Indexes in MongoDB

Team 3: Joseph Caltabiano, Shine Lin Thant

CS585 Fall 2020

Overview

MongoDB provides a wide variety of indexing mechanisms, such as Single, Compound, Text, Wildcard, 2d, geoHaystack etc. Indexing in MongoDB can be done over documents and collections, which in turn enhances query performance. Among these indexes, we will be focusing on Single field and compound indexes as we study the differences and advantages.

When deciding which fields to index, it is crucial to index the fields that will ensure index selectivity. Indexes are selective if they have a large number of unique values or few duplicate values. For example, a gender field is not selective since there are many other fields with the same value whereas a date of birth field is selective since it is less likely to have many other fields with the same values.

Single field index allows us to create an index on a single field, either ascending or descending. We can also create a single field index on a field inside a nested document, or even the nested document itself as a whole. In this case, you can treat the embedded document as a top-level field of the document.

```
"_id": ObjectId("570c04a4ad233577f97dc459"),

"score": 1034,

"location": { state: "NY", city: "New York" }

db.records.createIndex( { location: 1 } )
```

Single Field Index on a nested document as a whole

```
db.records.createIndex( { "location.state": 1 } )
```

Single Field Index on a field inside nested document

You can also index a field which is an array of embedded documents as a whole, or you can also index the fields inside the embedded document inside the array. For example:

```
"title" : "How the west was won",
"comments" : [{"text" : "great!" , "author" : "sam"},
              {"text" : "ok" , "author" : "julie"}],
"_id" : "497ce79f1ca9ca6d3efca325"}
```

```
db.articles.ensureIndex( { comments : 1 } )
```

Single Field Index on a field containing array of nested documents

```
db.posts.ensureIndex({"comments.author" : 1})
```

Single Field Index on a subfield inside an array of nested documents

Similar to Single field indices, compound indices allow us to create indices over multiple fields, either ascending or descending, and they support queries that match on multiple fields. However, with compound indices, we have to be aware of the order of the declared indices.

The advantage of compound indexes over single indexes is that compound indexes are more selective than single field indexes due to having more attributes. Compound indexes also support queries where not all attributes are specified.

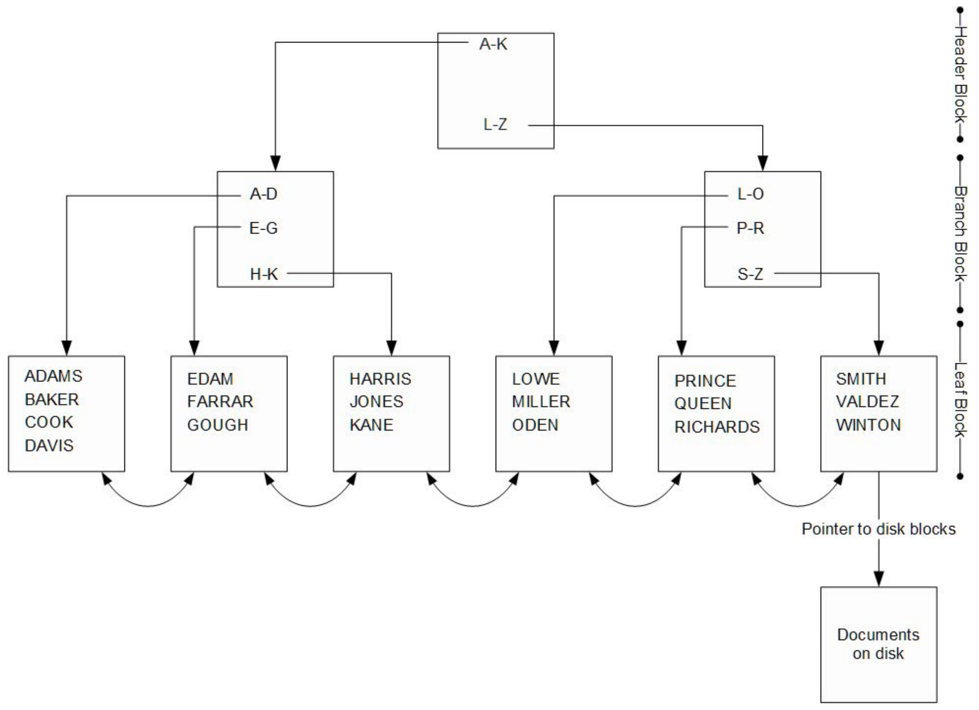
```
db.records.createIndex( { score: 1 } )
```

Single Field Index

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

Compound Index

All indexes in Mongo follow the B-tree structure by default, and the advantage of this structure is that it allows fast look-up, since the blocks have pointers to preceding and succeeding blocks which point to the location of these documents on disk. As shown in the figure below, the maximum depth level a leaf block can have is always 3-4, and all these leaf blocks are at the same level. However, one drawback of this structure is that when a new entry needs to be inserted and there is no free space in the desired block, we would need to allocate a new block and move half the entries to the newly created block, and the operation is costly.



Experimental Plan

Our experimental plan was to test the efficiency of different queries on single vs compound indices. Then we examined the execution time for each query, and storage overhead when using each type of index. We decided upon the following combination of indices to execute the queries with:

- single unindexed field
- same field but indexed
- one indexed field and one unindexed field
- a compound index on both fields
- a compound index on three fields but only use first and last fields

The next step was to implement the above combination of queries using find(), aggregate(), sort(), group() etc. With each query we set multiple indices upon each execution to compare the performance of different types of queries using different indices.

Execution & Results

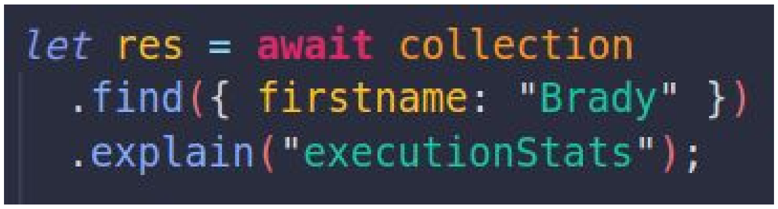
We created our own datasets of three different sizes: 1 million, 10 million, and 21 million respectively, using our own data generator. The collection structure is as follows:

- name, sex, email, phone number, salary
- [str, str], categorical M/F/NB, str, str, int

We ran each of our queries on all three datasets to compare and contrast not only the running time but also the effectiveness of these indexes. We have two types of queries: find and aggregate. And we collected the execution time (in milliseconds) of the queries to analyze the performance with and without indexes.

The data we will present about the execution time of find() was gathered from our Node script, and not from the shell. Doing all of the necessary database commands to carry out all these experiments by hand in the shell would have been incredibly time consuming and impractical. However, we tested a subset of the commands manually in the shell. Not only did we get results that were different, they tell the reverse story of what the Node scripts claimed for some of the queries. We believe the results gathered from the shell to be the most accurate as they are more in line with patterns you should expect, and others have tested, for using different indices.

When developing the code, we found that Mongo does not actually have any support for tracking the execution time for aggregate() queries. This is a known issue in the Mongo community, but we were not able to find any conventional solution.



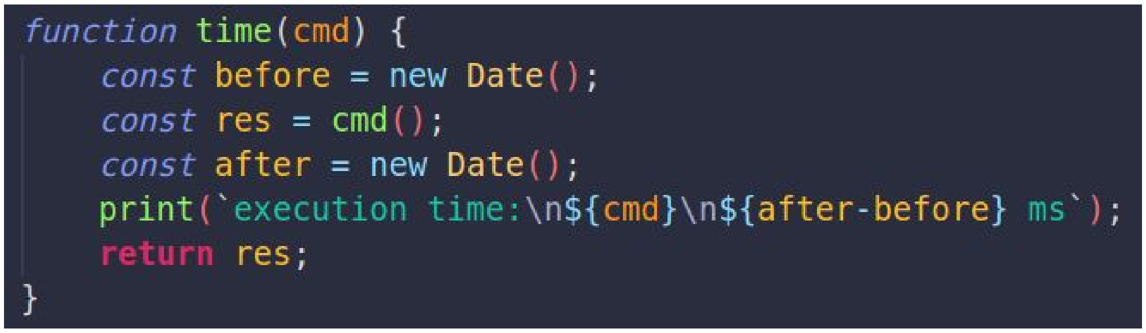
```
let res = await collection
  .find({ firstname: "Brady" })
  .explain("executionStats");
```

These are examples of how we gathered the data. The above figure shows how we use the explain() function. Using it like this is the conventional way for gathering statistics about your query, especially execution time. Using it in this way, we saw consistent results across trials for the same queries. We noticed some minor issues, for example a call to .explain() would return 0ms for a query we knew should take 500ms. Say the following query also took 500ms on average, it's call to explain() would return 1000ms, including its own time and the unreported time of the previous query.

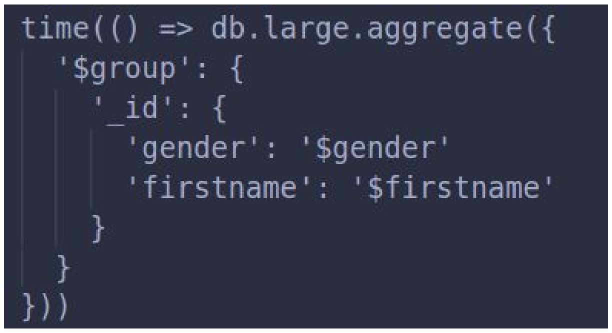
After we had finished our experimentation and started to analyze our data, we noticed some patterns that went against expectations. We decided to run a sample of the queries by hand in the mongo console and started to get completely different results.

Measuring the execution time for aggregate() queries was quite difficult. Since we cannot actually measure execution time on the database side, we were forced to try to take the measure on the client side by wrapping the statement in Date()s. This already introduces error into the measurements as we are adding time for communication between client and backend and the execution time of the js function as well. We also found that calls to aggregate() return a cursor, an iterator over the results, instead of the results themselves. In order to force the query to run on the collection, wait for the results, and return them, a function had to be called on the cursor, specifically .toArray(). At the scale we were working with, dedicated heap space and RAM size were far too small even when manually expanded.

So, we were forced to work in the console for the aggregate() queries. The solution we eventually turned to involves writing the above function time() in the mongorc.js configuration file. We could not test this robustly but the results from this appear to be accurate and consistent amongst themselves. Even if they may not represent the true execution time, the data is comparable.



```
function time(cmd) {
    const before = new Date();
    const res = cmd();
    const after = new Date();
    print(`execution time:\n${cmd}\n${after-before} ms`);
    return res;
}
```



```
time(() => db.large.aggregate({
  '$group': {
    '_id': {
      'gender': '$gender'
      'firstname': '$firstname'
    }
  }
}))
```

The following are the results from our experiments on execution time comparing using no index versus an index on the name.

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x)	none	607	14440	44238
	name	3917	49145	128354
groupby(name)	none	1210	11697	24720
	name	81	10455	24986

Notice how the find time was much longer with a name index than with none. Running the exact same find query in the shell provides the opposite results. With no index, all 1mil documents are scanned and the operation takes around 600ms. With the index however, only about 300 documents are actually read, and takes less than 1ms. The same goes for the collection with 10mil documents - with no index the search takes about 10k ms, but with name indexed only about 3000 documents are scanned and the operation completes in under 1ms.

For the aggregation query, we saw that the advantage of indexing the name appeared to diminish as the size of the collection grew. There is a sizable reduction in time for the small collection, less for the medium and none for the larger collection.

The following are the results from experimentation with the find() query on different compound indexes.

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x, phone:y})	name	1	12	123
	name, phone	5619	72935	171310

	name, email, phone	6468	86153	271421
	name, gender	5579	73145	226728

Here again we see some results that are not consistent with the actual behavior. Using the name only index should be comparable to the previous slide where search only took a few ms, which is true. We cannot be certain about the accuracy of the following numbers - however, we can compare their values relative to each other. Querying using an index on name and phone, and name and gender, are roughly similar. Here, since name is the first in the index and the search, it is the most important. Indexing on phone should not provide a huge improvement over just using name.

Using an index on name, email, and phone, this query will only be able to access the index on nam due to the nature of the compound index. A query can leverage the name index, the name and email index, or all three. But name and phone are not sequential in the index so only name index can be used. Similarly, using an index on name and gender, the query will only access the index on name, as gender is not part of the search criteria.

From our research, we believe these results should be similar, and we cannot explain why using the name, email, phone index takes consistently longer. We also must consider the possibility that these inconsistencies were the result of quirks with the Node driver script.

The following are results from experimenting with indexes on fields of differing cardinality.

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)
find({name:x, gender:y})	name	1	10	26
	gender	2917	27996	157612
find({gender:y, name:x})	gender, name	8538	71291	172200

Again, assuming that we can compare these results, we can see that finding a matching name and gender is much easier when indexed on name. The name field had many more unique values, and indexing it can allow us to ignore large parts of data as we search down the tree. However, gender has far less values. Indexing on a field like this does not make much sense, as there are not enough unique values.

Searching by gender first instead of name also displays this concept. It is much less efficient than searching by name first.

The following results are from experimenting with the aggregate($group) query using different indexes.

Query	**Index**	**1M**(ms)	**10M**(ms)	**21M**(ms)

groupyby({gender:y, name:x})	name	1374	15344	41631
	gender, name	1679	16431	42219
	name, gender	1531	17418	43199
	name: 1, gender: -1	1500	18719	43391
	name: -1, gender: 1	1611	17085	49344

We continued to experiment here with how different indexes would affect an aggregation like this. We also explored how the sort order changed times. For these specific experiments, there is very little change in execution time. This continues to prove that an index is only as effective as the query that uses it. For this query, the sort order of the indices has no effect. This is most useful is you are searching for an integer amount greater than some value. If you index that field and order it as descending, results will typically be faster than if it is of ascending order.

Here are the actual storage sizes of all the indexes we experimented with.

Index	**1M**(MB)	**10M**(MB)	**21M**(MB)
name	6	65.6	147.5
name, phone	25.8	202.3	285.2
name, email, phone	53.8	393.6	449.9
gender	4.6	50.3	113
gender, name	6.4	69.6	156.4
name, gender	6.4	69.6	156.4
name:1, gender:-1	6.4	69.6	156.4
name:-1, gender:1	6.4	69.6	156.4

For reference, the small collection had a total size of 150MB, the medium was 1.5GB, and the large over 3GB. The index sizes scale up as expected. It is interesting to note how large indexes can become when they are over large collections. For just a 3.1GB collection, a compound index on 3 fields takes up almost half a GB.

Conclusions

Indexing can be a very powerful tool for optimizing queries in MongoDB. They can offer large reductions in time to query by cutting down on the number of documents that must be examined. Mongo’s B-Tree structure for indexes allows a collection to be searched using a structured set. Indexes can take up a large amount of storage space, however. They are also not always helpful if not created properly. It is critical to understand the queries you are trying to run and to ensure that your indexes will actually speed up the search by indexing fields with high cardinality and structuring your queries around that. Indexes are most helpful when you can anticipate a certain set of queries will be run repeatedly on the data, then index the appropriate fields to optimize the execution. From a software side, ideally there should be more support in Mongo and the Node driver for programmatically gathering execution statistics from any type of query with more accurate timing on the database side for execution

References

1. https://docs.mongodb.com/manual/core/index-compound/#index-ascending-and-descending
2. https://docs.mongodb.com/manual/tutorial/sort-results-with-indexes/#sort-on-multiple-fields
3. https://docs.mongodb.com/manual/applications/indexes/
4. https://stackoverflow.com/questions/33545339/how-does-the-order-of-compound-indexes-matter-in-mongodb-performance-wise
5. https://emptysqua.re/blog/optimizing-mongodb-compound-indexes/