Lecure 6 - MongoDB performance, fault tolerance, and deployment part 1

Instructor: Suresh Melvin Sigera
Email: suresh.sigera@cuny.csi.edu

Agenda

- Indexes
- Indexing rules
- Index efficiency
- measuring performance
- Simple index
- Compound indexes
- Reusing compound indexes
- Indexes for sorting

Agenda

- Unique index
- What influences performance
- Improving performance
- QA

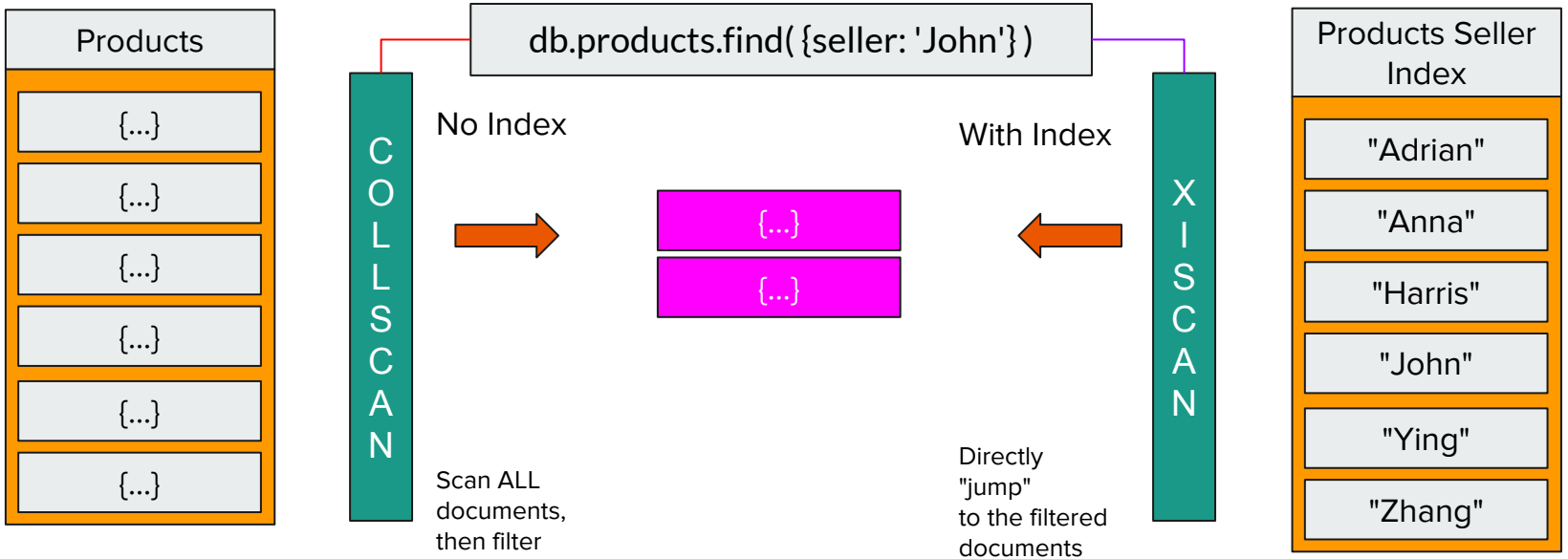
MongoDB - indexes

Indexes are enormously important. With the right indexes in place, MongoDB can use its hardware efficiently and serve your application's queries quickly. But the wrong indexes produce the opposite result: slow queries, slow writes, and poorly utilized hardware. It stands to reason that anyone wanting to use MongoDB effectively must understand indexing.

There are three index types supported by MongoDB: **single field, compound, and multi-key**.

Each of these can be defined as ascending or descending. In addition, there is an auto-generated default index on the _id field.

MongoDB - indexes



MongoDB - indexes

Creating indexes is an easy way to improve MongoDB performance at the collection level. Indexes can be created on a **single field, multiple fields, or embedded fields within arrays or objects**. When you issue a query which involves the indexed field, MongoDB is able to use information stored in the index rather than having to do a full scan of all database documents.

In this sense, you can think of the index as a shortcut which saves time when producing query results.

MongoDB - a thought experiment

Imagine a cookbook. And not just any cookbook — a massive cookbook: 5,000 pages long with the most delicious recipes for every occasion, cuisine, and season, with all the good ingredients you might find at home. This is the cookbook to end them all. Let's call it The Cookbook Omega.

Although this might be the best of all possible cookbooks, there are two tiny problems with The Cookbook Omega.

- The recipes are in random order (On page 3,475 you have Australian Braised Duck, and on page 2 you'll find Zacatecan Tacos.)
- The Cookbook Omega has no index

MongoDB - a thought experiment

Here's the first question to ask yourself: with no index, how do you find the recipe for Rosemary Potatoes in The Cookbook Omega? Your only choice is to scan through every page of the book until you find the recipe. If the recipe is on page 3,973, that's how many pages you have to look through. In the worst case, where the recipe is on the last page, you have to look at every single page. (That would be madness. The solution is to build an index.)

MongoDB - a thought experiment (simple index)

There are several ways you can imagine searching for a recipe, but the recipe's name is probably a good place to start. If you create an alphabetical listing of each recipe name followed by its page number, you'll have **indexed the book by recipe name**. A few entries might look like this:

- Pad Thai: 45
- Toasted Sesame Dumplings: 4,011
- Turkey à la King: 943

As long as you know the name of the recipe (or even the first few letters of that name), you can use this index to quickly find any recipe in the book. If that's the only way you expect to search for recipes, your work is done.

MongoDB - a thought experiment (simple index)

But this is unrealistic because you can also imagine wanting to find recipes based on, say, the ingredients you have in your pantry. Or perhaps you want to search by cuisine. For those cases, you need more indexes.

Here's a second question: with only one index on the recipe name, how do you find all the cauliflower recipes? Again, lacking the proper indexes, you'd have to scan the entire book, all 5,000 pages. This is true for any search on ingredients or cuisine.

You need to build another index, this time on ingredients. In this index, you have an alphabetical listing of ingredients, each pointing to all the page numbers of recipes containing that ingredient.

MongoDB - a thought experiment (simple index)

The most basic index on ingredients would look like this:

- Cashews: 3; 20; 42; 88; 103; 1,215…
- Cauliflower: 2; 47; 88; 89; 90; 275…
- Currants: 1,001; 1,050; 2,000; 2,133…

MongoDB - a thought experiment (simple index)

This index is good if all you need is a list of recipes for a given ingredient. But if you want to include any other information about the recipe in your search, you still have some scanning to do once you know the page numbers where cauliflower is referenced, you then need to go to each of those pages to get the name of the recipe and what type of cuisine it is.

This is better than paging through the whole book, but you can do better.

MongoDB - a thought experiment (compound indexes)

What can you do? Happily, there's a solution to the long-lost cauliflower recipe, and its answer lies in the use of compound indexes.

The two indexes you've created so far are single-key indexes: they both order only one key from each recipe. You're going to build yet another index for The Cookbook Omega, but this time, instead of using one key per index, you'll use two. **Indexes that use more than one key like this are called compound indexes**.

MongoDB - a thought experiment (compound indexes)

This compound index uses both **ingredients** and **recipe name**, in that order. You'll notate the index like this: ingredient-name.

Cashews

- **Cashews marinade**
  - 1,215
- **Chicken with cashews**
  - 88

Cauliflower

- **Lemon-baked cauliflower**
  - 2000

MongoDB - a thought experiment (compound indexes)

One thing to notice: with **compound indexes, order matters**. Imagine the reverse compound index on name-ingredient. Would this index be interchangeable with the compound index we just explored?

Definitely not. With the new index, once you have the recipe name, your search is already limited to a single recipe; a single page in your cookbook.

If this index were used on a search for the recipe Cashew Marinade and the ingredient Bananas, the index could confirm that no such recipe exists. But this use case is the opposite one: you know the ingredient, but not the recipe name.

MongoDB - a thought experiment (compound indexes)

Note: **The cookbook now has three indexes: one on recipe name, one on ingredient, and one on ingredient-name**. **This means that you can safely eliminate the single-key index on ingredient**. Why? Because a search on a single ingredient can use the index on ingredient-name.

That is, if you know the ingredient, you can traverse this compound index to get a list of all page numbers containing said ingredient.

MongoDB - indexing rules

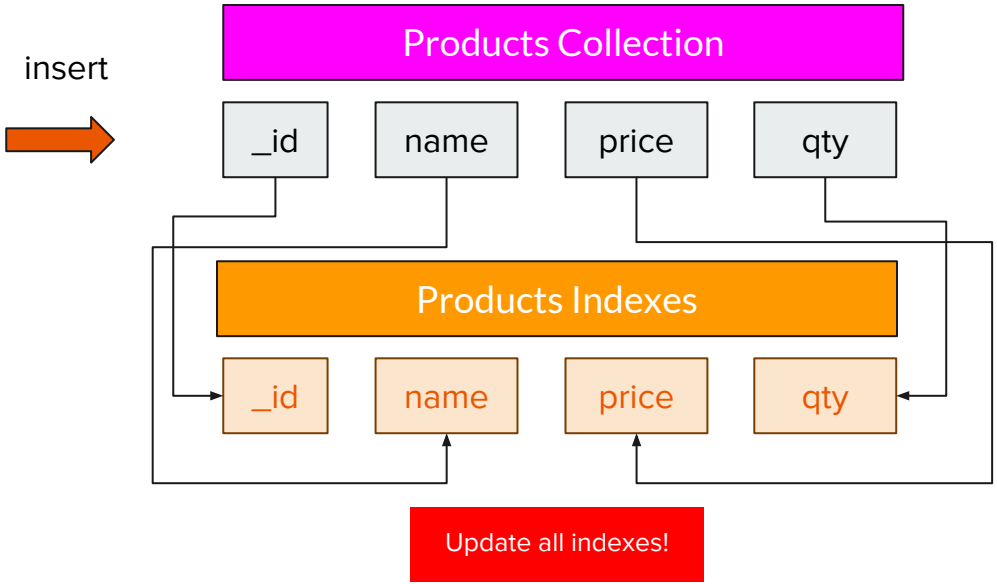
- Indexes significantly reduce the amount of work required to fetch documents. Without the proper indexes, **the only way to satisfy a query is to scan all documents linearly until the query conditions are met**. This frequently means scanning entire collections.
- Only one single-key index will be used to resolve a query. For queries containing multiple keys (say, ingredient and recipe name), a compound index containing those keys will best resolve the query.
- An index on ingredient can and should be eliminated if you have a second index on ingredient-name. More generally, if you have a compound index on a-b, then a second index on a alone will be redundant, but not one on b
- Compound indexes are useful when you wish to create an index on more than one field.

MongoDB - index efficiency

Although indexes are essential for good query performance, each new index imposes a small maintenance cost. Whenever you add a document to a collection, each index on that collection must be modified to include the new document. If a particular collection has 10 indexes, that makes 10 separate structures to modify on each insert, in addition to writing the document itself. This holds for any write operation, whether you're removing a document, relocating a document because the allocated space isn't enough, or updating a given document's indexed keys.

Note: Even with all the right indexes in place, it's still possible that those indexes won't result in faster queries. **This occurs when indexes and a working data set don't fit in RAM**.

MongoDB - don't use too many indexes!



Index all the fields in the collections for the best performance?

MongoDB - measuring performance

Before you proceed, download the person.json database from https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/persons.json and import the data using the following command,

mongoimport persons.json -d people -c contacts --jsonArray --drop

connected to: localhost
dropping: people.contacts
imported 5000 documents

MongoDB - measuring performance

To optimize, one must understand first. The first step towards understanding the performance of our indexes is to learn how to use the `explain()` command. The `explain()` command when used in conjunction with a query will return the query plan that MongoDB would use for this query instead of the actual results. It can take three options: `queryPlanner` (the default), `executionStats`, and `allPlansExecution`.

```
db.contacts.find({ "dob.age": {$gt: 50} }).count()
```

```
db.contacts.explain().find({ "dob.age": {$gt: 50} }).count()
```

MongoDB - measuring performance

Here, we can get information for both the winning query plan and also some partial information about query plans that were considered during the planning phase but rejected because the query planner considered them slower. The `explain()` command returns a rather verbose output anyway, allowing for deep insights into how the query plan works to return our results.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.2

MongoDB - adding a single field index

Here we create an index on the field name, in ascending order of index creation. For descending order, the same index would be created like this:

```
db.contacts.createIndex({ "dob.age": 1 })
```

```
{
     "createdCollectionAutomatically" : false,
     "numIndexesBefore" : 1,
     "numIndexesAfter" : 2,
     "ok" : 1
}
```

MongoDB - adding a single field index

Here we create an index on the field name, in ascending order of index creation. For descending order, the same index would be created like this:

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.3

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.4

MongoDB - adding a single field index

Over time, you may come across scenarios that require redesigning your indexing strategy. This may be by adding a new feature in your application or simply by identifying a more appropriate key that can be indexed. In either case, it is highly advisable to remove older (unused) indexes to ensure you do not have any unnecessary overhead on the database.

In order to remove an index, you can use the `db.<collection>.dropIndex(<index_name>)` If you are not sure about your index name, use the `db.<collection>.getIndexes()` function.

```
db.contacts.dropIndex({"dob.age":1})
```

```
{ "nIndexesWas" : 2, "ok" : 1 }
```

MongoDB - adding a single field index

```
db.contacts.createIndex( { "gender":1 } )
{
     "createdCollectionAutomatically" : false,
     "numIndexesBefore" : 1,
     "numIndexesAfter" : 2,
     "ok" : 1
}

db.contacts.explain("executionStats").find({ "gender": "male" })
```

MongoDB - adding a compound indexes

The beauty of indexes is that they can be used with multiple keys. A single key index can be thought of as a table with one column. A **multi-key index or compound index can be visualized as a multi column table where the first column is sorted first, and then the next, and so on**.

Compound indexes are a **generalization of single-key indexes**, **allowing for multiple fields to be included in the same index**. They are useful when we expect our queries to span multiple fields in our documents and also for consolidating our indexes when we start having too many of them in our collection.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.8

MongoDB - adding a compound indexes

A compound index is declared in a similar way to single indexes, by defining the fields we want to index and the order of indexing:

```
db.books.createIndex({ "dob.age": 1, gender: 1 })
```

```
{
     "createdCollectionAutomatically" : true,
     "numIndexesBefore" : 1,
     "numIndexesAfter" : 2,
     "ok" : 1
}
```

MongoDB - adding a compound indexes

An important attribute of compound indexes is that they can be used for multiple queries on prefixes of the fields indexed. This is useful when we want to consolidate indexes that over time pile up in our collections. This can be used for queries on name or `{dob.age, gender}`:

```
db.contacts.explain().find({ "dob.age":35, gender: "male" })
```

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.8

MongoDB - reusing compound indexes

The order of fields in our query doesn't matter, MongoDB will rearrange fields to match our query. However, the order of fields in our index does matter. A query just for the `gender` field cannot use our index:

```
db.contacts.explain().find({ "gender": "male" })
```

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.9

MongoDB - reusing compound indexes

The underlying reason is that our field's values are stored in the index as secondary, tertiary, and so on indexes; **each one is embedded inside the previous ones**, just like a matryoshka, the Russian nesting doll.

This means that, **when we query on the first field of our multi-field index, we can use the outermost doll to find our pattern**, whereas searching for the first two fields, **we can match the pattern on the outermost doll** and **then dive into the inner one**.

This concept is called **prefix indexing** and together with index intersection is the most powerful tool for index consolidation.

MongoDB - indexes for sorting

In multi-field indexes, though, ordering can determine whether we can use this index to sort or not. In our preceding example, a query matching the sorting direction of our index creation will use our index:

```
db.contacts.explain().find({ "dob.age": 35 }).sort({ gender:1 })
```

It will also use a sort query with all of the sort fields reversed (descending order):

```
db.contacts.explain().find({ "dob.age": 35 }).sort({ gender: -1 })
```

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.10

MongoDB - unique index

A unique index is similar to an RDBMS unique index, forbidding duplicate values for the indexed field. MongoDB creates a unique index by default on the `_id` field for every inserted document:

```
db.contacts.createIndex({ email:1 }, { unique: true })
```

This will create a unique index on a email field. A unique index can also be a compound embedded field or embedded document index. In a compound index, the uniqueness is enforced across the combination of values in all fields of the index.

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.11

MongoDB - partial indexes

If we need to index a subset of the documents in a collection, partial indexes can help us minimize the index set and improve performance. **A partial index will include a condition on the filter that we use in the desired query**.

```
db.contacts.createIndex({ "dob.age" : 1  }, { partialFilterExpression: { gender: "male" }})
```

```
db.contacts.find({ "dob.age": {$gt: 60}, gender: "male" }).count()
```

https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week6/index-part1.12

MongoDB - what influences performance?

- Developer / DB Admin
  - Efficient queries / operations
  - Indexes
  - Use fitting data schema
- DB Admin / System Admin
  - Hardware & Network
  - Sharding
  - Replica sets

MongoDB - improving performance

The general idea is that we need indexes when we expect or already have repeatable queries that are starting to run slow. Indexes do not come for free, as they impose a performance penalty in creation and maintenance but they are more than worth it for frequent queries and can reduce the lock % in our database if designed correctly.

Recapping on our suggestions from previous section we want our indexes to:

- Fit in RAM
- Ensure selectivity
- Be used to sort our query results
- Be used in our most common and important queries

MongoDB - improving performance

Ensuring that our indexes are used to sort our query results is a combination of using compound indexes (which will be used as a whole and also for any **prefix-based query**) and also declaring the direction of our indexes to be in accordance with our most common queries.

Finally, aligning indexes with our query needs is a matter of **application usage patterns** which can uncover which queries are used most of the time and then using `explain()` on these queries to identify the query plan that is being used each time.

MongoDB - special considerations

The following are a few limitations to keep in mind around indexing:

- Index entries have to be < 1024 bytes. This is mostly an internal consideration but we can keep it in mind if we run into issues with indexing.
- A collection can have up to 64 indexes.
- A compound index can have up to 31 fields.
- Special indexes cannot be combined in queries. This includes special query operators that have to use special indexes such as $text for text indexes and $near for geospatial indexes. This is because MongoDB can use multiple indexes to fulfill a query but not in all cases. More about this issue in the Index intersection section.

MongoDB - special considerations

- Multikey and geospatial indexes cannot cover a query. This means that index data alone will not be enough to fulfill the query and the underlying documents will need to be processed by MongoDB to get back the complete set of results.
- Indexes have a unique constraint on fields. We cannot create multiple indexes on the same fields, differing only in options. This is a limitation for sparse and partial indexes, as we cannot create multiple variations of these indexes differing only in the filtering query.
- A fully qualified index name has to be <=128 characters. That also includes database_name, collection_name, and the dots separating them.

QA