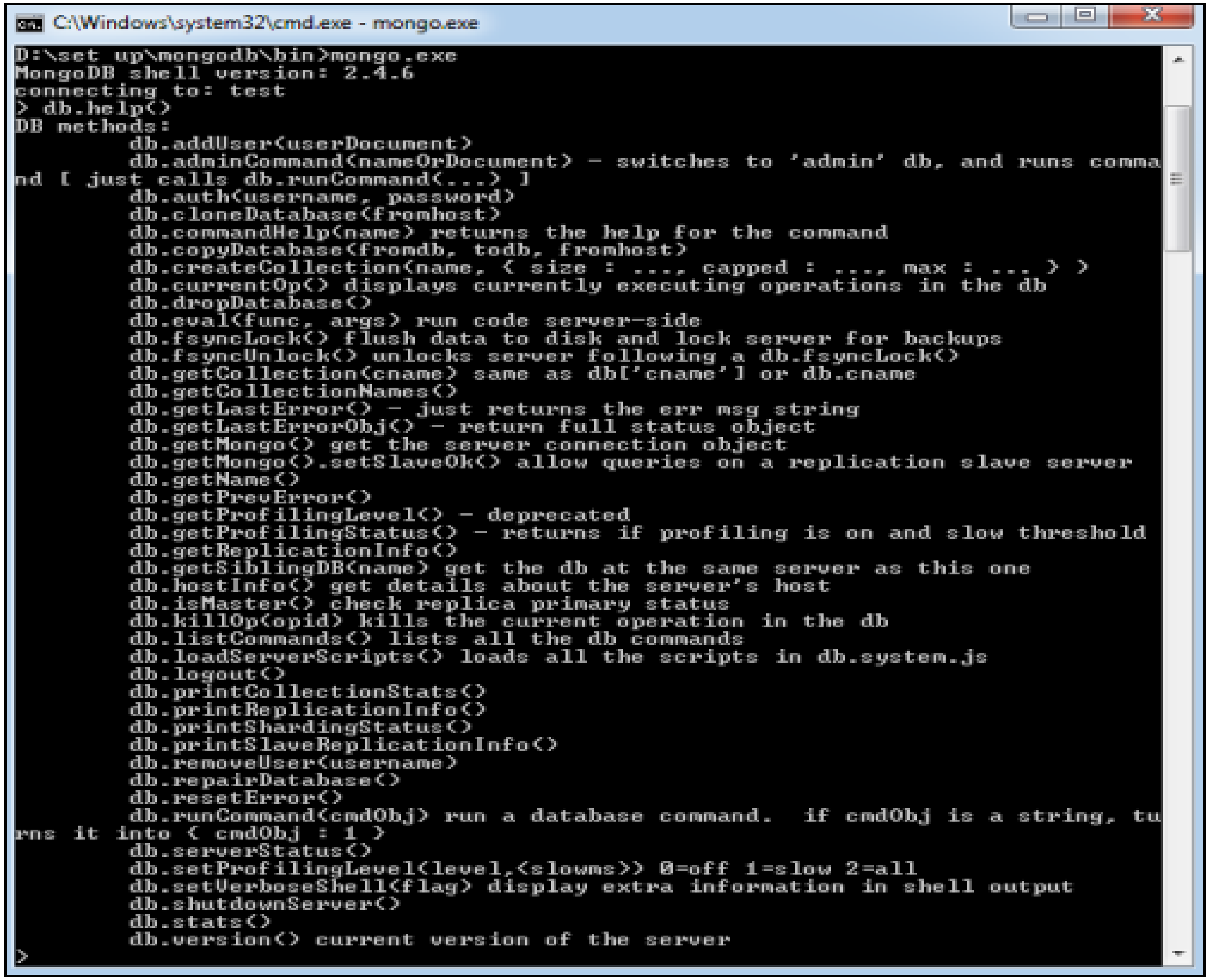
Following example shows the document structure of a blog site, which is simply a comma separated key value pair.



```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

- **_id** is a 12 bytes hexadecimal number which assures the uniqueness of every document.
- We can provide _id while inserting the document.
- If we don’t provide then MongoDB provides a unique id for every document.
- These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

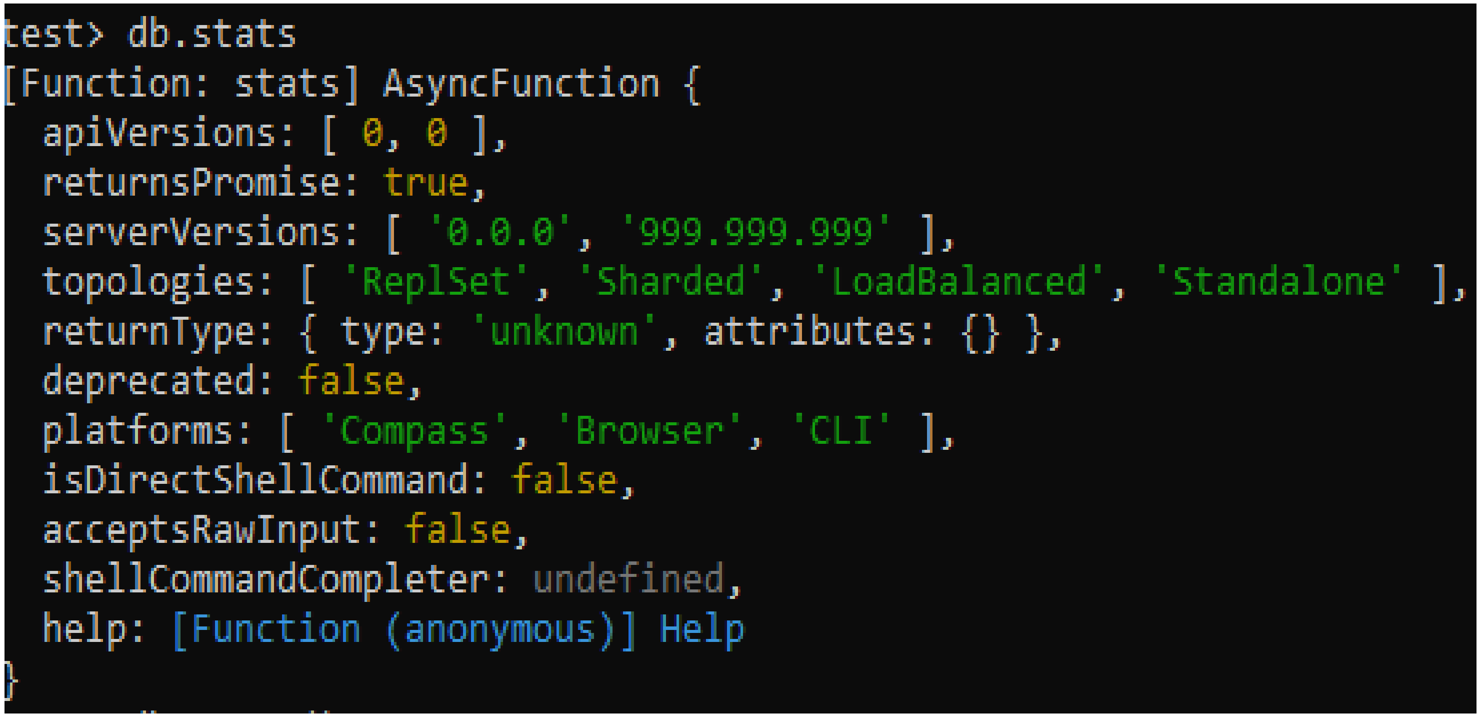
MongoDB Help



```
D:\set up\mongodb\bin>mongo.exe
MongoDB shell version: 2.4.6
connecting to: test
> db.help()
DB methods:
        db.addUser(userDocument)
        db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs comma
nd [ just calls db.runCommand(...) ]
        db.auth(username, password)
        db.cloneDatabase(fromhost)
        db.commandHelp(name) returns the help for the command
        db.copyDatabase(fromdb, todb, fromhost)
        db.createCollection(name, { size : ..., capped : ..., max : ... } )
        db.currentOp() displays currently executing operations in the db
        db.dropDatabase()
        db.eval(func, args) run code server-side
        db.fsyncLock() flush data to disk and lock server for backups
        db.fsyncUnlock() unlocks server following a db.fsyncLock()
        db.getCollection(cname) same as db['cname'] or db.cname
        db.getCollectionNames()
        db.getLastError() - just returns the err msg string
        db.getLastErrorObj() - return full status object
        db.getMongo() get the server connection object
        db.getMongo().setSlaveOk() allow queries on a replication slave server
        db.getName()
        db.getPrevError()
        db.getProfilingLevel() - deprecated
        db.getProfilingStatus() - returns if profiling is on and slow threshold
        db.getReplicationInfo()
        db.getSiblingDB(name) get the db at the same server as this one
        db.hostInfo() get details about the server's host
        db.isMaster() check replica primary status
        db.killOp(opid) kills the current operation in the db
        db.listCommands() lists all the db commands
        db.loadServerScripts() loads all the scripts in db.system.js
        db.logout()
        db.printCollectionStats()
        db.printReplicationInfo()
        db.printShardingStatus()
        db.printSlaveReplicationInfo()
        db.removeUser(username)
        db.repairDatabase()
        db.resetError()
        db.runCommand(cmdObj) run a database command.  if cmdObj is a string, tu
rns it into { cmdObj : 1 }
        db.serverStatus()
        db.setProfilingLevel(level,<slowms>) 0=off 1=slow 2=all
        db.setVerboseShell(flag) display extra information in shell output
        db.shutdownServer()
        db.stats()
        db.version() current version of the server
>
```

MongoDB Statistics

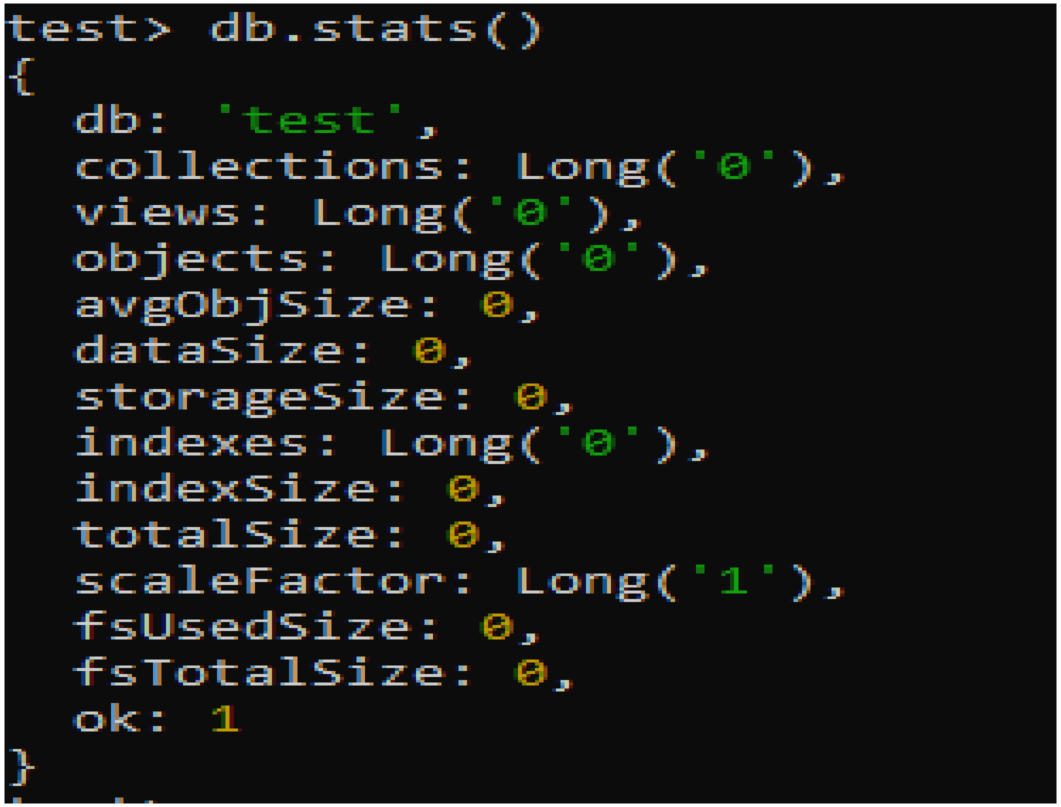
- **db.stats** : attributes configured MongoDB server.



```
test> db.stats
[Function: stats] AsyncFunction {
  apiVersions: [ 0, 0 ],
  returnsPromise: true,
  serverVersions: [ '0.0.0', '999.999.999' ],
  topologies: [ 'ReplSet', 'Sharded', 'LoadBalanced', 'Standalone' ],
  returnType: { type: 'unknown', attributes: {} },
  deprecated: false,
  platforms: [ 'Compass', 'Browser', 'CLI' ],
  isDirectShellCommand: false,
  acceptsRawInput: false,
  shellCommandCompleter: undefined,
  help: [Function (anonymous)] Help
}
```

MongoDB Statistics

- **db.stats()** : To get stats about MongoDB server.
- This shows the database name, number of collection and documents in the database.



```
test> db.stats()
{
  db: 'test',
  collections: Long('0'),
  views: Long('0'),
  objects: Long('0'),
  avgObjSize: 0,
  dataSize: 0,
  storageSize: 0,
  indexes: Long('0'),
  indexSize: 0,
  totalSize: 0,
  scaleFactor: Long('1'),
  fsUsedSize: 0,
  fsTotalSize: 0,
  ok: 1
}
```

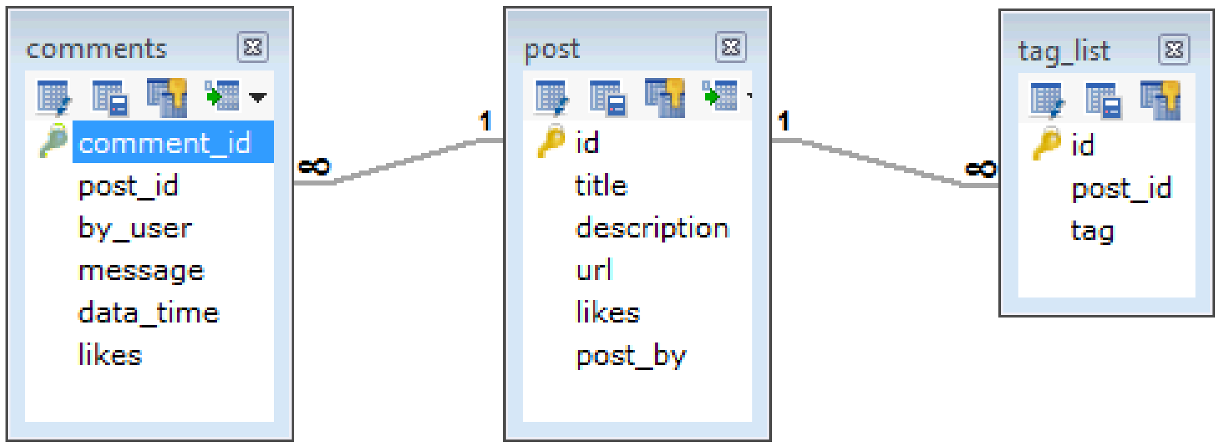
Example

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

Example

In RDBMS schema, design for above requirements will have minimum three tables.

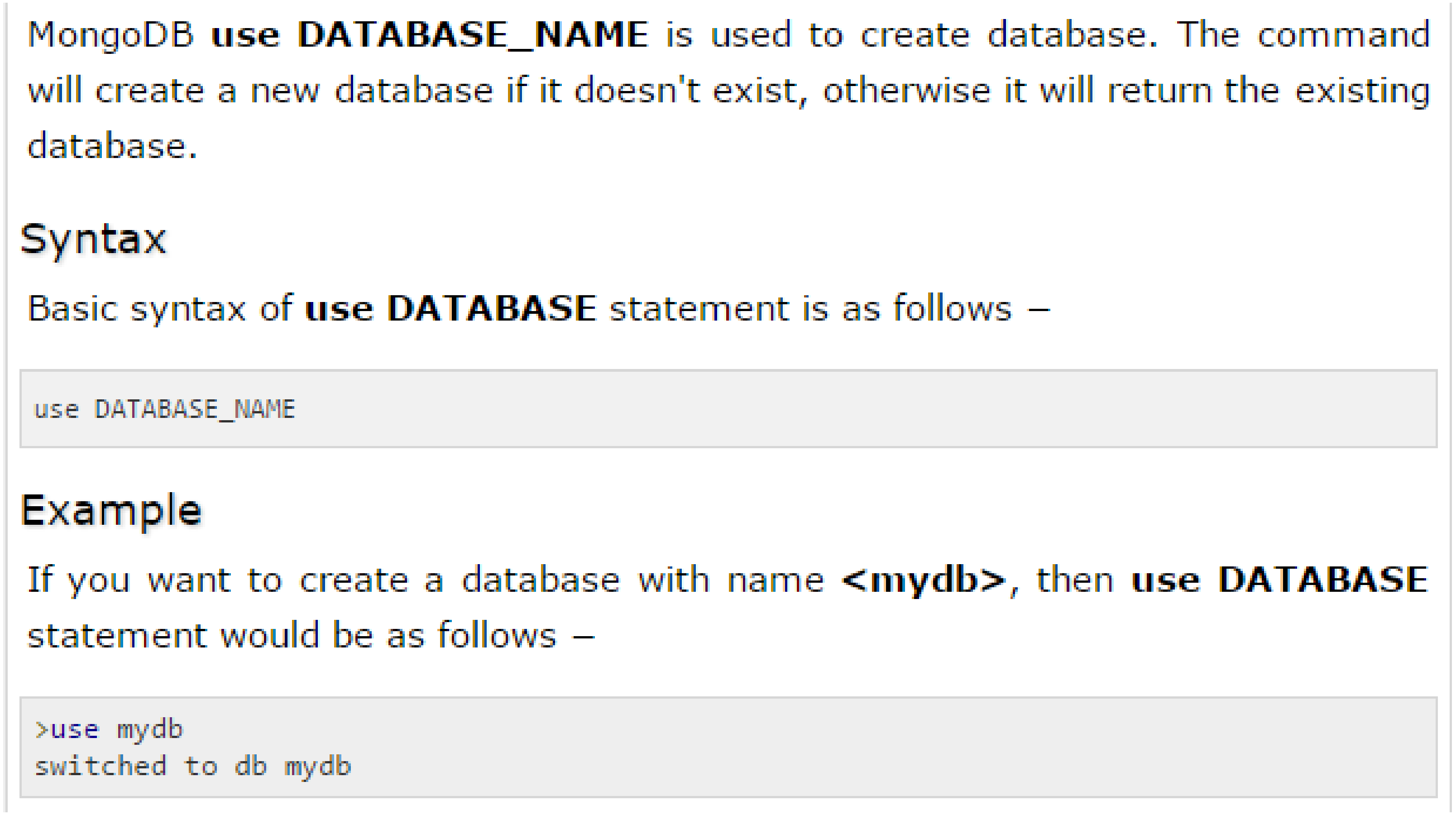


While in MongoDB schema, design will have one collection post and the following structure

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

So while showing the data, in RDBMS we need to join three tables and in MongoDB, data will be shown from one collection only.

The use Command



MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows –

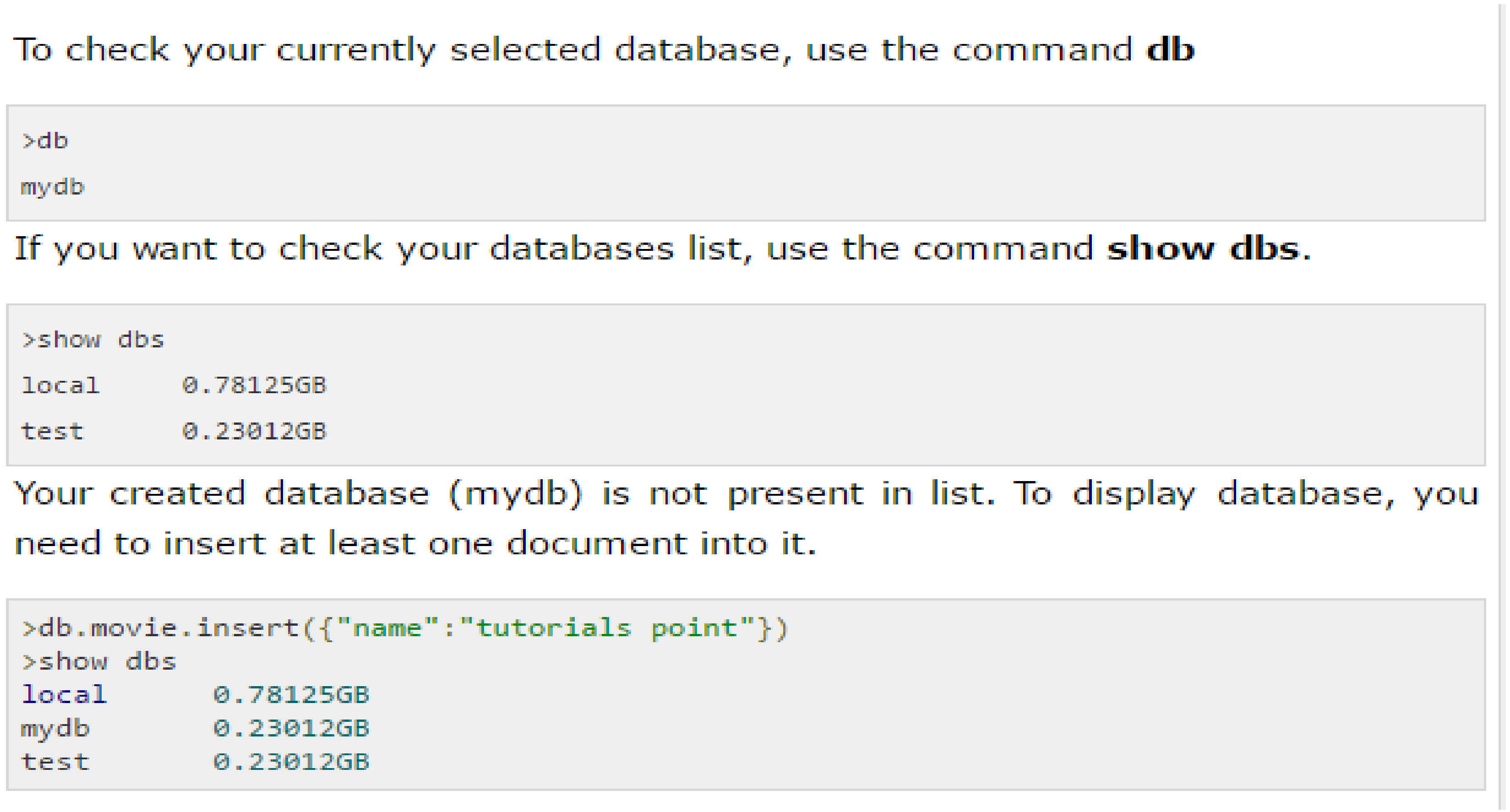
```
use DATABASE_NAME
```

Example

If you want to create a database with name **<mydb>**, then **use DATABASE** statement would be as follows –

```
>use mydb
switched to db mydb
```

The use Command



To check your currently selected database, use the command **db**

```
>db
mydb
```

If you want to check your databases list, use the command **show dbs**.

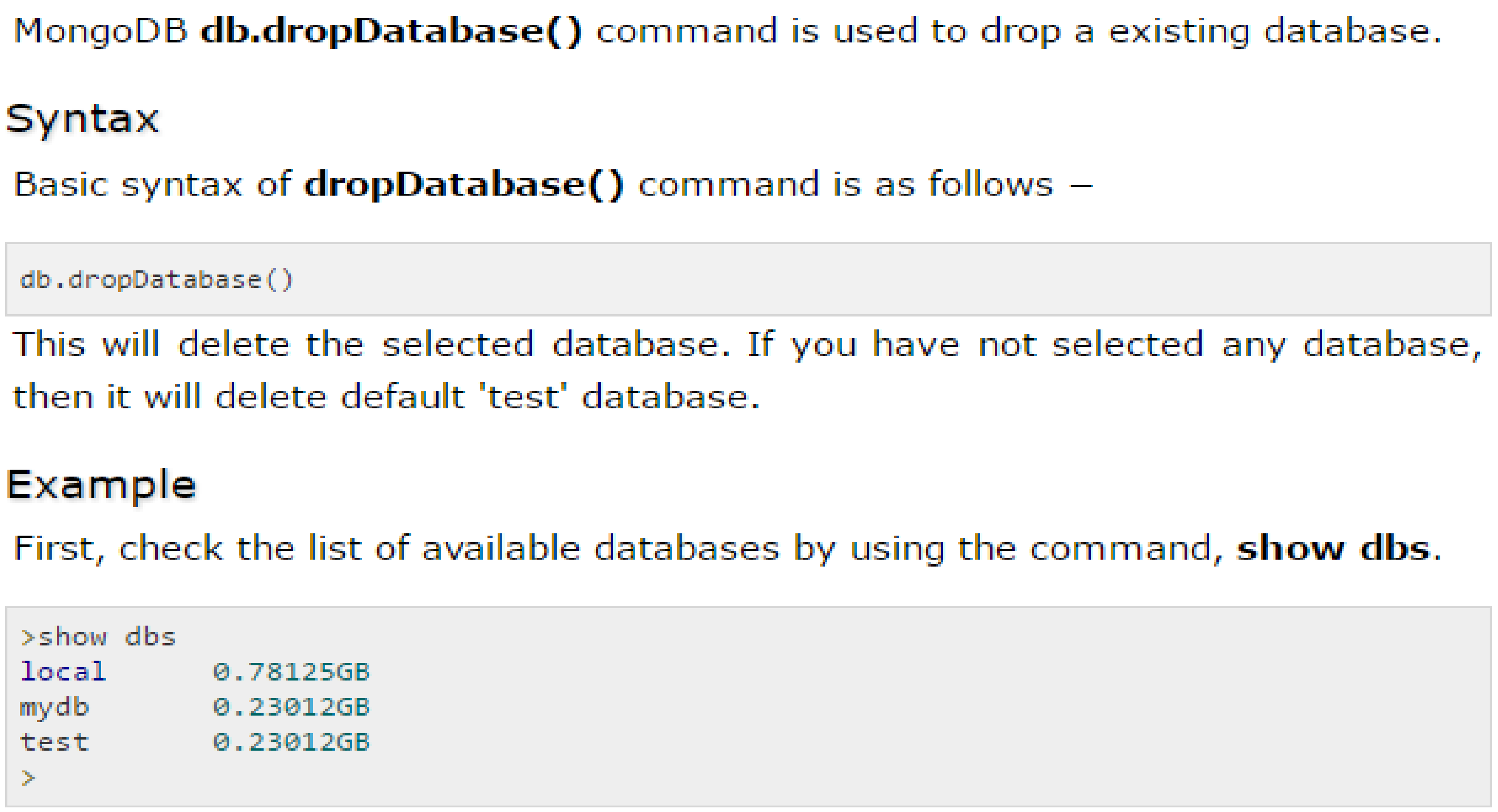
```
>show dbs
local     0.78125GB
test      0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

The dropDatabase() Method



MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows −

```
db.dropDatabase()
```

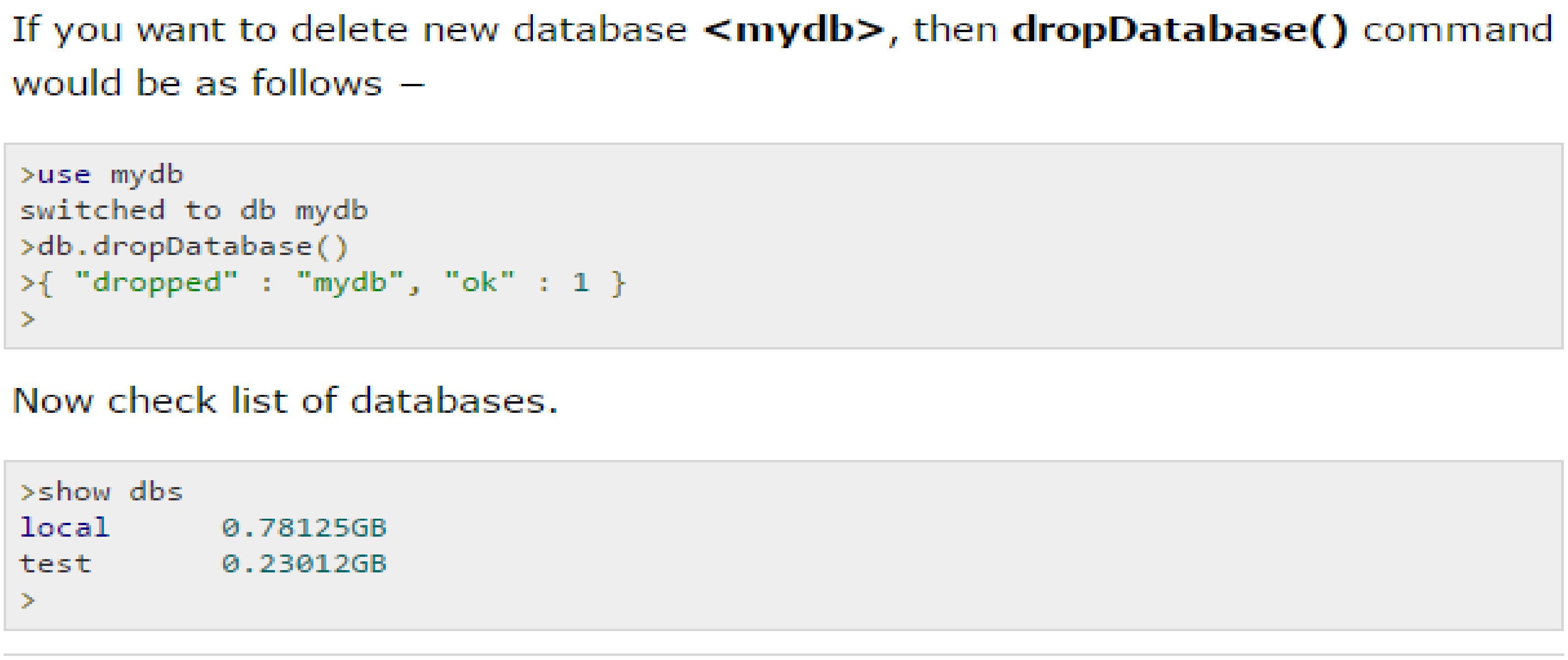
This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, **show dbs**.

```
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
>
```

The dropDatabase() Method



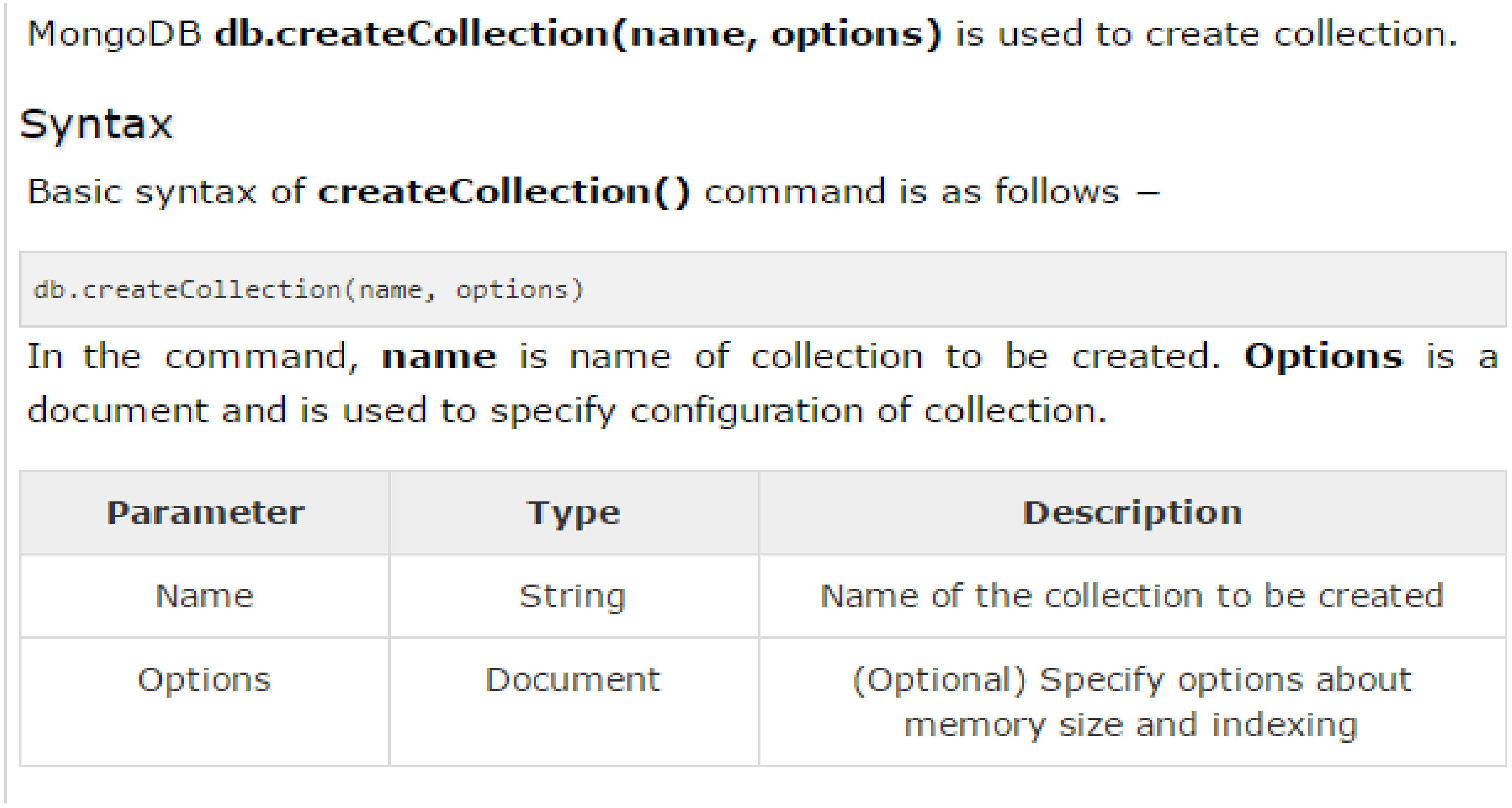
If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows −

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local      0.78125GB
test       0.23012GB
>
```

The createCollection() Method



MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

Basic syntax of **createCollection()** command is as follows −

```
db.createCollection(name, options)
```

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional

Examples

Basic syntax of **createCollection()** method without options is as follows –

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

You can check the created collection by using the command **show collections.**

```
>show collections
mycollection
system.indexes
```

In MongoDB, we don't need to create collection. MongoDB creates collection automatically, when we insert some document.

```
>db.mitcollection.insert({"name" : “ICT"})


>show collections
mycol
mycollection
system.indexes
mitcollection
>
```

The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

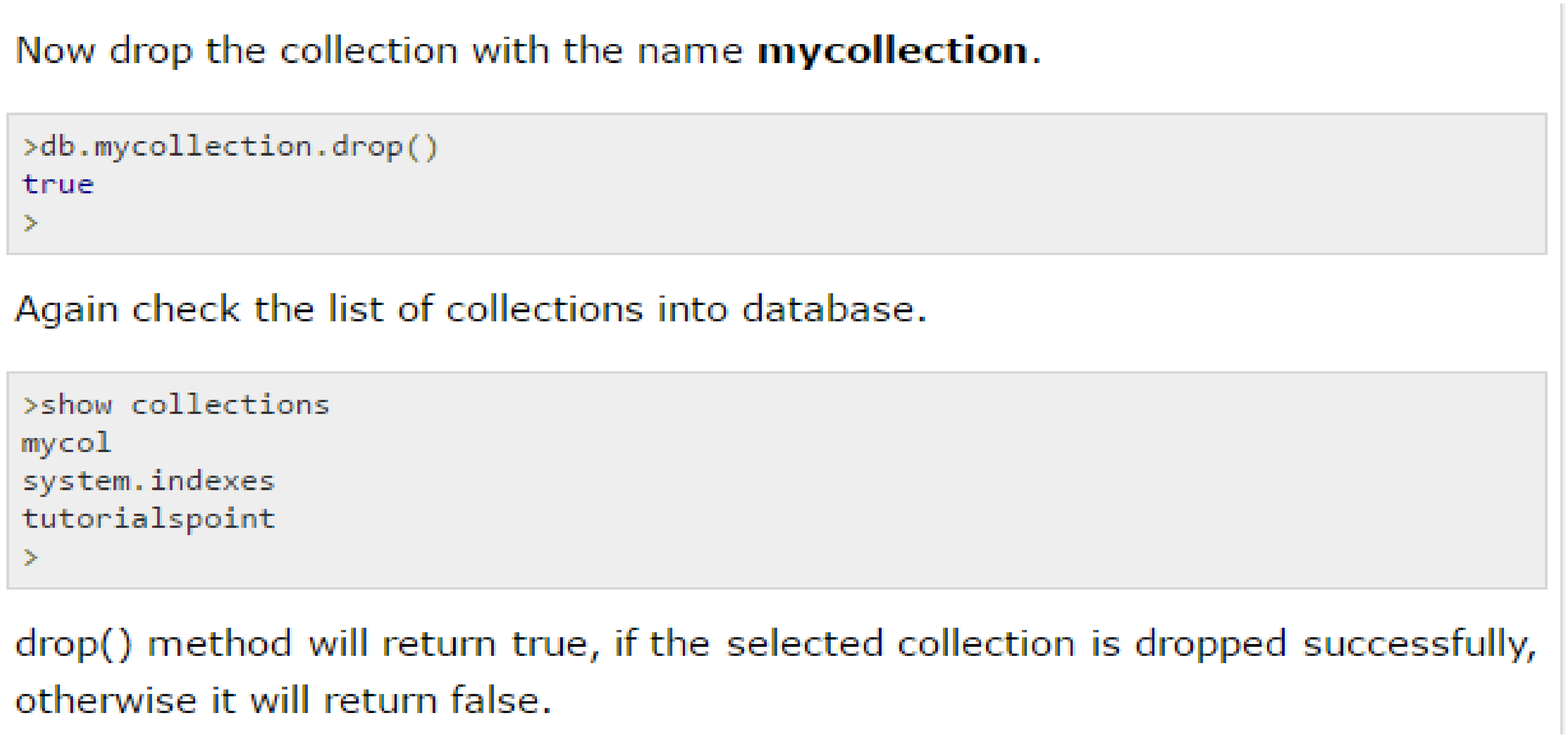
Basic syntax of **drop()** command is as follows −

```
db.COLLECTION_NAME.drop()
```

Example

First, check the available collections into your database **mydb**.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```



Now drop the collection with the name **mycollection**.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
tutorialspoint
>
```

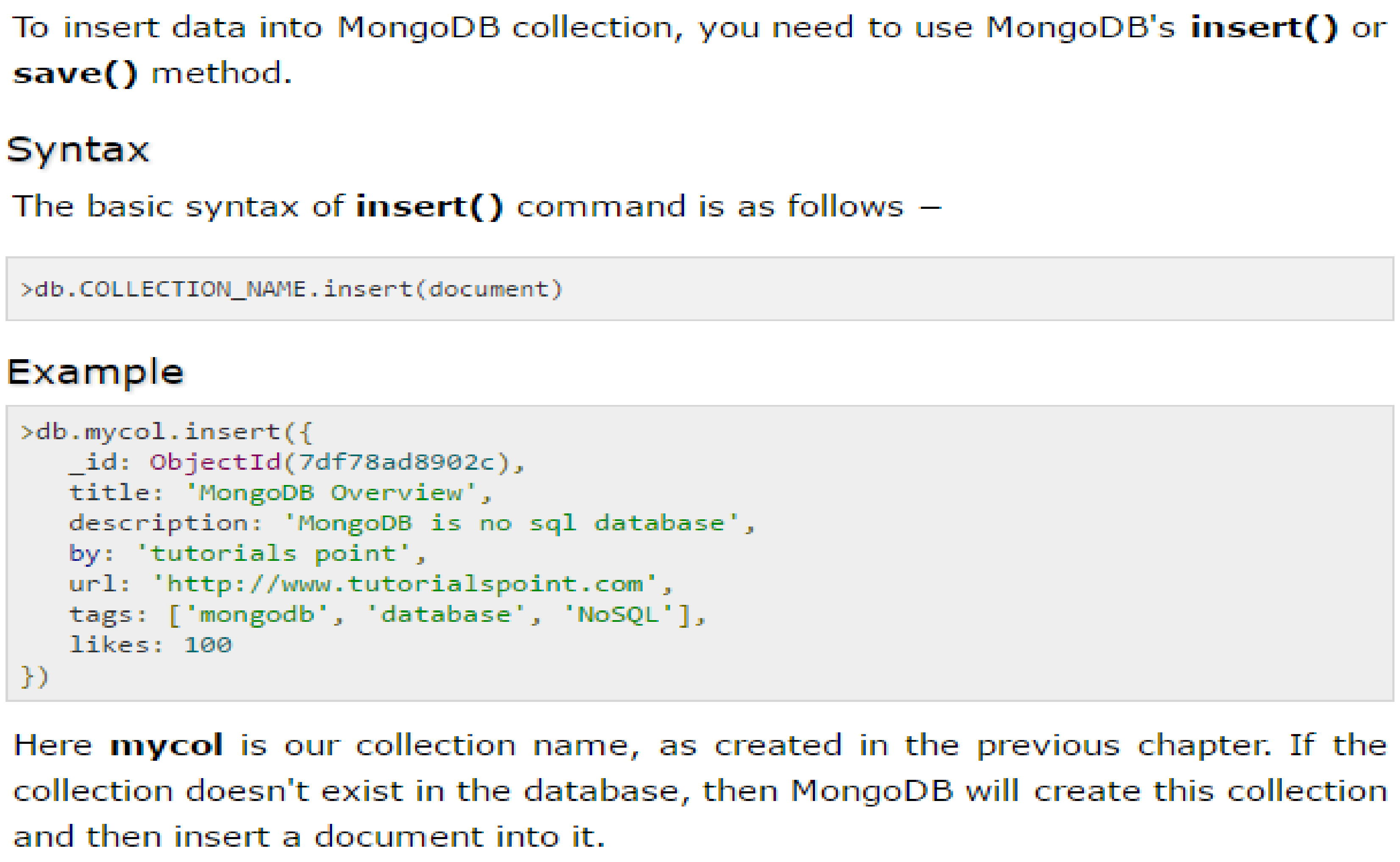
drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

MongoDB - Datatypes

- String
- Integer
- Boolean
- Double
- Arrays
- Timestamp
- Object.

And more

The insert() Method



To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

Syntax

The basic syntax of **insert()** command is as follows −

```
>db.COLLECTION_NAME.insert(document)
```

Example

```
>db.mycol.insert({
   _id: ObjectId(7df78ad8902c),
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
})
```

Here **mycol** is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.

_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows –

```
_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id,
   3 bytes incrementer)
```

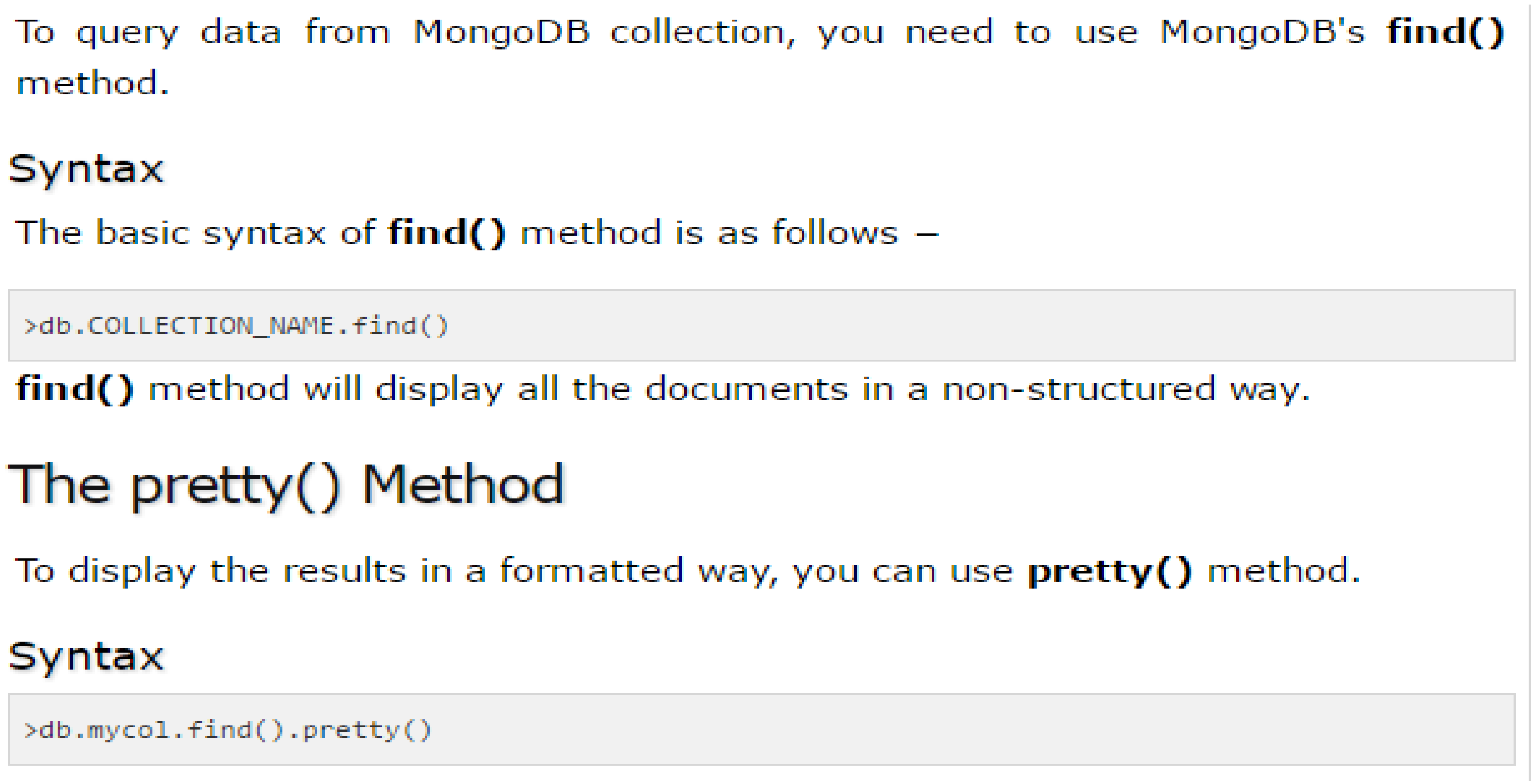
To insert multiple documents in a single query, you can pass an array of documents in insert() command.

Example

```
>db.post.insert([
   {
      title: 'MongoDB Overview',
      description: 'MongoDB is no sql database',
      by: 'tutorials point',
      url: 'http://www.tutorialspoint.com',
      tags: ['mongodb', 'database', 'NoSQL'],
      likes: 100
   },

   {
      title: 'NoSQL Database',
      description: 'NoSQL database doesn't have tables',
      by: 'tutorials point',
      url: 'http://www.tutorialspoint.com',
      tags: ['mongodb', 'database', 'NoSQL'],
      likes: 20,
      comments: [
         {
            user:'user1',
            message: 'My first comment',
            dateCreated: new Date(2013,11,10,2,35),
            like: 0
         }
      ]
   }
])
```

MongoDB - Query Document

The find() Method



To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows −

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

The pretty() Method

To display the results in a formatted way, you can use **pretty()** method.

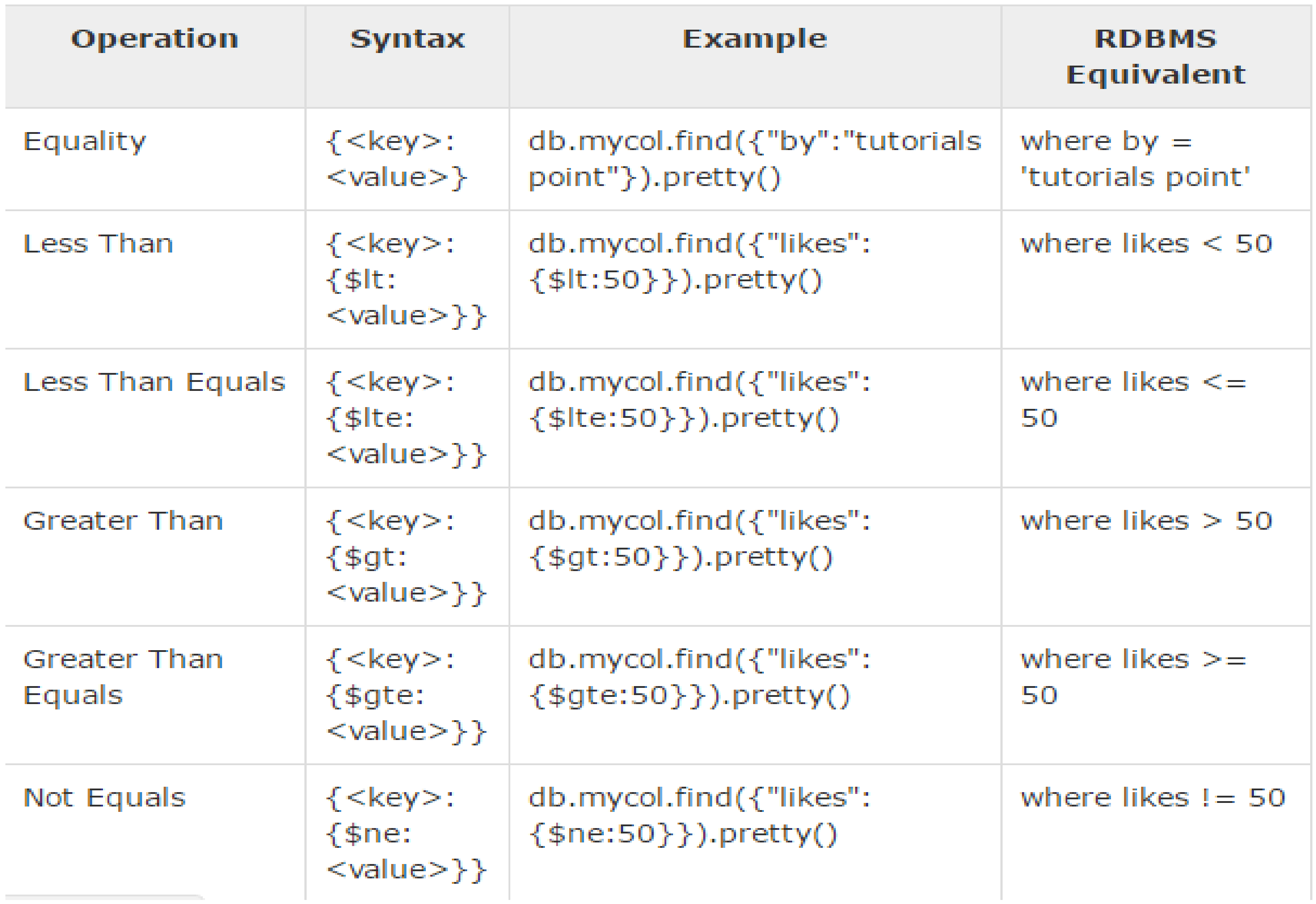
Syntax

```
>db.mycol.find().pretty()
```

Example

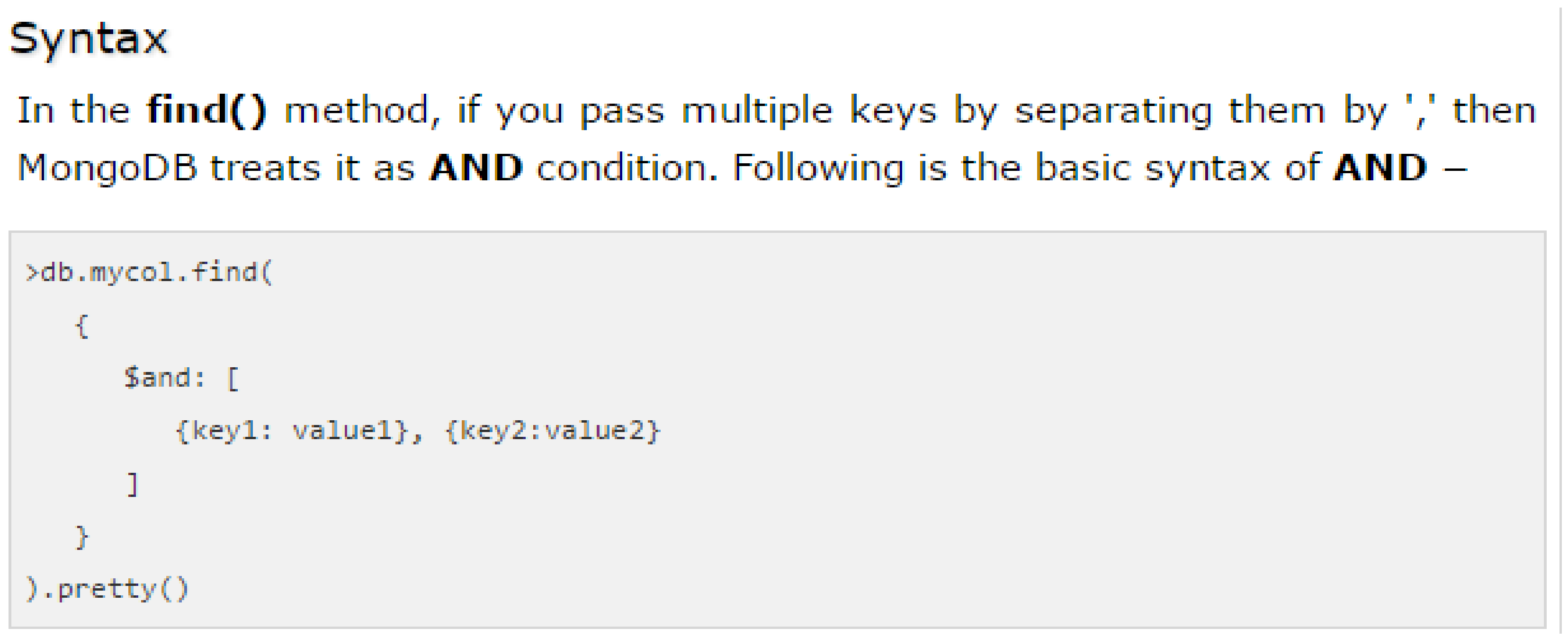
```
>db.mycol.find().pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

RDBMS Where Clause Equivalents in MongoDB



Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

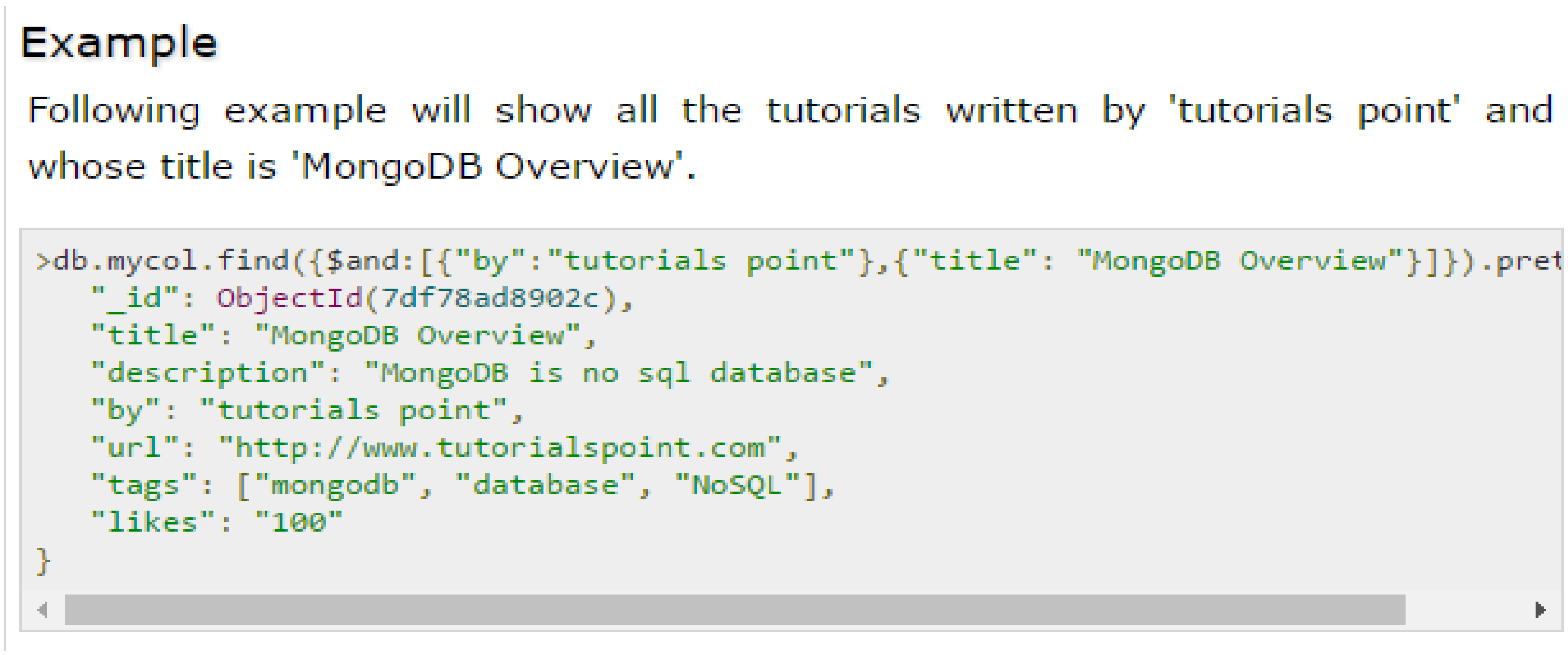
AND in MongoDB



Syntax

In the **find()** method, if you pass multiple keys by separating them by ',' then MongoDB treats it as **AND** condition. Following is the basic syntax of **AND** –

```
>db.mycol.find(
   {
      $and: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```

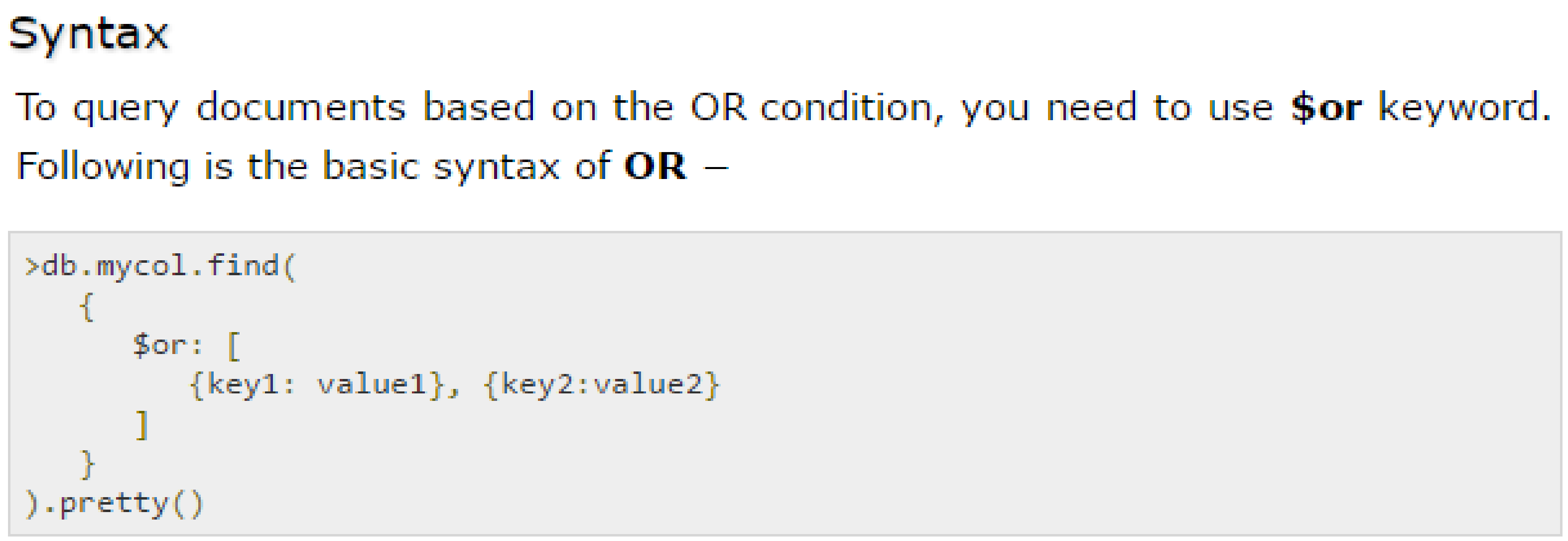


Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
>db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pret
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
```

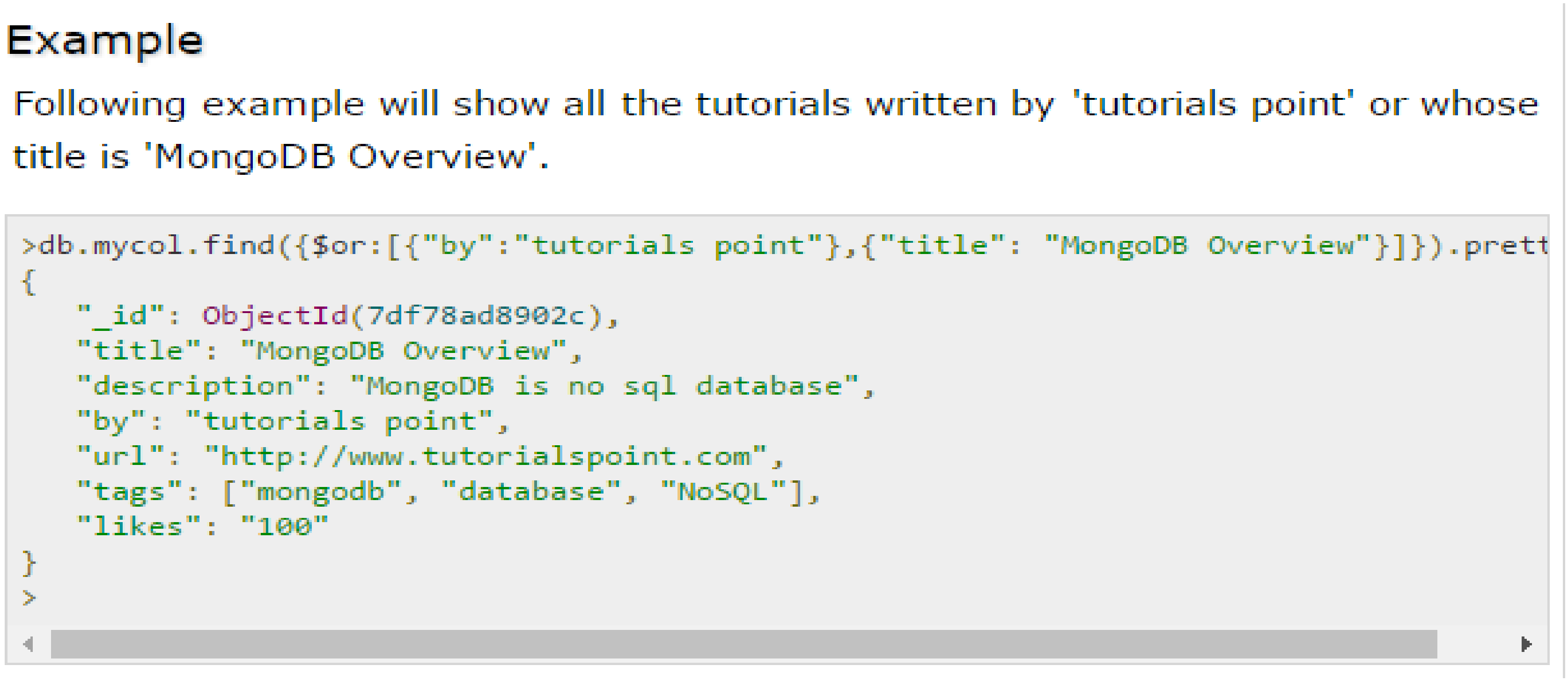
OR in MongoDB



Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** –

```
>db.mycol.find(
   {
      $or: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```



Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).prett
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Using AND and OR Together

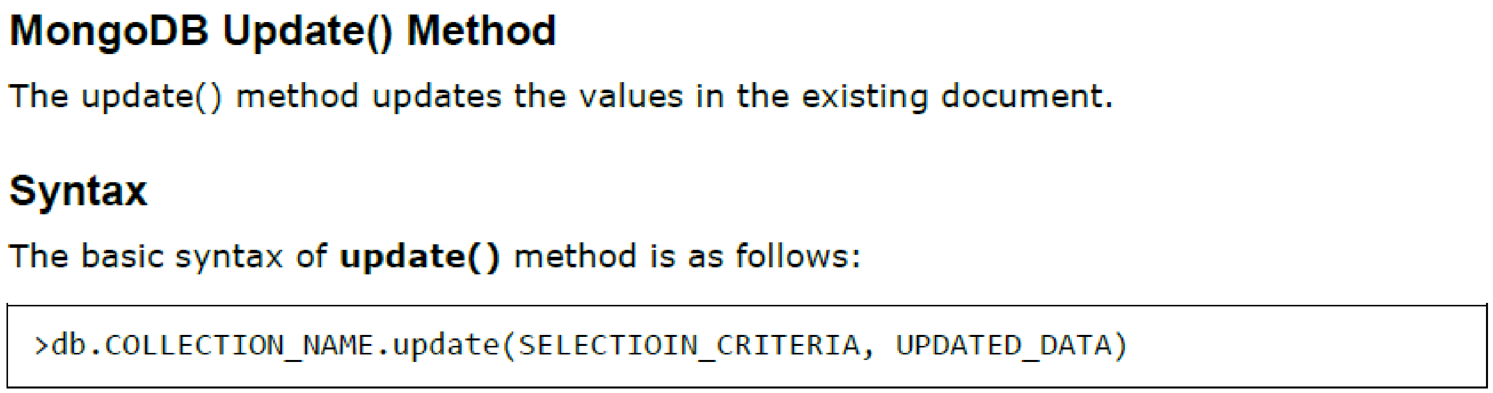
The following example will show the documents that have likes greater than 100 and whose title is either 'MongoDB Overview' or by is 'tutorials point'.

Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100" }
>
```

Update Document

MongoDB's **update()** and **save()** methods are used to update document into a collection.
The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.



MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of **update()** method is as follows:

```
>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)
```

Example

Consider the mycol collection has the following data.

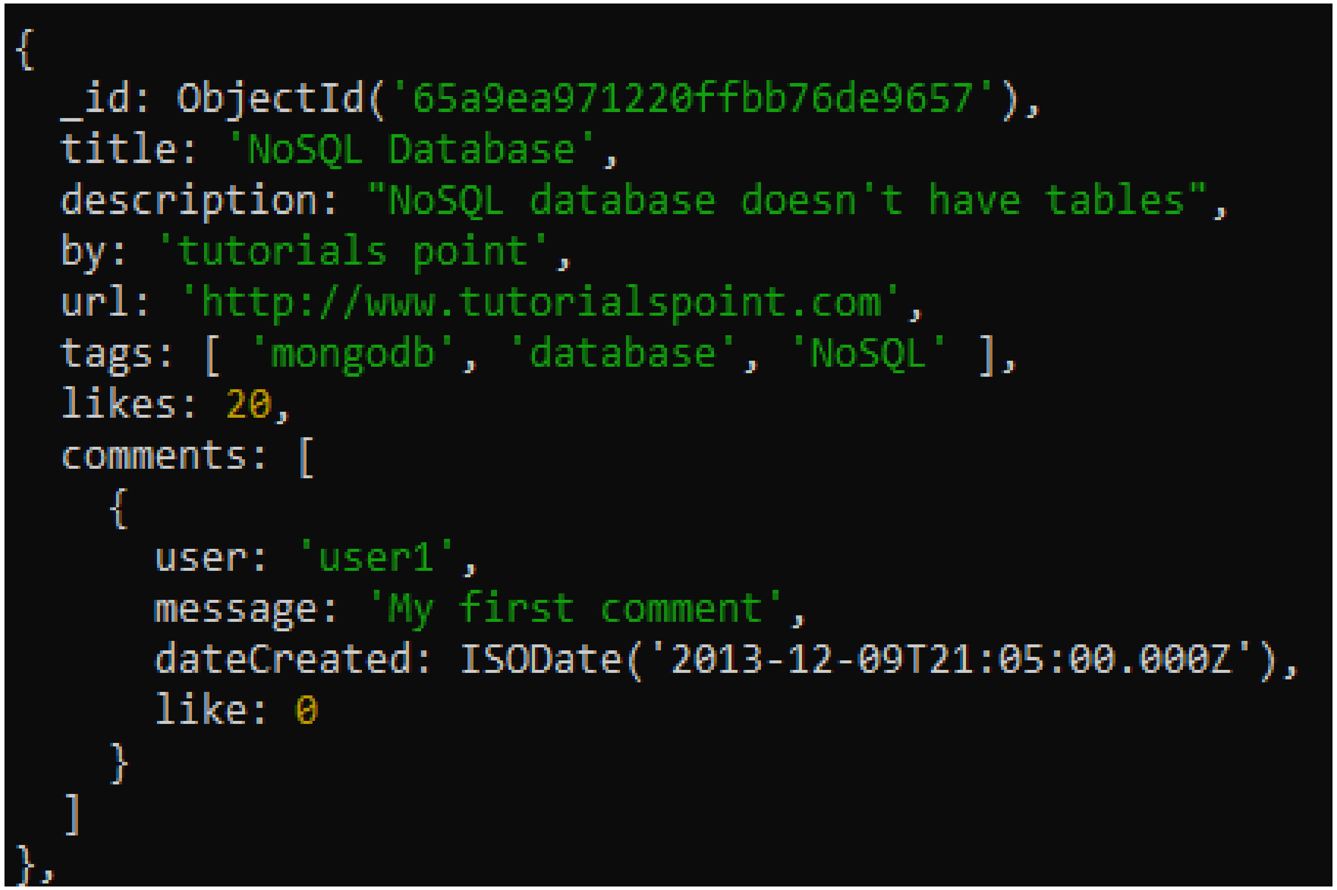
```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB
Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

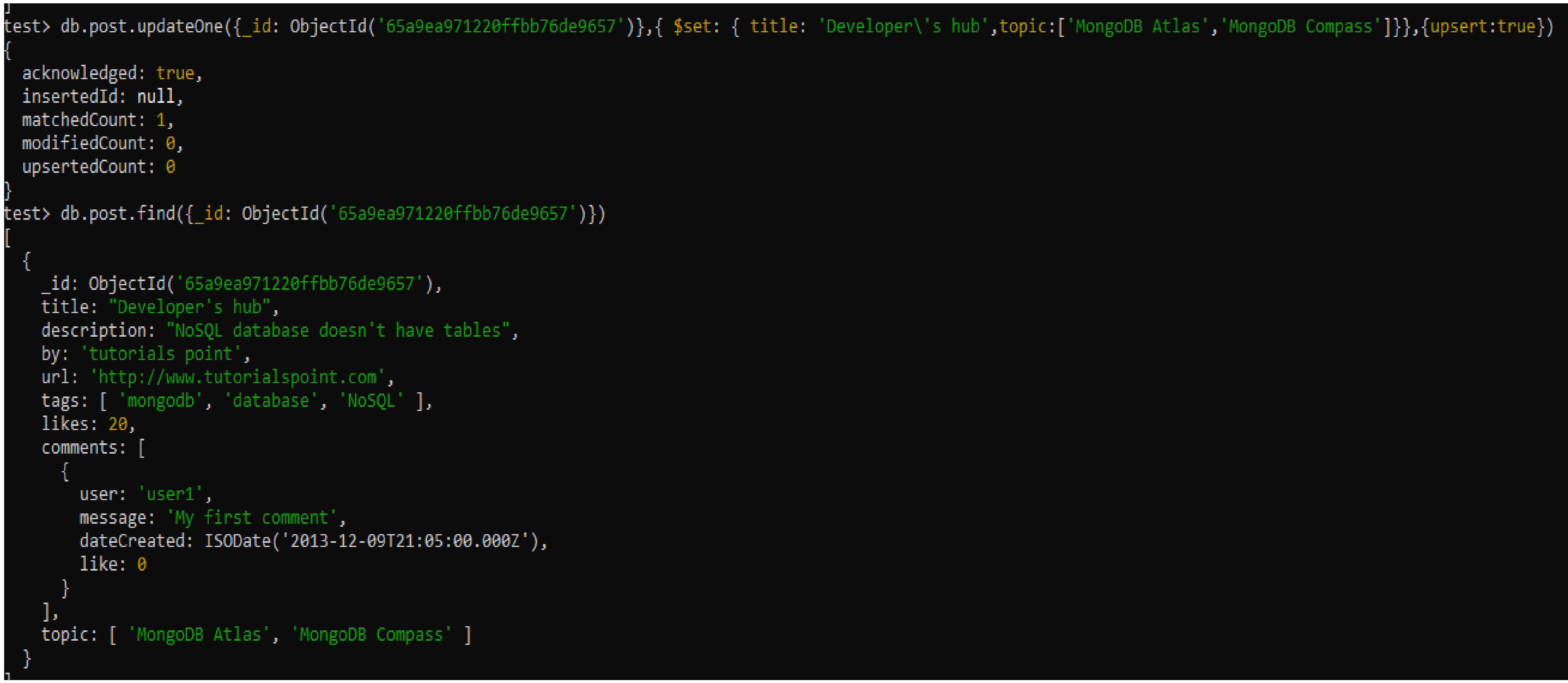
By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
   {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```



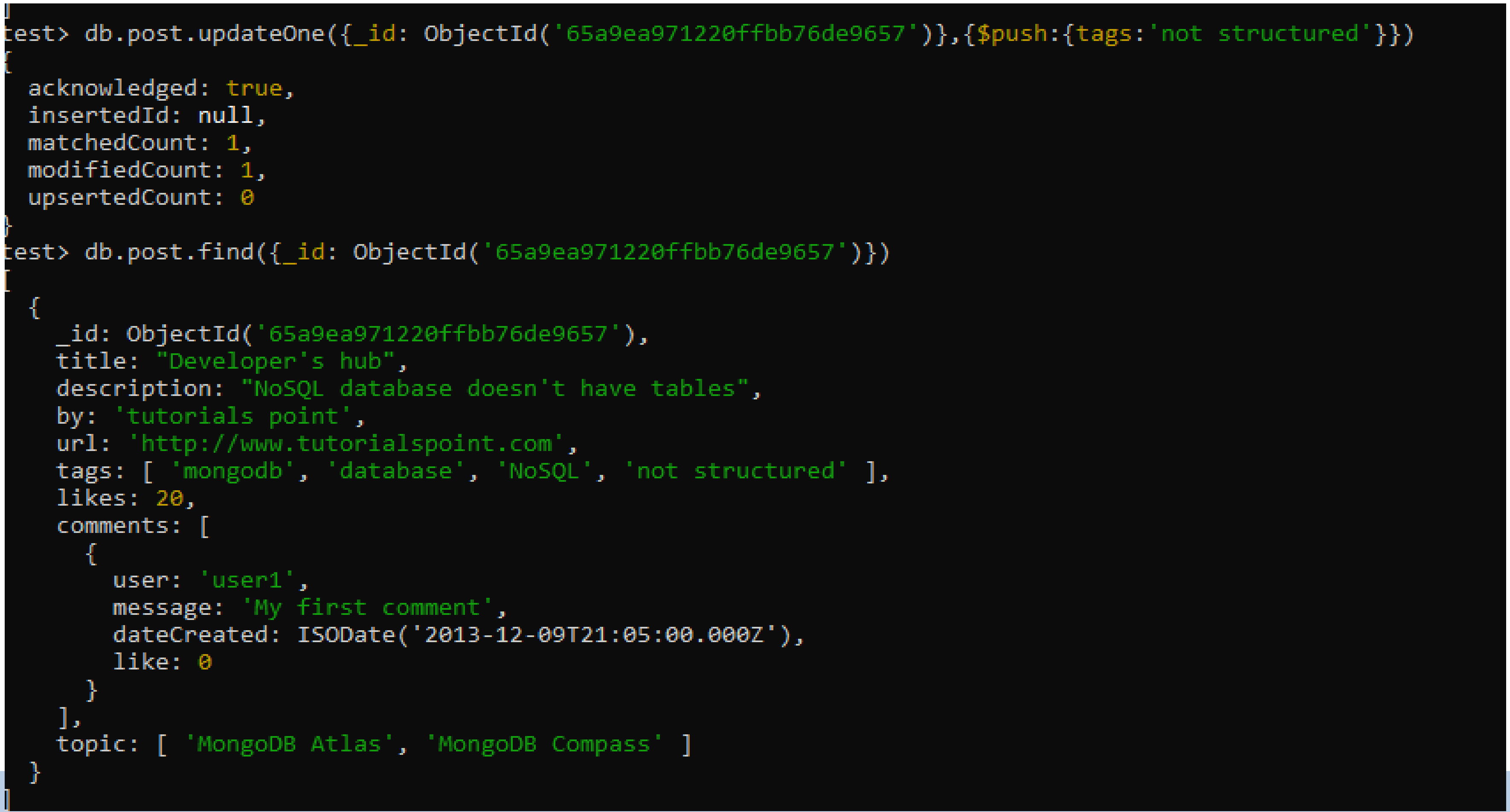
```
{
  _id: ObjectId('65a9ea971220ffbb76de9657'),
  title: 'NoSQL Database',
  description: "NoSQL database doesn't have tables",
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: [ 'mongodb', 'database', 'NoSQL' ],
  likes: 20,
  comments: [
    {
      user: 'user1',
      message: 'My first comment',
      dateCreated: ISODate('2013-12-09T21:05:00.000Z'),
      like: 0
    }
  ]
},
```

db.post.updateOne({_id: ObjectId('65a9ea971220ffbb76de9657')},{ $set: { title: 'Developer\'s hub',topic:['MongoDB Atlas','MongoDB Compass']}},{upsert:true})



```
test> db.post.updateOne({_id: ObjectId('65a9ea971220ffbb76de9657')},{ $set: { title: 'Developer\'s hub',topic:['MongoDB Atlas','MongoDB Compass']}},{upsert:true})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 0,
  upsertedCount: 0
}
test> db.post.find({_id: ObjectId('65a9ea971220ffbb76de9657')})
[
  {
    _id: ObjectId('65a9ea971220ffbb76de9657'),
    title: "Developer's hub",
    description: "NoSQL database doesn't have tables",
    by: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: [ 'mongodb', 'database', 'NoSQL' ],
    likes: 20,
    comments: [
      {
        user: 'user1',
        message: 'My first comment',
        dateCreated: ISODate('2013-12-09T21:05:00.000Z'),
        like: 0
      }
    ],
    topic: [ 'MongoDB Atlas', 'MongoDB Compass' ]
  }
```

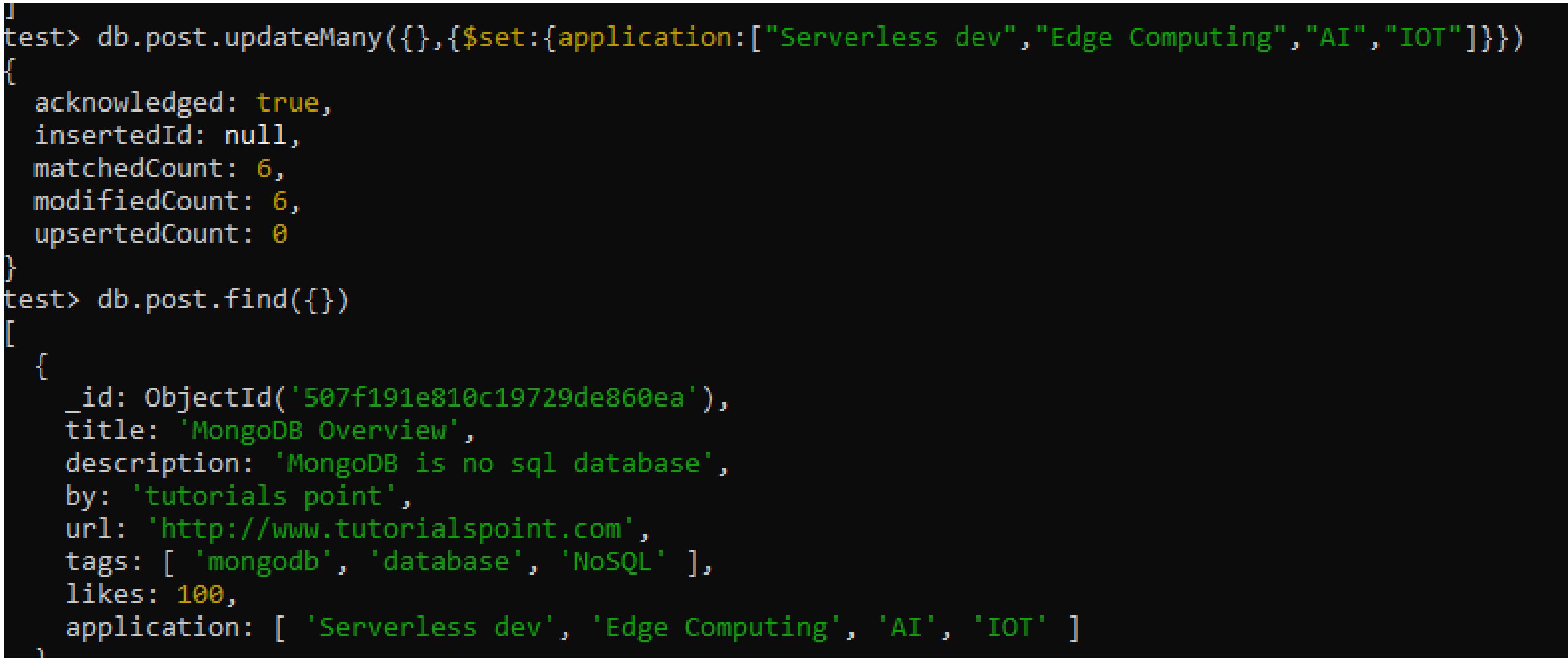
db.post.updateOne({_id: ObjectId('65a9ea971220ffbb76de9657')},{$push:{tags:'not structured'}})



```
test> db.post.updateOne({_id: ObjectId('65a9ea971220ffbb76de9657')},{$push:{tags:'not structured'}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
test> db.post.find({_id: ObjectId('65a9ea971220ffbb76de9657')})
[
  {
    _id: ObjectId('65a9ea971220ffbb76de9657'),
    title: "Developer's hub",
    description: "NoSQL database doesn't have tables",
    by: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: [ 'mongodb', 'database', 'NoSQL', 'not structured' ],
    likes: 20,
    comments: [
      {
        user: 'user1',
        message: 'My first comment',
        dateCreated: ISODate('2013-12-09T21:05:00.000Z'),
        like: 0
      }
    ],
    topic: [ 'MongoDB Atlas', 'MongoDB Compass' ]
  }
]
```

- **The updateOne() method accepts a filter document, an update document, and an optional options object. MongoDB provides update operators and options to help you update documents.**
- The **$set** operator replaces the value of a field with the specified value
- The **upsert** option creates a new document if no documents match the filtered criteria
- The **$push** operator adds a new value to the hosts array field

- db.post.updateMany({},{$set:{application:["Serverless dev","Edge Computing","AI","IOT"]}})



```
test> db.post.updateMany({},{$set:{application:["Serverless dev","Edge Computing","AI","IOT"]}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 6,
  modifiedCount: 6,
  upsertedCount: 0
}
test> db.post.find({})
[
  {
    _id: ObjectId('507f191e810c19729de860ea'),
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: [ 'mongodb', 'database', 'NoSQL' ],
    likes: 100,
    application: [ 'Serverless dev', 'Edge Computing', 'AI', 'IOT' ]
```

MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB **save()** method is –

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Following example will replace the document with the _id '5983548781331adf45ec7'.

```
>db.mycol.save(

   {

      "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point New
Topic",

         "by":"Tutorials Point"

   }

)
```

Delete Document

The remove() Method

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria:** (Optional) deletion criteria according to documents will be removed.

- **justOne**: (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of **remove()** method is as follows:

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'.

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

Remove Only One

If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

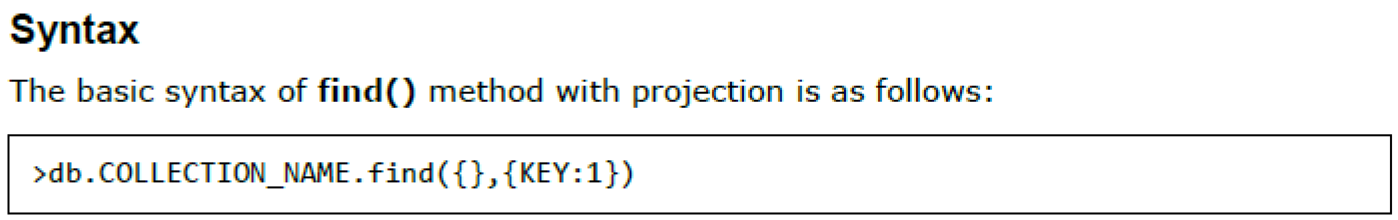
```
>db.mycol.remove()

>db.mycol.find()

>
```

Projection

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

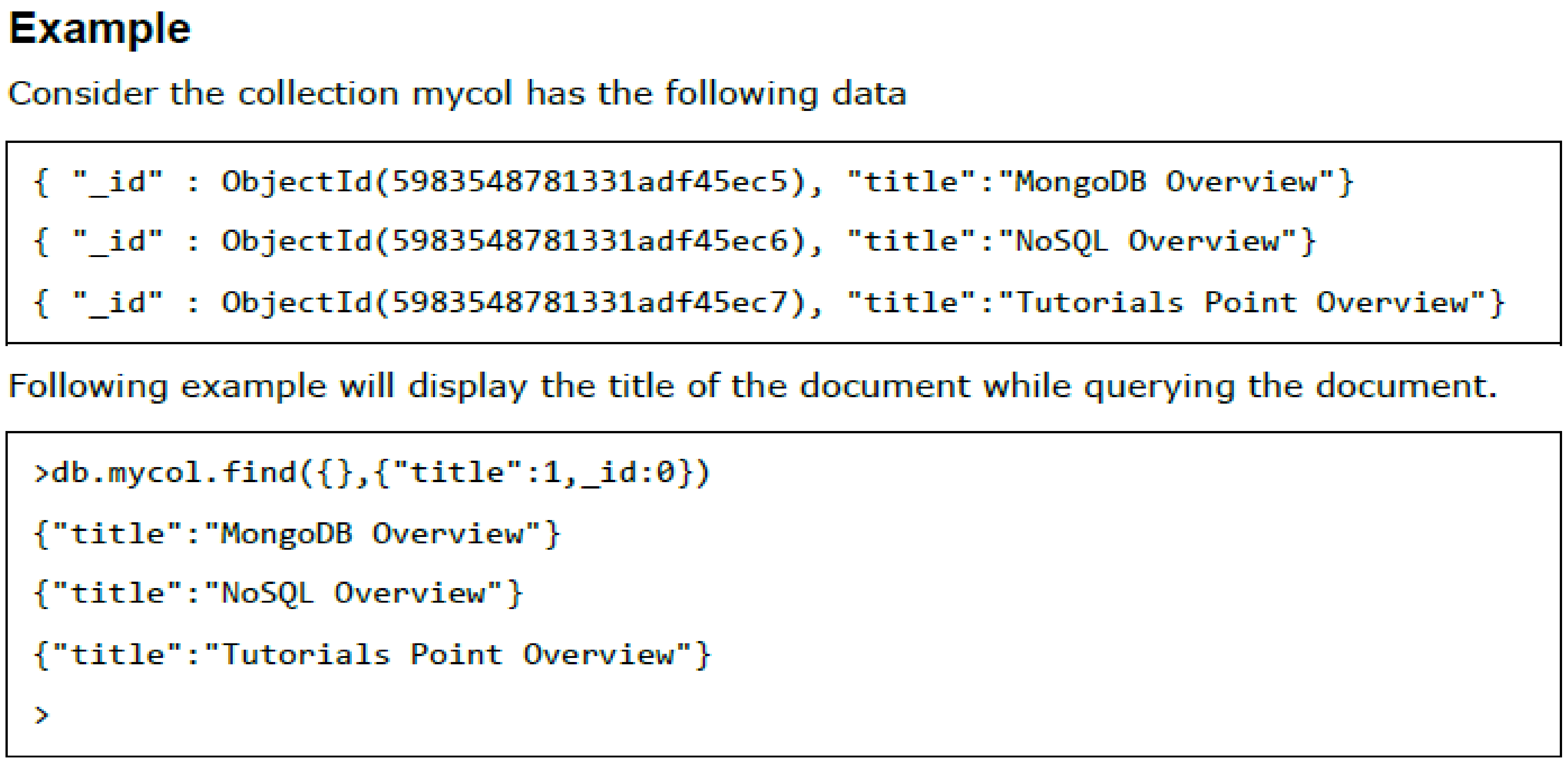
MongoDB's **find()** method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.



Syntax

The basic syntax of **find()** method with projection is as follows:

```
>db.COLLECTION_NAME.find({},{KEY:1})
```



Example

Consider the collection mycol has the following data

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

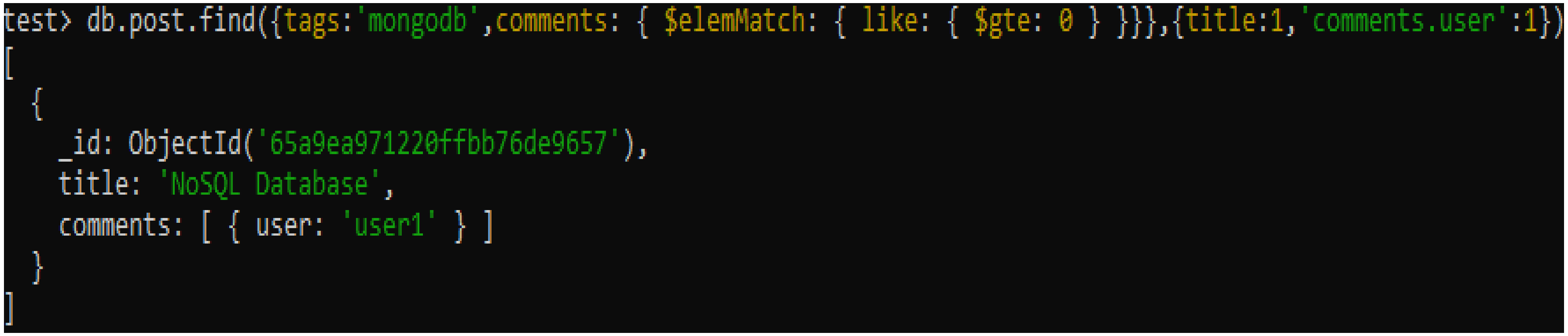
Following example will display the title of the document while querying the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

Please note **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

Querying on Array Elements

- { <field>: { $elemMatch: { <query1>, <query2>, ... } } }
- The $elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria
- db.post.find({tags:'mongodb',comments: { $elemMatch: { like: { $gte: 0 } }}})
- db.post.find({tags:'mongodb',comments: { $elemMatch: { like: { $gte: 0 } }}},{title:1,'comments.user':1})



```
test> db.post.find({tags:'mongodb',comments: { $elemMatch: { like: { $gte: 0 } }}},{title:1,'comments.user':1})
[
  {
    _id: ObjectId('65a9ea971220ffbb76de9657'),
    title: 'NoSQL Database',
    comments: [ { user: 'user1' } ]
  }
]
```

Limit Records

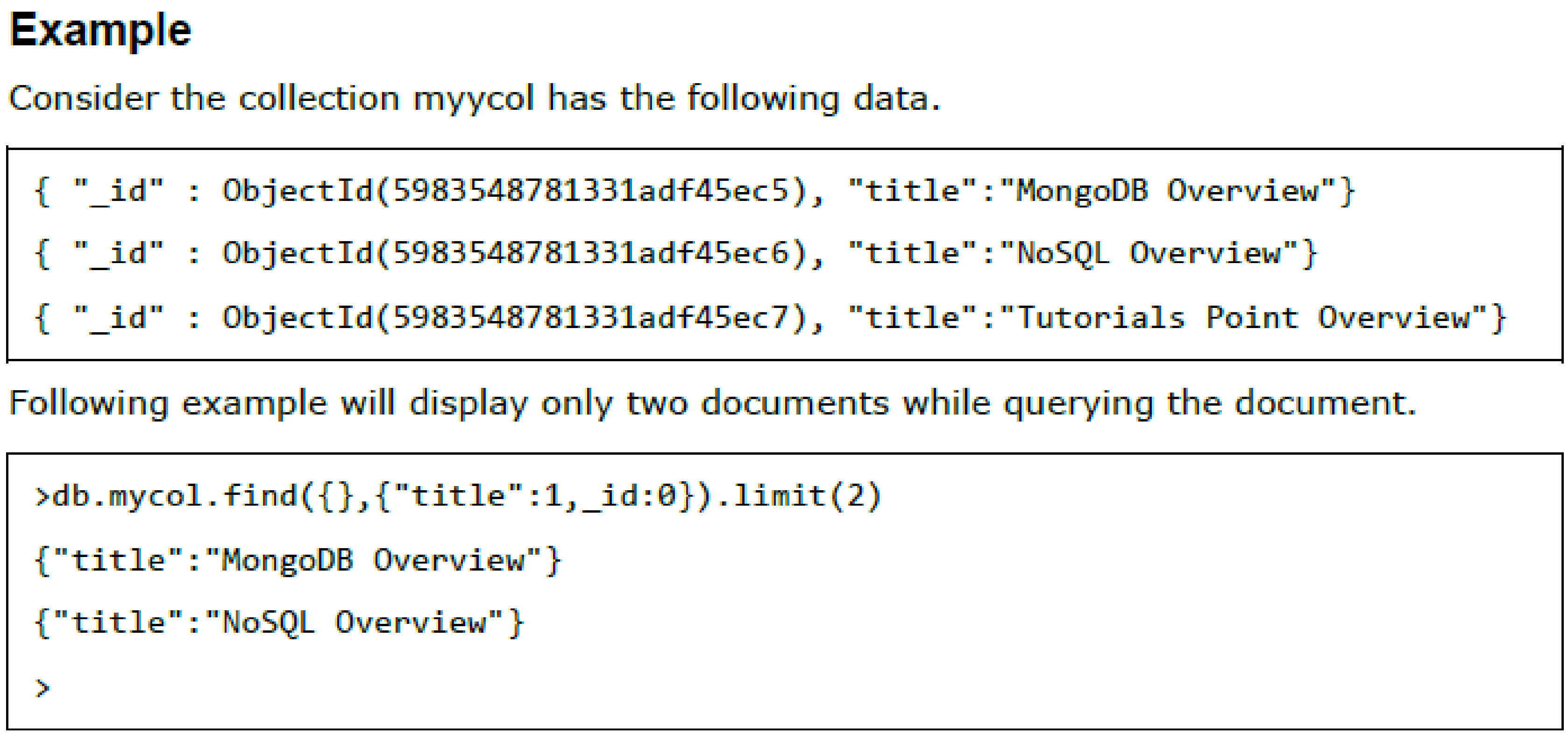
The Limit() Method

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of **limit()** method is as follows:

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```



Example

Consider the collection myycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

If you don't specify the number argument in **limit()** method then it will display all documents from the collection.

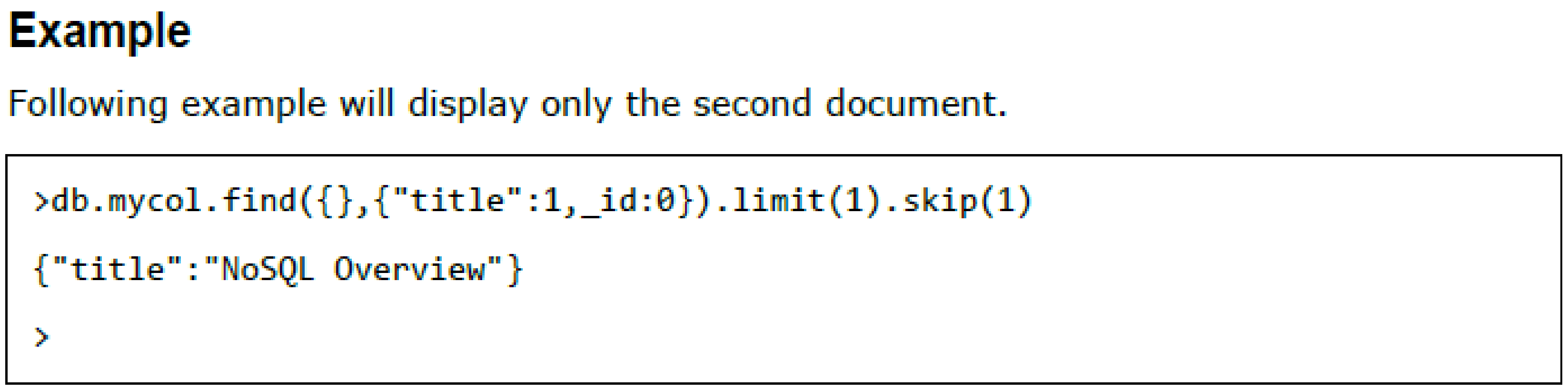
MongoDB Skip() Method

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows:

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```



Example

Following example will display only the second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

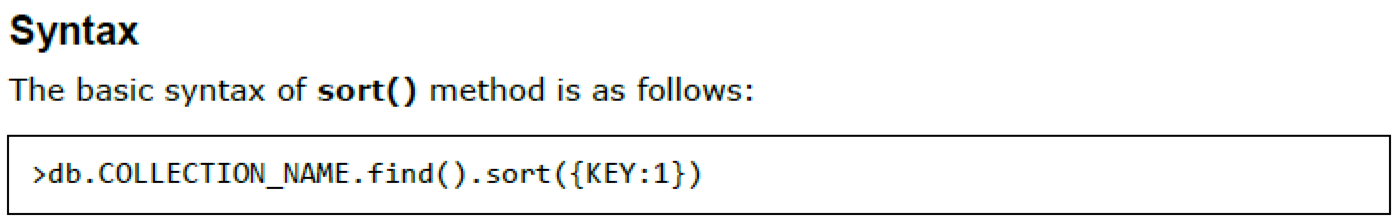
Please note, the default value in **skip()** method is 0.

Sort Records

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order.

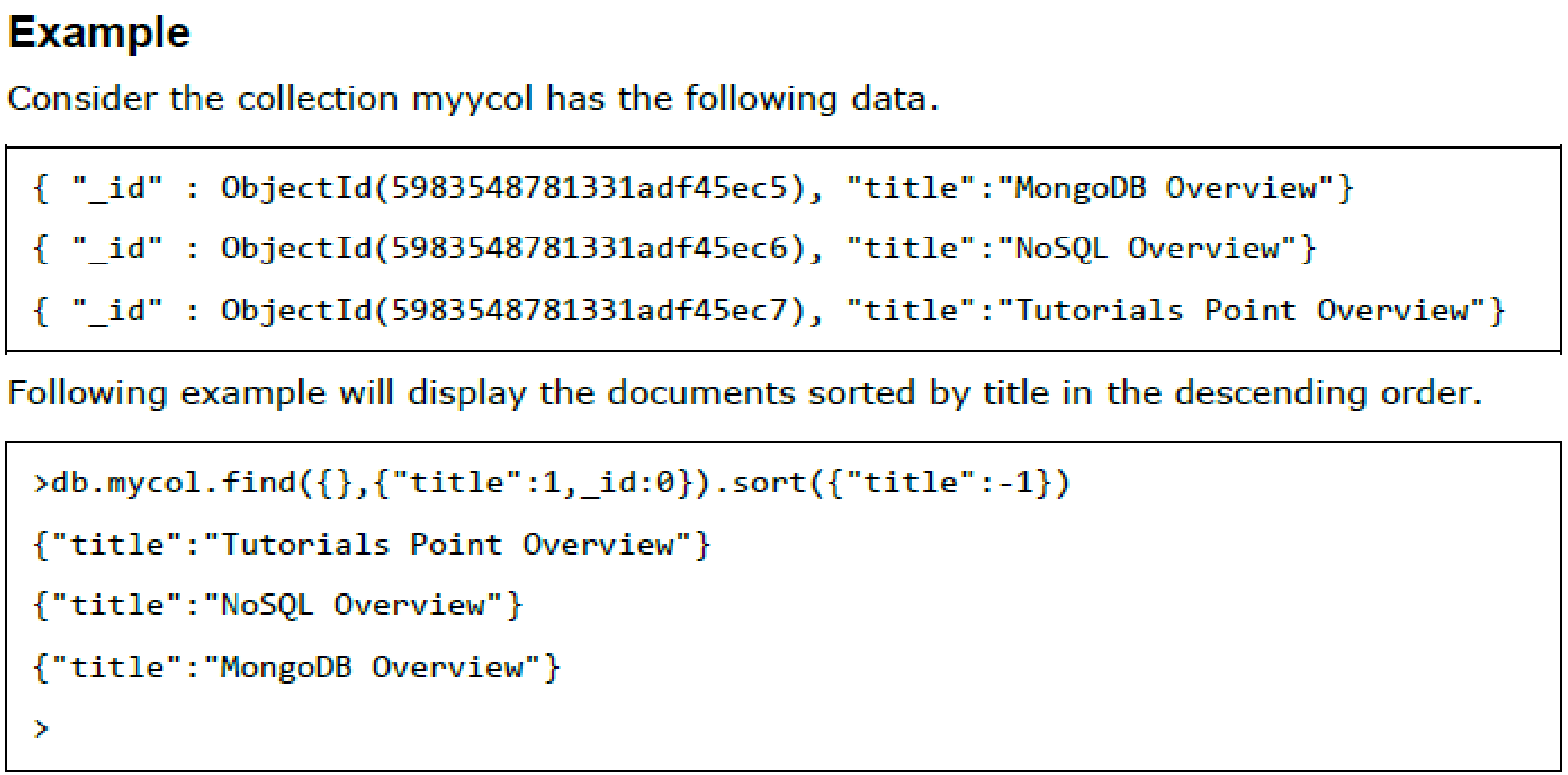
To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.



Syntax

The basic syntax of **sort()** method is as follows:

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```



Example

Consider the collection myycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

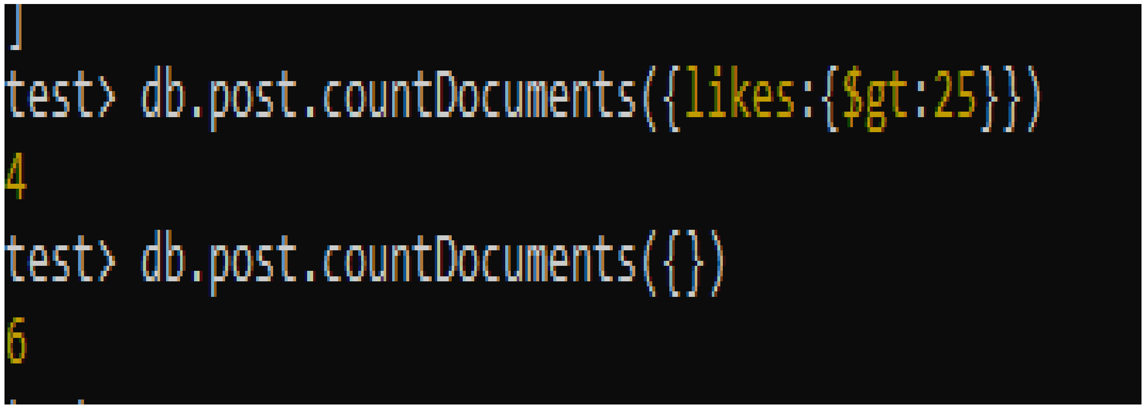
Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Please note, if you don't specify the sorting preference, then **sort()** method will display the documents in ascending order.

Count Documents

- **db.collection.countDocuments(<query>, <options>)**
- db.post.countDocuments({likes:{$gt:25}})



```
]
test> db.post.countDocuments({likes:{$gt:25}})
4
test> db.post.countDocuments({})
6
```

Aggregation Pipeline

- **Aggregation**: Collection and summary of data
- **Stage**: One of the built-in methods that can be completed on the data, but does not permanently alter it
- **Aggregation pipeline**: A series of stages completed on the data in order

```
db.collection.aggregate([
  {
    $stage1: {
      { expression1 },
      { expression2 }...
    },
    $stage2: {
      { expression1 }...
    }
  }
])
```

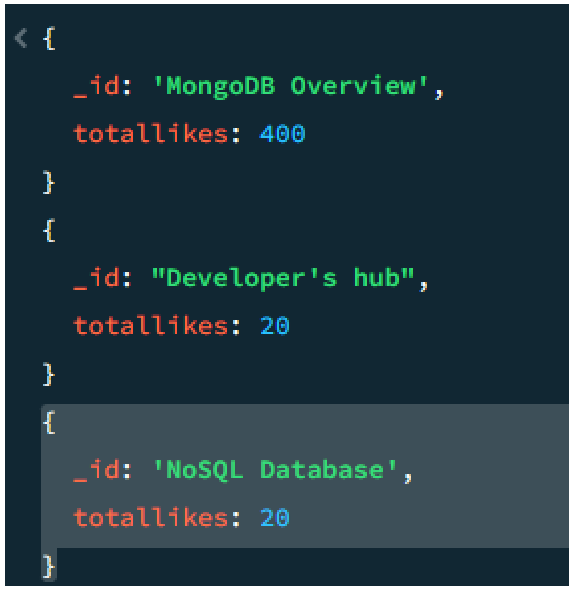
- **$match and $group aggregation**
- The **$match** stage filters for documents that match specified conditions
- The **$group** stage groups documents by a group key

```
{
  $match: {
    "field_name": "value"
  }
}
```

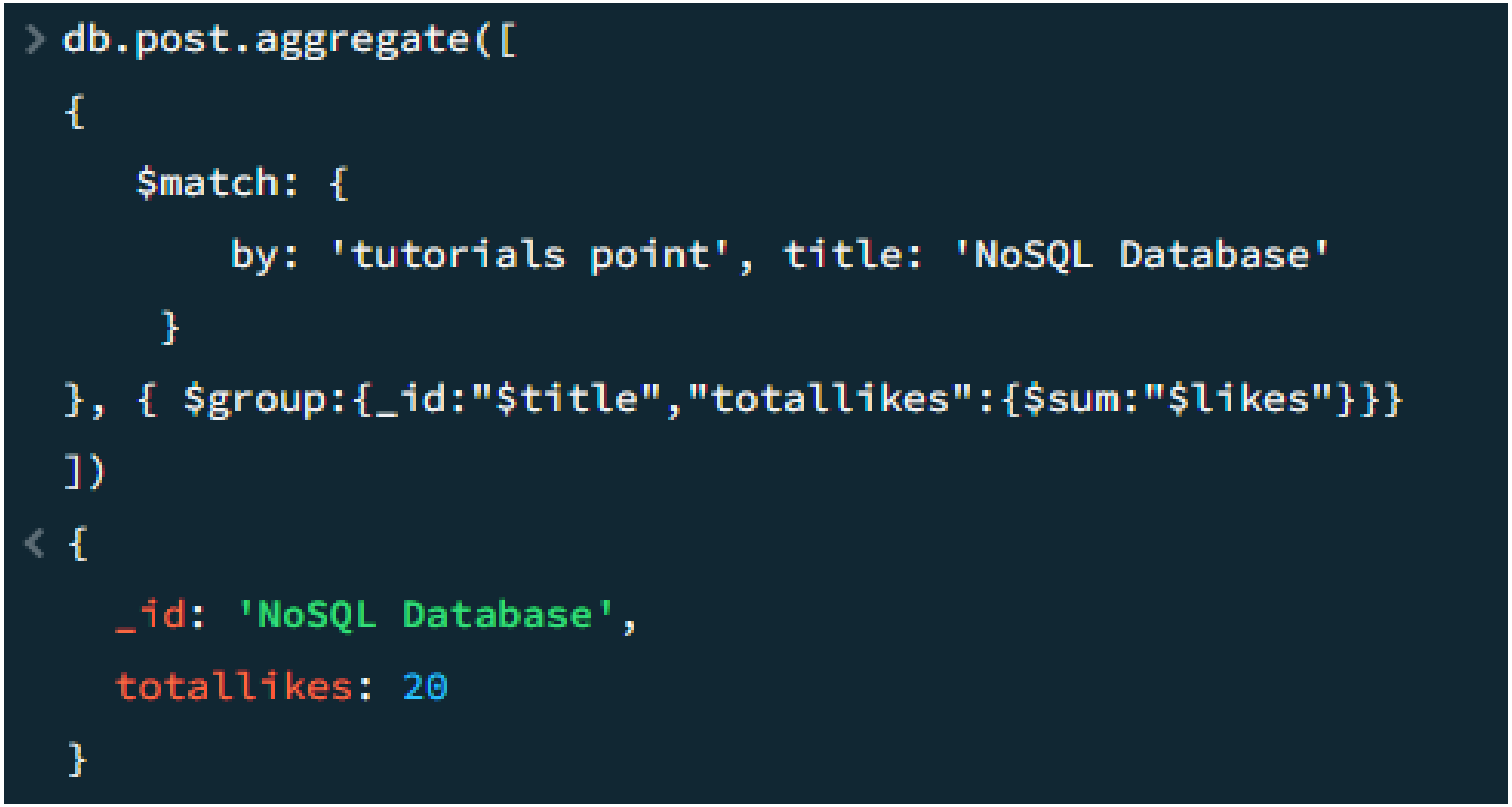
- {
- $group:
- {
- _id: <expression>, // Group key
- <field>: { <accumulator> : <expression> }
- }
- }

- db.post.aggregate([
- {
-    $match: {
-        by: 'tutorials point'
-     }
- }, { $group:{_id:null,"totallikes":{$sum:"$likes"}}}
-])
- *< { _id: null,*
- *totallikes: 440}*

- db.post.aggregate([
- {
-    $match: {
-        by: 'tutorials point'
-     }
- }, { $group:{_id:"$title","totallikes":{$sum:"$likes"}}}
-])



```
< {
    _id: 'MongoDB Overview',
    totallikes: 400
  }
  {
    _id: "Developer's hub",
    totallikes: 20
  }
  {
    _id: 'NoSQL Database',
    totallikes: 20
  }
```

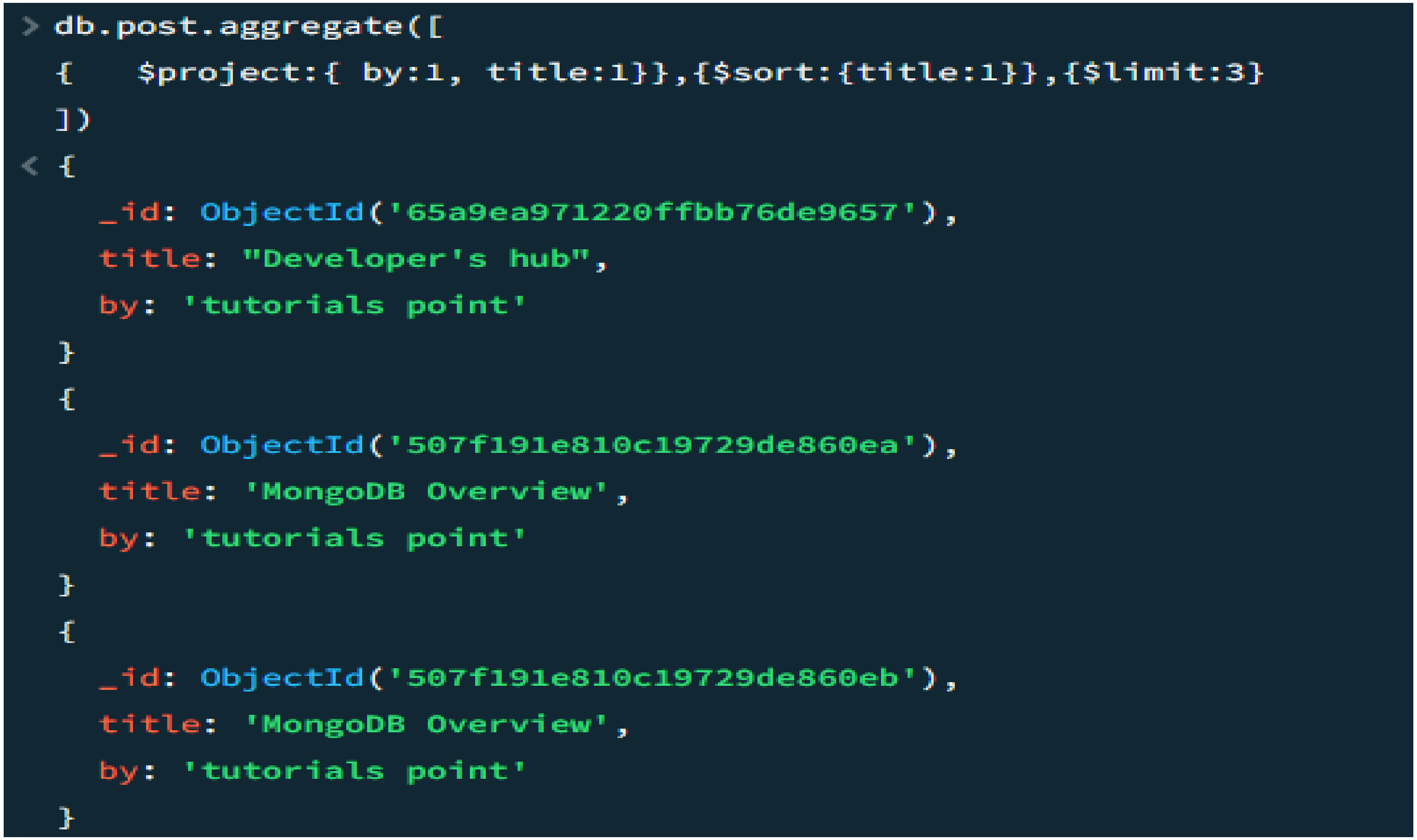


```
> db.post.aggregate([
  {
     $match: {
         by: 'tutorials point', title: 'NoSQL Database'
      }
  }, { $group:{_id:"$title","totallikes":{$sum:"$likes"}}}
  ])
< {
    _id: 'NoSQL Database',
    totallikes: 20
  }
```

Sort and limit

- The **$sort** stage sorts all input documents and returns them to the pipeline in sorted order. Use 1 to represent ascending order, and -1 to represent descending order.
- The **$limit** stage returns only a specified number of records.

Sort and limit

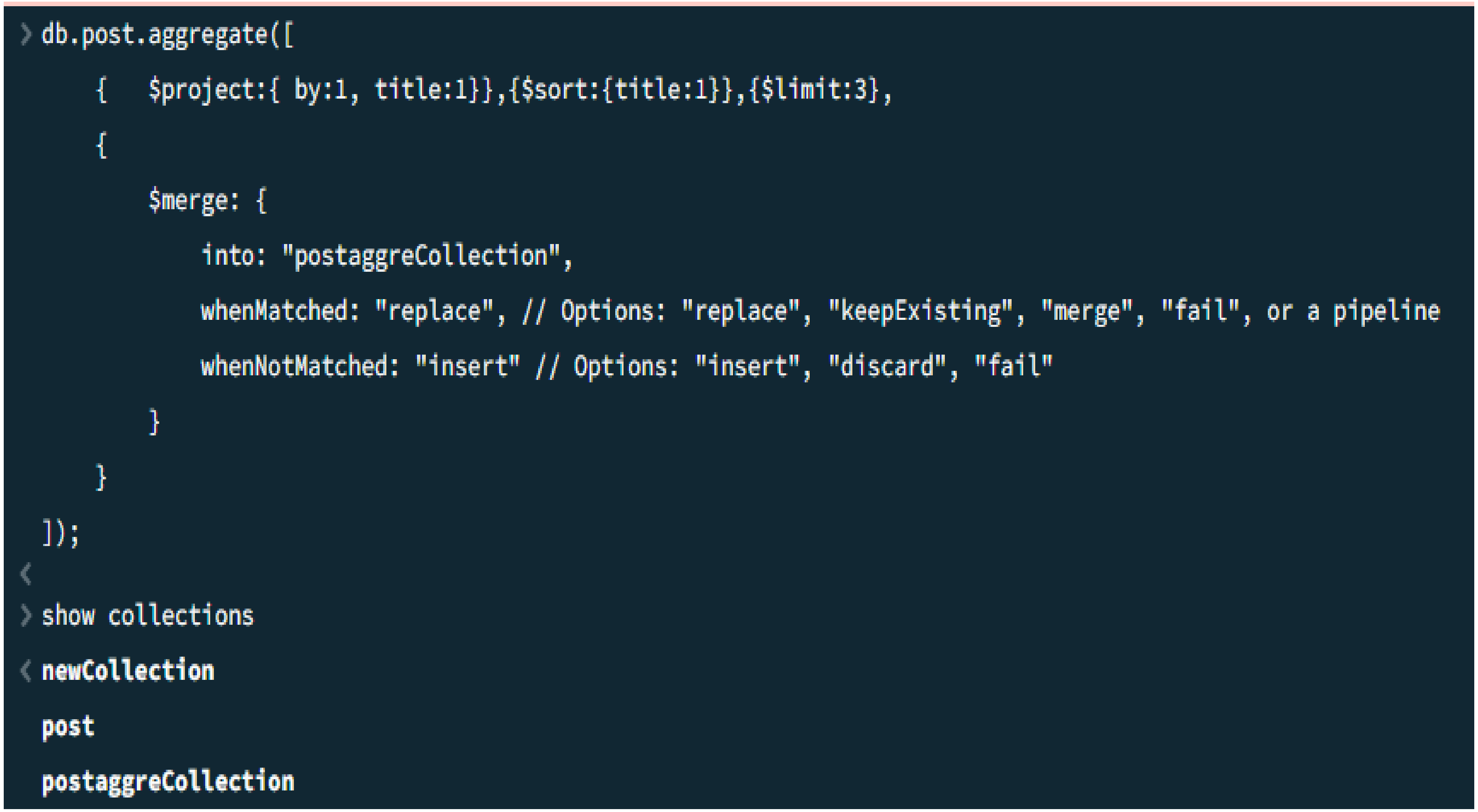


```
> db.post.aggregate([
  {   $project:{ by:1, title:1}},{$sort:{title:1}},{$limit:3}
  ])
< {
    _id: ObjectId('65a9ea971220ffbb76de9657'),
    title: "Developer's hub",
    by: 'tutorials point'
  }
  {
    _id: ObjectId('507f191e810c19729de860ea'),
    title: 'MongoDB Overview',
    by: 'tutorials point'
  }
  {
    _id: ObjectId('507f191e810c19729de860eb'),
    title: 'MongoDB Overview',
    by: 'tutorials point'
  }
```

$merge

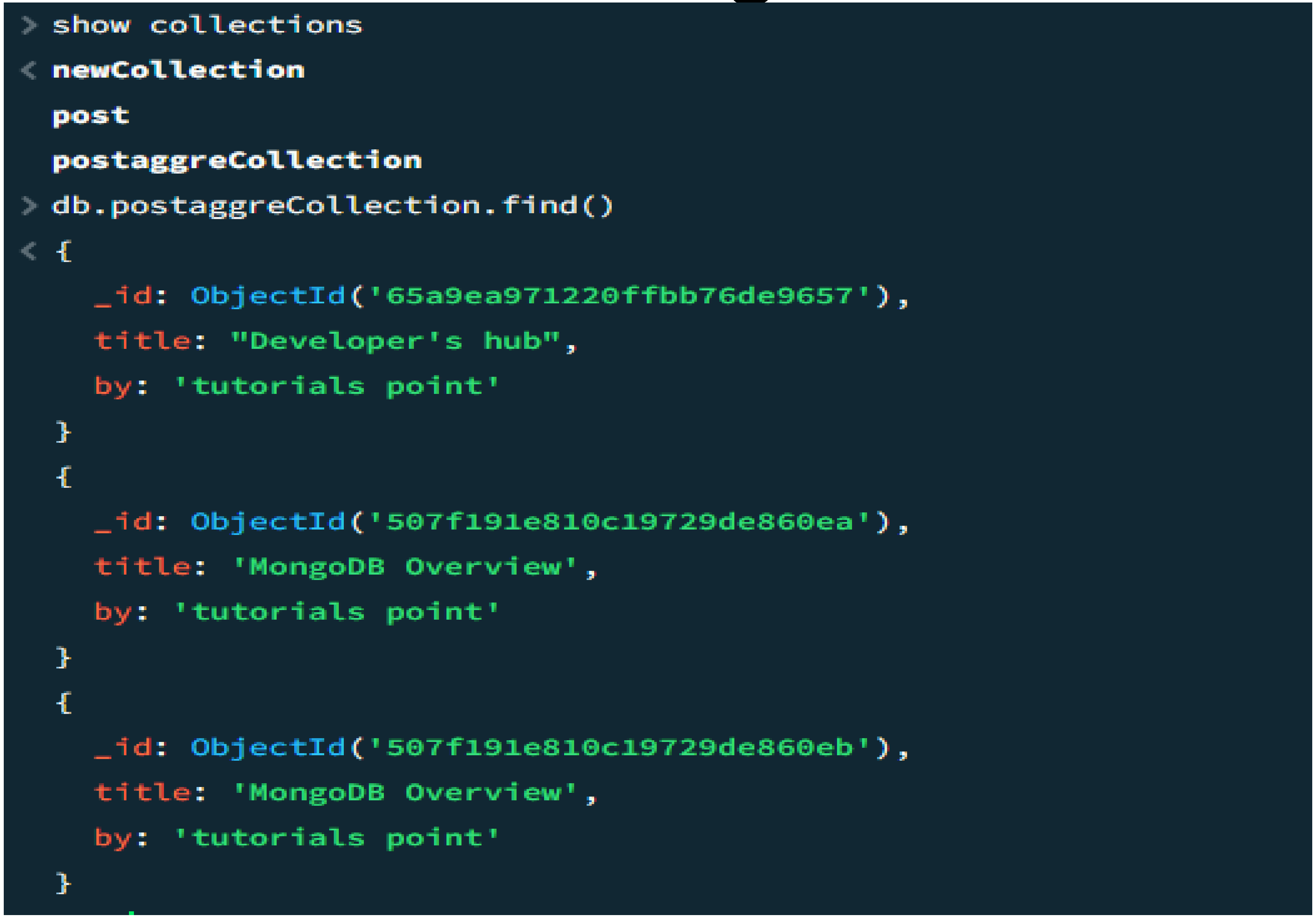
- Merges the output with a specified collection
- The $merge stage provides more flexibility. It can merge the results of the aggregation with an existing collection.
- It allows specifying how the merging should occur, with options like overwriting existing documents, merging them, or even keeping the existing ones if there's a conflict

$merge



```
> db.post.aggregate([
      {   $project:{ by:1, title:1}},{$sort:{title:1}},{$limit:3},
      {
          $merge: {
              into: "postaggreCollection",
              whenMatched: "replace", // Options: "replace", "keepExisting", "merge", "fail", or a pipeline
              whenNotMatched: "insert" // Options: "insert", "discard", "fail"
          }
      }
  ]);
<
> show collections
< newCollection
  post
  postaggreCollection
```

$merge



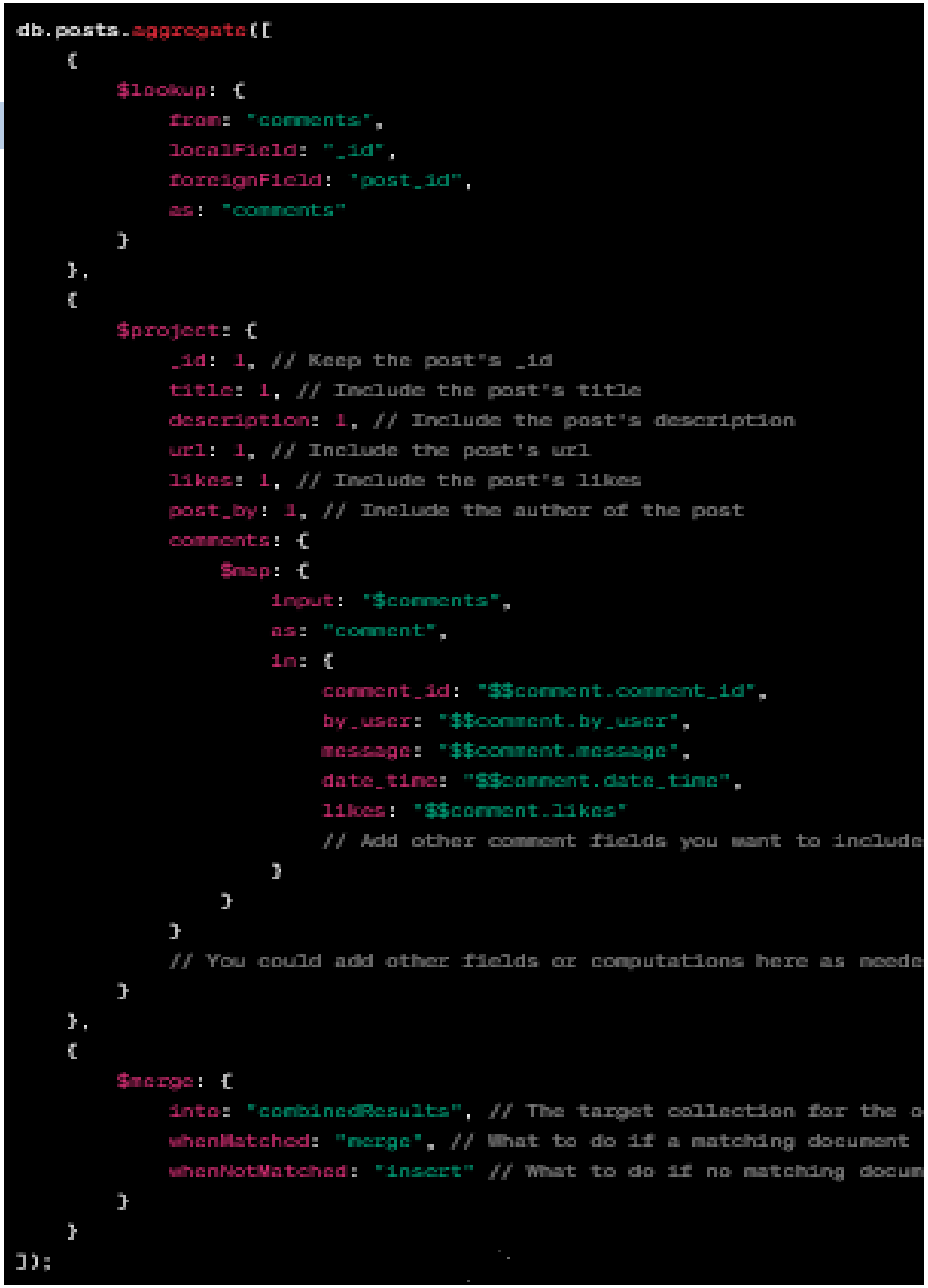
```
> show collections
< newCollection
  post
  postaggreCollection
> db.postaggreCollection.find()
< {
    _id: ObjectId('65a9ea971220ffbb76de9657'),
    title: "Developer's hub",
    by: 'tutorials point'
  }
  {
    _id: ObjectId('507f191e810c19729de860ea'),
    title: 'MongoDB Overview',
    by: 'tutorials point'
  }
  {
    _id: ObjectId('507f191e810c19729de860eb'),
    title: 'MongoDB Overview',
    by: 'tutorials point'
  }
```

$lookup and $map

- The $lookup stage adds a new array field to each input document.
- $map Applies an expression to each item in an array and returns an array with the applied results

- db.posts.insertMany([{ _id: 1, title: "The Joy of MongoDB", description: "Introduction to MongoDB", url: "http://example.com/mongodb", likes: 100, post_by: "Author1" },
- { _id: 2, title: "Aggregation Framework", description: "Deep Dive into Aggregation", url: "http://example.com/aggregation", likes: 150, post_by: "Author2" },
- { _id: 3, title: "Sharding Strategies", description: "How to shard effectively", url: "http://example.com/sharding", likes: 75, post_by: "Author3" }]);

- db.comments.insertMany([// Comments for post with _id: 1   { comment_id: 1, post_id: 1, by_user: "User1", message: "Great post!", date_time: new Date(), likes: 5 },   { comment_id: 2, post_id: 1, by_user: "User2", message: "Very informative!", date_time: new Date(), likes: 3 },
- // Comments for post with _id: 2   { comment_id: 3, post_id: 2, by_user: "User3", message: "I love the Aggregation Framework!", date_time: new Date(), likes: 8 },   { comment_id: 4, post_id: 2, by_user: "User4", message: "Can't wait to try this out.", date_time: new Date(), likes: 4 },
- // Comments for post with _id: 3   { comment_id: 5, post_id: 3, by_user: "User5", message: "Sharding is complex, but this helps.", date_time: new Date(), likes: 2 }]);



```
db.posts.aggregate([
    {
        $lookup: {
            from: "comments",
            localField: "_id",
            foreignField: "post_id",
            as: "comments"
        }
    },
    {
        $project: {
            _id: 1, // Keep the post's _id
            title: 1, // Include the post's title
            description: 1, // Include the post's description
            url: 1, // Include the post's url
            likes: 1, // Include the post's likes
            post_by: 1, // Include the author of the post
            comments: {
                $map: {
                    input: "$comments",
                    as: "comment",
                    in: {
                        comment_id: "$$comment.comment_id",
                        by_user: "$$comment.by_user",
                        message: "$$comment.message",
                        date_time: "$$comment.date_time",
                        likes: "$$comment.likes"
                        // Add other comment fields you want to include
                    }
                }
            }
            // You could add other fields or computations here as neede
        }
    },
    {
        $merge: {
            into: "combinedResults", // The target collection for the o
            whenMatched: "merge", // What to do if a matching document
            whenNotMatched: "insert" // What to do if no matching docum
        }
    }
]);
```

```
• db.accounts.insertMany([   {      accountId: 1,      accountType: "Savings",
  customerName: "John Doe",      amount: 1000,      previousTransactionDetails: [         {
  transactionId: "T1001", amount: 100, type: "Deposit" },         { transactionId: "T1002",
  amount: 50, type: "Withdrawal" }      ]   },
 {      accountId: 2,      accountType: "Checking",      customerName: "Jane Smith",      amount:
1500,      previousTransactionDetails: [         { transactionId: "T2001", amount: 200, type:
"Deposit" },         { transactionId: "T2002", amount: 100, type: "Withdrawal" }      ]   },
  {      accountId: 3,      accountType: "Savings",      customerName: "Alice Johnson",
amount: 500,      previousTransactionDetails: [         { transactionId: "T3001", amount: 500, type:
"Deposit" }      ]   },
{      accountId: 4,      accountType: "Checking",      customerName: "Bob Brown",      amount:
800,      previousTransactionDetails: [         { transactionId: "T4001", amount: 300, type:
"Deposit" },         { transactionId: "T4002", amount: 100, type: "Withdrawal" }      ]   },
 {      accountId: 5,      accountType: "Savings",      customerName: "Charlie Davis",      amount:
1200,      previousTransactionDetails: [         { transactionId: "T5001", amount: 1000, type:
"Deposit" },         { transactionId: "T5002", amount: 200, type: "Withdrawal" }      ]   }]);
```