Logical operartors in mongodb

And operator checks two or more conditions and finds the matching documents

Or operators checks two or more expressions and finds documents which match at least one expression

Nor operator checks two or more expressions and finds documents which does not match any expression or condition

Not operator it is used in combination with other operators or expressions

db.student.find({$and:[{id:102},

... {"marks.c":{$gt:80}}

... ]})

Or example

db.student.find({$and:[{id:{$gt:102, $lt:105},

... {"marks.java":{$gt:80}}

... ]})

Nor example

db.student.find({$nor:[{id:{$lt:102},

... {"marks.dbms":{$gt:80}}

... ]})

Not example

Gets the documents where marks in c are not less than 60

Db.student.find({“marks.c”:{$not:{$lt:60}}})

db.student.find( { name: { $not: { $regex: "^p.\*" } } } )

db.student.find( { name: { $not: { $regex: /^p.\*/ } } } )

Use Indexes to Sort Query Results

Since indexes contain ordered records, MongoDB can obtain the results of a sort from an index that includes the sort fields. MongoDB may use multiple indexes to support a sort operation if the sort uses the same indexes as the query predicate.

If MongoDB cannot use an index or indexes to obtain the sort order, MongoDB must perform a blocking sort operation on the data. A blocking sort indicates that MongoDB must consume and process all input documents to the sort before returning results. Blocking sorts do not block concurrent operations on the collection or database.

If MongoDB needs more than 100 megabytes of system memory for the blocking sort operation, MongoDB returns an error unless the query specifies cursor.allowDiskUse() (New in MongoDB 4.4). allowDiskUse() allows MongoDB to use temporary files on disk to store data exceeding the 100 megabyte system memory limit while processing a blocking sort operation.

Sort operations that use an index often have better performance than blocking sorts.

NOTE

As a result of changes to sorting behavior on array fields in MongoDB 4.4, when you sort on an array indexed with a multikey index, the query plan includes a blocking sort stage, unless:

The index boundaries for all sort fields are [MinKey, MaxKey], and

No boundaries for any multikey-indexed field have the same path prefix as the sort pattern.

Sort with a Single Field Index

If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.

For example, create an ascending index on the field a for a collection records:

db.records.createIndex( { a: 1 } )

This index can support an ascending sort on a:

db.records.find().sort( { a: 1 } )

The index can also support the following descending sort on a by traversing the index in reverse order:

db.records.find().sort( { a: -1 } )

Sort on Multiple Fields

Create a compound index to support sorting on multiple fields.

You can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the same order as they appear in the index. For example, an index key pattern { a: 1, b: 1 } can support a sort on { a: 1, b: 1 } but not on { b: 1, a: 1 }.

For a query to use a compound index for a sort, the specified sort direction for all keys in the cursor.sort() document must match the index key pattern or match the inverse of the index key pattern. For example, an index key pattern { a: 1, b: -1 } can support a sort on { a: 1, b: -1 } and { a: -1, b: 1 } but not on { a: -1, b: -1 } or {a: 1, b: 1}.

Sort and Index Prefix

If the sort keys correspond to the index keys or an index prefix, MongoDB can use the index to sort the query results. A prefix of a compound index is a subset that consists of one or more keys at the start of the index key pattern.

For example, create a compound index on the data collection:

db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )

Then, the following are prefixes for that index:

{ a: 1 }

{ a: 1, b: 1 }

{ a: 1, b: 1, c: 1 }

The following query and sort operations use the index prefixes to sort the results. These operations do not need to sort the result set in memory.

| **Example** | **Index Prefix** |
| --- | --- |
| db.data.find().sort( { a: 1 } ) | { a: 1 } |
| Example | Index Prefix |
| db.data.find().sort( { a: 1 } ) | { a: 1 } |
| db.data.find().sort( { a: -1 } ) | { a: 1 } |
| db.data.find().sort( { a: 1, b: 1 } ) | { a: 1, b: 1 } |
| db.data.find().sort( { a: -1, b: -1 } ) | { a: 1, b: 1 } |
| db.data.find().sort( { a: 1, b: 1, c: 1 } ) | { a: 1, b: 1, c: 1 } |
| db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } ) | { a: 1, b: 1 } |
| db.data.find().sort( { a: -1 } ) | { a: 1 } |
| db.data.find().sort( { a: 1, b: 1 } ) | { a: 1, b: 1 } |
| db.data.find().sort( { a: -1, b: -1 } ) | { a: 1, b: 1 } |
| db.data.find().sort( { a: 1, b: 1, c: 1 } ) | { a: 1, b: 1, c: 1 } |
| db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } ) | { a: 1, b: 1 } |

The author selected the Open Internet/Free Speech Fund to receive a donation as part of the Write for DOnations program.

Introduction

MongoDB is a document-oriented database management system that allows you to store large amounts of data in documents that can vary in terms of size and structure. MongoDB features a powerful query mechanism that allows you to filter documents based on specific criteria. As a MongoDB collection grows, though, searching for documents can become like searching for a needle in a haystack.

The flexibility that MongoDB offers with regard to queries can make it difficult for the database engine to predict what kinds of queries will be used most frequently; it must be ready to find documents regardless of the size of the collection. Because of this, the amount of data held in a collection directly impacts search performance: the bigger the data set, the more difficult it is for MongoDB to find the documents matching the query.

Indexes are one of the most essential tools the database administrator can use to consciously aid the database engine and improve its performance. In this tutorial, you’ll learn what indexes are, how to create them, and how to check how they’re used when the database performs queries.

Prerequisites

To follow this tutorial, you will need:

A server with a regular, non-root user with sudo privileges and a firewall configured with UFW. This tutorial was validated using a server running Ubuntu 20.04, and you can prepare your server by following this initial server setup tutorial for Ubuntu 20.04.

MongoDB installed on your server. To set this up, follow our tutorial on How to Install MongoDB on Ubuntu 20.04.

Your server’s MongoDB instance secured by enabling authentication and creating an administrative user. To secure MongoDB like this, follow our tutorial on How To Secure MongoDB on Ubuntu 20.04.

Familiarity with MongoDB CRUD operations and retrieving objects from collections in particular. To learn how to use MongoDB shell to perform CRUD operations, follow the tutorial How To Perform CRUD operations in MongoDB.

Note: The linked tutorials on how to configure your server, install, and then secure MongoDB installation refer to Ubuntu 20.04. This tutorial concentrates on MongoDB itself, not the underlying operating system. It will generally work with any MongoDB installation regardless of the operating system as long as authentication has been enabled.

Understanding Indexes

Typically, when you query a MongoDB database to retrieve documents that match a particular condition — such as mountain peaks with a height greater than 8000 meters — the database must perform a collection scan to find them. This means that it will retrieve every document from the collection to verify whether they match the condition. If a document does match the condition, it will be added to the list of returned documents. If a document doesn’t match the specified condition, MongoDB will move on to scanning the next document until it has finished scanning the entire collection.

This mechanism works well for many use cases, but it can become noticeably slow when the collection grows larger. This becomes more pronounced if the documents stored in the collection are complex; if a collection’s documents are more than just a few fields, it can be an expensive operation to read and then analyze their contents.

Indexes are special data structures that store only a small subset of the data held in a collection’s documents separately from the documents themselves. In MongoDB, they are implemented in such a way that the database can quickly and efficiently traverse them when searching for values.

To help understand indexes, imagine a database collection storing products in an online store. Each product is represented by a document containing images, detailed descriptions, category relationships, and many other fields. The application frequently runs a query against this collection to check which products are in stock.

Without any indexes, MongoDB would need to retrieve every product from the collection and check the stock information in the document structure. With an index, though, MongoDB will maintain a separate, smaller list containing only pointers to products in stock. MongoDB can then use this structure to find which products are in stock much more quickly.

In the following steps, you’ll prepare a sample database and use it to create indexes of different types. You’ll learn how to verify if the indexes are used when doing a query. Finally, you’ll learn how to list previously-defined indexes and remove them, if desired.

Step 1 — Preparing the Sample Database

In order to learn how indexes work and how to create them, this step outlines how to open the MongoDB shell to connect to your locally-installed MongoDB instance. It also explains how to create a sample collection and insert a few sample documents into it. This guide will use this sample data to illustrate different types of indexes that MongoDB can use to improve the query performance.

To create this sample collection, connect to the MongoDB shell as your administrative user. This tutorial follows the conventions of the prerequisite MongoDB security tutorial and assumes the name of this administrative user is AdminSammy and its authentication database is admin. Be sure to change these details in the following command to reflect your own setup, if different:

mongo -u AdminSammy -p --authenticationDatabase admin

Enter the password you set during installation to gain access to the shell. After providing the password, your prompt will change to a greater-than sign (>).

Note: On a fresh connection, the MongoDB shell will automatically connect to the test database by default. You can safely use this database to experiment with MongoDB and the MongoDB shell.

Alternatively, you could also switch to another database to run all of the example commands given in this tutorial. To switch to another database, run the use command followed by the name of your database:

use database\_name

To illustrate how indexes work, we’ll need a collection of documents with multiple fields of different types. We’ll be using the sample collection of the five highest mountains in the world. The following is an example document representing Mount Everest:

The Everest document

{

"name": "Everest",

"height": 8848,

"location": ["Nepal", "China"],

"ascents": {

"first": {

"year": 1953,

},

"first\_winter": {

"year": 1980,

},

"total": 5656,

}

}

This document contains the following information:

name: the peak’s name.

height: the peak’s elevation, in meters.

location: the countries in which the mountain is located. This field stores values as an array to allow for mountains located in more than one country.

ascents: this field’s value is another document. When one document is stored within another document like this, it’s known as an embedded or nested document. Each ascents document describes successful ascents of the given mountain. Specifically, each ascents document contains a total field that lists the total number of successful ascents of each given peak. Additionally, each of these nested documents contain two fields whose values are also nested documents:.

first: this field’s value is a nested document that contains one field, year, which describes the year of the first overall successful ascent.

first\_winter: this field’s value is a nested document that also contains a year field, the value of which represents the year of the first successful winter ascent of the given mountain.

Run the following insertMany() method in the MongoDB shell to simultaneously create a collection named peaks and insert five sample documents into it. These documents describe the five tallest mountain peaks in the world:

db.peaks.insertMany([

{

"name": "Everest",

"height": 8848,

"location": ["Nepal", "China"],

"ascents": {

"first": {

"year": 1953

},

"first\_winter": {

"year": 1980

},

"total": 5656

}

},

{

"name": "K2",

"height": 8611,

"location": ["Pakistan", "China"],

"ascents": {

"first": {

"year": 1954

},

"first\_winter": {

"year": 1921

},

"total": 306

}

},

{

"name": "Kangchenjunga",

"height": 8586,

"location": ["Nepal", "India"],

"ascents": {

"first": {

"year": 1955

},

"first\_winter": {

"year": 1986

},

"total": 283

}

},

{

"name": "Lhotse",

"height": 8516,

"location": ["Nepal", "China"],

"ascents": {

"first": {

"year": 1956

},

"first\_winter": {

"year": 1988

},

"total": 461

}

},

{

"name": "Makalu",

"height": 8485,

"location": ["China", "Nepal"],

"ascents": {

"first": {

"year": 1955

},

"first\_winter": {

"year": 2009

},

"total": 361

}

}

])

The output will contain a list of object identifiers assigned to the newly inserted objects.

Output

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("61212a8300c8304536a86b2f"),

ObjectId("61212a8300c8304536a86b30"),

ObjectId("61212a8300c8304536a86b31"),

ObjectId("61212a8300c8304536a86b32"),

ObjectId("61212a8300c8304536a86b33")

]

}

You can verify that the documents were properly inserted by running the find() method with no arguments, which will retrieve all documents:

db.peaks.find()

Output

{ "\_id" : ObjectId("61212a8300c8304536a86b2f"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first\_winter" : { "year" : 1980 }, "total" : 5656 } }

...

Please note that this example collection is not big enough to directly illustrate the performance impact of indexes or lack thereof. However, this guide will outline how MongoDB uses indexes to limit the amount of traversed documents by higlighting query details as reported by the database engine.

With the sample data in place, you can continue on to the next step to learn how to create an index based on a single field.

Step 2 — Creating a Single Field Index and Evaluating Index Usage

This step explains how to create a single field index in order to speed up document queries that filter data using that field as part of the filtering condition. It also outlines how you can verify whether MongoDB used an index to boost the query performance or resorted to a full collection scan instead.

To begin, run the following query. Normally, the query document { "height": { $gt: 8700 } } would cause this query to retrieve any documents that describe a mountain peak with a height value greater than 8700. However, this operation includes the explain(executionStats) method, which will cause the query to instead return information about how the query is performed. Because you haven’t yet created any indexes, this will provide you with a benchmark which you can use to compare against the performance of queries that do use indexes:

db.peaks.find(

{ "height": { $gt: 8700 } }

).explain("executionStats")

This operation returns a lot of information. The following example output removes a number of lines that aren’t important for the purposes of this tutorial:

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

"stage" : "COLLSCAN",

. . .

},

},

. . .

"executionStats" : {

. . .

"nReturned" : 1,

"executionTimeMillis" : 0,

"totalKeysExamined" : 0,

"totalDocsExamined" : 5,

. . .

},

. . .

}

The following fields returned in this output are particularly relevant to understanding how indexes work:

winningPlan: This document within the queryPlanner section describes how MongoDB decided to execute the query. Depending on the query type, the detailed structure of the winningPlan may differ, but here the key thing to note is COLLSCAN. The presence of this value means that MongoDB needed to go through the full collection without any aids to find the requested documents.

nReturned: This value tells you how many documents were returned by a given query. Here, just a single mountain peak matches the query.

executionTimeMillis: This value represents the execution time. With such a small collection, its importance is negligible. However, when analyzing the performance of queries against larger or more complex collections, it is an important metric to keep in mind.

totalKeysExamined: This tells you how many index entries MongoDB checked to find the requested documents. Because the collection scan was used and you haven’t created any indexes yet, the value is 0.

totalDocsExamined: This value indicates how many documents MongoDB had to read from the collection. Because MongoDB performed a collection scan, its value is 5, the total count of all documents in the collection. The larger the collection, the bigger the value in this field when indexes are not used.

Notice the discrepancy between the total examined documents and returned documents counts: MongoDB had to inspect 5 documents in order to return one.

This tutorial will reference these values in later sections to analyze how indexes affect the way that queries are executed.

To that end, create an index on the height field in the peaks collection using the createIndex() method. This method accepts a JSON document describing the index you want to create. This example will create a single field index, meaning that the document contains a single key (height in this example) for the field we want to use. This key accepts either 1 or -1 as a value. These values denote the index’s sorting order, with 1 indicating ascending order and -1 indicating descending:

db.peaks.createIndex( { "height": 1 } )

Note: With single field indexes, the ordering is not important, since the index structure can be traversed in both directions efficiently. Choosing the order for index fields becomes more important with compound indexes based on multiple fields, as described in Step 4.

MongoDB returns a confirmation indicating how many indexes have been defined on the collection now, and how that differs from the previous state.

Output

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

Now try executing the same query you ran previously. This time, though, the information returned by the explain("executionStats") method will differ because there is an index in place:

db.peaks.find(

{ "height": { $gt: 8700 } }

).explain("executionStats")

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

. . .

"inputStage" : {

"stage" : "IXSCAN",

. . .

"indexName" : "height\_1",

. . .

}

},

. . .

},

"executionStats" : {

. . .

"nReturned" : 1,

"executionTimeMillis" : 0,

"totalKeysExamined" : 1,

"totalDocsExamined" : 1,

. . .

},

. . .

}

Notice that winningPlan no longer shows COLLSCAN. Instead, IXSCAN is present, indicating that the index was used as part of the query execution. MongoDB also informs you which index was used through the indexName value. By default, MongoDB constructs index names from the field names to which the index is bound and the ordering applied. From { "height": 1 }, MongoDB automatically generated the name height\_1.

The most important change is in the executionStats section. Once again, this query only returned a single document, as denoted by nReturned. However, this time the totalDocsExamined is only 1. This means that the database retrieved just one document from the collection to satisfy the query. The totalKeysExamined reveals that the index was checked just one time because it provided enough information to compile the result.

By creating this index, you’ve reduced the number of documents MongoDB had to inspect from 5 to 1, a five-fold reduction. If the peaks collection contained thousands of entries, the impact of using an index would be even more apparent.

Step 3 — Creating Unique Indexes

In MongoDB, it’s impossible to insert two documents into the collection if they both have the same \_id values. This is because the database automatically maintains a single-field index on the \_id field that, in addition to helping speed up document lookups, ensures the uniqueness of the \_id field value. This step explains how you can create indexes to ensure the values of a given field will be unique for every document in a collection.

To illustrate, run the following createIndex() method. This command’s syntax is similar to the one used in the previous step except, this time, a second parameter is passed to createIndex() with additional settings for the index. The { "unique": true } indicates that the created index will ensure that the values of the specified field (name) can’t repeat:

db.peaks.createIndex( { "name": 1 }, { "unique": true } )

Once again, MongoDB will confirm that the index was created successfully:

Output

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 2,

"numIndexesAfter" : 3,

"ok" : 1

}

Next, check whether the index serves its primary purpose and runs any queries against mountain names faster by avoiding collection scans. To do so, run the following equality query with the explain("executionStats") method:

db.peaks.find(

{ "name": "Everest" }

).explain("executionStats")

The returned query plan uses the IXSCAN strategy with the newly-created index, just like with mountain height query from the previous step:

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

. . .

"inputStage" : {

"stage" : "IXSCAN",

. . .

"indexName" : "name\_1",

. . .

}

},

. . .

},

. . .

}

Next check whether you’re able to add a second document representing Mt. Everest to the collection now that the index is in place. Do so by running the following insertOne() method:

db.peaks.insertOne({

"name": "Everest",

"height": 9200,

"location": ["India"],

"ascents": {

"first": {

"year": 2020

},

"first\_winter": {

"year": 2021

},

"total": 2

}

})

MongoDB will not create the document and will instead return an error message:

Output

WriteError({

"index" : 0,

"code" : 11000,

"errmsg" : "E11000 duplicate key error collection: test.peaks index: name\_1 dup key: { name: \"Everest\" }",

"op" : {

. . .

This duplicatye key error message refers to the name\_1 index, indicating that it’s enforcing a uniqueness constraint on this field.

With that, you’ve learned how to create a unique index to prevent a given field from containing duplicate values. Continue reading to learn how to use indexes with embedded documents.

Step 4 — Creating an Index on an Embedded Field

Whenever you query a collection using a field within a nested document that doesn’t have an index, MongoDB not only has to retrieve all documents from the collection, but it must also traverse each nested document.

As an example, run the following query. This will return any documents whose total — a field nested within the ascents document found in each document in the peaks collection — is greater than 300 and sorts the results in descending order:

db.peaks.find(

{ "ascents.total": { $gt: 300 } }

).sort({ "ascents.total": -1 })

This query will return four peaks from the collection, with Mt. Everest being the peak with the most ascents, followed by Lhotse, Makalu, and K2:

Output

{ "\_id" : ObjectId("61212a8300c8304536a86b2f"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first\_winter" : { "year" : 1980 }, "total" : 5656 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b32"), "name" : "Lhotse", "height" : 8516, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1956 }, "first\_winter" : { "year" : 1988 }, "total" : 461 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b33"), "name" : "Makalu", "height" : 8485, "location" : [ "China", "Nepal" ], "ascents" : { "first" : { "year" : 1955 }, "first\_winter" : { "year" : 2009 }, "total" : 361 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b30"), "name" : "K2", "height" : 8611, "location" : [ "Pakistan", "China" ], "ascents" : { "first" : { "year" : 1954 }, "first\_winter" : { "year" : 1921 }, "total" : 306 } }

Now run the same query, but include the explain("executionStats") method used previously:

db.peaks.find(

{ "ascents.total": { $gt: 300 } }

).sort({ "ascents.total": -1 }).explain("executionStats")

As the COLLSCAN value in this section of the output indicates, MongoDB resorted to a full collection scan and traversed all the documents from the peaks collection to compare them against the query conditions:

Output

{

. . .

"winningPlan" : {

"stage" : "COLLSCAN",

. . .

},

. . .

}

Because this collection only has five entries, the lack of an index didn’t significantly affect performance and this query executed immediately. However, the more complex the documents stored in the database, the greater the performance impact queries can have. This step outlines how to create single-field indexes on fields inside embedded documents to help mitigate this issue.

To help MongoDB execute this query, let’s create an index on the total field within the ascents document. Because the total field is nested within ascents, it’s not possible to specify total as the field name when creating this index. Instead, MongoDB provides a dot notation to access fields in nested documents. To refer to total field inside ascents nested document, you can use the ascents.total notation, like this:

db.peaks.createIndex( { "ascents.total": 1 } )

MongoDB will reply with a success message letting you know that you now have four indexes defined.

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 3,

"numIndexesAfter" : 4,

"ok" : 1

}

Note: In this tutorial, we add additional indexes from step to step to illustrate how different types of indexes can be used. However, it is important to be aware that adding too many indexes can be as bad for performance as having too few.

For every index in the database, MongoDB must keep each properly updated whenever a new document is inserted to the collection or any are changed. The performance penalty of having many indexes can counter the benefits they provide through increasing query speed. Make sure to add indexes only for fields that are queried often or have the most impact on performance.

Run the previous query once again to check whether the index helped MongoDB avoid performing a full collection scan:

db.peaks.find(

{ "ascents.total": { $gt: 300 } }

).sort({ "ascents.total": -1 }).explain("executionStats")

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

. . .

"inputStage" : {

"stage" : "IXSCAN",

. . .

"indexName" : "ascents.total\_-1",

. . .

}

},

. . .

},

"executionStats" : {

. . .

"nReturned" : 4,

"executionTimeMillis" : 0,

"totalKeysExamined" : 4,

"totalDocsExamined" : 4,

. . .

"direction" : "backward",

. . .

},

. . .

}

Notice that now IXSCAN is used against the newly created ascents.total\_-1 index, and only four documents have been examined. This is the same number of documents returned and examined in the index, so no additional documents have been retrieved to complete the query.

direction, another field in the executionStats section, indicates which direction MongoDB decided to traverse the index. Because the index was created as ascending, using the { "ascents.total": 1 } syntax, and the query requested mountain peaks sorted in descending order, the database engine decided to go backwards. When retrieving documents in a particular order based on a field that’s part of an index, MongoDB will use the index to provide final ordering without the need to further sort documents after retrieving them in full.

Step 5 — Creating a Compound Field Index

The examples so far in this guide are helpful for understanding the benefits of using indexes, but document filtering queries used in real world applications are rarely this simple. This step explains how MongoDB uses indexes when executing queries on more than one field and how to use compound field indexes to target such queries specifically.

Recall from Step 2 when you created a single field index on the height field in order to more efficiently query the peaks collection to find the highest mountain peaks. With this index in place, let’s analyze how MongoDB will perform a similar but slightly more complex query. Try finding mountains with a height of less than 8600 meters whose first winter ascent occurred after the year 1990:

db.peaks.find(

{

"ascents.first\_winter.year": { $gt: 1990 },

"height": { $lt: 8600 }

}

).sort({ "height": -1 })

Only a single mountain — Makalu — satisfies both of these conditions:

Output

{ "\_id" : ObjectId("61212a8300c8304536a86b33"), "name" : "Makalu", "height" : 8485, "location" : [ "China", "Nepal" ], "ascents" : { "first" : { "year" : 1955 }, "first\_winter" : { "year" : 2009 }, "total" : 361 } }

Now add the explaion("executionStats") method to find how MongoDB performed this query:

db.peaks.find(

{

"ascents.first\_winter.year": { $gt: 1990 },

"height": { $lt: 8600 }

}

).sort({ "height": -1 }).explain("executionStats")

Even though there is no index that might affect the first winter ascent date, MongoDB used a previously-created index instead of doing a full collection scan:

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

"stage" : "IXSCAN",

. . .

"indexName" : "height\_1",

. . .

}

},

. . .

},

"executionStats" : {

. . .

"nReturned" : 1,

"executionTimeMillis" : 0,

"totalKeysExamined" : 3,

"totalDocsExamined" : 3,

. . .

},

. . .

}

Notice that this time, different from previous index-backed query executions, the nReturned value denoting the number of returned documents is different than both totalKeysExamined and totalDocsExamined. MongoDB used the single field index on the height field to narrow down the results from 5 to 3, but then had to scan the remaining documents to check the first winter ascent date.

If an index is only available for one part of a query, MongoDB will use it to narrow down the results first before doing a collection scan. It will traverse only the list of documents it initially filtered in order to satisfy the rest of the query.

In many situations, this is entirely sufficient. If the most common queries examine a single indexed field and must only occasionally perform additional filtering, having a single field index is usually good enough. When queries against multiple fields are common, though, it might be beneficial to define an index spanning all these fields to make sure no additional scans must be performed.

Imagine that you query the database for mountain peaks satisfying conditions related to their first winter ascent and height regularly enough that it becomes a performance concern and would benefit from having an index. To create an index based on both of these fields fields, run the following createIndex(0) method:

db.peaks.createIndex(

{

"ascents.first\_winter.year": 1,

"height": -1

}

)

Notice this operation’s syntax is similar to the single field index creation, but this time both fields are listed in the index definition object. The index is created as ascending regarding the peaks’ first winter ascents and descending with regards to their heights.

MongoDB will acknowledge that the index was successfully created:

Output

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 4,

"numIndexesAfter" : 5,

"ok" : 1

}

With single field indexes, the database engine can freely traverse the index either forwards or backwards. However, with compound indexes this is not always the case. If a particular sorting order for a combination of fields is queried more often, it can further increase performance to include that order in the index definition. MongoDB will then satisfy the requested ordering using the index directly, rather than doing additional sorting on the list of returned documents.

Run the previous query once again to test whether there was any change in how the query was performed:

db.peaks.find(

{

"ascents.first\_winter.year": { $gt: 1990 },

"height": { $lt: 8600 }

}

).sort({ "height": -1 }).explain("executionStats")

This time the query again used an index scan, but the index is different. Now, the ascents.first\_winter.year\_1\_height\_-1 index that you just created is chosen over the previously-used height\_1 index:

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

"stage" : "IXSCAN",

. . .

"indexName" : "ascents.first\_winter.year\_1\_height\_-1",

. . .

}

},

. . .

},

"executionStats" : {

. . .

"nReturned" : 1,

"executionTimeMillis" : 0,

"totalKeysExamined" : 1,

"totalDocsExamined" : 1,

. . .

},

. . .

}

The important difference lies in executionStats. With the new index, a single document was examined directly from the index and then returned, as opposed to three documents requiring further document scans to narrow the results down. If this was a larger collection, the difference between the new compound index and using a single field index with further filtering would be even more pronounced.

Now that you’ve learned how to create indexes that span more than one field, you can move on to learning about multi-key indexes and how they’re used.

Step 6 — Creating a Multi-key Index

In previous examples, the fields used in indexes had single values stored in them, like a height, a year, or a name. In these cases, MongoDB stored the field value directly as the index key, making the index quickly traversable. This step outlines how MongoDB behaves when the field used to create the index is a field storing multiple values, such as an array.

To begin, try finding all the mountains in the collection that are located in Nepal:

db.peaks.find(

{ "location": "Nepal" }

)

Four peaks are returned:

Output

{ "\_id" : ObjectId("61212a8300c8304536a86b2f"), "name" : "Everest", "height" : 8848, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1953 }, "first\_winter" : { "year" : 1980 }, "total" : 5656 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b31"), "name" : "Kangchenjunga", "height" : 8586, "location" : [ "Nepal", "India" ], "ascents" : { "first" : { "year" : 1955 }, "first\_winter" : { "year" : 1986 }, "total" : 283 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b32"), "name" : "Lhotse", "height" : 8516, "location" : [ "Nepal", "China" ], "ascents" : { "first" : { "year" : 1956 }, "first\_winter" : { "year" : 1988 }, "total" : 461 } }

{ "\_id" : ObjectId("61212a8300c8304536a86b33"), "name" : "Makalu", "height" : 8485, "location" : [ "China", "Nepal" ], "ascents" : { "first" : { "year" : 1955 }, "first\_winter" : { "year" : 2009 }, "total" : 361 } }

Notice that none of these peaks are only in Nepal. Each of these four peaks span more than one country as indicated by their location fields, all of which are an array of multiple values. What is more, these values can appear in different orders. For example, Lhotse is listed as being in [ "Nepal", "China" ], whereas Makalu is listed as being in [ "China", "Nepal" ].

Because there is no index available spanning the location field, MongoDB currently does a full collection scan to execute that query. Let’s create a new index for the location field:

db.peaks.createIndex( { "location": 1 } )

Notice that this syntax does not differ from any other single field index. MongoDB will return a success message, and the index is now available to use:

Output

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 5,

"numIndexesAfter" : 6,

"ok" : 1

}

Now that you’ve created an index for the location field, run the previous query again with the explain("executionStats") method to understand how it executes:

db.peaks.find(

{ "location": "Nepal" }

).explain("executionStats")

The resulting output indicates that MongoDB used an index scan as the strategy, referring to the newly-created location\_1 index:

Output

{

"queryPlanner" : {

. . .

"winningPlan" : {

. . .

"inputStage" : {

"stage" : "IXSCAN",

. . .

"indexName" : "location\_1",

"isMultiKey" : true,

. . .

}

},

. . .

},

"executionStats" : {

. . .

"nReturned" : 4,

"executionTimeMillis" : 0,

"totalKeysExamined" : 4,

"totalDocsExamined" : 4,

. . .

}

. . .

}

The number of returned documents matches the total number of examined index keys and examined documents. This means that the index was used as the sole source of information for the query. How was that possible if the field values are arrays of more than one value, and the query asked for mountains with one of the locations matching Nepal?

Notice the isMultiKey property listed as true in the output. MongoDB automatically created a multi-key index for the location field. If you create an index for a field holding arrays, MongoDB automatically determines it needs to create a multi-key index and creates separate index entries for every element of these arrays.

So, for a document that has a location field storing the array [ "China", "Nepal" ], two separate index entries appear for the same document, one for China and another for Nepal. This way, MongoDB can use the index efficiently even if the query requests a partial match against the array contents.

Step 7 — Listing and Removing Indexes on a Collection

In the previous steps, you’ve learned how to create different types of indexes. When the database grows or requirements change, it’s important to be able to know what indexes are defined and sometimes remove unwanted ones. Indexes that are no longer useful can have a negative impact on the database’s performance, since MongoDB must still maintain them any time you add or change data.

To list all the indexes you’ve defined on the peaks collection throughout this tutorial, you can use the getIndexes() method:

db.peaks.getIndexes()

MongoDB will return the list of indexes, describing their nature and listing their names:

Output

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"key" : {

"height" : 1

},

"name" : "height\_1"

},

{

"v" : 2,

"unique" : true,

"key" : {

"name" : 1

},

"name" : "name\_1"

},

{

"v" : 2,

"key" : {

"ascents.total" : 1

},

"name" : "ascents.total\_1"

},

{

"v" : 2,

"key" : {

"ascents.first\_winter.year" : 1,

"height" : -1

},

"name" : "ascents.first\_winter.year\_1\_height\_-1"

},

{

"v" : 2,

"key" : {

"location" : 1

},

"name" : "location\_1"

}

]

Throughout this tutorial, you have defined 6 indexes altogether. For each one, the key property lists the index definition, matching the way the index was created before. For each index, the name property contains the name MongoDB generated automatically when creating the index.

To delete an existing index, you can use either of these properties with the dropIndex() method. The following example will delete the height\_1 index by using the definition of its contents:

db.peaks.dropIndex( { "height": 1 } )

Since the { "height": 1 } matches the single field index on height named height\_1, MongoDB will remove that index and reply with a success message indicating how many indexes there were prior to removing this one:

Output

{ "nIndexesWas" : 6, "ok" : 1 }

This way of specifying the index to remove can become unwieldy if the index definition is more complex, as can be the case with compound indexes. As an alternative, you can remove indexes using an index’s name. To remove the index created on the first winter ascent and height in Step 5 using its name, run the following operation:

db.peaks.dropIndex("ascents.first\_winter.year\_1\_height\_-1")

Once again, MongoDB will remove the index and return a success message:

Output

{ "nIndexesWas" : 5, "ok" : 1 }

You can confirm that these two indexes have been indeed removed from the list of collection indexes by calling getIndexes() again:

db.peaks.getIndexes()

This time, only the four remaining indexes are listed:

Output

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_"

},

{

"v" : 2,

"unique" : true,

"key" : {

"name" : 1

},

"name" : "name\_1"

},

{

"v" : 2,

"key" : {

"ascents.total" : 1

},

"name" : "ascents.total\_1"

},

{

"v" : 2,

"key" : {

"location" : 1

},

"name" : "location\_1"

}

]

As a final note, be aware that it’s not possible to modify an existing index in MongoDB. If you ever need to change an index, you must first drop that index and create a new one.