جامعة طرابلس

كلية تقنية المعلومات

قواعد البيانات النقالة والغير متجانسة

Heterogeneous and Mobile Databases

ITMC322

أستاذ المادة / محمد أوهيبة

المحاضرة الثامنة

Review of topics

- *Introducing the write operations*
  - *Inserts*
  - *Updates*
- *Write concerns*
- *Bulk writing documents*
- ***Indexing documents***
  - ***Indexing a single field***
  - ***Indexing more than one field***

Indexing

- indexes are important structures when we think of a performance boost.

- indexes are so important that for most database administrators, they are a critical tool in their search for the continuous improvement of database performance.

- In NoSQL databases such as MongoDB, indexing is part of a bigger strategy that will allow us to achieve many gains in performance and assign important behaviors to our database, which will be essential to the data model's maintenance.

- in MongoDB, we have indexes with very special properties.

- For example we can define an index of a date typed field that will control when a document should be removed from the collection.

- in this chapter we will see:
- **Indexing documents**
- **Index types**
- **Special index properties**

Indexing documents

- indexes are data structures that hold part of the data from our main data source.
- In relational databases, indexes hold parts of a table, while in MongoDB, since indexes are on a collection level, these will hold part of a document.
- Similar to relational databases, indexes use a **B-Tree** data structure at implementation level.
- we can create indexes of fields or fields of embedded documents. When we create an index, it will hold a sorted set of values of the fields we choose.
- when we execute a query, if there is an index that covers the query criteria, MongoDB will use the index to limit the number of documents to be scanned.
- We have the **customers** collection that we used in *Querying Documents*, which contains these documents:

```
{
"_id" : ObjectId("54aecd26867124b88608b4c9"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1"
}
```

Indexing documents (Cont.)

- We can create an index in the mongo shell on the username field, by using the createIndex method:

  ```
  db.customers.createIndex({username: 1})
  ```

- The following query will use the previously created index:

  ```
  db.customers.find({username: "customer1"})
  ```

- This is the simplest way to create and use an index in MongoDB. In addition to this, we can create indexes on multikey fields or in embedded documents' fields.

Note
Since Version 3.0.0, the ensureIndex method is deprecated and is an alias for the createIndex method.

Indexing a single field

- The index could be created on a field of any type in the collection of documents.
- Consider the customers collection we used before, with some modification to work in this section:

```
{
"_id" : ObjectId("54aecd26867124b88608b4c9"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1",
"age" : 25,
"address" : {
        "street" : "Street 1",
        "zipcode" : "87654321",
        "state" : "RJ"
  }
}
```

- The following command creates an ascending index in the username field:

```
db.customers.createIndex({username: 1})
```

- In the preceding code, we just passed a single document as a parameter to the createIndex method. The document {username: 1} contains a reference to the field that the index should be creating and the order: 1 for ascending or -1 for descending.

Indexing a single field (Cont.)

- Another way to create the same index, but in descending order, is:

```
db.customers.createIndex({username: -1})
```

- In the following query, MongoDB will use the index created in the username field to reduce the number of documents in the customers collection that it should inspect:

```
db.customers.find({username: "customer1"})
```

- we could create an index of a field in an embedded document. Therefore, queries such as this will use the created index:

```
db.customers.createIndex({"address.state": 1})
```

- MongoDB will use the index created in the username field to reduce the number of documents in the customers collection.

```
db.customers.find({"address.state": "RJ"})
```

- a bit more complex, we can also create an index of the entire embedded document:

```
db.customers.createIndex({address: 1})
```

Indexing a single field (Cont..)

- The following query will use the index:

```
db.customers.find(
{
    "address" :
    {
        "street" : "Street 1",
        "zipcode" : "87654321",
        "state" : "RJ"
    }
}
)
```

- But none of these queries will do this:

```
db.customers.find({state: "RJ"})

db.customers.find({address: {zipcode: "87654321"}})
```

- This happens because in order to match an embedded document, we have to match exactly the entire document, including the field order.

Indexing a single field (Cont...)

- The following query will not use the index either:

```
db.customers.find(
    {
        "address" :
        {
        "state" : "RJ",
        "street" : "Street 1",
        "zipcode" : "87654321"
        }
    }
    )
```

- Although the document contains all the fields, these are in a different order.

- By reviewing a concept that you learned in Chapter 3, Querying Documents, the **_id** field.
  For every new document created in a collection, we should specify the _id field. If we do not specify it, MongoDB automatically creates one **ObjectId** typed for us. Furthermore, every collection automatically creates a unique ascending index of the **_id** field. That being said, we can state that the **_id** field is the document's **primary key**.

Indexing more than one field

- In MongoDB, we can create an index that holds values for more than one field.
- Call this kind of index a **compound index.**
- There is no big difference between a single field index and a compound index. the biggest difference is in the sort order.
- let's use the customers collection to create our first compound index:

```
{
"_id" : ObjectId("54aecd26867124b88608b4c9"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1",
"age" : 25,
"address" : {
      "street" : "Street 1",
      "zipcode" : "87654321",
      "state" : "RJ"
      }
}
```

Indexing more than one field (Cont.)

- We can imagine that an application that wants to authenticate a customer uses the username and password fields together in a query like this:

```
db.customers.find(
{ username: "
customer1",
password: "b1c5098d0c6074db325b0b9dddb068e1"
})
```

- To enable better performance when executing this query, we can create an index of both the username and password fields:

```
db.customers.createIndex({username: 1, password: 1})
```

- Nevertheless, for the following queries, does MongoDB use the compound index?

```
#Query 1
db.customers.find({username: "customer1"})
#Query 2
db.customers.find({password: "b1c5098d0c6074db325b0b9dddb068e1"})
#Query 3
db.customers.find(
{
password: "b1c5098d0c6074db325b0b9dddb068e1",
username: "customer1"
})
```

Indexing more than one field (Cont..)

- The answer is yes for Query 1 and Query 3. As mentioned before, the order is very important in the creation of a compound index.

- The index created will have references to the documents sorted by the username field, and within each username entry, sorted by password entries. Thus, a query with only the password field as the criteria will not use the index.

- assume for a moment that we have the following index in the customers collection:

```
db.customers.createIndex(
{
"address.state":1,
"address.zipcode": 1,
"address.street": 1
})
```

- You might be asking which queries will use our new compound index?

- we need to understand a compound index concept in MongoDB: the **prefix**.

- The **prefix** in a compound index is a subset of the indexed fields. As its name suggests, it is the fields that take precedence over other fields in the index.
- In our example, both {"address.state":1} and {"address.state":1, "address.zipcode": 1} are index prefixes.

Indexing more than one field (Cont...)

- A query that has any index prefix will use the compound index. Therefore, we can deduce that:

- Queries that include the address.state field will use the compound index

- Queries that include both the address.state and address.zipcode fields will also use the compound index

- Queries with address.state, address.zipcode and address.street will also use the compound index

- Queries with both address.state and address.street will also use the compound
- Index

- The compound index will not be used on queries that:
- Have only the address.zipcode field
- Have only the address.street field
- Have both the address.zipcode and address.street fields

Indexing more than one field (Cont....)

- **Note**
- despite a query that has both **address.state** and **address.street** fields using the index, we could achieve a better performance in this query if we have single indexes for each field.
- This is explained by the fact that the compound index will be first sorted by **address.state**, followed by a sort on the **address.zipcode** field, and finally a sort on the **address.street** field. Thus, it is much more expensive for MongoDB to inspect this index than to inspect the other two indexes individually.

- So, for this query:

```
db.customers.find(
{
"address.state": "RJ",
"address.street": "Street 1"
})
```

- It would be more efficient if we have this index:

```
db.customers.createIndex({"address.state": 1, "address.street": 1})
```

Indexing multikey fields

- Another way to create indexes in MongoDB is to create an index of an array field. These indexes can hold arrays of primitive values, such as strings and numbers, or even arrays of documents.

- We must be particularly attentive while creating multikey indexes. Especially when we want to create a compound multikey index. It is not possible to create a compound index of two array fields.

Note
The main reason why we could not create an index of parallel arrays is because they will require the index to include an entry in the Cartesian product of the compound keys, which will result in a large index.

Indexing multikey fields (Cont.)

- Consider the customers collection with documents like this one:

```
{
"_id" : ObjectId("54aecd26867124b88608b4c9"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1",
"age" : 25,
"address" : {
"street" : "Street 1",
"zipcode" : "87654321",
"state" : "RJ"
        },
        "followedSellers" : [
        "seller1",
        "seller2",
        "seller3"
],
"wishList" : [
        {
        "sku" : 123,
        "seller" : "seller1"
        },
        {
        "sku" : 456,
        "seller" : "seller2"
        },
        {
        "sku" : 678,
        "seller" : "seller3"
        }
]
}
```

Indexing multikey fields (Cont..)

- We can create the following indexes for this collection:

```
db.customers.createIndex({followedSellers: 1})

db.customers.createIndex({wishList: 1})

db.customers.createIndex({"wishList.sku": 1})
db.customers.createIndex({"wishList.seller": 1})
```

- But the following index cannot be created:

```
db.customers.createIndex({followedSellers: 1, wishList: 1}
```

Indexing for text search

- MongoDB gives us the chance to create indexes that will help us in a text search. Although there are a wide variety of specialized tools for this, such Apache Solr, Sphinx, and ElasticSearch, most of the relational and NoSQL databases have full text searching natively.

Reference Book

- **MongoDB Data Modeling Wilson da Rocha França**
  Copyright © 2015 Packt Publishing

The End

Thanks for listening ..

Any questions ?