Module 5

Application Designs

Introduction to Indexing

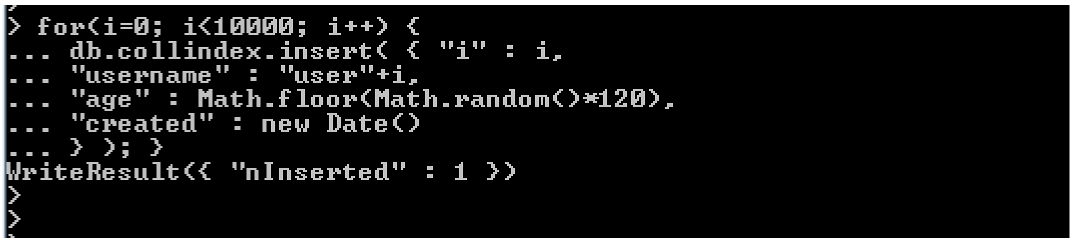
Table of Contents

Introduction to indexing:

A database is similar to a book's index. Instead of looking through the whole book, the database takes a shortcut and just looks at an ordered list that points to the content, which allows it to query orders of magnitude faster.

For example, let's create a collection with 10 thousand documents in a collection

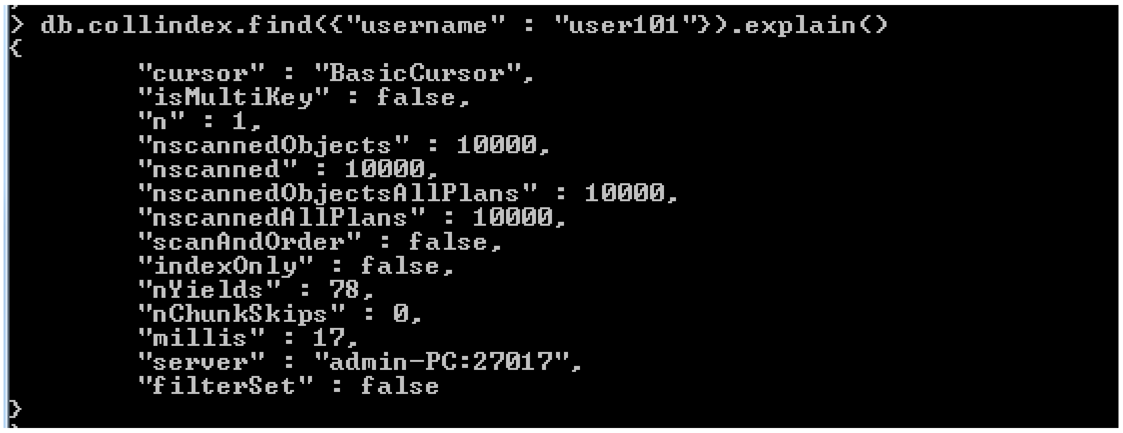
Like:



```
> for(i=0; i<10000; i++) {
... db.collindex.insert( { "i" : i,
... "username" : "user"+i,
... "age" : Math.floor(Math.random()*120),
... "created" : new Date()
... } ); }
WriteResult({ "nInserted" : 1 })
>
>
```

If we do a query on this collection, we can use the explain() function to see what MongoDB is doing when it execute the query.

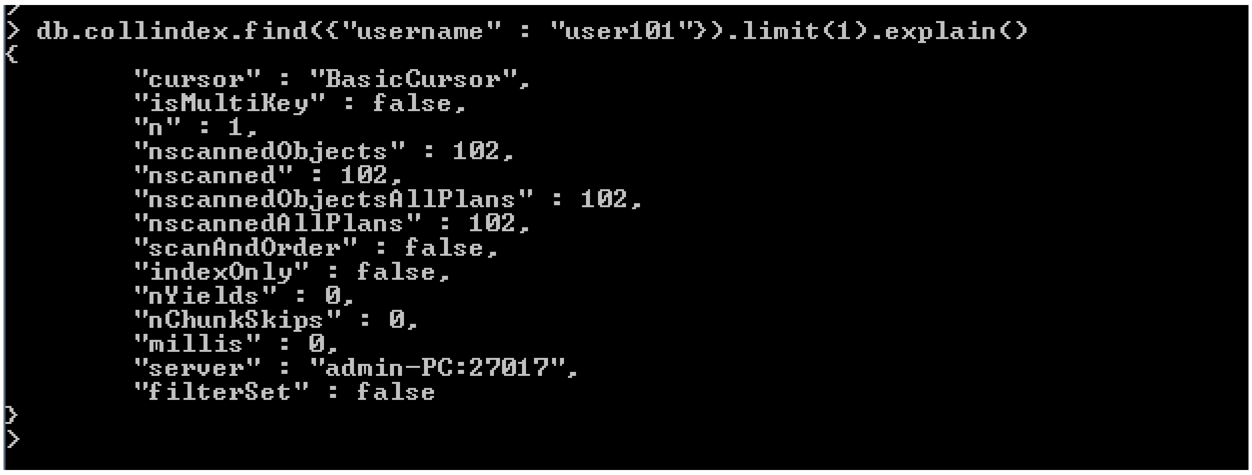
db.collindex.find({“username” : “user101”}).explain()



```
> db.collindex.find({"username" : "user101"}).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 10000,
        "nscanned" : 10000,
        "nscannedObjectsAllPlans" : 10000,
        "nscannedAllPlans" : 10000,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 78,
        "nChunkSkips" : 0,
        "millis" : 17,
        "server" : "admin-PC:27017",
        "filterSet" : false
}
```

Here you can see that “nscanned” is 10000, means the number of documents mongodb looked at while trying to satisfy the query, which, as you can see, is every document in the collection. That is mongoDB had to look through every field in every document. The “n” field shows the number of results returned: 1, which makes sense because there is only one user with the username “user101”.

To optimize this query, we could limit it to one result so that MongoDB would stop looking after it found “user101”.

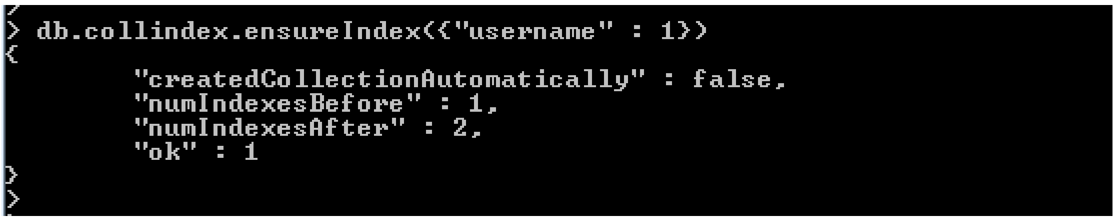

```
> db.collindex.find({"username" : "user101"}).limit(1).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 102,
        "nscanned" : 102,
        "nscannedObjectsAllPlans" : 102,
        "nscannedAllPlans" : 102,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "server" : "admin-PC:27017",
        "filterSet" : false
}
>
```

The number scanned has now been cut way down and the query is almost instantaneous. However, this is an impractical solution in general: what if we were looking for user9999? Then we would still have to traverse the entire collection and our service would just go slower.

Indexes are a great way to fix queries like this because they organize data by a given field to let MongoDB find it quickly.

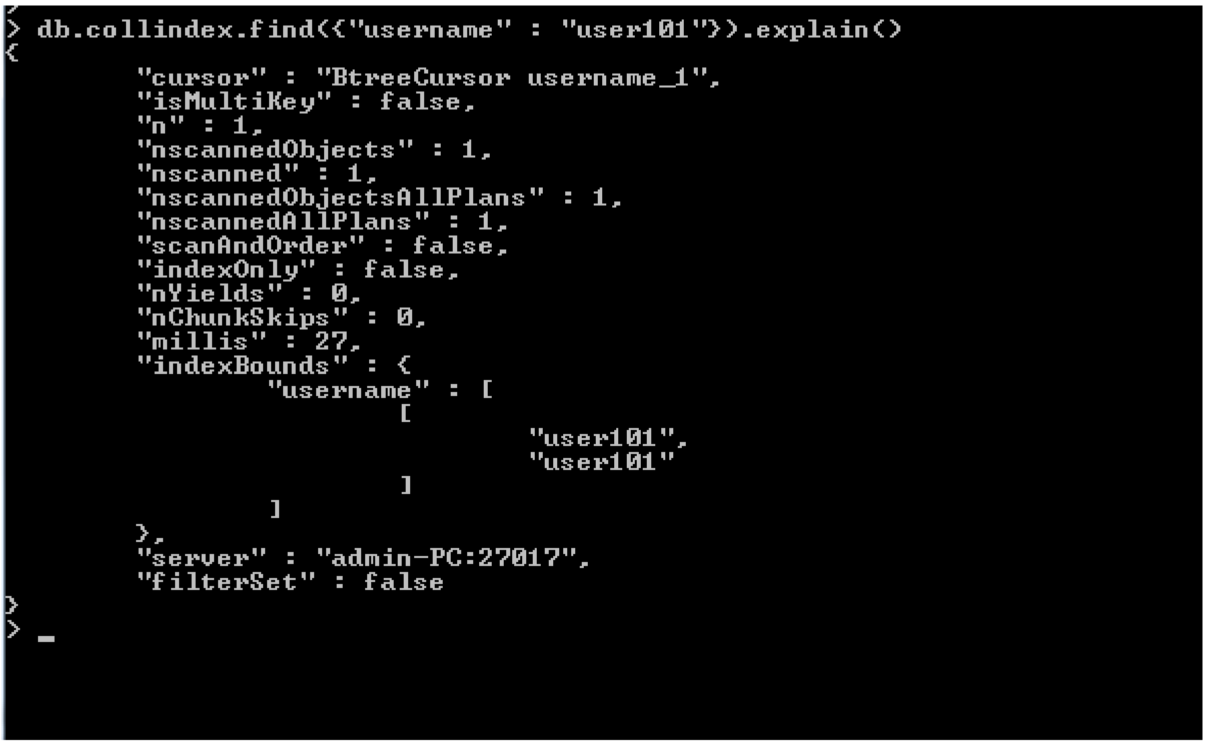
Try creating an index on the username field:

db.collindex.ensureIndex({“username” : 1})


```
> db.collindex.ensureIndex({"username" : 1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Here username is the name of field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Once the index build is complete, try repeating the original query:



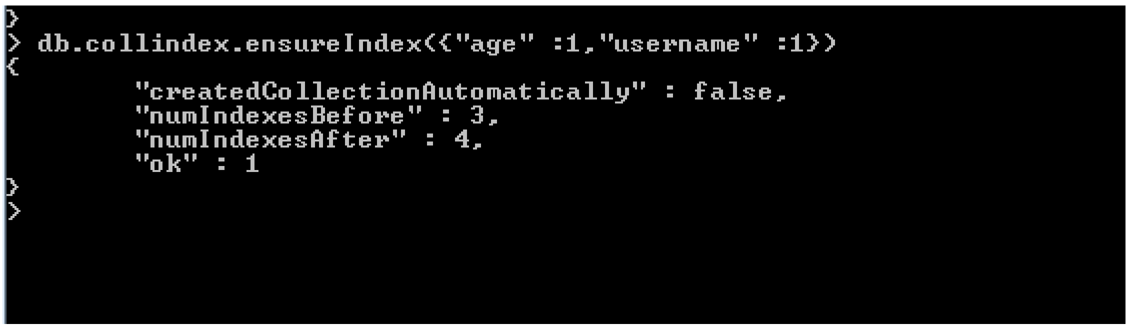
```
> db.collindex.find({"username" : "user101"}).explain()
{
        "cursor" : "BtreeCursor username_1",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" : 1,
        "nscannedAllPlans" : 1,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 27,
        "indexBounds" : {
                "username" : [
                        [
                                "user101",
                                "user101"
                        ]
                ]
        },
        "server" : "admin-PC:27017",
        "filterSet" : false
}
> _
```

From the above query, you can see that now MongoDB has scanned only one object. However, indexes have their price: every write (insert, update and delete) will take longer for every index you add. This is because MongoDB has to update all your indexes whenever your data changes, as well as the document itself. MongoDB limits you to 64 indexes per collection.

Introduction to Compound Indexes

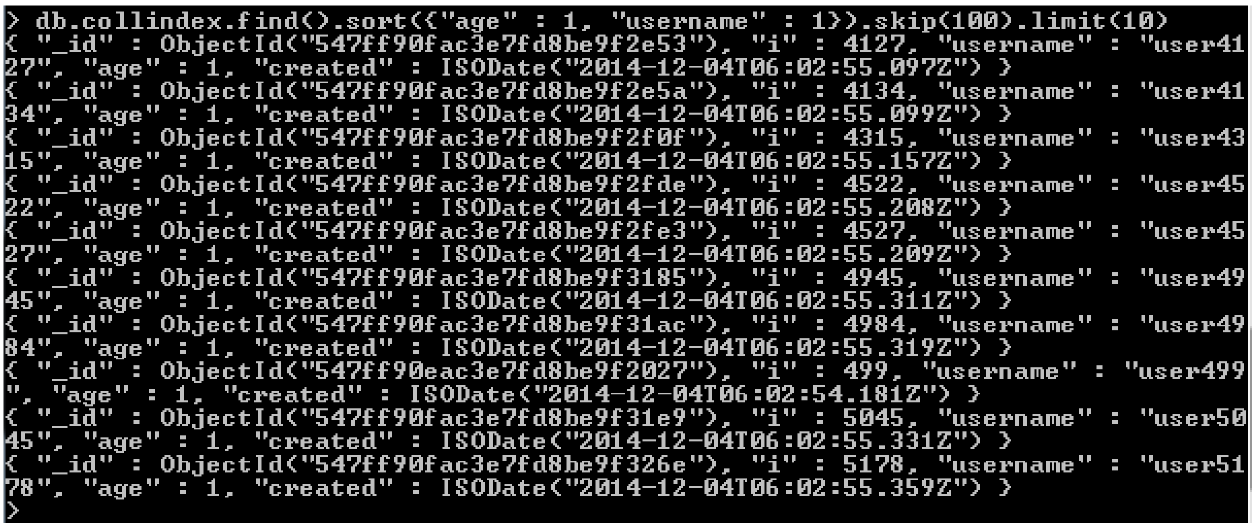
MongoDB also supports user-defined indexes on multiple fields. These compound field indexes behave like single-field indexes; however, the query can select documents based on additional fields. The order of fields listed in a compound index has significance.

For instance, if a compound index consists of { “username” : 1, “age” : -1}, the index sorts first by “username” and then, within each username value, sort by “age”.



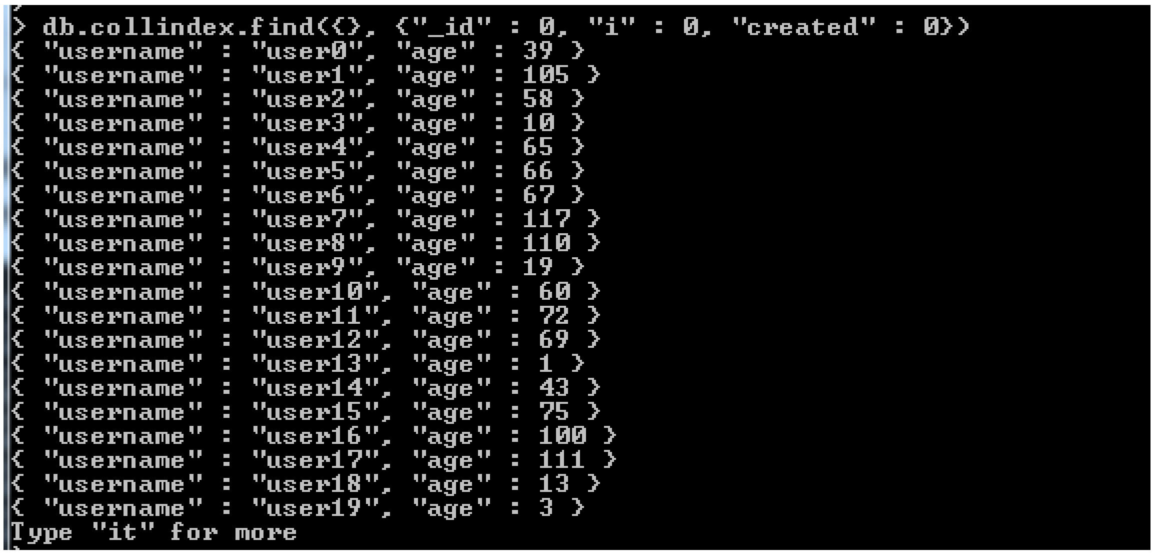
```
>
> db.collindex.ensureIndex({"age" :1,"username" :1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 3,
        "numIndexesAfter" : 4,
        "ok" : 1
}
>
```

This sorts by "age" and then "username", this is called compound index and is useful if your query has multiple sort directions or multiple keys in the criteria. A compound index is an index on more than one field.



```
> db.collindex.find().sort({"age" : 1, "username" : 1}).skip(100).limit(10)
{ "_id" : ObjectId("547ff90fac3e7fd8be9f2e53"), "i" : 4127, "username" : "user41
27", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.097Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f2e5a"), "i" : 4134, "username" : "user41
34", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.099Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f2f0f"), "i" : 4315, "username" : "user43
15", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.157Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f2fde"), "i" : 4522, "username" : "user45
22", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.208Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f2fe3"), "i" : 4527, "username" : "user45
27", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.209Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f3185"), "i" : 4945, "username" : "user49
45", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.311Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f31ac"), "i" : 4984, "username" : "user49
84", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.319Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f2027"), "i" : 499, "username" : "user499
", "age" : 1, "created" : ISODate("2014-12-04T06:02:54.181Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f31e9"), "i" : 5045, "username" : "user50
45", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.331Z") }
{ "_id" : ObjectId("547ff90fac3e7fd8be9f326e"), "i" : 5178, "username" : "user51
78", "age" : 1, "created" : ISODate("2014-12-04T06:02:55.359Z") }
>
```

Suppose we have a **collindex collection** that looks something like this, if we run a query with no sorting (called natural order):

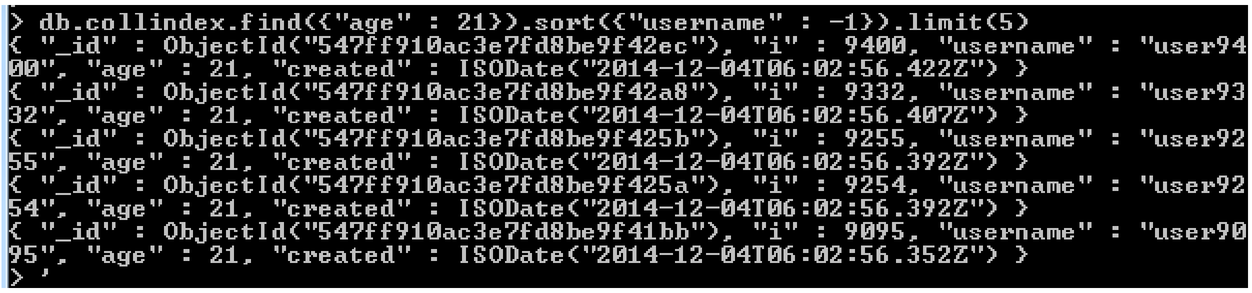


```
> db.collindex.find({}, {"_id" : 0, "i" : 0, "created" : 0})
{ "username" : "user0", "age" : 39 }
{ "username" : "user1", "age" : 105 }
{ "username" : "user2", "age" : 58 }
{ "username" : "user3", "age" : 10 }
{ "username" : "user4", "age" : 65 }
{ "username" : "user5", "age" : 66 }
{ "username" : "user6", "age" : 67 }
{ "username" : "user7", "age" : 117 }
{ "username" : "user8", "age" : 110 }
{ "username" : "user9", "age" : 19 }
{ "username" : "user10", "age" : 60 }
{ "username" : "user11", "age" : 72 }
{ "username" : "user12", "age" : 69 }
{ "username" : "user13", "age" : 1 }
{ "username" : "user14", "age" : 43 }
{ "username" : "user15", "age" : 75 }
{ "username" : "user16", "age" : 100 }
{ "username" : "user17", "age" : 111 }
{ "username" : "user18", "age" : 13 }
{ "username" : "user19", "age" : 3 }
Type "it" for more
```

The way MongoDB uses this index depends on the type of query you're doing. These are the three most common ways:

1. **db.collindex.find({“age” : 21}).sort({“username” : -1})**

MongoDB can start with the last match for {“age” : 21} and traverse the index in order:



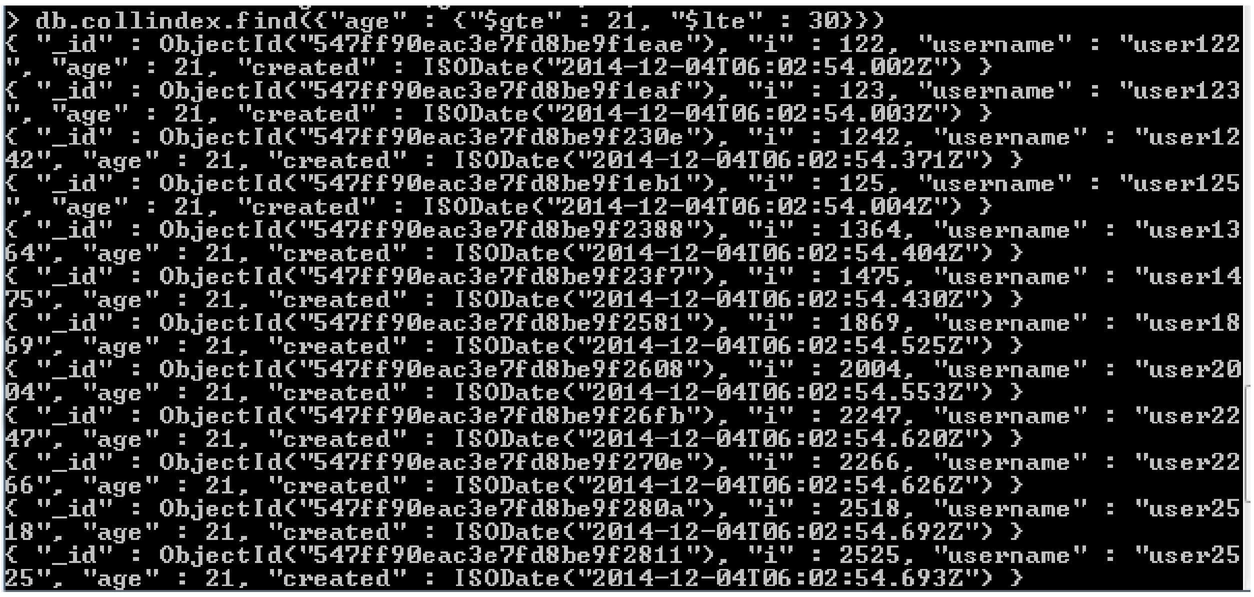
```
> db.collindex.find({"age" : 21}).sort({"username" : -1}).limit(5)
{ "_id" : ObjectId("547ff910ac3e7fd8be9f42ec"), "i" : 9400, "username" : "user94
00", "age" : 21, "created" : ISODate("2014-12-04T06:02:56.422Z") }
{ "_id" : ObjectId("547ff910ac3e7fd8be9f42a8"), "i" : 9332, "username" : "user93
32", "age" : 21, "created" : ISODate("2014-12-04T06:02:56.407Z") }
{ "_id" : ObjectId("547ff910ac3e7fd8be9f425b"), "i" : 9255, "username" : "user92
55", "age" : 21, "created" : ISODate("2014-12-04T06:02:56.392Z") }
{ "_id" : ObjectId("547ff910ac3e7fd8be9f425a"), "i" : 9254, "username" : "user92
54", "age" : 21, "created" : ISODate("2014-12-04T06:02:56.392Z") }
{ "_id" : ObjectId("547ff910ac3e7fd8be9f41bb"), "i" : 9095, "username" : "user90
95", "age" : 21, "created" : ISODate("2014-12-04T06:02:56.352Z") }
> '
```

This is a point query, which searches for a single value (although there may be multiple documents with that value). This type of query is very efficient: MongoDB can jump directly to the correct age and does not need to sort the results as traversing the index returns the data in the correct order.

Note: that sort direction does not matter, MongoDB is comfortable traversing the index in either direction.

2. **db.collindex.find({“age” : {“$gte” : 21, “$lte” : 30}})**

This is a multi-value query, which looks for documents matching multiple values (in this case, all ages between 21 and 30). MongoDB will use the first key in the index, “age”, to return the matching documents.


```
> db.collindex.find({"age" : {"$gte" : 21, "$lte" : 30}})
{ "_id" : ObjectId("547ff90eac3e7fd8be9f1eae"), "i" : 122, "username" : "user122
", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.002Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f1eaf"), "i" : 123, "username" : "user123
", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.003Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f230e"), "i" : 1242, "username" : "user12
42", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.371Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f1eb1"), "i" : 125, "username" : "user125
", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.004Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f2388"), "i" : 1364, "username" : "user13
64", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.404Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f23f7"), "i" : 1475, "username" : "user14
75", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.430Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f2581"), "i" : 1869, "username" : "user18
69", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.525Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f2608"), "i" : 2004, "username" : "user20
04", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.553Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f26fb"), "i" : 2247, "username" : "user22
47", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.620Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f270e"), "i" : 2266, "username" : "user22
66", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.626Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f280a"), "i" : 2518, "username" : "user25
18", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.692Z") }
{ "_id" : ObjectId("547ff90eac3e7fd8be9f2811"), "i" : 2525, "username" : "user25
25", "age" : 21, "created" : ISODate("2014-12-04T06:02:54.693Z") }
```

3. **db.collindex.find({“age” : {“$gte” : 21, “$lte” : 30}}).sort({“username” : 1})**

This is a multi-value query, like the previous one, but this time it has a sort. However, the index does not return the usernames in sorted order and the query requested that the results be sorted by username, so MongoDB has to sort the results in memory before returning them. Thus, this query is usually less efficient than the queries above.

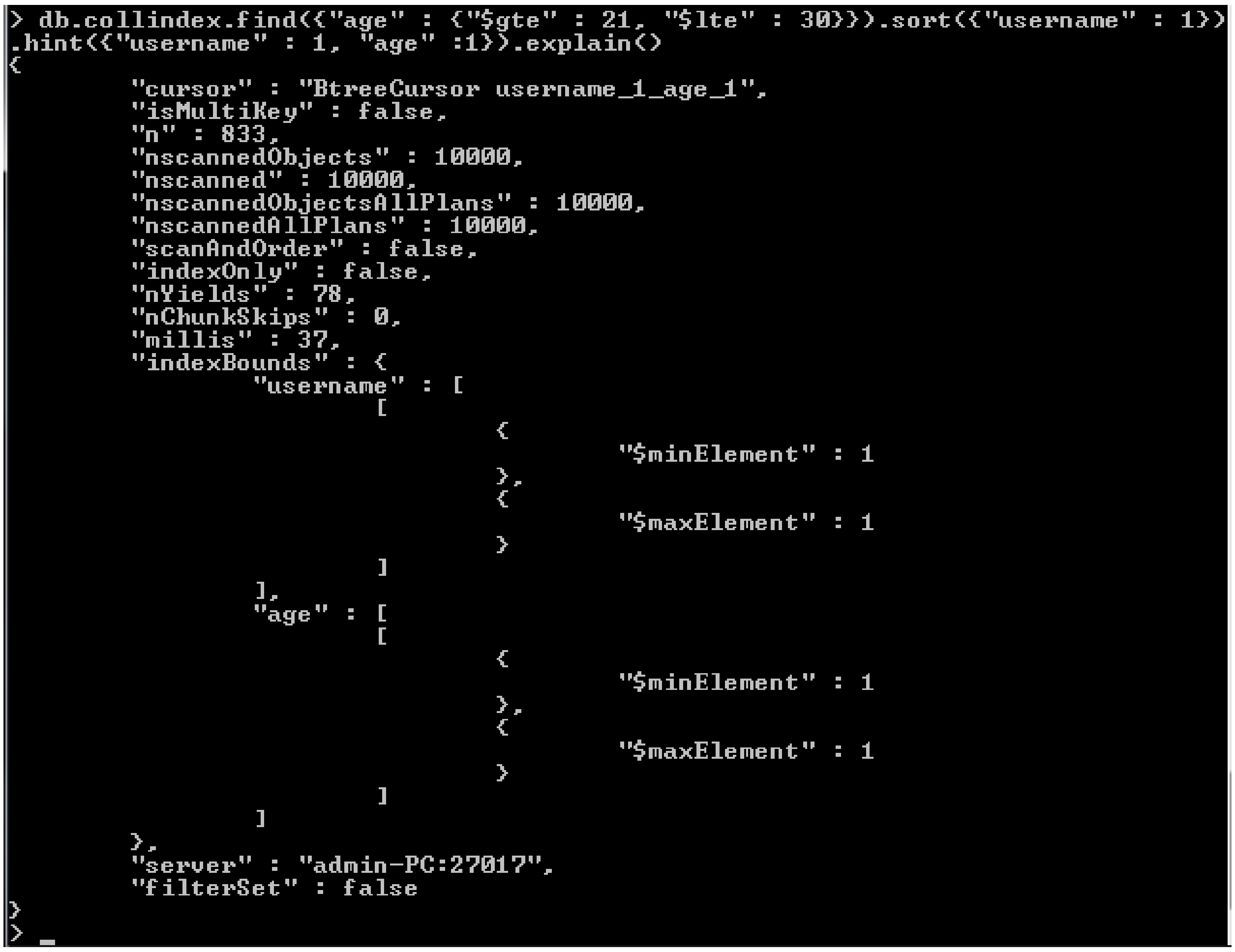
We can use hint to force MongoDB to use a certain index, so try the same query again using the {“username”: 1 , ”age” : 1 } index, instead. This query scans more documents, but does not have to do an in-memory sort.

db.collindex.find({“age” : {“$gte” : 21, “$lte” : 30}}).

sort({“username” : 1}).

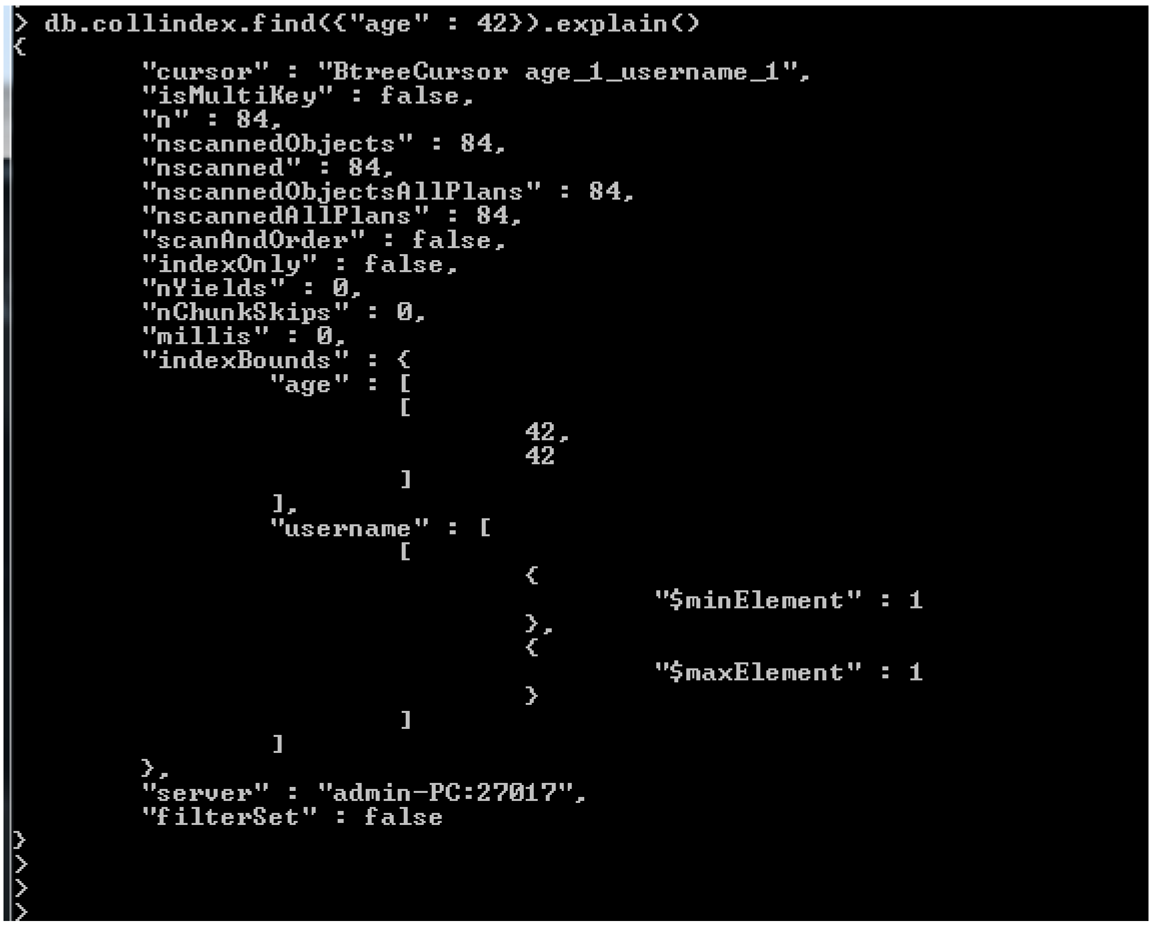
hint({“username” : 1, “age” : 1}).

explain()



```
> db.collindex.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" : 1})
.hint({"username" : 1, "age" :1}).explain()
{
        "cursor" : "BtreeCursor username_1_age_1",
        "isMultiKey" : false,
        "n" : 833,
        "nscannedObjects" : 10000,
        "nscanned" : 10000,
        "nscannedObjectsAllPlans" : 10000,
        "nscannedAllPlans" : 10000,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 78,
        "nChunkSkips" : 0,
        "millis" : 37,
        "indexBounds" : {
                "username" : [
                        [
                                {
                                        "$minElement" : 1
                                },
                                {
                                        "$maxElement" : 1
                                }
                        ]
                ],
                "age" : [
                        [
                                {
                                        "$minElement" : 1
                                },
                                {
                                        "$maxElement" : 1
                                }
                        ]
                ]
        },
        "server" : "admin-PC:27017",
        "filterSet" : false
}
> _
```

Overview of all the fields:



```
> db.collindex.find({"age" : 42}).explain()
{
        "cursor" : "BtreeCursor age_1_username_1",
        "isMultiKey" : false,
        "n" : 84,
        "nscannedObjects" : 84,
        "nscanned" : 84,
        "nscannedObjectsAllPlans" : 84,
        "nscannedAllPlans" : 84,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {
                "age" : [
                        [
                                42,
                                42
                        ]
                ],
                "username" : [
                        [
                                {
                                        "$minElement" : 1
                                },
                                {
                                        "$maxElement" : 1
                                }
                        ]
                ]
        },
        "server" : "admin-PC:27017",
        "filterSet" : false
}
>
>
>
>
```

“cursor” : “BtreeCursor age_1_username_1”

BtreeCursor means that an index was used, specifically, the index on age and username: {“age” : 1, “username” : 1}.

“isMultiKey” : false

If the query uses a multikey index

“n” : 84

The number of documents returned by a query.

“nscannedObjects” : 84

This is the count of number of times MongoDB had to follow an index pointer to the actual document on the disk. If the query contains criteria that is not part of the index or requests fields back that aren't contained in the index, MongoDB must look up the document each index entry points to.

"nscanned" : 84

The number of index looked at if an index was used. If this was a table scan, it is the number of documents examined.

"scanAndOrder" : false

If MongoDB had to sort results in memory.

"indexOnly" : false

If MongoDB was able to fulfill this query using only the index entries.

"nYields" : 0

The number of times this query yielded (paused) to allow a write request to proceed. If there are writes waiting to go, queries will periodically release their lock and allow them to do so. Here, there are no writes waiting so the query never yielded.

"millis" : 0

The number of milliseconds it took the database to execute the query. The lower this number is, the better.

"indexBounds" : {…}

This field describes how the index was used, giving ranges of the index traversed. As the first clause in the query was an exact match, the index only needed to look at that value: 42. The second key was a free variable, as the query didn't specify any restrictions to it. Thus, the database looked for values between negative infinity ("$minElement" : 1) and infinity ("$maxElement" : 1) for usernames within "age" : 42.

MultiKey Indexes

To index a field that holds an array value, MongoDB adds index items for each item in the array. These multikey indexes allow MongoDB to return documents from queries using the value of an array. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Multikey indexes support arrays that hold both values (e.g. strings, numbers) and nested documents.

MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

Let's take an example:

{

"_id" : 1,

"name" : "mary",

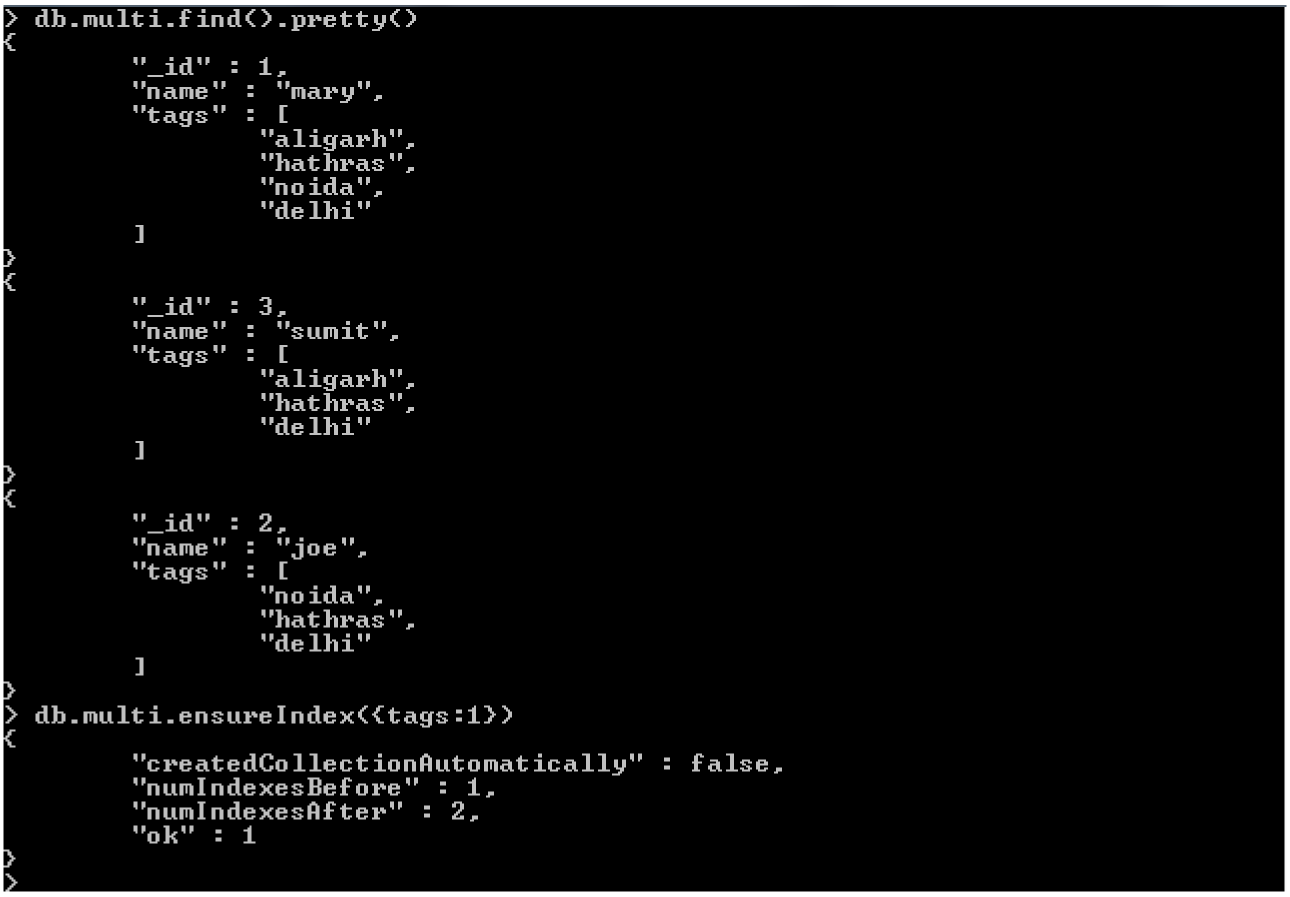
"tags" : ["aligarh", "hathras" ,"noida", "delhi"]

}

In this document, an index on the tags field, {tags : 1}, would be a multikey index and would include these given below four separate entries for that document:

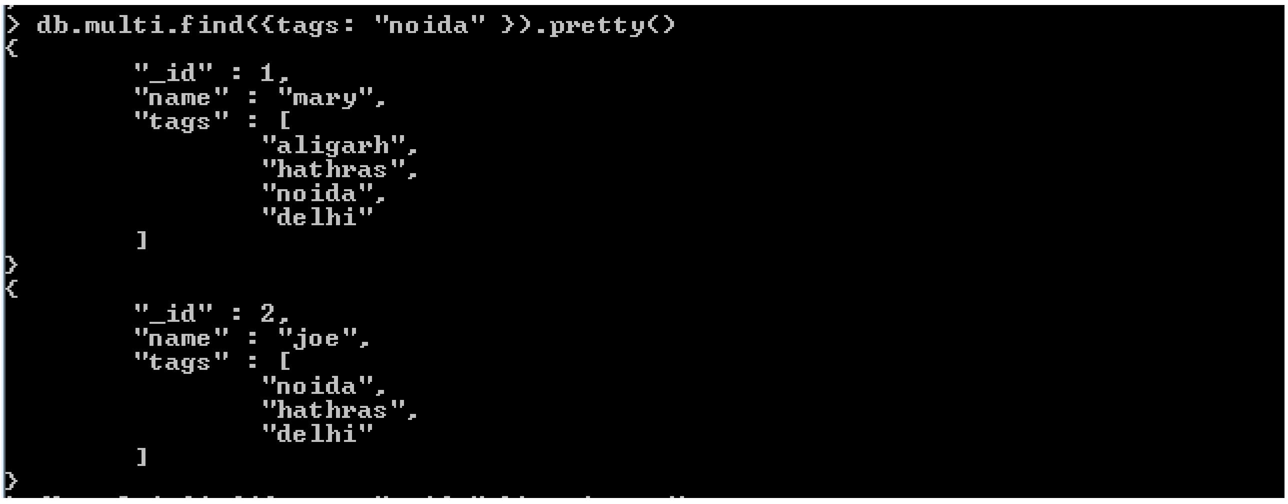
"aligarh", "hathras", "noida", "delhi"

In this condition, queries could use the multikey index to return the queries for any of the above values.



```
> db.multi.find().pretty()
{
        "_id" : 1,
        "name" : "mary",
        "tags" : [
                "aligarh",
                "hathras",
                "noida",
                "delhi"
        ]
}
{
        "_id" : 3,
        "name" : "sumit",
        "tags" : [
                "aligarh",
                "hathras",
                "delhi"
        ]
}
{
        "_id" : 2,
        "name" : "joe",
        "tags" : [
                "noida",
                "hathras",
                "delhi"
        ]
}
> db.multi.ensureIndex({tags:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Here the documents are displayed using the multikey index on tags field.



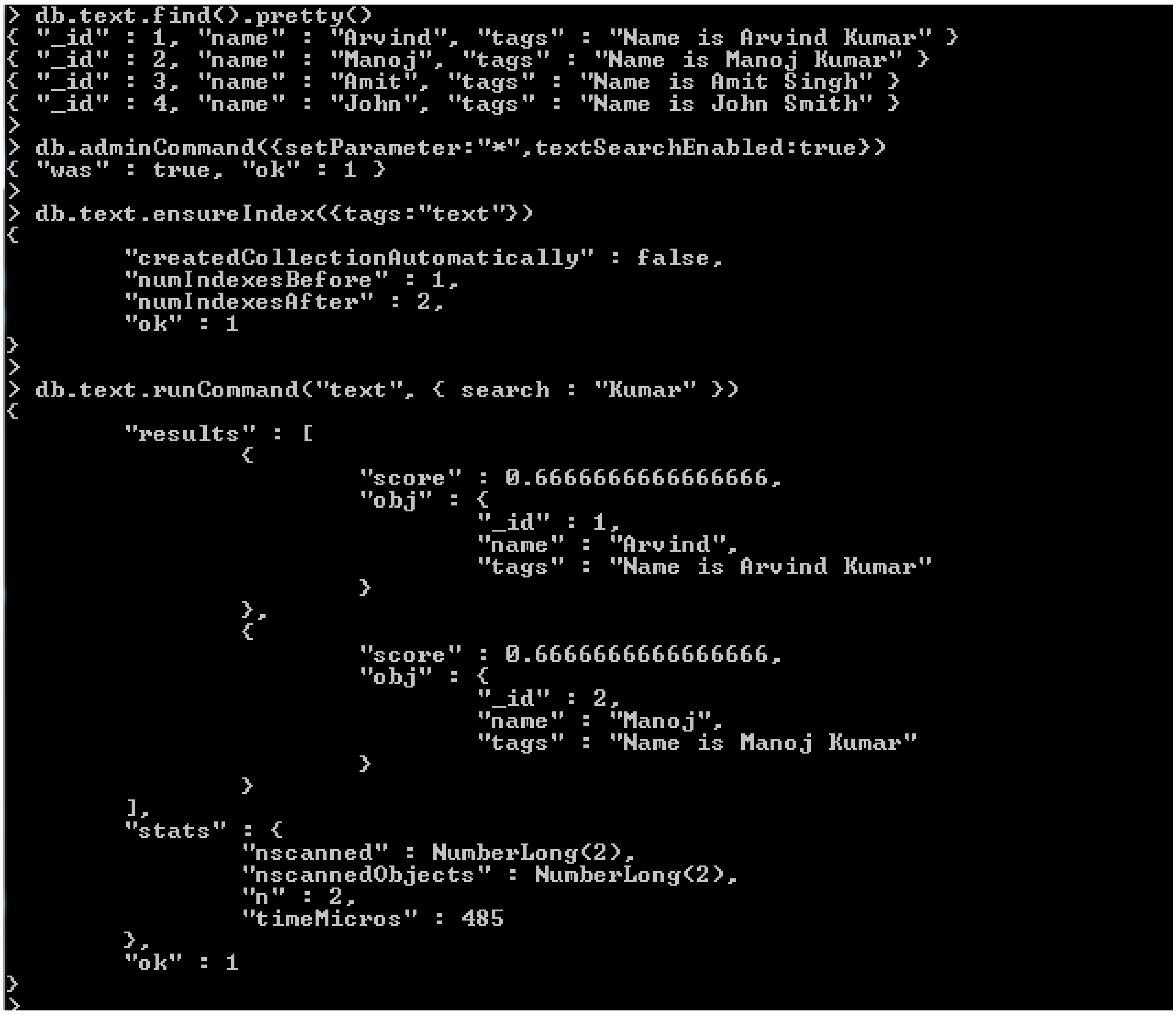
```
> db.multi.find({tags: "noida" }).pretty()
{
        "_id" : 1,
        "name" : "mary",
        "tags" : [
                "aligarh",
                "hathras",
                "noida",
                "delhi"
        ]
}
{
        "_id" : 2,
        "name" : "joe",
        "tags" : [
                "noida",
                "hathras",
                "delhi"
        ]
}
```

Text Indexes: MongoDB provides a text index type that supports searching for string content in a collection. Text indexes can include any field whose value is a string or an array of string elements. To perform queries that access the text index, use the $text query operator.

Here a text collection has some pre-defined documents. Then we have to manually enable the text search, using **db.adminCommand({setParameter: "*", textSearchEnabled:true}).**

Then create a text index on "tags" field, using **ensureIndex** command.

And finally run the command using text indexes to search the documents.



```
> db.text.find().pretty()
{ "_id" : 1, "name" : "Arvind", "tags" : "Name is Arvind Kumar" }
{ "_id" : 2, "name" : "Manoj", "tags" : "Name is Manoj Kumar" }
{ "_id" : 3, "name" : "Amit", "tags" : "Name is Amit Singh" }
{ "_id" : 4, "name" : "John", "tags" : "Name is John Smith" }
>
> db.adminCommand({setParameter:"*",textSearchEnabled:true})
{ "was" : true, "ok" : 1 }
>
> db.text.ensureIndex({tags:"text"})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
> db.text.runCommand("text", { search : "Kumar" })
{
        "results" : [
                {
                        "score" : 0.6666666666666666,
                        "obj" : {
                                "_id" : 1,
                                "name" : "Arvind",
                                "tags" : "Name is Arvind Kumar"
                        }
                },
                {
                        "score" : 0.6666666666666666,
                        "obj" : {
                                "_id" : 2,
                                "name" : "Manoj",
                                "tags" : "Name is Manoj Kumar"
                        }
                }
        ],
        "stats" : {
                "nscanned" : NumberLong(2),
                "nscannedObjects" : NumberLong(2),
                "n" : 2,
                "timeMicros" : 485
        },
        "ok" : 1
}
>
```

Types of Indexes: **there are a few index options you can specify when building an index that changes the way the index behaves.**

1. *Unique indexes*: Unique indexes guarantee that each value will appear at most once in the index. They causes MongoDB to reject all documents that contain a duplicate value for indexed field.

   For example: if you want to make sure no two documents can have the same value in the "username" key, you can create a unique index:

   ```
   db.collindex.ensureIndex({"username" : 1},{"unique" : true})
   ```

   Suppose that we try to insert the following documents on the collection above:

   ```
   db.collindex.insert({"username" : "bob"})
   db.collindex.insert({"username" : "bob"})
   E11000 duplicate key error index
   ```

   If you check the collection, you'll see that only the first "bob" was stored. A unique index that you are probably already familiar with is the index on "_id", which is automatically created whenever you create a collection. This is a normal unique index (aside from the fact that it cannot be dropped as other unique indexes can be).

2. *Sparse Indexes*: Only contain entries for documents that have the indexed field, even if the indexed field contains a null value.

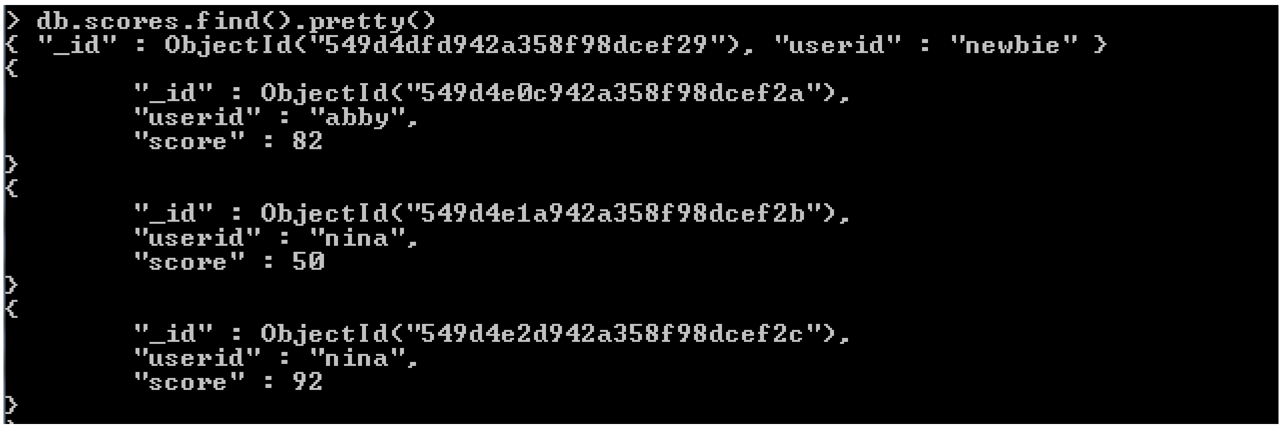
   As mentioned in an earlier section, unique indexes count null as a value, so you cannot have a unique index with more than one document missing the key. However, there are lots of cases where you want the unique index to be enforced only if the key exists. If you have a field that may or may not exists but must be unique when it does, you can combine the unique option with the sparse option.

   MongoDB sparse indexes are different from indexes on relational databases, they are basically indexes that need not include every document as an entry.

   To create a sparse index, include the sparse option. For example, if providing an email address was optional but, if provided, should be unique:

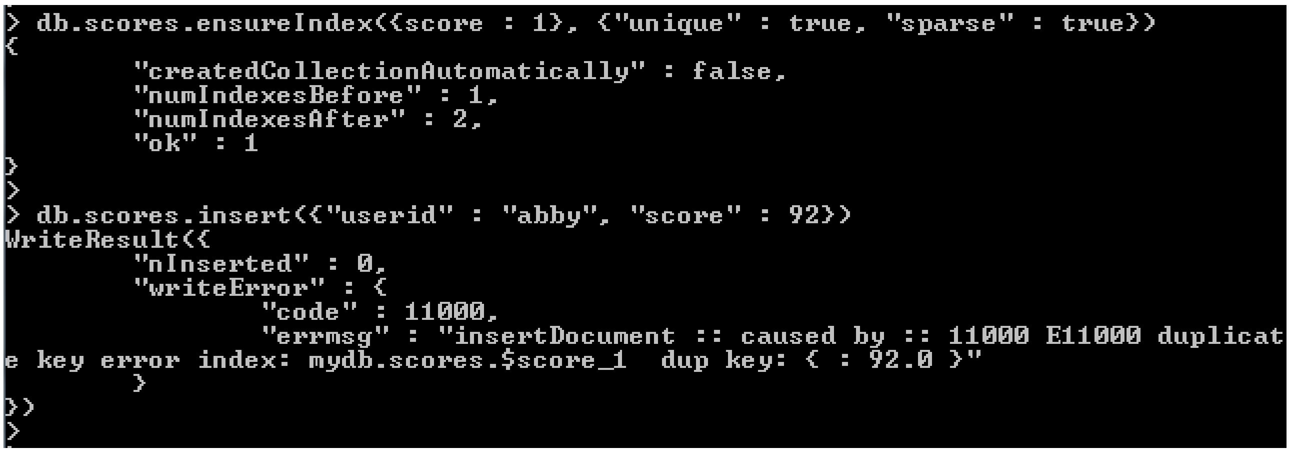
   db.collindex.ensureIndex({"email" : 1}, {"unique" : true, "sparse" : true})

Suppose we have a collection called scores, which has the following documents containing userid and score field.



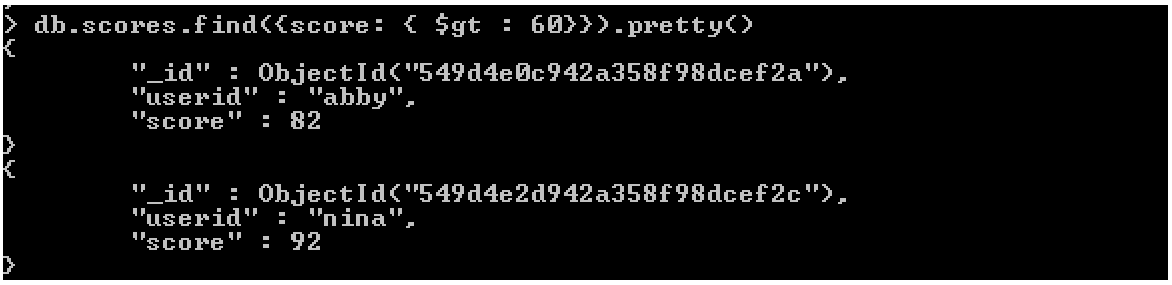
```
> db.scores.find().pretty()
{ "_id" : ObjectId("549d4dfd942a358f98dcef29"), "userid" : "newbie" }
{
        "_id" : ObjectId("549d4e0c942a358f98dcef2a"),
        "userid" : "abby",
        "score" : 82
}
{
        "_id" : ObjectId("549d4e1a942a358f98dcef2b"),
        "userid" : "nina",
        "score" : 50
}
{
        "_id" : ObjectId("549d4e2d942a358f98dcef2c"),
        "userid" : "nina",
        "score" : 92
}
```

We put the index on score field with unique and sparse as true.



```
> db.scores.ensureIndex({score : 1}, {"unique" : true, "sparse" : true})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
> db.scores.insert({"userid" : "abby", "score" : 92})
WriteResult({
        "nInserted" : 0,
        "writeError" : {
                "code" : 11000,
                "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicat
e key error index: mydb.scores.$score_1  dup key: { : 92.0 }"
        }
})
>
```

Then, the following query on the scores collection uses the sparse index to return the documents that have the score field greater than 60. Because the document for the userid "newbie" does not contain the score field and thus does not meet the query criteria, the query can use the sparse index to return the results:



```
> db.scores.find({score: { $gt : 60}}).pretty()
{
        "_id" : ObjectId("549d4e0c942a358f98dcef2a"),
        "userid" : "abby",
        "score" : 82
}
{
        "_id" : ObjectId("549d4e2d942a358f98dcef2c"),
        "userid" : "nina",
        "score" : 92
}
```

3. ***Time-to-Live Indexes*:** These indexes allow you to set a timeout for each document. When a document reaches a preconfigured age, it will be deleted. This type of index is useful for caching problems like session storage.
   You can create a TTL index by specifying the expireAfterSecs option in the second argument to ensureIndex:

   db.collindex.ensureIndex({“created” : 1}, {“expireAfterSeconds” : 60*60*24})

   This creates a TTL index on the “created” field. If a documents “created” field exists and is a date, the document will be removed once the server time expireAfterSecs seconds ahead of the document’s time.
   Limitations of TTL indexes:
   1. Compound indexes are not supported.
   2. The indexed field must be a date type.
   3. If the field holds an array, and there are multiple date-typed data in the index, the document will expire when the lowest (i.e. earliest) matches the expiration threshold.

The TTL index does not guarantee that expired data will be deleted immediately. There may be a delay between the time a document expires and the time that MongoDB removes the document from the database.

Limitations of Indexing:

1. **Extra Overhead:**
   Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

2. **RAM Usage:**
   Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes and hence causing performance loss.

3. **Query Limitations:** indexing cannot be used in queries which use:
   Regular expressions or negation operations like $nin, $not, etc.
   Arithmetic operators like $mod, etc.
   $Where clause

4. **Index Key Limits:**
   Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

5. **Inserting Documents exceeding Index Key Limit:**
   MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

6. **Maximum Ranges:**
   A collection cannot have more than 64 indexes
   The length of the index name cannot be longer than 125 characters
   A compound index can have maximum 31 fields indexed

Geospatial Indexing

MongoDB has a few types of geospatial indexes. The most commonly used ones are 2dsphere, for surface-of-the-earth-type maps, and 2d, for flat maps (and time series data).
2dsphere allows you to specify points, lines, and polygons in GeoJSON format.
A point is given by a two-element array, representing [longitude, latitude]:

```
{
"name" : "New York City",
"loc" : {
        "type" : "Point",
        "coordinates" : [50,2]
        }
}
```

A line is given by an array of points:

```
{
"name" : "Hudson River",
"loc" : {
        "type" : "Line",
        "coordinates" : [[0,1], [0,2], [1,2]]
        }
}
```

A polygon is specified the same way a line is (an array of points), but with a different "type":

```
{
"name" : "New England",
"loc" : {
        "type" : "Polygon",
        "coordinates" : [[0,1], [0,2], [1,2]]
        }
}
```

The "loc" field can be called anything, but the field names within its subobject are specified by GeoJSON and cannot be changed.
You can create a geospatial index using the "2dsphere" type with ensureIndex:
db.world.ensureIndex({"loc" : "2dsphere"})

Index Management

Remove indexes: to drop or remove an index from a collection use the dropIndex() method.

To remove a specific index, use db.collection.dropIndex() method.

For example: following operation removes an ascending index on the username filed in the sales collection:

db.sales.dropIndex ({ “username” : 1 })

And if you do not pass any value in dropIndex(), all the indexes will be removed except the _id index from a collection.

Modify the index: to modify an index, you just cannot change the existing index. First you have to delete all the indexes and then create the index again.

Rebuild Indexes: if you need to rebuild indexes for a collection you can use the db.collection.reIndex() method to rebuild all indexes on a collection in a single operation.

To return a list of all indexes: if you want to check which indexes exist on a collection. Every index on a collection has a corresponding document in the system.indexes collection, and you can use standard queries (i.e. find()) to list the indexes, or in the mongo shell, the getIndexes() method to return a list of the indexes on a collection.

For example: to view all indexes on the people collection, db.people.getIndexes().

And to list all indexes for a database, use db.system.indexes.find() operation.

Capped Collections

Capped collections are fixed-size collections that support high throughput operations that insert, retrieve and delete documents based on insertion order.

“Normal” collections in MongoDB are created dynamically and automatically grow in size and fit additional data. MongoDB also supports a different type of collection that is fixed in size, called capped collections.
With MongoDB one may create collections of a predefined size, where old data automatically ages out on a least recently inserted basis.

db.createCollection(“mycoll”, {capped : true, size : 100000})

When capped, a MongoDB collection has a couple of interesting properties. **First**, the data automatically ages out when the collection is full on a least recently inserted basis.
Second, for capped collections, MongoDB automatically keeps the objects in the collection in their insertion order. To retrieve items in their insertion order:
db.mycoll.find().sort({$natural : 1}); // oldest to newest
db.mycoll.find().sort({$natural : -1}); // newest to oldest

The implementation of the above two properties in the database is done at a low level and is very fast and efficient.

Procedures

Create a Capped Collection

You must create capped collections explicitly using the createCollection() method. When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection.
db.createCollection(“log” ,{capped : true, size : 100000 })

Or you can specify a maximum number of documents for the collection using the max field as in the following document:
db.createCollection(“log” , { capped : true , size : 5242880, max : 5000 })

Query a capped collection

If you query a find() on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.
To retrieve documents in reverse insertion order specified, issue find() along with the sort() method with the **$natural** parameter set to -**1**.
db.cappedCollection.find().sort ({ $natural : -1})

To check if a Collection is Capped:

isCapped() method is used to determine if a collection is capped.
db.collection.isCapped()

Convert a Collection to Capped:

You can convert a non-capped collection with the **convertToCapped** command.
db.runCommand({ “convertToCapped” : “mycoll” , size : 100000 });

Restrictions on Capped Collections:

- You can only make in-place updates of documents. If the update operation causes the document to grow beyond their original size, the update operation will fail. If you plan to update documents in a capped collection, create an index so that these update operations do not require a table scan.
- You cannot delete documents from a capped collection. To remove all documents from a collection, use the drop() method to drop the collection.
- You cannot shard a capped collection.
- Use natural ordering to retrieve the most recently inserted elements from the collection efficiently.
- The aggregation pipeline operator **$out** cannot write results to a capped collection.