Large-Scale and Multi-Structured Databases

Indexes in MongoDB

Prof Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Copyright Issues

Most of the information included this presentation have been extracted from the official documentation of MongoDB and from the book “MongoDB the Definitive Guide”

Check on the last slide the sources used to extract the material for this class.

Introduction

Indexes support the ***efficient execution*** of read queries in MongoDB.

Without indexes, MongoDB must perform a collection scan, i.e. scan ***every document*** in a collection, to ***select*** those documents that ***match*** a ***query*** statement.

If an appropriate index exists for a query, MongoDB can use the index to ***limit*** the number of documents it must inspect.

What are Indexes?

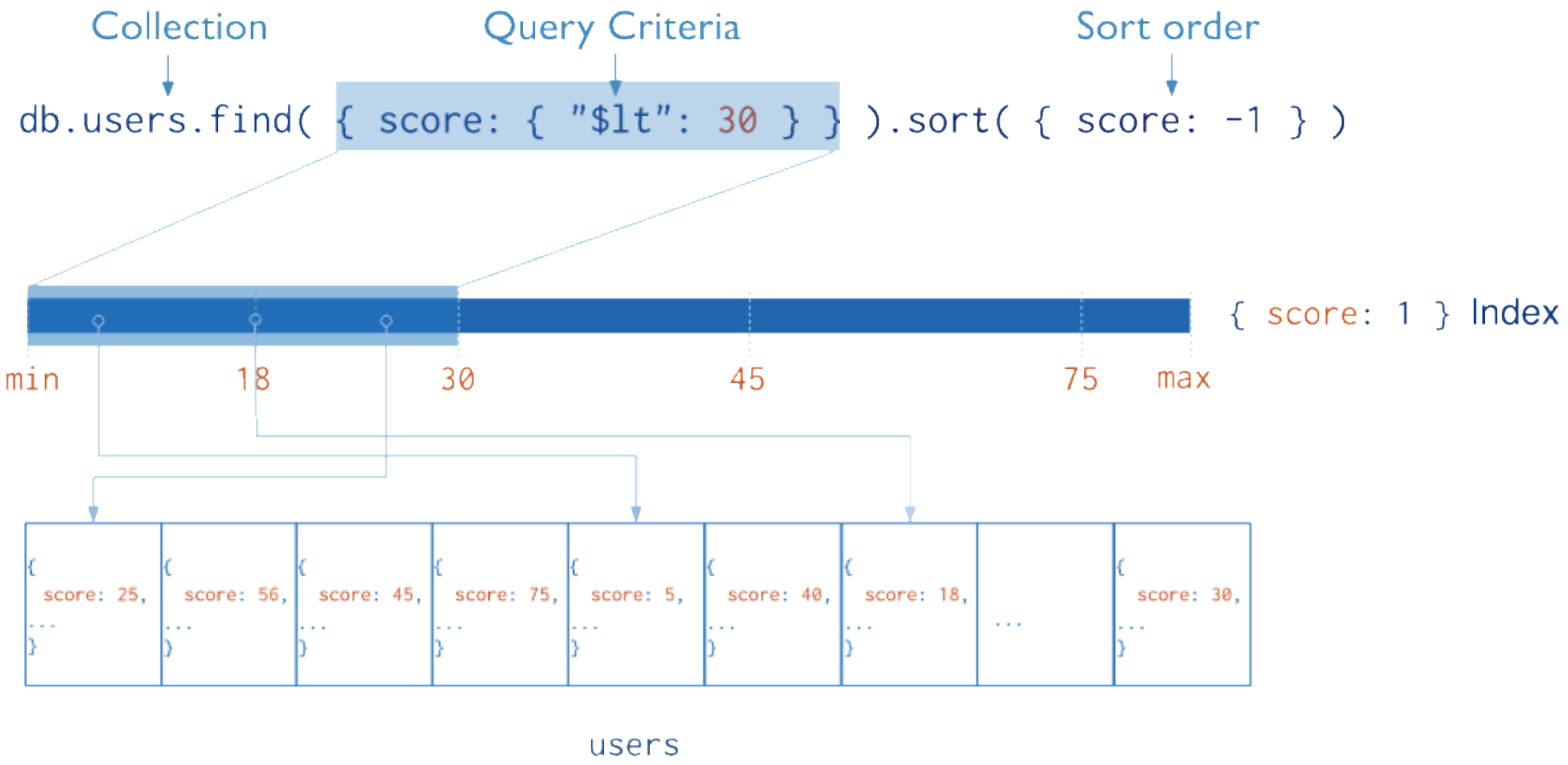
Indexes are ***special data structures*** that store a ***small portion*** of the collection's data set in an easy to traverse form.

The index stores the value of a ***specific field*** or ***set of fields***, ***ordered*** by the value of the field.

The ordering of the index entries supports efficient ***equality matches*** and ***range-based*** query operations.

In addition, MongoDB can ***return sorted results*** by using the ordering in the index.

An Example of Index



MongoDB defines indexes at the ***collection level*** and supports indexes on ***any field or sub-field*** of the documents in a MongoDB collection.

Default _id Index

MongoDB creates a ***unique index*** on the _id field during the creation of a collection.

A unique index ensures that the indexed fields ***do not store duplicate values.***

The _id index ***prevents*** clients from ***inserting*** two ***documents*** with the **same** value for the ***_id*** field.

The index on the _id field cannot be dropped.

Creating an Index

The following method can be used for creating and index:

*db.collection.createIndex**(keys, options)***

keys-> a document that contains the field and value pairs where:

- the field is the index key and
- the value describes the type of index for that field. For an ascending index on a field, specify a value of 1; for descending index, specify a value of -1.

options-> a document which contains a set of options that controls the creation of the index.

*The method **only creates** an index if an index of the same specification **does not already exist.***

An example of Index (single field)

```
> db.zipcodes.createIndex(
...    { pop: 1} ,
...    { name: "query for population" }
... )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

For a ***single-field index*** and sort operations, the ***sort order*** (i.e. ascending or descending) of the index key ***does not matter***.

We can view index names using the ***db.collection.getIndexes()*** method.

We ***cannot rename*** an index once created. Instead, we must drop and re-create the index with a new name.

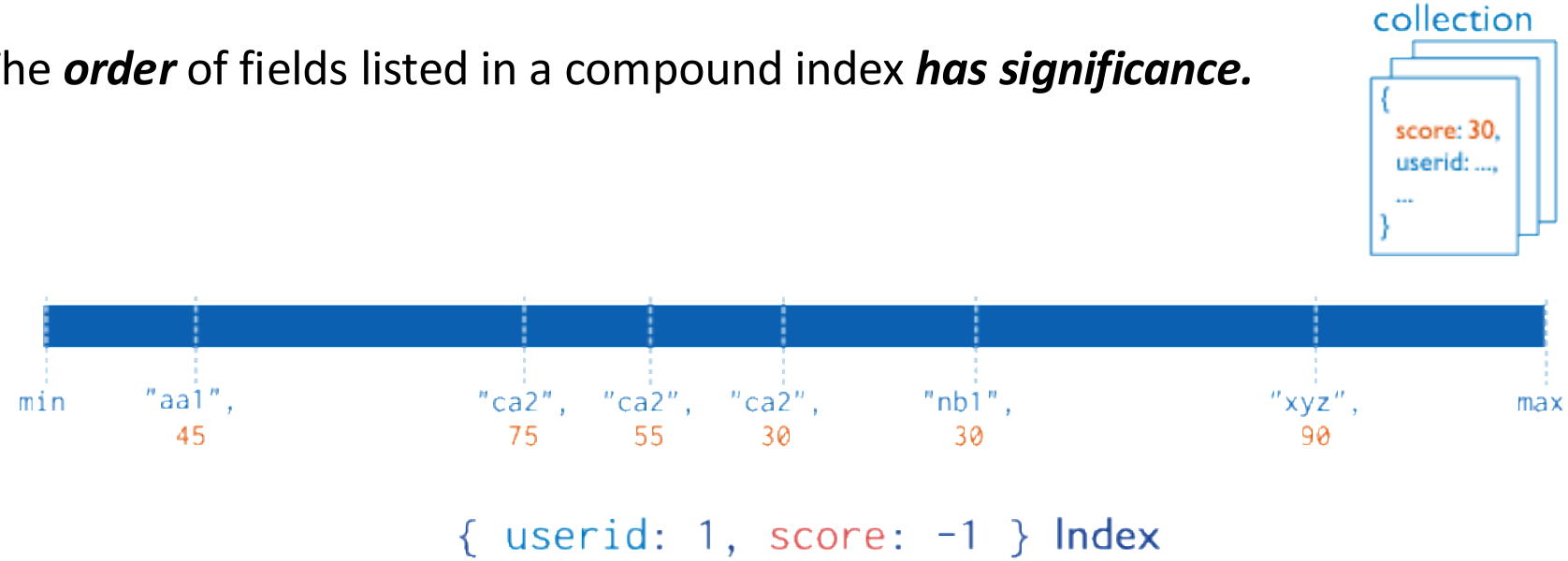
An example of Dropping an Index

```
> db.zipcodes.dropIndex("query for population")
{ "nIndexesWas" : 2, "ok" : 1 }
> db.zipcodes.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "exer.zipcodes"
        }
]
```

Compound Index

In the following image we can see an example of user-defined index ***on multiple fields***.

The ***order*** of fields listed in a compound index ***has significance.***



Sorting with Compound Index

We can specify a sort query 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 ***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}.

Indexing Embedded Documents (I)

Indexes can be created on keys in ***embedded documents*** in the same way that they are created on normal keys.

Let suppose to have a collection of document like the following one:

```
{
    "username" : "sid",
    "loc" : {
        "ip" : "1.2.3.4",
        "city" : "Springfield",
        "state" : "NY"
}}
```

We can define an index on the subfield ”city” as follows:

db.users.createIndex({"loc.city" : 1})

Indexing Embedded Documents (II)

What happens if for the previous example

```
{
      "username" : "sid",
      "loc" : {
         "ip" : "1.2.3.4",
         "city" : "Springfield",
         "state" : "NY"
}}
```

we define and index as follows *db.users.createIndex({loc : 1}) ?*

Indexing the ***entire subdocument*** will only help queries that are querying for the entire subdocument, such as
db.users.find({"loc" : {"ip" : "123.456.789.000", "city" : "Shelby ville", "state" : "NY"}}})).

The following query: db.users.find({"loc.city" : "Shelbyville"})
will be not supported by the the index on the entire document.

Indexing arrays (I)

Let suppose to have a collection of posts extracted from a social network. Each post includes a set of comments modelled as an array of documents:

```
{
   _id =: 1,
   title : "post title”,
   body : "post body”,
   comments = [
{text: “this is the first comment”, date: "2016-05-18T16:00:00Z" },
{text: “this is the second comment”, date: "2016-05-18T16:10:00Z"},
        ]
}
```

We can create an index on the date subfield of the field comments as follows:

db.blog.createIndex({"comments.date" : 1})

Indexing arrays (II)

Notice that indexing an array ***creates an index entry for each element of the array,*** so if a post had 20 comments, it would have 20 index entries.

This makes array indexes ***more expensive*** than single-value ones: for a single insert, update, or remove, every array entry of the index might have to be updated.

Consider a collection composed by documents like the following one:

{ _id: 1, item: "ABC", ratings: [2, 5, 9] }

If we create this index: *db.survey.createIndex({ ratings: 1 })*

The index contains three keys, each pointing to the same document.

Indexes on array elements ***do not keep any notion of position***: we cannot use an index for a query that is looking for a specific array element .

Indexing arrays (III)

```
db.students.insertMany( [
    {
        "name": "Andre Robinson",
        "test_scores": [ 88, 97 ]
    },
    {
        "name": "Wei Zhang",
        "test_scores": [ 62, 73 ]
    },
    {
        "name": "Jacob Meyer",
        "test_scores": [ 92, 89 ]
    }
] )
```

We can define this index:

- *db.students.createIndex({ test_scores: 1 })*

And the keys are stored in the following order:

- *[62, 73, 88, 89, 92, 97]*

The index support the following query:

db.students.find(
{test_scores: { $elemMatch: { $gt: 90 } } })

Advanced Indexes

MongoDB offers the possibility of defining the following advanced indexes:

- ***Geospatial Index***: supports efficient queries of geospatial coordinate data. MongoDB provides two special indexes: ***2d indexes*** that uses ***planar geometry*** when returning results and ***2dsphere indexes*** that use ***spherical geometry*** to return results. Details: https://www.mongodb.com/docs/manual/core/indexes/index-types/index-geospatial/

- ***Text Index***: MongoDB provides text indexes to support ***text search*** queries on string content. Text indexes can include ***any field*** whose value is a ***string*** or an ***array of string elements***. Details: https://docs.mongodb.com/manual/core/index-text/

- Hash index: supports ***hash based sharding*** and indexes the hash of the value of a field. Only support equality matches and cannot support range-based queries. Details: https://docs.mongodb.com/manual/core/index-hashed/

Types of Indexes

- ***Unique Indexes:*** The unique property for an index causes MongoDB to ***reject duplicate*** values for the indexed field (https://docs.mongodb.com/manual/core/index-unique/).

- ***Partial Indexes***: only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance (https://docs.mongodb.com/manual/core/index-partial/).

- ***Sparse Indexes:*** the sparse property of an index ensures that the index ***only contain entries for documents that have the indexed field***. The index skips documents that do not have the indexed field (https://docs.mongodb.com/manual/core/index-sparse/).

- ***TTL Indexes:*** are special indexes that MongoDB can use to ***automatically remove*** documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to ***persist in a database for a finite amount of time (https://docs.mongodb.com/manual/core/index-ttl/)***.

EVALUATE THE PERFORMANCE OF A QUERY

The Collection

Let suppose to have the following collection:

{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 }
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 }
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 }
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 }
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 }
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 }
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 }
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 }
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 }
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }

Query Performance without Indexes

To evaluate the performance of a query chain the ***cursor.explain("executionStats")*** cursor method to the end of the find command as follows:

db.inventory.find({ quantity: { $gte: 100, $lte: 200 } }).explain("executionStats") which returns the following document*:*

```
{
  "queryPlanner" : {
},
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 3,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 10,
    "executionStages" : {
      "stage" : "COLLSCAN",
      ...
    },
    ...
  },
  ...
}
```

Query Performance with Indexes

Let define the following index: *db.inventory.createIndex({ quantity: 1 })*

Let use the command:
db.inventory.find({ quantity: { $gte: 100, $lte: 200 } }).explain("executionStats")
which returns the following document*:*

```
{
  "queryPlanner"

"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 3,
        "executionStages" : {
          ...
        },
        ...
  },
  ...
}
```

Compare Performance of Indexes

Consider the following query:

db.inventory.find({ quantity: { $gte: 100, $lte: 300 }, type: "food" }) which returns*:*

{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 5, "item" : "f3", "type" : "food", "quantity" : 300 }

We can define the following ***two compound*** indexes:

db.inventory.createIndex({ quantity: 1, type: 1 })
db.inventory.createIndex({ type: 1, quantity: 1 })

With compound indexes, ***the order of the fields matter.***

- The first index orders by quantity field first, and then the type field.
- The second index orders by type first, and then the quantity field.

Which is the best compound index?

```
test> db.inventory.find()
[
  { _id: 1, item: 'f1', type: 'food', quantity: 500 },
  { _id: 2, item: 'f2', type: 'food', quantity: 100 },
  { _id: 3, item: 'p1', type: 'paper', quantity: 200 },
  { _id: 4, item: 'p2', type: 'paper', quantity: 150 },
  { _id: 5, item: 'f3', type: 'food', quantity: 300 },
  { _id: 6, item: 't1', type: 'toys', quantity: 500 },
  { _id: 7, item: 'a1', type: 'apparel', quantity: 250 },
  { _id: 8, item: 'a2', type: 'apparel', quantity: 400 },
  { _id: 9, item: 't2', type: 'toys', quantity: 50 },
  { _id: 10, item: 'f4', type: 'food', quantity: 75 }
]
```

db.inventory.find({ quantity: { $gte: 100, $lte: 300 }, type: "food"

db.inventory.createIndex({ quantity: 1, type: 1 })
db.inventory.createIndex({ type: 1, quantity: 1 })

With compound indexes, ***the order of the fields matter.***

- The first index orders by quantity field first, and then the type field.
- The second index orders by type first, and then the quantity field.

Evaluate the performance of the using the first index

We can use the ***hint()*** method to force MongoDB to use a specific index for the query.

Let use the command:
db.inventory.find({ quantity: { $gte: 100, $lte: 300 }, type: "food" }).hint({ quantity: 1, type: 1 }).explain("executionStats")

which returns the following document*:*

```
{
  "queryPlanner" : {
},
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 2,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 6,
    "totalDocsExamined" : 2,
    "executionStages" : {
    ...
    }
  },
  ...
}
```

Evaluate the performance of the using the second index

Let use the command:
db.inventory.find({ quantity: { $gte: 100, $lte: 300 }, type: "food" }).hint({type: 1, quantity: 1}).explain("executionStats")

which returns the following document*:*

```
{
  "queryPlanner" : {
},
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 2,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 2,
    "totalDocsExamined" : 2,
    "executionStages" : {
    ...
    }
  },
  ...
}
```

Suggested Readings

Students are invited to read the official documentation of MongoDB.

The documentation is available at:
https://docs.mongodb.com/manual/indexes/
https://docs.mongodb.com/manual/tutorial/analyze-query-plan/

Chapter 6 of the Book “MongoDB the Definitive Guide, 3rd Edition”