MONGODB INDEXES: CREATING, FINDING & DROPPING TOP INDEX TYPES

Indexes provide users with an efficient way of querying data. When querying data without indexes, the query will have to search for all the records within a database to find data that match the query.

In MongoDB, querying without indexes is called a collection scan. A collection scan will:

- Result in various performance bottlenecks
- Significantly slow down your application

Fortunately, using indexes fixes both these issues. By limiting the number of documents to be queried, you'll increases the overall performance of the application.

In this tutorial, I'll walk you through different types of indexes and show you how to create and manage indexes in MongoDB.

(This article is part of our MongoDB Guide. Use the right-hand menu to navigate.)

What are indexes in MongoDB?

Indexes are special data structures that store a small part of the Collection's data in a way that can be queried easily.

In simplest terms, indexes store the values of the indexed fields outside the table or collection and keep track of their location in the disk. These values are used to order the indexed fields. This ordering helps to perform equality matches and range-based query operations efficiently. In MongoDB, indexes are defined in the collection level and indexes on any field or subfield of the

documents in a collection are supported.

For this tutorial, we'll use the following data set to demonstrate the indexing functionality of MongoDB.

```
use students
db.createCollection("studentgrades")
db.studentgrades.insertMany(

)
db.studentgrades.find({},{_id:0})
```

Result

```
> db.studentgrades.find({},{_id:0})
{ "name" : "Barry", "subject" : "Maths", "score" : 92 }
{ "name" : "Kent", "subject" : "Physics", "score" : 87 }
{ "name" : "Harry", "subject" : "Maths", "score" : 99, "notes" : "Exceptional Performance" }
{ "name" : "Alex", "subject" : "Literature", "score" : 78 }
{ "name" : "Tom", "subject" : "History", "score" : 65, "notes" : "Adequate" }
>
```

Creating indexes

When creating documents in a collection, MongoDB creates a unique index using the _id field. MongoDB refers to this as the **Default _id Index**. This default index cannot be dropped from the collection.

When querying the test data set, you can see the _id field which will be utilized as the default index:

```
db.studentgrades.find().pretty()
```

Result:

```
> db.studentgrades.find().pretty()
{
        "_id" : ObjectId("6026a176de0e65dbecbef031"),
        "name" : "Barry",
        "subject" : "Maths",
        "score" : 92
}
{
        "_id" : ObjectId("6026a176de0e65dbecbef032"),
        "name" : "Kent",
        "subject" : "Physics",
        "score" : 87
}
{
        "_id" : ObjectId("6026a176de0e65dbecbef033"),
        "name" : "Harry",
        "subject" : "Maths",
        "score" : 99,
        "notes" : "Exceptional Performance"
}
{
        "_id" : ObjectId("6026a176de0e65dbecbef034"),
        "name" : "Alex",
        "subject" : "Literature",
        "score" : 78
}
{
        "_id" : ObjectId("6026a176de0e65dbecbef035"),
        "name" : "Tom",
        "subject" : "History",
        "score" : 65,
        "notes" : "Adequate"
}
>
```

Now let's create an index. To do that, you can use the **createIndex** method using the following syntax:

```
db.<collection>.createIndex(<Key and Index Type>, <Options>)
```

When creating an index, you need to define the field to be indexed and the direction of the key (1 or -1) to indicate ascending or descending order.

Another thing to keep in mind is the index names. By default, MongoDB will generate index names by concatenating the indexed keys with the direction of each key in the index using an underscore as the separator. For example: {name: 1} will be created as name_1.

The best option is to use the name option to define a custom index name when creating an index. Indexes cannot be renamed after creation. (The only way to rename an index is to first drop that index, which we show below, and recreate it using the desired name.)

Let's create an index using the name field in the studentgrades collection and name it as **student name index**.

```
db.studentgrades.createIndex(
{name: 1},
```

```
{name: "student name index"}
)
```

Result:

```
> db.studentgrades.createIndex(
...     {name: 1},
...     {name: "student name index"}
... )
{
        "createdCollectionAutomatically" : true,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

Finding indexes

You can find all the available indexes in a MongoDB collection by using the **getIndexes** method. This will return all the indexes in a specific collection.

`db.<collection>.getIndexes()`

Let's view all the indexes in the studentgrades collection using the following command:

`db.studentgrades.getIndexes()`

Result:

```
> db.studentgrades.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
        {
                "v" : 2,
                "key" : {
                        "name" : 1
                },
                "name" : "student name index"
        }
]
>
```

The output contains the **default _id index** and the user-created index **student name index**.

Dropping indexes

To delete an index from a collection, use the **dropIndex** method while specifying the index name to

be dropped.

```
db.<collection>.dropIndex(<Index Name / Field Name>)
```

Let's remove the user-created index with the index name **student name index**, as shown below.

```
db.studentgrades.dropIndex("student name index")
```

Result:

```
> db.studentgrades.dropIndex("student name index")
{ "nIndexesWas" : 2, "ok" : 1 }
>
> db.studentgrades.getIndexes()
[ { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ]
>
```

You can also use the index field value for removing an index without a defined name:

```
db.studentgrades.dropIndex({name:1})
```

Result:

```
> db.studentgrades.dropIndex({name:1})
{ "nIndexesWas" : 2, "ok" : 1 }
>
>
```

The **dropIndexes** command can also drop all the indexes excluding the default _id index.

```
db.studentgrades.dropIndexes()
```

Result:

```
> db.studentgrades.dropIndexes()
{
        "nIndexesWas" : 1,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
>
```

Common MongoDB index types

MongoDB provides different types of indexes that can be utilized according to user needs. Here are the most common ones:

- Single field index
- Compound index
- Multikey index

Single field index

These user-defined indexes use a single field in a document to create an index in an ascending or descending sort order (1 or -1). In a single field index, the sort order of the index key does not have

an impact because MongoDB can traverse the index in either direction.

```
db.studentgrades.createIndex({name: 1})
```

Result:

```
> db.studentgrades.createIndex({name: 1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

The above index will sort the data in ascending order using the name field. You can use the **sort()** method to see how the data will be represented in the index.

```
db.studentgrades.find({},{_id:0}).sort({name:1})
```

Result:

```
> db.studentgrades.find({},{_id:0}).sort({name:1})
{ "name" : "Alex", "subject" : "Literature", "score" : 78 }
{ "name" : "Barry", "subject" : "Maths", "score" : 92 }
{ "name" : "Harry", "subject" : "Maths", "score" : 99, "notes" : "Exceptional Performance" }
{ "name" : "Kent", "subject" : "Physics", "score" : 87 }
{ "name" : "Tom", "subject" : "History", "score" : 65, "notes" : "Adequate" }
>
```

Compound index

You can use multiple fields in a MongoDB document to create a compound index. This type of index will use the first field for the initial sort and then sort by the preceding fields.

```
db.studentgrades.createIndex({subject: 1, score: -1})
```

```
> db.studentgrades.createIndex({subject: 1, score: -1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
>
```

In the above compound index, MongoDB will:

- First sort by the subject field
- Then, within each subject value, sort by grade

The index would create a data structure similar to the following:

```
db.studentgrades.find({},{_id:0}).sort({subject:1, score:-1})
```

Result:

```
> db.studentgrades.find({},{_id:0}).sort({subject:1, score:-1})
{ "name" : "Tom", "subject" : "History", "score" : 65, "notes" : "Adequate" }
{ "name" : "Alex", "subject" : "Literature", "score" : 78 }
{ "name" : "Harry", "subject" : "Maths", "score" : 99, "notes" : "Exceptional Performance" }
{ "name" : "Barry", "subject" : "Maths", "score" : 92 }
{ "name" : "Kent", "subject" : "Physics", "score" : 87 }
>
```

Multikey index

MongoDB supports indexing array fields. When you create an index for a field containing an array, MongoDB will create separate index entries for every element in the array. These multikey indexes enable users to query documents using the elements within the array.

MongoDB will automatically create a multikey index when encountered with an array field without requiring the user to explicitly define the multikey type.

Let's create a new data set containing an array field to demonstrate the creation of a multikey index.

```
db.createCollection("studentperformance")
db.studentperformance.insertMany(
 },
{name: "Kent", school: "FX High School", grades: },
{name: "Alex", school: "XYZ High", grades: },
]
)
db.studentperformance.find({},{_id:0}).pretty()
```

Result:

```
> db.studentperformance.find({},{_id:0}).pretty()
{
        "name" : "Barry",
        "school" : "ABC Academy",
        "grades" : [
                85,
                75,
                90,
                99
        ]
}
{
        "name" : "Kent",
        "school" : "FX High School",
        "grades" : [
                74,
                66,
                45,
                67
        ]
}
{
        "name" : "Alex",
        "school" : "XYZ High",
        "grades" : [
                80,
                78,
                71,
                89
        ]
}
>
```

Now let's create an index using the grades field.

```
db.studentperformance.createIndex({grades:1})
```

Result:

```
> db.studentperformance.createIndex({grades:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
>
```

The above code will automatically create a Multikey index in MongoDB. When you query for a document using the array field (grades), MongoDB will search for the first element of the array defined in the **find()** method and then search for the whole matching query.

For instance, let's consider the following find query:

```
db.studentperformance.find({grades: }, {_id: 0})
```

Initially, MongoDB will use the multikey index for searching documents where the grades array contains the first element (80) in any position. Then, within those selected documents, the documents with all the matching elements will be selected.

```
> db.studentperformance.find({grades: [80, 78, 71, 89]}, {_id: 0})
{ "name" : "Alex", "school" : "XYZ High", "grades" : [ 80, 78, 71, 89 ] }
>
```

Other MongoDB index types

In addition to the popular Index types mentioned above, MongoDB also offers some special index types for targeted use cases:

- Geospatial index
- Test index
- Hashed index

Geospatial Index

MongoDB provides two types of indexes to increase the efficiency of database queries when dealing with geospatial coordinate data:

- 2d indexes that use planar geometry which is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.
- 2dsphere indexes that use spherical geometry.

```
db.<collection>.createIndex( { <location Field> : "2dsphere" } )
```

Text index

The text index type enables you to search the string content in a collection.

```
db.<collection>.createIndex( { <Index Field>: "text" } )
```

Hashed index

MongoDB Hashed index type is used to provide support for hash-based sharding functionality. This would index the hash value of the specified field.

```
db.<collection>.createIndex( { <Index Field> : "hashed" } )
```

MongoDB index properties

You can enhance the functionality of an index further by utilizing index properties. In this section, you will get to know these commonly used index properties:

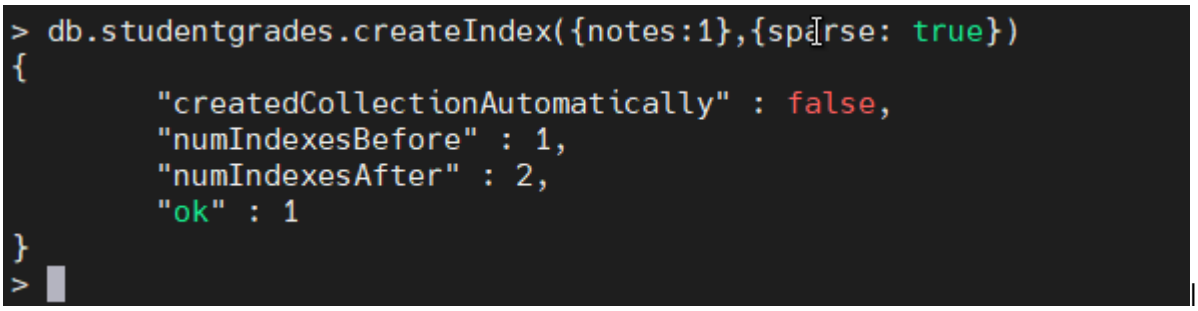
- Sparse index
- Partial index
- Unique index

Sparse index

The MongoDB sparse property allows indexes to omit indexing documents in a collection if the indexed field is unavailable in a document and create an index containing only the documents which contain the indexed field.

```
db.studentgrades.createIndex({notes:1},{sparse: true})
```

Result:



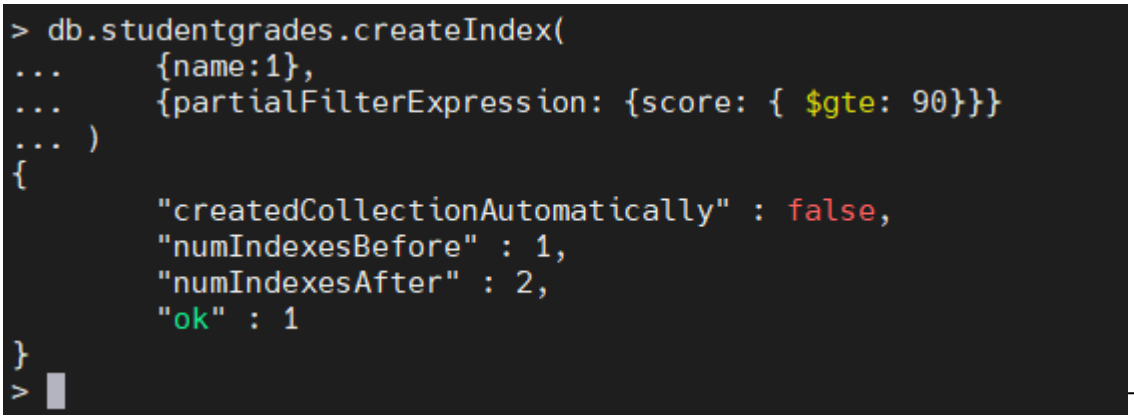
In the previous **studentgrades** collection, if you create an index using the notes field, it will index only two documents as the notes field is present only in two documents.

Partial index

The partial index functionality allows users to create indexes that match a certain filter condition. Partial indexes use the **partialFilterExpression** option to specify the filter condition.

```
db.studentgrades.createIndex(
{name:1},
{partialFilterExpression: {score: { $gte: 90}}}
)
```

Result:



The above code will create an index for the name field but will only include documents in which the value of the score field is greater than or equal to 90.

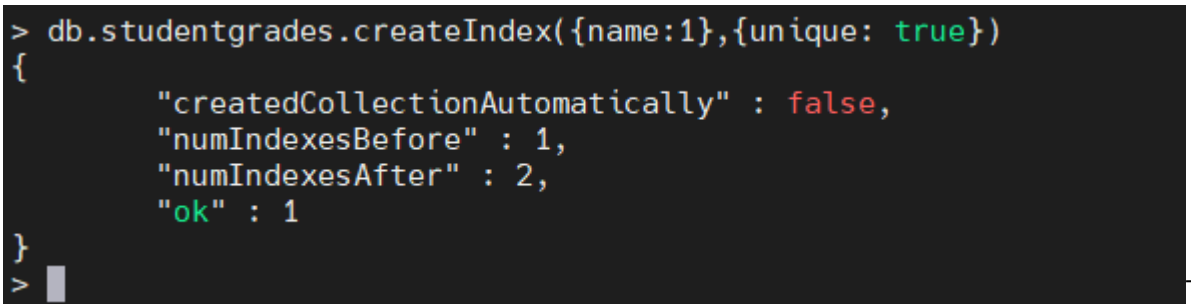
Unique index

The unique property enables users to create a MongoDB index that only includes unique values. This will:

- Reject any duplicate values in the indexed field
- Limit the index to documents containing unique values

```
db.studentgrades.createIndex({name:1},{unique: true})
```

Result:



The above-created index will limit the indexing to documents with unique values in the name field.

Indexes recap

That concludes this MongoDB indexes tutorial. You learned how to create, find, and drop indexes, use different index types, and create complex indexes. These indexes can then be used to further enhance the functionality of the MongoDB databases increasing the performance of applications which utilize fast database queries.

Related reading

- BMC Machine Learning & Big Data Blog
- MongoDB Guide, a series of articles and tutorials
- MongoDB: The Mongo Shell & Basic Commands
- Data Storage Explained: Data Lake vs Warehouse vs Database