# MongoDB - Indexing

# Agenda

1 Understand about Indexes

2 Understand different types of Indexes

3 Understand properties of Indexes

4 Explain Plan in MongoDB

5 Mongostat

6 Mongotop

7 Logging Slow queries

8 Profiling

# Module Objectives

ease

the
nt

## What you will learn

At the end of this module, you will learn:

- Indexing in MongoDB

## What you will be able to do

At the end of this module, you be able to:

- Understand what are Indexes
- List the different types of Indexes and their features
- Explain Plan in MongoDB
- Describe Mongostat and Mongotop
- State the features of Logging Slow queries and Profiling

# Index in MongoDB

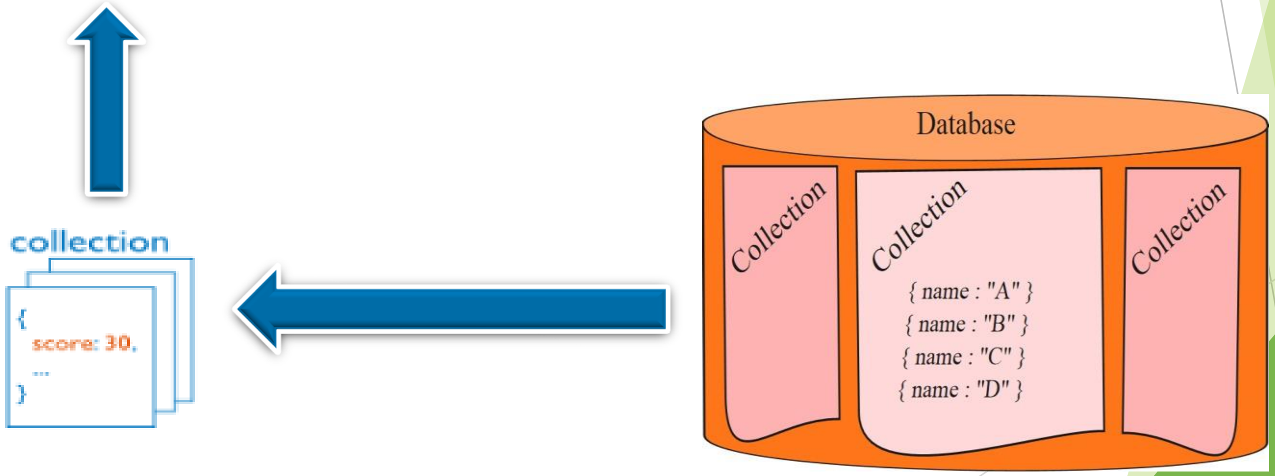
| | |
|---|---|
| **Creation Index** | • db.users.ensureIndex( { score: 1 } ) |
| **Show Existing Indexes** | • db.users.getIndexes() |
| **Drop Index** | • db.users.dropIndex( {score: 1} ) |
| **Explain—Explain** | • db.users.find().explain()<br>• Returns a document that describes the process and indexes. |
| **Hint** | • db.users.find().hint({score: 1})<br>• Overide MongoDB's default index selection. |

# Before Index

~~What does database normally do when we query?~~

- MongoDB must scan every document.
- Inefficient because process large volume of data.

```
db.users.find( { score: { “$lt” : 30} } )
```



# What is Index?

Indexes are special data structures that store a small portion of the collection’s data set in an easy to traverse form.

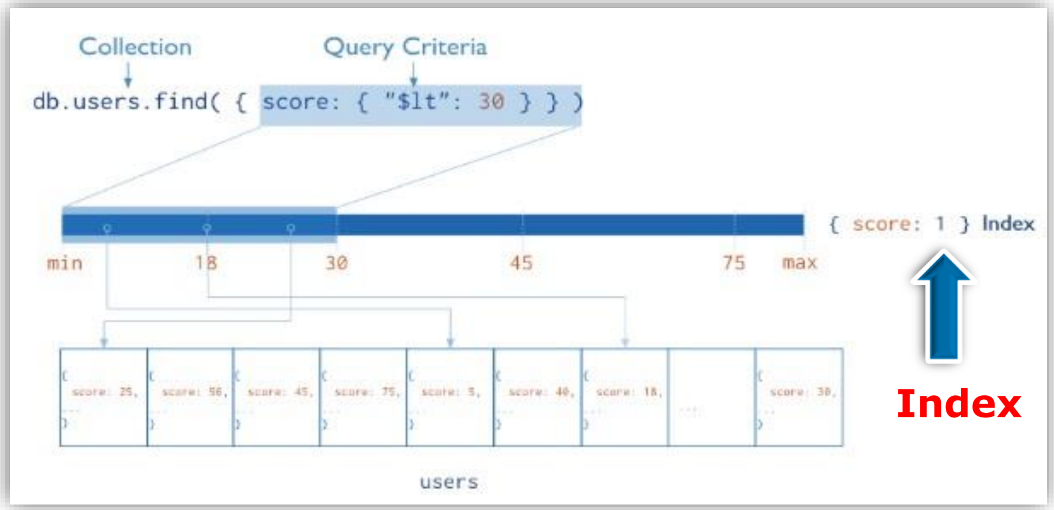
The index stores the value of a specific field or set of fields, ordered by the value of the field.

# Definition of Index

**Definition**

- **Indexes are special data structures that store a small portion of the collection’s data set in an easy to traverse form.**

collection
{
score: 30,
...
}



**Index**

**Diagram of a query that uses an index to select.**

# Indexes

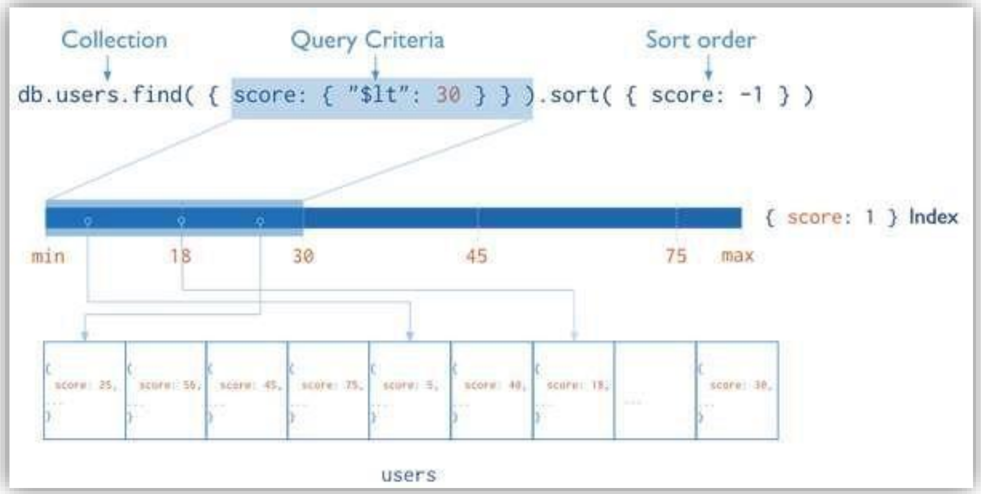
MongoDB can use indexes to return documents sorted by the index key directly from the index without requiring an additional sort phase.

Indexes help efficient execution of queries.

- Without indexes MongoDB must scan every document in a collection to select those documents that match the query statement.

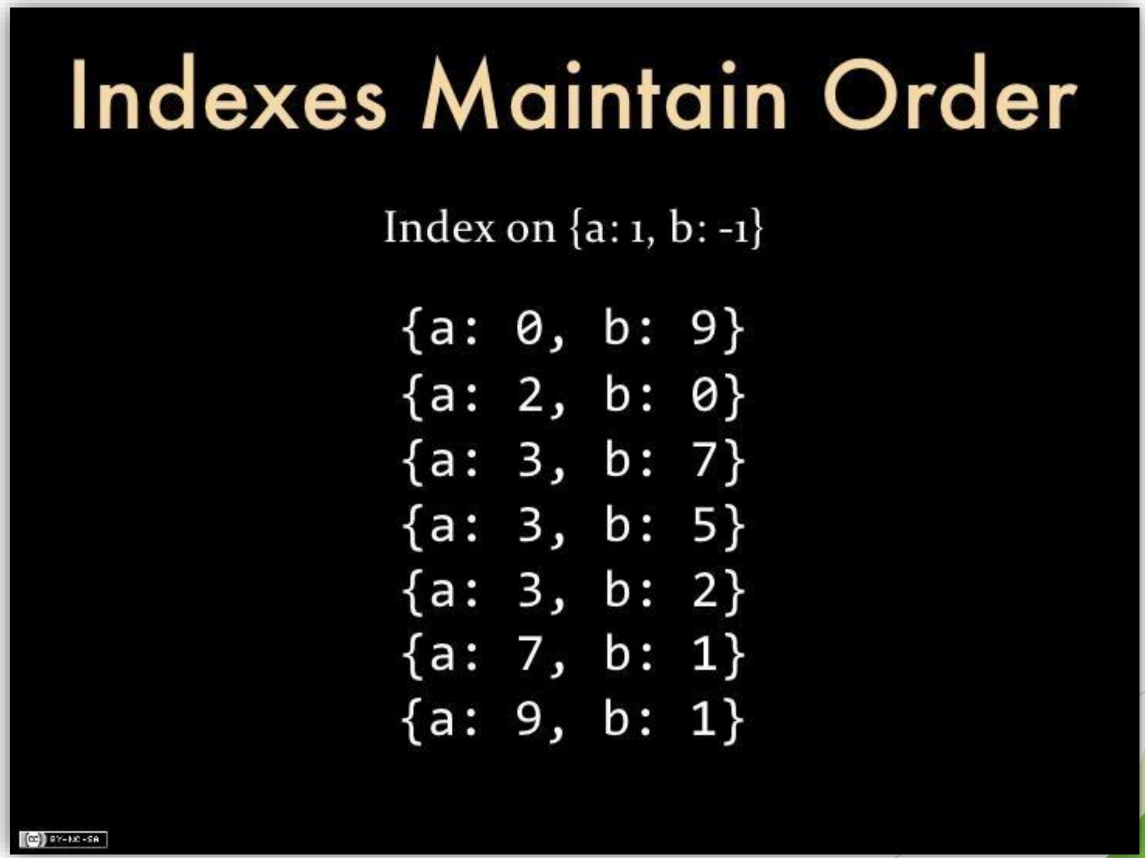
These collection scans are inefficient because they require Mongod to process a larger volume of data than an index for each operation.

# Diagrammatic Representation



```
db.users.find( { score: { "$lt": 30 } } ).sort( { score: -1 } )
```

# Indexes



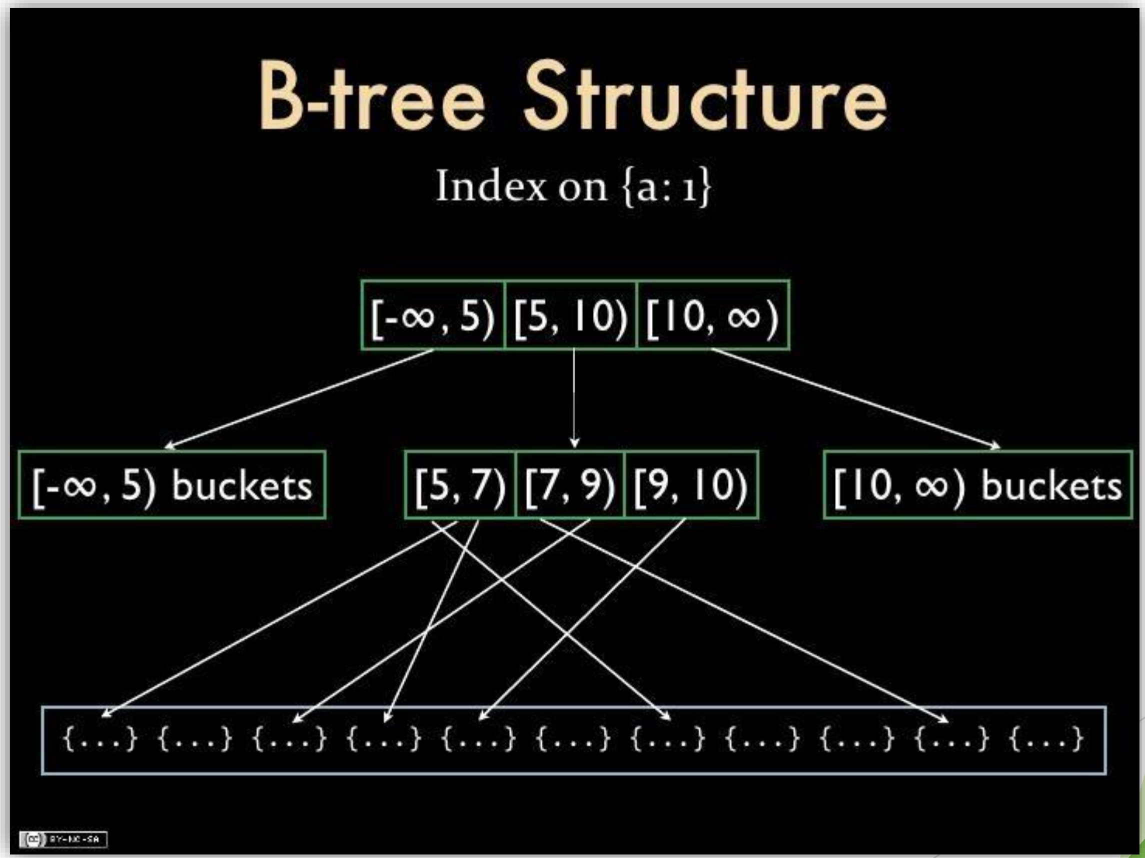
## Indexes Maintain Order

Index on {a: 1, b: -1}

```
{a: 0, b: 9}
{a: 2, b: 0}
{a: 3, b: 7}
{a: 3, b: 5}
{a: 3, b: 2}
{a: 7, b: 1}
{a: 9, b: 1}
```

BY-NC-SA

# B-tree Index Structure

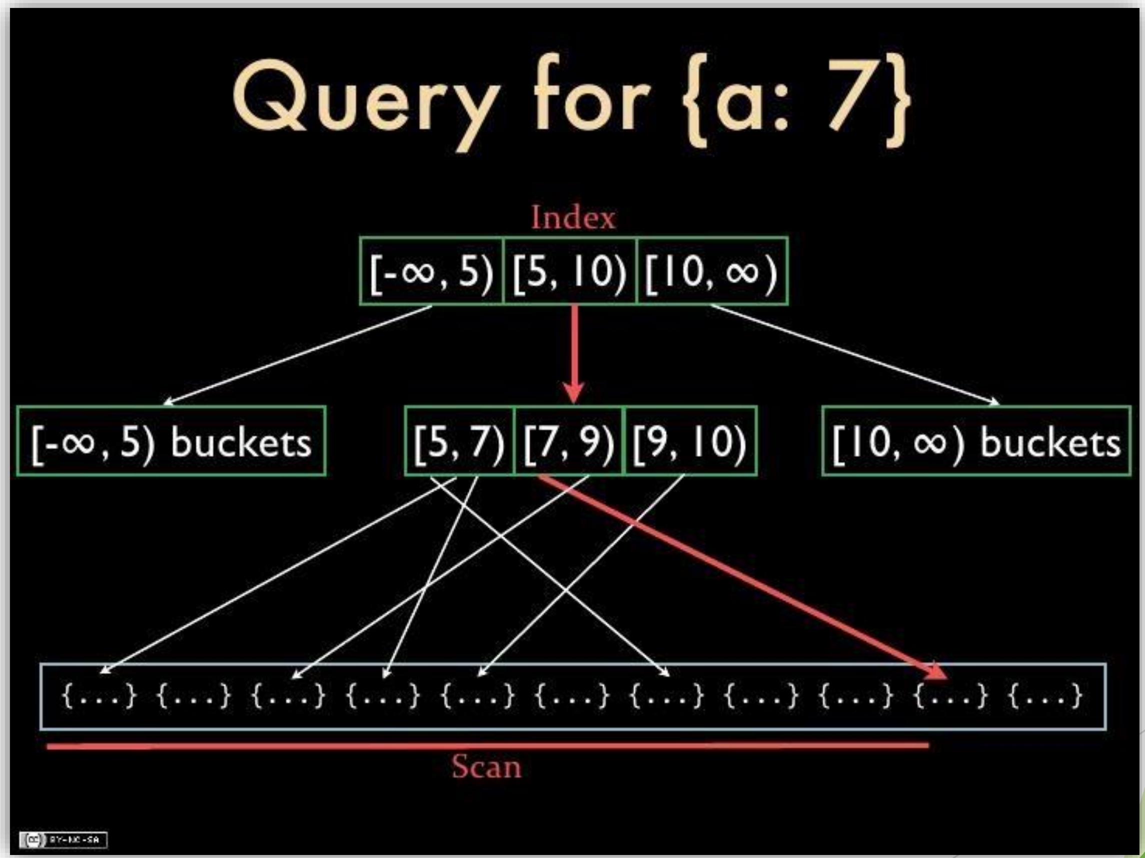


## B-tree Structure

Index on {a: 1}

BY-NC-SA

# Index Querying



# Types of Indexes

| | | |
|---|---|---|
| _id (default) | Single field index | Compound index |
| Multikey index | Geospatial index | Text index |
| Hashed index | Unique index | Sparse index |

# About Indexes

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.

The ordering of the index entries supports efficient equality matches and range-based query operations.

The index stores the value of a specific field or set of fields, ordered by the value of the field.

MongoDB can return sorted results by using the ordering in the index.

# Index Types

### Default _id

- All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an _id field with an objectid value.

### Single Field

- In addition to the MongoDB-defined _id index, MongoDB supports the creation of user-defined ascending / descending indexes on a single field of a document.

# Example

{ "_id" : ObjectId(...), "name" : "Alice", "age" : 27 }

The following command creates an index on the name field:

- db.friends.createIndex( { "name" : 1 } )

- **Single Field Indexes**
  - db.users.ensureIndex( { score: 1 } )

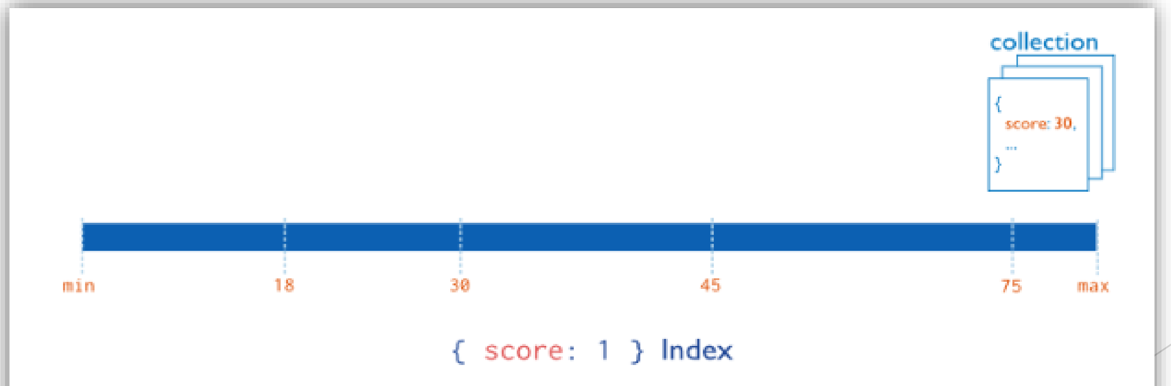


Diagram of an index on the score field (ascending).

# Compound Indexes

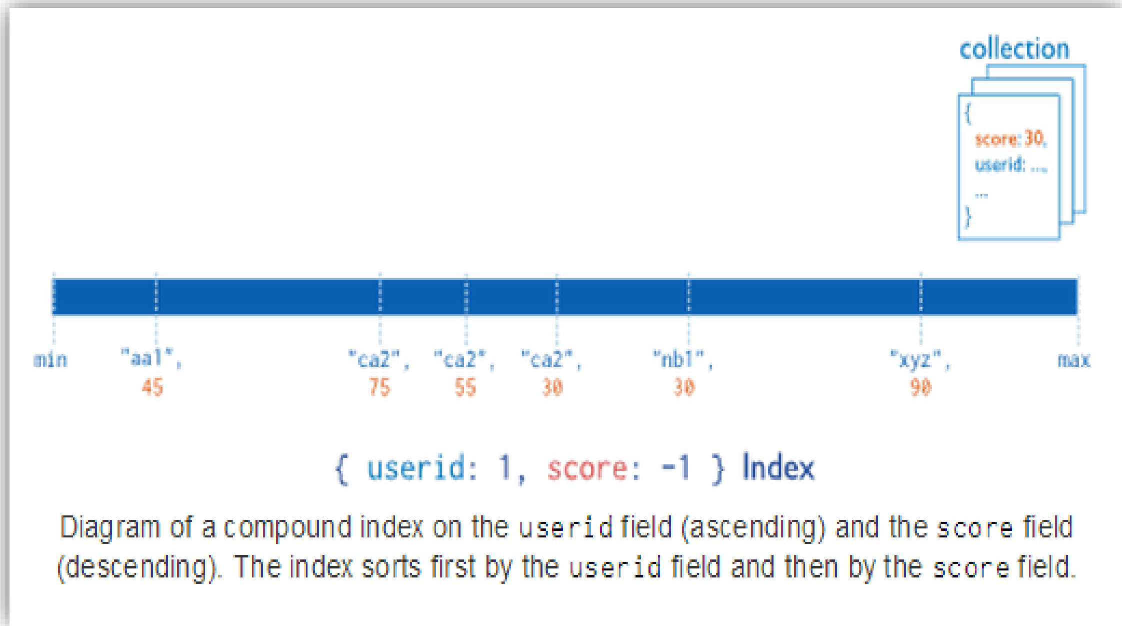
MongoDB also supports user-defined indexes on multiple fields, i.e. Compound indexes.

The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each user id value, sorts by score.

# Compound Indexes (contd.)

## Compound Field Indexes

- db.users.ensureIndex( { userid:1, score: -1 } )



{ userid: 1, score: -1 } Index

Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

# Example

Consider a collection named products that holds documents that resemble the following document:

- { "_id": ObjectId(...), "item": "Banana", "category": ["food", "produce", "grocery"], "location": "4th Street Store", "stock": 4, "type": "cases", "arrival": Date(...) }

If applications query on the item field as well as query on both the item field and the stock field, you can specify a single compound index to support both of these queries:

- db.products.createIndex( { "item": 1, "stock": 1 } )

# Multikey Indexes

When indexing is done on an array field, it is called a multikey index.

To index a field that holds an array value ,MongoDB creates an index key for each element in the array.

These *multikey* indexes support efficient queries against array fields.

Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) *and* nested documents.

# Example

- Lets consider a document:

```
{
"title": "Superman",
"tags": ["comic", "action", "xray"],
"issues": [ { "number": 1, "published_on": "June 1938"
 }
 ]
}
```

- Multikey indexes lets us search on the values in the tags array as well as in the issues array. Let’s create two indexes to cover both.

# Example (contd.)

```
comics.ensureIndex({tags: 1});
comics.ensureIndex({issues: 1});
```

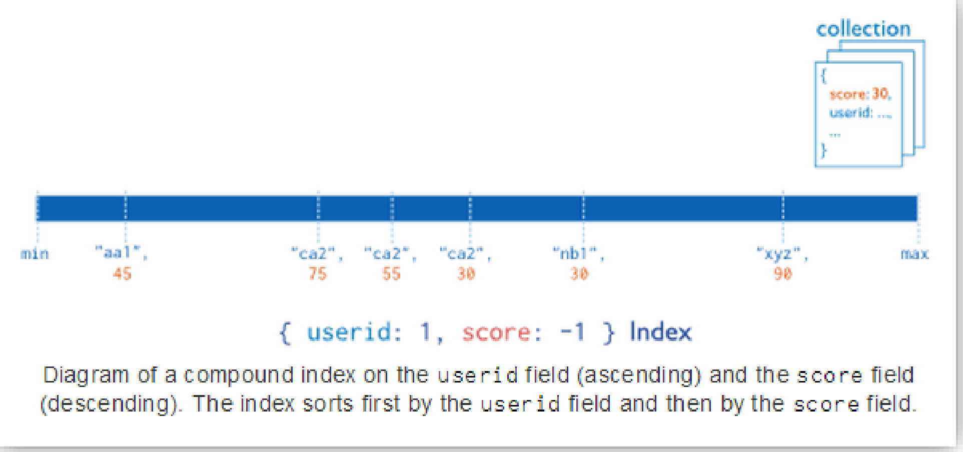
If the document changes structure it would be better to create a specific compound index on the fields needed in sub element document.

```
comics.ensureIndex({"issues.number":1,
"issues.published_on":1});
```

# Example (contd.)

## Multikey Indexes

- db.users.ensureIndex( { addr.zip:1} )



{ userid: 1, score: -1 } Index

Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

# Limitations

Consider a collection that contains the following document:

- { _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }

You cannot create a compound multikey index { a: 1, b: 1 } on the collection since both the a and b fields are arrays.

# Demo of Indexes in MongoDB

**Import Da**

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

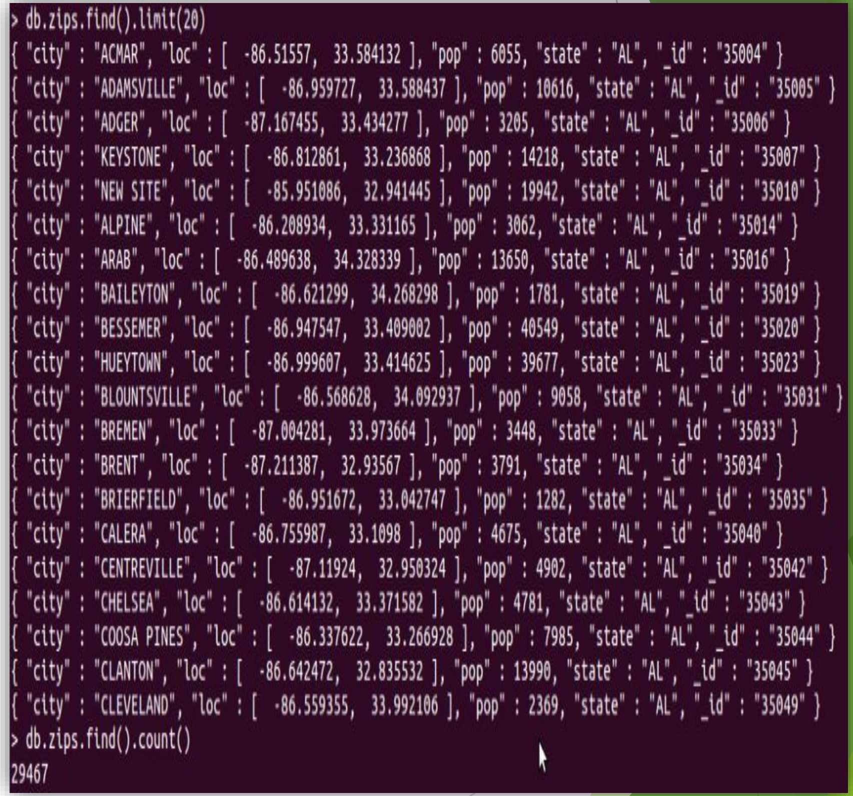
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

**Compare with data without indexes**



```
> db.zips.find().limit(20)
{ "city" : "ACMAR", "loc" : [  -86.51557,  33.584132 ], "pop" : 6055, "state" : "AL", "_id" : "35004" }
{ "city" : "ADAMSVILLE", "loc" : [  -86.959727,  33.588437 ], "pop" : 10616, "state" : "AL", "_id" : "35005" }
{ "city" : "ADGER", "loc" : [  -87.167455,  33.434277 ], "pop" : 3205, "state" : "AL", "_id" : "35006" }
{ "city" : "KEYSTONE", "loc" : [  -86.812861,  33.236868 ], "pop" : 14218, "state" : "AL", "_id" : "35007" }
{ "city" : "NEW SITE", "loc" : [  -85.951086,  32.941445 ], "pop" : 19942, "state" : "AL", "_id" : "35010" }
{ "city" : "ALPINE", "loc" : [  -86.208934,  33.331165 ], "pop" : 3062, "state" : "AL", "_id" : "35014" }
{ "city" : "ARAB", "loc" : [  -86.489638,  34.328339 ], "pop" : 13650, "state" : "AL", "_id" : "35016" }
{ "city" : "BAILEYTON", "loc" : [  -86.621299,  34.268298 ], "pop" : 1781, "state" : "AL", "_id" : "35019" }
{ "city" : "BESSEMER", "loc" : [  -86.947547,  33.409002 ], "pop" : 40549, "state" : "AL", "_id" : "35020" }
{ "city" : "HUEYTOWN", "loc" : [  -86.999607,  33.414625 ], "pop" : 39677, "state" : "AL", "_id" : "35023" }
{ "city" : "BLOUNTSVILLE", "loc" : [  -86.568628,  34.092937 ], "pop" : 9058, "state" : "AL", "_id" : "35031" }
{ "city" : "BREMEN", "loc" : [  -87.004281,  33.973664 ], "pop" : 3448, "state" : "AL", "_id" : "35033" }
{ "city" : "BRENT", "loc" : [  -87.211387,  32.93567 ], "pop" : 3791, "state" : "AL", "_id" : "35034" }
{ "city" : "BRIERFIELD", "loc" : [  -86.951672,  33.042747 ], "pop" : 1282, "state" : "AL", "_id" : "35035" }
{ "city" : "CALERA", "loc" : [  -86.755987,  33.1098 ], "pop" : 4675, "state" : "AL", "_id" : "35040" }
{ "city" : "CENTREVILLE", "loc" : [  -87.11924,  32.950324 ], "pop" : 4902, "state" : "AL", "_id" : "35042" }
{ "city" : "CHELSEA", "loc" : [  -86.614132,  33.371582 ], "pop" : 4781, "state" : "AL", "_id" : "35043" }
{ "city" : "COOSA PINES", "loc" : [  -86.337622,  33.266928 ], "pop" : 7985, "state" : "AL", "_id" : "35044" }
{ "city" : "CLANTON", "loc" : [  -86.642472,  32.835532 ], "pop" : 13990, "state" : "AL", "_id" : "35045" }
{ "city" : "CLEVELAND", "loc" : [  -86.559355,  33.992106 ], "pop" : 2369, "state" : "AL", "_id" : "35049" }
> db.zips.find().count()
29467
```

# Demo of Indexes in MongoDB (contd.)

Import Da

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

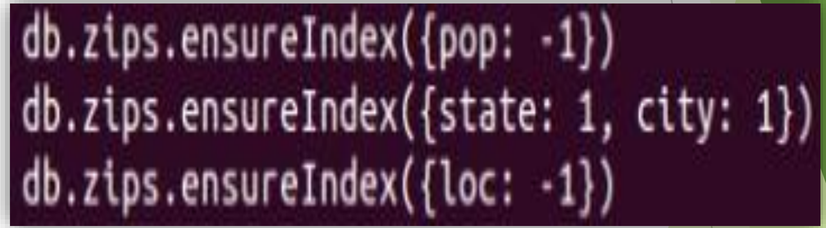
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

**~~Compare with data without indexes~~**



```
db.zips.ensureIndex({pop: -1})
db.zips.ensureIndex({state: 1, city: 1})
db.zips.ensureIndex({loc: -1})
```

# Demo of Indexes in MongoDB (contd.)

Import Da

**Create Index**

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

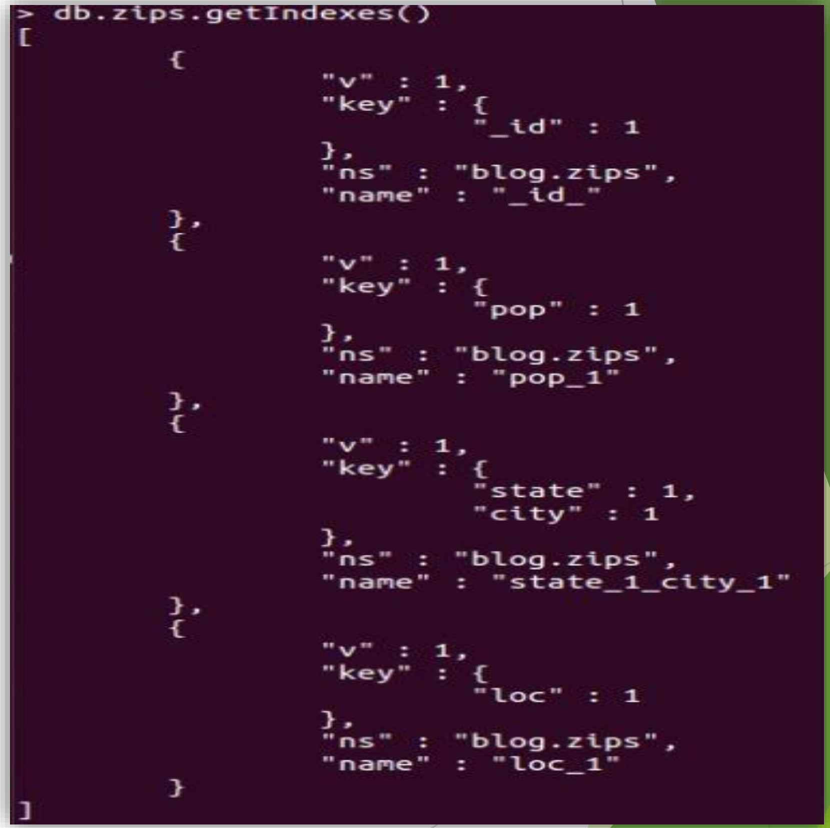
**Show Existing Index**

**Hint**

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

**Explain**

**Compare with data without indexes**



```
> db.zips.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "ns" : "blog.zips",
                "name" : "_id_"
        },
        {
                "v" : 1,
                "key" : {
                        "pop" : 1
                },
                "ns" : "blog.zips",
                "name" : "pop_1"
        },
        {
                "v" : 1,
                "key" : {
                        "state" : 1,
                        "city" : 1
                },
                "ns" : "blog.zips",
                "name" : "state_1_city_1"
        },
        {
                "v" : 1,
                "key" : {
                        "loc" : 1
                },
                "ns" : "blog.zips",
                "name" : "loc_1"
        }
]
```

# Demo of Indexes in MongoDB (contd.)

**Import Da**

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

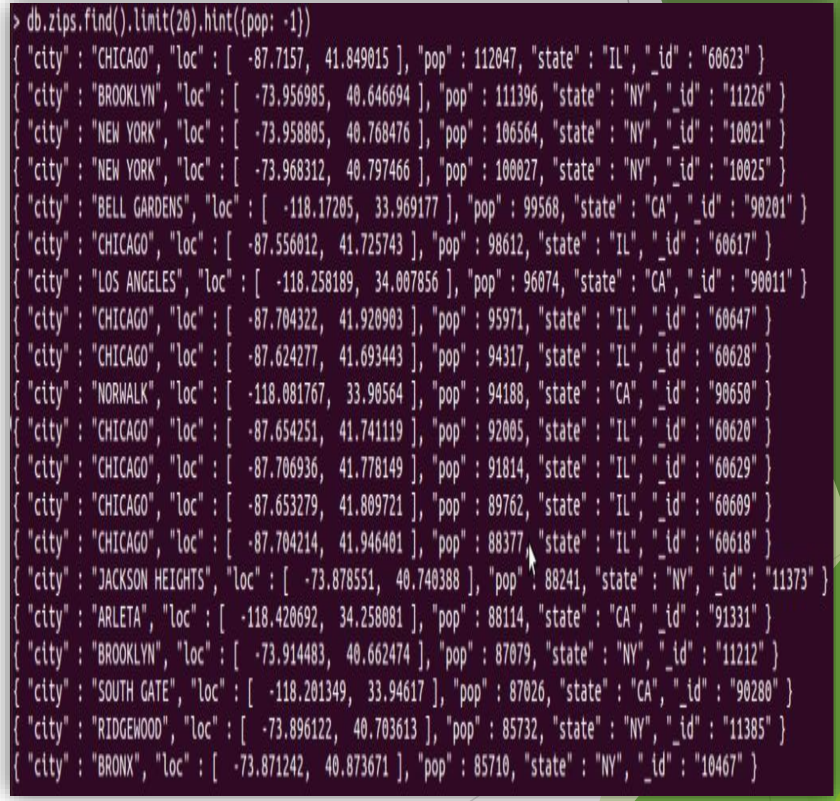
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

~~**Compare with data without indexes**~~



```
> db.zips.find().limit(20).hint({pop: -1})
{ "city" : "CHICAGO", "loc" : [  -87.7157,  41.849015 ], "pop" : 112047, "state" : "IL", "_id" : "60623" }
{ "city" : "BROOKLYN", "loc" : [  -73.956985,  40.646694 ], "pop" : 111396, "state" : "NY", "_id" : "11226" }
{ "city" : "NEW YORK", "loc" : [  -73.958805,  40.768476 ], "pop" : 106564, "state" : "NY", "_id" : "10021" }
{ "city" : "NEW YORK", "loc" : [  -73.968312,  40.797466 ], "pop" : 100027, "state" : "NY", "_id" : "10025" }
{ "city" : "BELL GARDENS", "loc" : [  -118.17205,  33.969177 ], "pop" : 99568, "state" : "CA", "_id" : "90201" }
{ "city" : "CHICAGO", "loc" : [  -87.556012,  41.725743 ], "pop" : 98612, "state" : "IL", "_id" : "60617" }
{ "city" : "LOS ANGELES", "loc" : [  -118.258189,  34.007856 ], "pop" : 96074, "state" : "CA", "_id" : "90011" }
{ "city" : "CHICAGO", "loc" : [  -87.704322,  41.920903 ], "pop" : 95971, "state" : "IL", "_id" : "60647" }
{ "city" : "CHICAGO", "loc" : [  -87.624277,  41.693443 ], "pop" : 94317, "state" : "IL", "_id" : "60628" }
{ "city" : "NORWALK", "loc" : [  -118.081767,  33.90564 ], "pop" : 94188, "state" : "CA", "_id" : "90650" }
{ "city" : "CHICAGO", "loc" : [  -87.654251,  41.741119 ], "pop" : 92005, "state" : "IL", "_id" : "60620" }
{ "city" : "CHICAGO", "loc" : [  -87.706936,  41.778149 ], "pop" : 91814, "state" : "IL", "_id" : "60629" }
{ "city" : "CHICAGO", "loc" : [  -87.653279,  41.809721 ], "pop" : 89762, "state" : "IL", "_id" : "60609" }
{ "city" : "CHICAGO", "loc" : [  -87.704214,  41.946401 ], "pop" : 88377, "state" : "IL", "_id" : "60618" }
{ "city" : "JACKSON HEIGHTS", "loc" : [  -73.878551,  40.740388 ], "pop" : 88241, "state" : "NY", "_id" : "11373" }
{ "city" : "ARLETA", "loc" : [  -118.420692,  34.258081 ], "pop" : 88114, "state" : "CA", "_id" : "91331" }
{ "city" : "BROOKLYN", "loc" : [  -73.914483,  40.662474 ], "pop" : 87079, "state" : "NY", "_id" : "11212" }
{ "city" : "SOUTH GATE", "loc" : [  -118.201349,  33.94617 ], "pop" : 87026, "state" : "CA", "_id" : "90280" }
{ "city" : "RIDGEWOOD", "loc" : [  -73.896122,  40.703613 ], "pop" : 85732, "state" : "NY", "_id" : "11385" }
{ "city" : "BRONX", "loc" : [  -73.871242,  40.873671 ], "pop" : 85710, "state" : "NY", "_id" : "10467" }
```

# Demo of Indexes in MongoDB (contd.)

**Import Da**

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

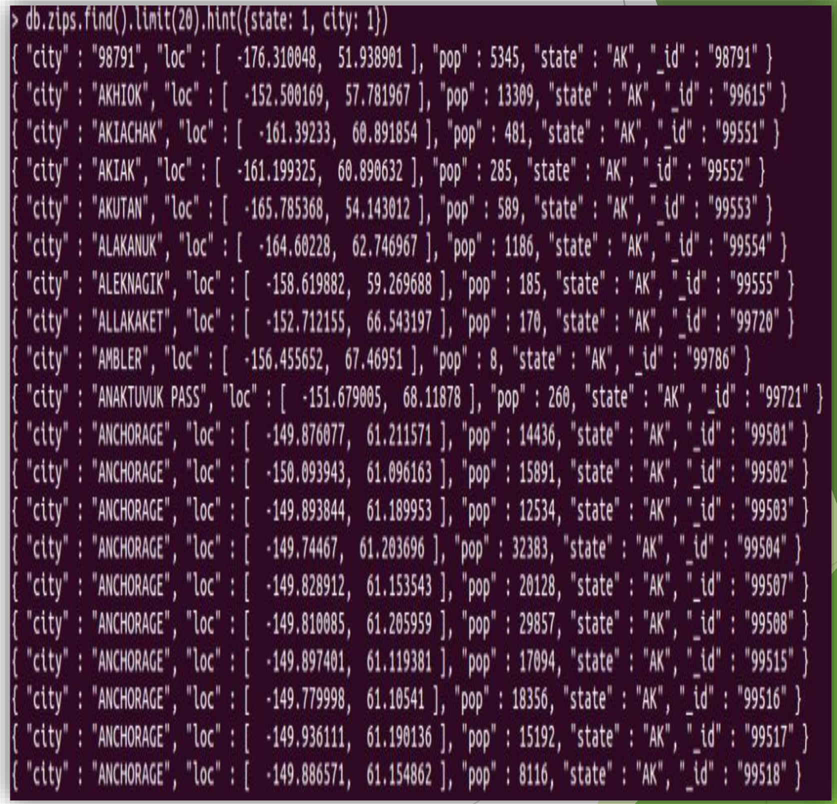
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

~~**Compare with data without indexes**~~



```
> db.zips.find().limit(20).hint({state: 1, city: 1})
{ "city" : "98791", "loc" : [ -176.310048, 51.938901 ], "pop" : 5345, "state" : "AK", "_id" : "98791" }
{ "city" : "AKHIOK", "loc" : [ -152.500169, 57.781967 ], "pop" : 13309, "state" : "AK", "_id" : "99615" }
{ "city" : "AKIACHAK", "loc" : [ -161.39233, 60.891854 ], "pop" : 481, "state" : "AK", "_id" : "99551" }
{ "city" : "AKIAK", "loc" : [ -161.199325, 60.890632 ], "pop" : 285, "state" : "AK", "_id" : "99552" }
{ "city" : "AKUTAN", "loc" : [ -165.785368, 54.143012 ], "pop" : 589, "state" : "AK", "_id" : "99553" }
{ "city" : "ALAKANUK", "loc" : [ -164.60228, 62.746967 ], "pop" : 1186, "state" : "AK", "_id" : "99554" }
{ "city" : "ALEKNAGIK", "loc" : [ -158.619882, 59.269688 ], "pop" : 185, "state" : "AK", "_id" : "99555" }
{ "city" : "ALLAKAKET", "loc" : [ -152.712155, 66.543197 ], "pop" : 170, "state" : "AK", "_id" : "99720" }
{ "city" : "AMBLER", "loc" : [ -156.455652, 67.46951 ], "pop" : 8, "state" : "AK", "_id" : "99786" }
{ "city" : "ANAKTUVUK PASS", "loc" : [ -151.679005, 68.11878 ], "pop" : 260, "state" : "AK", "_id" : "99721" }
{ "city" : "ANCHORAGE", "loc" : [ -149.876077, 61.211571 ], "pop" : 14436, "state" : "AK", "_id" : "99501" }
{ "city" : "ANCHORAGE", "loc" : [ -150.093943, 61.096163 ], "pop" : 15891, "state" : "AK", "_id" : "99502" }
{ "city" : "ANCHORAGE", "loc" : [ -149.893844, 61.189953 ], "pop" : 12534, "state" : "AK", "_id" : "99503" }
{ "city" : "ANCHORAGE", "loc" : [ -149.74467, 61.203696 ], "pop" : 32383, "state" : "AK", "_id" : "99504" }
{ "city" : "ANCHORAGE", "loc" : [ -149.828912, 61.153543 ], "pop" : 20128, "state" : "AK", "_id" : "99507" }
{ "city" : "ANCHORAGE", "loc" : [ -149.810085, 61.205959 ], "pop" : 29857, "state" : "AK", "_id" : "99508" }
{ "city" : "ANCHORAGE", "loc" : [ -149.897401, 61.119381 ], "pop" : 17094, "state" : "AK", "_id" : "99515" }
{ "city" : "ANCHORAGE", "loc" : [ -149.779998, 61.10541 ], "pop" : 18356, "state" : "AK", "_id" : "99516" }
{ "city" : "ANCHORAGE", "loc" : [ -149.936111, 61.190136 ], "pop" : 15192, "state" : "AK", "_id" : "99517" }
{ "city" : "ANCHORAGE", "loc" : [ -149.886571, 61.154862 ], "pop" : 8116, "state" : "AK", "_id" : "99518" }
```

# Demo of Indexes in MongoDB (contd.)

**Import Da**

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

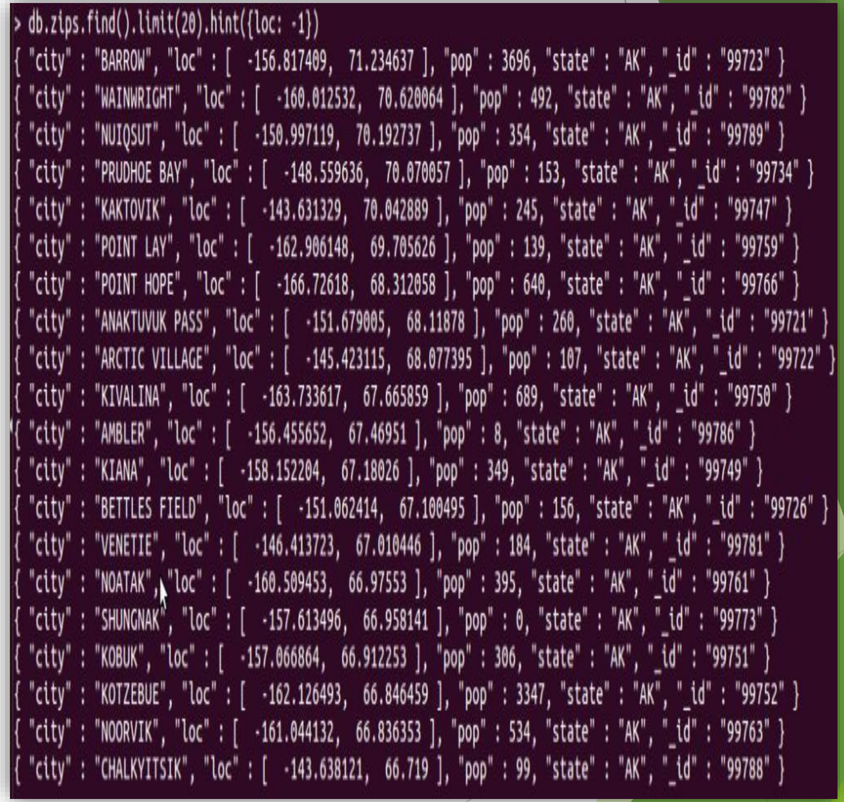
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

~~**Compare with data without indexes**~~



```
> db.zips.find().limit(20).hint({loc: -1})
{ "city" : "BARROW", "loc" : [  -156.817409,  71.234637 ], "pop" : 3696, "state" : "AK", "_id" : "99723" }
{ "city" : "WAINWRIGHT", "loc" : [  -160.012532,  70.620064 ], "pop" : 492, "state" : "AK", "_id" : "99782" }
{ "city" : "NUIQSUT", "loc" : [  -150.997119,  70.192737 ], "pop" : 354, "state" : "AK", "_id" : "99789" }
{ "city" : "PRUDHOE BAY", "loc" : [  -148.559636,  70.070057 ], "pop" : 153, "state" : "AK", "_id" : "99734" }
{ "city" : "KAKTOVIK", "loc" : [  -143.631329,  70.042889 ], "pop" : 245, "state" : "AK", "_id" : "99747" }
{ "city" : "POINT LAY", "loc" : [  -162.906148,  69.705626 ], "pop" : 139, "state" : "AK", "_id" : "99759" }
{ "city" : "POINT HOPE", "loc" : [  -166.72618,  68.312058 ], "pop" : 640, "state" : "AK", "_id" : "99766" }
{ "city" : "ANAKTUVUK PASS", "loc" : [  -151.679005,  68.11878 ], "pop" : 260, "state" : "AK", "_id" : "99721" }
{ "city" : "ARCTIC VILLAGE", "loc" : [  -145.423115,  68.077395 ], "pop" : 107, "state" : "AK", "_id" : "99722" }
{ "city" : "KIVALINA", "loc" : [  -163.733617,  67.665859 ], "pop" : 689, "state" : "AK", "_id" : "99750" }
{ "city" : "AMBLER", "loc" : [  -156.455652,  67.46951 ], "pop" : 8, "state" : "AK", "_id" : "99786" }
{ "city" : "KIANA", "loc" : [  -158.152204,  67.18026 ], "pop" : 349, "state" : "AK", "_id" : "99749" }
{ "city" : "BETTLES FIELD", "loc" : [  -151.062414,  67.100495 ], "pop" : 156, "state" : "AK", "_id" : "99726" }
{ "city" : "VENETIE", "loc" : [  -146.413723,  67.010446 ], "pop" : 184, "state" : "AK", "_id" : "99781" }
{ "city" : "NOATAK", "loc" : [  -160.509453,  66.97553 ], "pop" : 395, "state" : "AK", "_id" : "99761" }
{ "city" : "SHUNGNAK", "loc" : [  -157.613496,  66.958141 ], "pop" : 0, "state" : "AK", "_id" : "99773" }
{ "city" : "KOBUK", "loc" : [  -157.066864,  66.912253 ], "pop" : 306, "state" : "AK", "_id" : "99751" }
{ "city" : "KOTZEBUE", "loc" : [  -162.126493,  66.846459 ], "pop" : 3347, "state" : "AK", "_id" : "99752" }
{ "city" : "NOORVIK", "loc" : [  -161.044132,  66.836353 ], "pop" : 534, "state" : "AK", "_id" : "99763" }
{ "city" : "CHALKYITSIK", "loc" : [  -143.638121,  66.719 ], "pop" : 99, "state" : "AK", "_id" : "99788" }
```

# Demo of Indexes in MongoDB (contd.)

**Import Da**

**Create Index**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

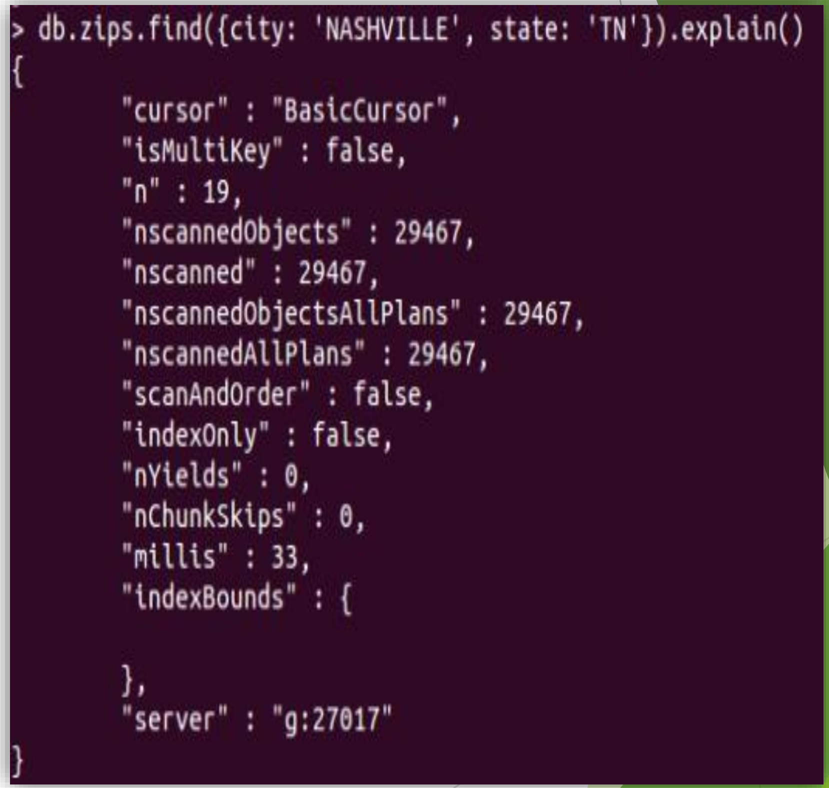
**Show Existing Index**

**Hint**

- **Single Field Index**
- **Compound Field Indexes**
- **Multikey Indexes**

**Explain**

~~**Compare with data without indexes**~~



```
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 29467,
        "nscanned" : 29467,
        "nscannedObjectsAllPlans" : 29467,
        "nscannedAllPlans" : 29467,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 33,
        "indexBounds" : {

        },
        "server" : "g:27017"
}
```

# MongoDB Advanced Usage

**Index**

- MongoDB’s indexes work almost identically to typical relational database indexes.
- Index optimization for MySQL / Oracle / SQLite will apply equally well to MongoDB.
- If an index has N keys, it will make queries on any prefix of those keys fast.

**Example**

- db.people.find({"username" : "mark"})
- db.people.ensureIndex({"username" : 1})
- db.people.find({"date" : date1}).sort({"date" : 1, "username" : 1})
- db.ensureIndex({"date" : 1, "username" : 1})
- db.people.find({"username" : "mark"}).explain()

# MongoDB Advanced Usage (contd.)

Indexes can be created on keys in embedded documents in the same

- db.blog.ensureIndex({"comments.date" : 1})
- db.people.ensureIndex({"username" : 1}, {"unique" : true})
- db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups" : true})
- autocomplete:PRIMARY> db.system.indexes.find()

```
{ "name" : "_id_", "ns" : "test.fs.files", "key" : { "_id" : 1 }, "v" : 0 }
{ "ns" : "test.fs.files", "key" : { "filename" : 1 }, "name" : "filename_1", "v" : 0 }
{ "name" : "_id_", "ns" : "test.fs.chunks", "key" : { "_id" : 1 }, "v" : 0 }
{ "ns" : "test.fs.chunks", "key" : { "files_id" : 1, "n" : 1 }, "name" : "files_id_1_n_1", "v" : 0 }
```

# MongoDB Advanced Usage (contd.)

```
{
        "cursor" : "BtreeCursor name_1",
        "nscanned" : 1,
        "nscannedObjects" : 1,
        "n" : 1,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
            "name" : [
                [
                        "user0",
                        "user0"
                ]
            ]
        }
}
```

# MongoDB Advanced Usage (contd.)

## Index continue: hint()

- If you find that Mongo is using different indexes than you want it to for a query, you can force it to use a certain index by using hint.
- db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})

## Index continue: change index

- db.runCommand({"dropIndexes" : "foo", "index" : "alphabet"})

- db.people.ensureIndex({"username" : 1}, {"background" : true})
  - Using the {"background" : true} option builds the index in the background, while handling incoming requests. If you do not include the background option, the database will block all other requests while the index is being built.

# MongoDB Advanced Usage (contd.)

Advanced Usage

- Index
- Aggregation

- db.foo.count()
- db.foo.count({"x" : 1})
- db.runCommand({"distinct" : "people", "key" : "age"})

- Group

- {"day" : "2010/10/03", "time" : "10/3/2010 03:57:01 GMT-400", "price" : 4.23}
- {"day" : "2010/10/04", "time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27}
- {"day" : "2010/10/03", "time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10}
- {"day" : "2010/10/06", "time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
- {"day" : "2010/10/04", "time" : "10/4/2010 08:34:50 GMT-400", "price" : 4.01}

- db.runCommand({"group" : {
- ... "ns" : "stocks",
- ... "key" : "day",
- ... "initial" : {"time" : 0},
- ... "$reduce" : function(doc, prev) {
- ... if (doc.time > prev.time) {
- ... prev.price = doc.price;
- ... prev.time = doc.time;
- ... } }}})

# Geospatial Indexes

Finding the nearest N things to a current location.

MongoDB provides a special type of index for coordinate plane queries, called a geospatial index.

The indexes makes it possible to perform efficient Geospatial queries.

The 2d Geospatial Sphere index allows to perform queries on a earth-like sphere making for better accuracy in matching locations.

MongoDB’sgeospatial indexes assumes that:

- Whatever you’re indexing is a flat plane.
- This means that results aren’t perfect for spherical shapes, like the earth, especially near the poles.

# Geospatial Indexes (contd.)

A geospatial index can be created using the ensureIndexfunction, but by passing "2d" as a value instead of 1 or -1:

- > db.map.ensureIndex({"gps" : "2d"})
- "gps" : [ 0, 100 ] }
- { "gps" : { "x" : -30, "y" : 30 } }
- { "gps" : { "latitude" : -180, "longitude" : 180 } }
- > db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000, "max" : 1000})
- > db.map.find({"gps" : {"$near" : [40, -73]}})
- > db.map.find({"gps" : {"$near" : [40, -73]}}).limit(10)

# Example

- Take the following example document.

```
{ loc: {
 type: "Point",
 coordinates: [60, 79] },
 type: "house" }

var locations = db.getSisterDB("geo").locations;
locations.ensureIndex({loc: "2dsphere", house: 1});
```

# Text Indexes

MongoDB provides text indexes to support query operations that perform a text search of string content. text indexes can include any field whose value is a string or an array of string elements.

To index a field that contains a string or an array of string elements, include the field and specify the string literal "text" in the index document, as in the following example:

- db.reviews.createIndex( { comments: "text" } )

# Properties of Indexes

Hashes Indexes

Unique Indexes

Sparse Indexes

# Hashed Indexes

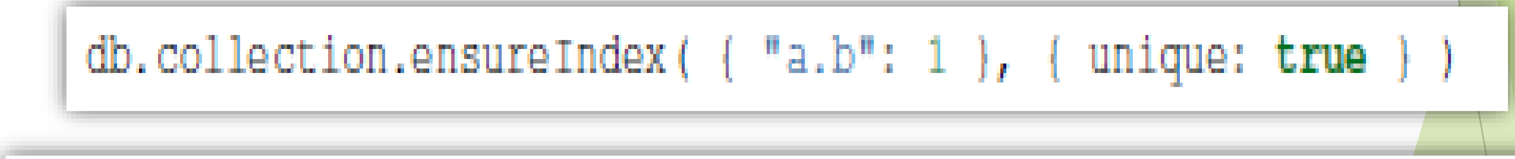
- Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses embedded documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

- Create a hashed index using an operation that resembles the
- following:

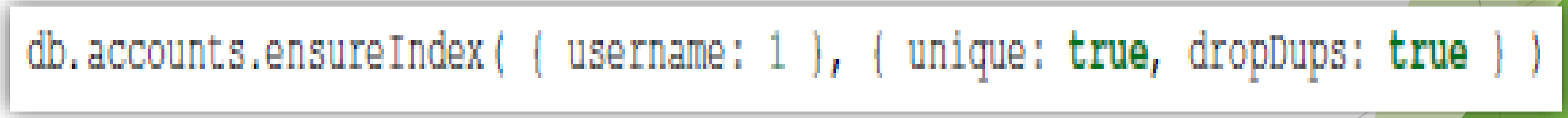
  - db.active.createIndex( { a: "hashed" } )
  - This operation creates a hashed index for the active collection
- on the a field.

# Hashed Indexes (contd.)

- The unique property for an index causes MongoDB to reject duplicate values for the indexed field. To create a unique index on a field that already has duplicate values.



```
db.collection.ensureIndex( { "a.b": 1 }, { unique: true } )
```



```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

# Unique Index

Unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

By default, unique is false on MongoDB indexes.

To solve this issue MongoDB allows to add the *unique* option to the *ensureIndex* statement when created, making the target field unique across the entire collection.

```
db.users.ensureIndex( { "userId": 1 }, { unique: true } )
```

# Sparse Index

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value.

The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection.

By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

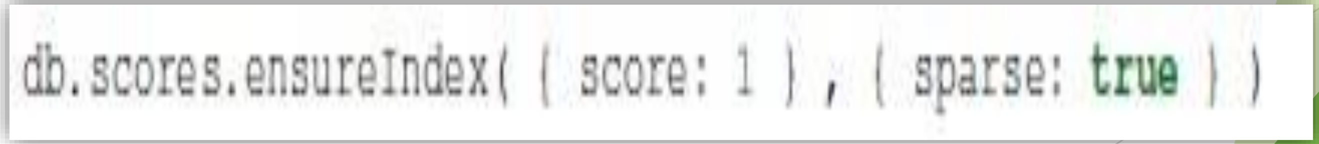
# Sparse Index (contd.)

The sparse property of an index ensures that the index only contain entries for documents that have the indexed field.

The index skips documents that do not have the indexed field.

You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.

If we create unique index on column of documents and it don’t exists in another documents then it will give error which will be taken care by **sparse : true**.

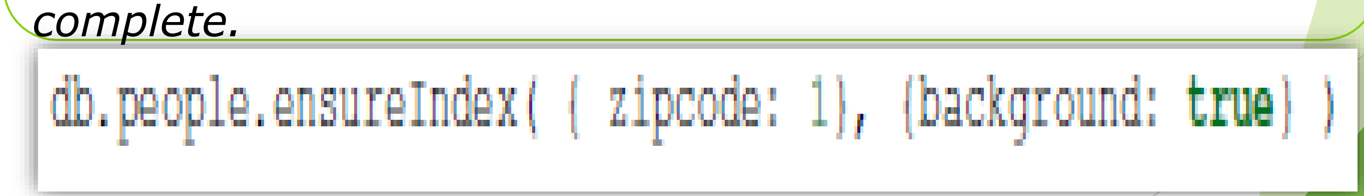


```
db.scores.ensureIndex( { score: 1 } , { sparse: true } )
```

# TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

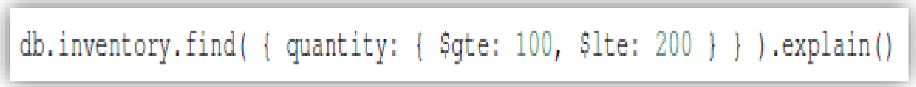
***Note :*** *By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any operation that requires a read or write lock databases and will wait for the foreground index build to complete.*



```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

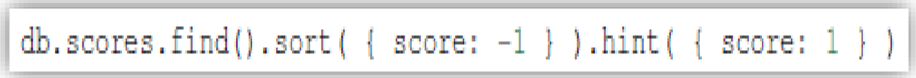
# Explain Plan in MongoDB

Like in other RDBMS database, MongoDB also provide Explain Plan option to find query is doing full table scan or Index scan.



```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } ).explain()
```

| | |
|---|---|
| **n** | No of document matches query selection criteria or query results. |
| **nscanned** | Indicate no of document Mongodb scanned index entry. |
| **nscannedObjects** | Total number of documents scanned. when this is equal to nscanned then collection scan. |
| **indexOnly** | True means that Mongodb used only the index to match and return result of query. |



```
db.scores.find().sort( { score: -1 } ).hint( { score: 1 } )
```

```
{
        "cursor" : "BtreeCursor
location_1",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" :
1,
        "nscannedAllPlans" : 1,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {
                "location" : [
                        [
                                null,
                                null
                        ]
                ]
        },
        "server" : "MongoDB_BIM",
        "filterSet" : false
```

# Why MongoDB: Performance

**No Joins + No multi-row transactions**

- = Fast Reads
- = Fast Writes (b/c you write to fewer tables, no trans. log)

**Async writes**

- = you don’t wait for inserts to complete.
- (optional, though)

- = Index on embedded document fields for superfast ad-hoc queries.
- Indexes live in RAM.

# Monitoring for MongoDB

- **Monitoring Strategies**
- **MongoDB Reporting Tools**
- **Process Logging**

# MongoDB Strategies

There are three methods for collecting data about the state of a running MongoDB instance:

First, there is a set of utilities distributed with MongoDB that provides real-time reporting of database activities.

Second, ***database commands*** return statistics regarding the current database state with greater fidelity.

Third, **MongoDB Cloud Manager**, a hosted service, and **Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced**, provide monitoring to collect data from running MongoDB deployments as well as providing visualization and alerts based on that data.

# MongoDB Reporting Tools

## Utilities

- The MongoDB distribution includes a number of utilities that quickly return statistics about instances' performance and activity. Typically, these are most useful for diagnosing issues and assessing normal operation.

# Mongostat

**Mongostat** captures and returns the counts of database operations by type (e.g. insert, query, update, delete, etc.).

These counts report on the load distribution on the server.

Use **mongostat** to understand the distribution of operation types and to inform capacity planning.

Mongostat is functionally similar to the UNIX/Linux file system utility vmstat, but provides data regarding **mongod** and **mongos** instances.

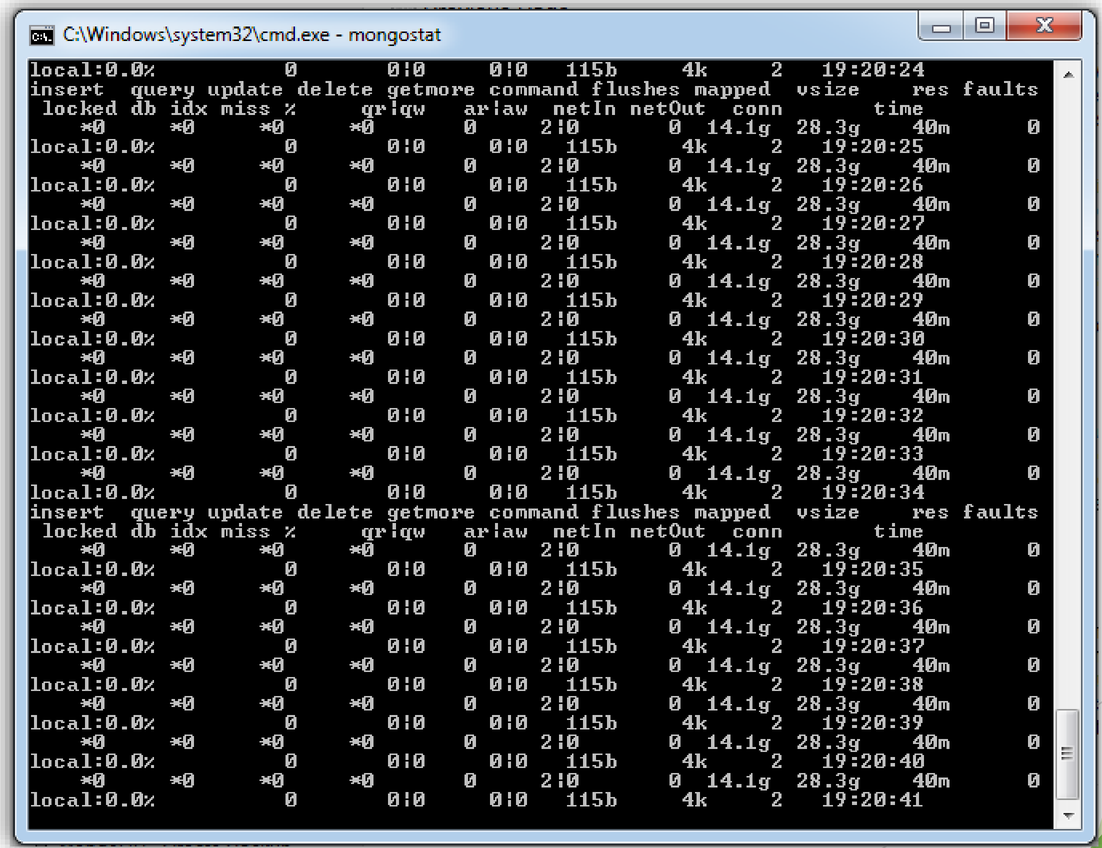
# Mongostat (contd.)

Command also shows when you’re hitting page faults, and showcase your lock percentage. This means that you're running low on memory, hitting write capacity or have some performance issue.

To run the command start your mongod instance. In another command prompt go to **bin directory of your mongodb** installation and type **mongostat.**

**Example:** D:\set up\mongodb\bin>mongostat

# Output of the Command



```
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:24
insert  query update delete getmore command flushes mapped  vsize    res faults
 locked db idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:25
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:26
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:27
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:28
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:29
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:30
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:31
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:32
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:33
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:34
insert  query update delete getmore command flushes mapped  vsize    res faults
 locked db idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:35
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:36
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:37
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:38
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:39
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:40
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0   115b     4k     2   19:20:41
```

# Mongotop Command

This command track and report the read and write activity of MongoDB instance on a collection basis.

By default mongotop returns information in each second, by you can change it accordingly.

We can check that this read and write activity matches your application intention, and you're not firing too many writes to the database at a time.

Reading too frequently from disk, or are exceeding your working set size.

# Mongotop Command (contd.)

To run the command start your mongod instance.

In another command prompt go to **bin directory of your mongodb** installation and type **mongotop.**

```
D:\set up\mongodb\bin>mongotop 30
```

The above example will return values every 30 seconds.

# ServerStatus Command

The **serverStatus** command, or **db.serverStatus()** from the shell, returns a general overview of the status of the database, detailing disk usage, memory use, connection, journaling, and index access.

The command returns quickly and does not impact MongoDB performance.

**serverStatus** outputs an account of the state of a MongoDB instance.

# Dbstats Command

The **dbStats** command, or **db.stats()** from the shell, returns a document that addresses storage use and data volumes.

The **dbStats** reflect the amount of storage used, the quantity of data contained in the database, and object, collection, and index counters.

# Collstats Command

- The **collStats** or **db.collection.stats()** from the shell that provides statistics that resemble **dbStats** on the collection level, including a count of the objects in the collection, the size of the collection, the amount of disk space used by the collection, and information about its indexes.

# ReplSetGetStatus Command

The **replSetGetStatus** command **(rs.status()** from the shell) returns an overview of your replica set's status. The ***replSetGetStatus*** document details the state and configuration of the replica set and statistics about its members.

# Process Logging

During normal operation, **mongod** and **mongos** instances report a live account of all server activity and operations to either standard output or a log file. The following runtime settings control these options.

**Verbosity:**

Increases the amount of information written to the log or output. You can also modify the logging verbosity during runtime with the **logLevel** parameter or the **db.setLogLevel()** method in the shell.

**Path:**

Enables logging to a file, rather than the standard output. You must specify the full path to the log file when adjusting this setting.

**LogAppend:**

Adds information to a log file instead of overwriting the file.

# Profiler

The database profiler collects fine grained data about MongoDB write operations, cursors, database commands on a running **mongod** instance.

You can enable profiling on a per-database or per-instance basis.

The ***profiling level*** is also configurable when enabling profiling. The profiler is *off* by default.

# Profiling Levels

The database profiler writes all the data it collects to the **system.profile** collection, which is a ***capped collection***.

**Profiling Levels -** The following profiling levels are available:

- **0:** The profiler is off, does not collect any data. **Mongod** always writes operations longer than the **slowOpThresholdMs** threshold to its log. This is the default profiler level.
- **1:** Collects profiling data for slow operations only. By default slow operations are those slower than 100 milliseconds.
- You can modify the threshold for "slow" operations with the **slowOpThresholdMs** runtime option or the **setParameter** command. See the ***Specify the Threshold for Slow Operations*** section for more information.
- **2:** Collects profiling data for all database operations.

# Enable Database Profiling and Set the Profiling Level

You can enable database profiling from the **mongo** shell or through a driver using the **profile** command.

When you enable profiling, you also set the ***profiling level***. The profiler records data in the **system.profile** collection.

MongoDB creates the **system.profile** collection in a database after you enable profiling for that database.

To enable profiling and set the profiling level, use the **db.setProfilingLevel()** helper in the **mongo** shell, passing the profiling level as a parameter.

# Enable Database Profiling and Set the Profiling Level (contd.)

To enable profiling for all database operations, consider the following operation in the **mongo** shell:

- **db.setProfilingLevel(2)**

The shell returns a document showing the *previous* level of profiling. The "ok" : 1 key-value pair indicates the operation succeeded:

- **{ "was" : 0, "slowms" : 100, "ok" : 1 }**

# Thresholds for Slow Operations

The threshold for slow operations applies to the entire **mongod** instance. When you change the threshold, you change it for all databases on the instance.

By default the slow operation threshold is 100 milliseconds. Databases with a profiling level of 1 will log operations slower than 100 milliseconds.

To change the threshold, pass two parameters to the **db.setProfilingLevel()** helper in the **mongo** shell. The first parameter sets the profiling level for the current database, and the second sets the default slow operation threshold *for the entire* **mongod** *instance*.

# Setting Profiling Level

The following command sets the profiling level for the current database to 0, which disables profiling, and sets the slow-operation threshold for the **mongod** instance to 20 milliseconds.

Any database on the instance with a profiling level of 1 will use this threshold:

- **db.setProfilingLevel(0,20)**

# Checking Profiling Level

To view the ***profiling level***, issue the following from the **mongo** shell:

- **db.getProfilingStatus()**

The shell returns a document similar to the following:

- **{ "was" : 0, "slowms" : 100 }**

The was field indicates the current level of profiling.

The slowms field indicates how long an operation must exist in milliseconds for an operation to pass the “slow” threshold. MongoDB will log operations that take longer than the threshold if the profiling level is 1.

# Get Profiling Level

To return only the profiling level, use the **db.getProfilingLevel()** helper in the **mongo** as in the following:

- db.getProfilingLevel()

# Disable Profiling

To disable profiling, use the following helper in the **mongo** shell:

- db.setProfilingLevel(0)

# Summary

Understand about Indexes

Understand different types of Indexes

Understand properties of Indexes

Explain Plan in MongoDB

Mongostat

Mongotop

Logging Slow queries

Profiling