جامعة طرابلس

كلية تقنية المعلومات

قواعد البيانات النقالة والغير متجانسة

Heterogeneous and Mobile Databases

ITMC322

أستاذ المادة / محمد أوهيبة

المحاضرة التاسعة

Review of topics

- ***Indexing for text search***
- ***Creating special indexes***
- ***Optimizing Queries***
  - ***Understanding the query plan***
  - ***Evaluating queries***
  - ***Covering a query***
  - ***The query optimizer***

Indexing for text search

- It is possible to create a text index of a string or an array of string fields in a collection.
- For the following examples, we will use the products collection.

```
{
        "_id" : ObjectId("54837b61f059b08503e200db"),
        "name" : "Product 1",
        "description" :
        "Product 1 description",
        "price" : 10,
        "supplier" : {
        "name" : "Supplier 1",
        "telephone" : "+552199998888"
},
"review" : [
        {
        "customer" : {
        "email" : "customer@customer.com"
        },
        "stars" : 5
        }
  ],
        "keywords" : [ "keyword1", "keyword2", "keyword3" ]
}
```

Indexing for text search (Cont.)

- We can create a text index just by specifying the text parameter in the createIndex
- method:

```
db.products.createIndex({name: "text"})

db.products.createIndex({description: "text"})

db.products.createIndex({keywords: "text"})
```

- All the preceding commands could create a text index of the products collection.
- MongoDB has a **limitation**, in that we can only have one text index per collection.
- Thus,only one of the previous commands could be executed for the products collection.
- it is possible to create a compound text index:

```
db.products.createIndex({name: "text", description: "text"})
```

- This command creates a text index field for the name and description fields.

Indexing for text search (Cont..)

- A common and useful way of creating a text index of a collection is to create an index for all text fields of the collection. There is a special syntax for creating this index, which you can see as follows:

```
db.products.createIndex({"$**","text"})
```

- For a query to use a text index, we should use the $text operator in it. And, to better understand how to create an effective query, it is good to know how the indexes are created.
- the same process is used to execute the query using the $text operator.
- To sum up the process, we can split it into three phases:
  - Tokenization
  - Removal of suffix and/or prefix, or stemming
  - Removal of stop words
- In order to optimize our queries, we can specify the language we are using in our text fields, and consequently in our text index, so that MongoDB will use a list of words in all three phases of the indexing process.
- **Since its 2.6 version, MongoDB supports the following languages:**
- **da or danish , nl or dutch , en or english , fi or finnish , fr or french , de or german**
- **hu or hungarian ,it or italian ,nb or norwegian , pt or portuguese , ro or romanian**
- **ru or russian , es or spanish, sv or swedish ,tr or turkish**

Indexing for text search (Cont...)

- An example of an index creation with language could be:

```
db.products.createIndex({name: "text"},{ default_language: "pt"})
```

- We can also opt to not use any language, by just creating the index with a none value:

```
db.products.createIndex({name: "text"},{ default_language: "none"})
```

- By using the none value option, MongoDB will simply perform tokenization and stemming; it will not load any stop words list.

Creating special indexes

- we have three more special indexes: time to live, unique, and sparse.
- **Time to live indexes**
- The time to live (TTL) index is an index based on lifetime.
- This index is created only in fields that are from the Date type. They cannot be compound and they will be automatically removed from the document after a given period of time.
- This type of index can be created from a date vector. The document will expire at the moment when the lower array value is reached.
- MongoDB is responsible for controlling the documents' expiration through a background task at intervals of 60 seconds.
- For an example, let's use the customers collection as we have seen previously :

```
{
"_id" : ObjectId("5498da405d0ffdd8a07a87ba"),
"username" : "customer1",
"email" : "customer1@customer.com",
"password" : "b1c5098d0c6074db325b0b9dddb068e1",
"accountConfirmationExpireAt" : ISODate("2015-01-11T20:27:02.138Z")
}
```

Creating special indexes (Cont.)

- The creation command that is based on the time to live index for the *accountConfirmationExpireAt* field will be the following:

```
db.customers.createIndex(
{accountConfirmationExpireAt: 1}, {expireAfterSeconds: 3600}
)
```

- This command indicates that every document that is older than the value in seconds requested in the expireAfterSeconds field will be deleted.

- There is also another way to create indexes based on lifetime, which is the scheduled way. The following example shows us this method of implementation:

```
db.customers.createIndex({
accountConfirmationExpireAt: 1}, {expireAfterSeconds: 0}
)
```

- This will make sure that the document you saw in the previous example expires on January 11 2015, 20:27:02.

- This type of index is very useful for applications that use machine-generated events, logs and session information, which need to be persistent only during a given period of time.

Creating special indexes (Cont..)

- **Unique indexes**
- MongoDB has a unique index. The unique index is responsible for rejecting duplicated values in the indexed field.
- The unique index can be created from a single or from a multikey field and as a compound index.
- When creating a unique compound index, there must be uniqueness in the values'
- combinations.
- The default value for a unique field will always be null if we don't set any value during the insert operation.
- Considering the last example of the customers collection, it's possible to create a unique index by executing the following:

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

- This command will create an index of the username field that will not allow duplicated values.

Creating special indexes (Cont...)

- **Sparse indexes**

- Sparse indexes are indexes that will be created only when the document has a value for the field that will be indexed.

- Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is "sparse" because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

- We can create sparse indexes using only one field from the document or using more fields. This last use is called **a compound index**

- When we create compound indexes, it is mandatory that at least one of the fields has a **not-null-value**.

Creating special indexes (Cont....)

- an example the following documents in the customers collection:

```
{ "_id" : ObjectId("54b2e184bc471cf3f4c0a314"), "username" :
   "customer1",
"email" : "customer1@customer.com", "password" :
"b1c5098d0c6074db325b0b9dddb068e1" }

{ "_id" : ObjectId("54b2e618bc471cf3f4c0a316"), "username" :
   "customer2",
"email" : "customer2@customer.com", "password" :
"9f6a4a5540b8ebdd3bec8a8d23efe6bb" }

{ "_id" : ObjectId("54b2e629bc471cf3f4c0a317"), "username" :
   "customer3",
"email" : "customer3@customer.com" }
```

- Using the following example command, we can create a sparse index in the customers collection:

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

- The following example query uses the created index:

```
db.customers.find({password: "9f6a4a5540b8ebdd3bec8a8d23efe6bb"})
```

- On the other hand, the following example query, which requests the descending order by the indexed field, will not use the index:

```
db.customers.find().sort({password: -1})
```

Optimizing Queries

- Now that we have taken a great step forward in comprehending how to improve read and write performance using indexes.
- we will study the concept of query plans and how MongoDB handles it.
- This includes understanding query covering and query selectivity, and how these plans behave when used in sharded environments and through replica sets.

Optimizing Queries

- **Understanding the query plan**

- When we run a query, MongoDB will internally figure out the best way to do it by choosing from a set of possibilities extracted after query analysis (performed by the MongoDB query optimizer). These possibilities are called **query plans**.

- To get a better understanding of a query plan, we have to use the cursor concept and one of the cursor methods: explain( ).

- The explain( ) method is one of the big changes in the MongoDB 3.0 release. It has been significantly enhanced due to the new query introspection system.

- We now can pass to the explain( ) method an option parameter that specifies the verbosity of the explain output. The possible modes are "queryPlanner", "executionStats", and "allPlansExecution". The default mode is "queryPlanner".

- **In the "queryPlanner"** mode, MongoDB runs the query optimizer to choose the winning plan under evaluation, and returns the information to the evaluated method.
- **In the "executionStats"** mode, MongoDB runs the query optimizer to choose the winning plan, executes it, and returns the information to the evaluated method. If we are executing the explain() method for a write operation, it returns the information about the operation that would be performed but does not actually execute it.
- **Finally, in the "allPlansExecution"** mode, MongoDB runs the query optimizer to choose the winning plan, executes it, and returns the information to the evaluated method as well as information for the other candidate plans.

Optimizing Queries

- **Understanding the query plan**

- The output of an explain execution shows us the query plans as a tree of stages.
- From the leaf to the root, each stage passes its results to the parent node.

- There are four stages:
- COLLSCAN: This means that a full collection scan happened during this stage
- IXSCAN: This indicates that an index key scan happened during this stage
- FETCH: This is the stage when we are retrieving documents
- SHARD_MERGE: This is the stage where results that came from each shard are merged and passed to the parent stage.

- Detailed information about the winning plan stages can be found in the explain.queryPlanner.winningPlan key of the explain() execution output.

- The explain.queryPlanner.winningPlan.stage key shows us the name of the root stage. If there are one or more child stages, the stage will have an inputStage or inputStages key depending on how many stages we have.

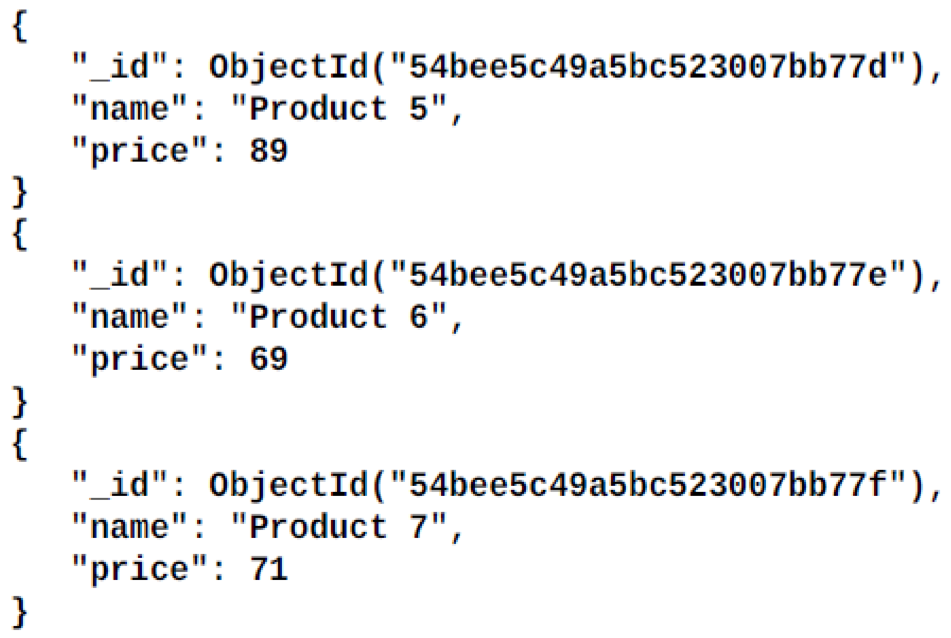
- The child stages will be represented by the keys explain.queryPlanner.winningPlan.inputStage and explain.queryPlanner.winningPlan.inputStages of the explain() execution output.

Optimizi

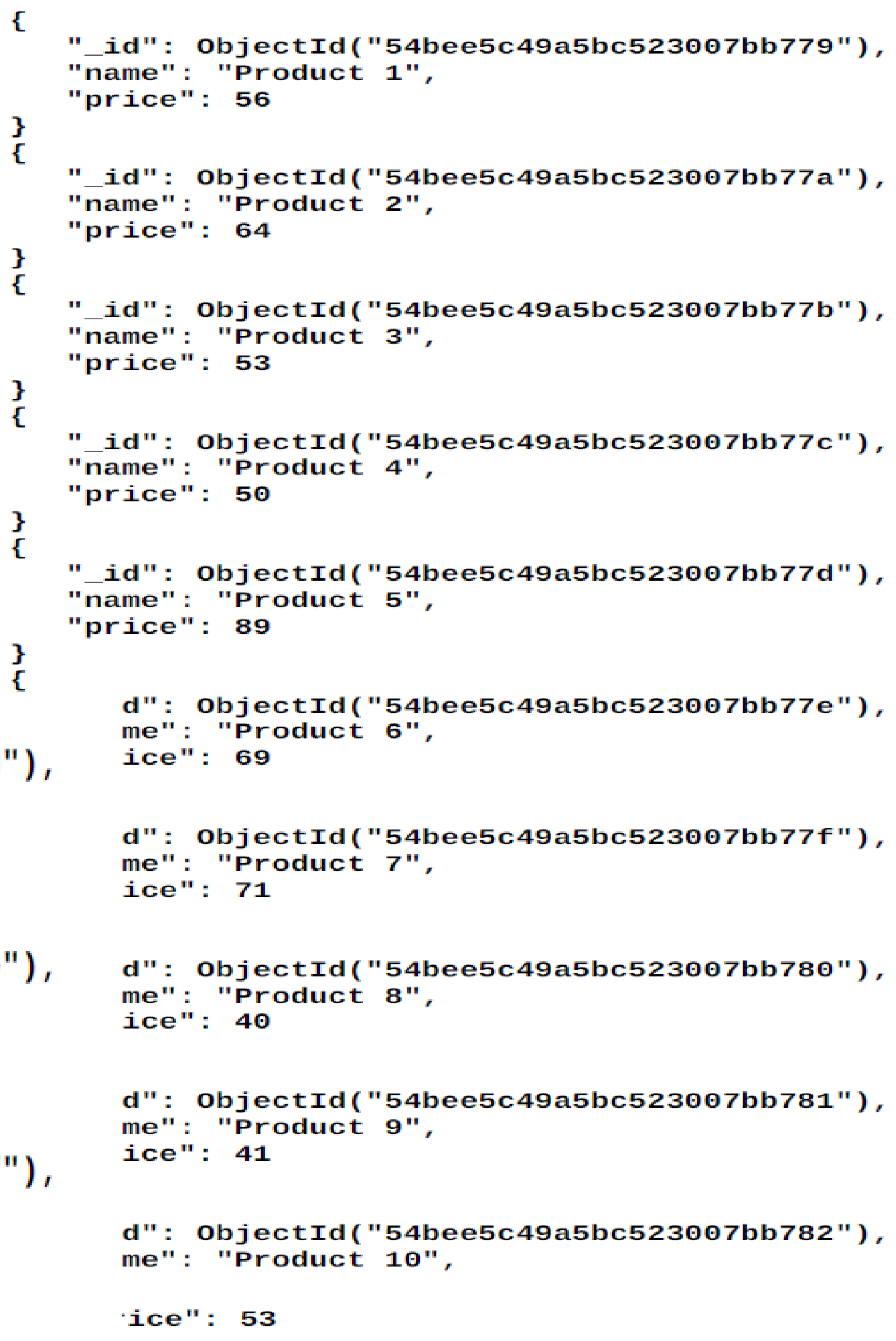
- **Evaluating queries**

- the explain method will give us statist
  will see in these statistics whether a cu
- Let's use the following products collec

- To get all the documents in the collect
   query in the mongod shell:
       db.products.find({price: {$gt: 65}})
- The result of the query will be the foll



```
{
   "_id": ObjectId("54bee5c49a5bc523007bb77d"),
   "name": "Product 5",
   "price": 89
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77e"),
   "name": "Product 6",
   "price": 69
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77f"),
   "name": "Product 7",
   "price": 71
}
```



```
{
   "_id": ObjectId("54bee5c49a5bc523007bb779"),
   "name": "Product 1",
   "price": 56
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77a"),
   "name": "Product 2",
   "price": 64
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77b"),
   "name": "Product 3",
   "price": 53
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77c"),
   "name": "Product 4",
   "price": 50
}
{
   "_id": ObjectId("54bee5c49a5bc523007bb77d"),
   "name": "Product 5",
   "price": 89
}
{
     d": ObjectId("54bee5c49a5bc523007bb77e"),
     me": "Product 6",
     ice": 69

     d": ObjectId("54bee5c49a5bc523007bb77f"),
     me": "Product 7",
     ice": 71

     d": ObjectId("54bee5c49a5bc523007bb780"),
     me": "Product 8",
     ice": 40

     d": ObjectId("54bee5c49a5bc523007bb781"),
     me": "Product 9",
     ice": 41

     d": ObjectId("54bee5c49a5bc523007bb782"),
     me": "Product 10",

     ice": 53
```

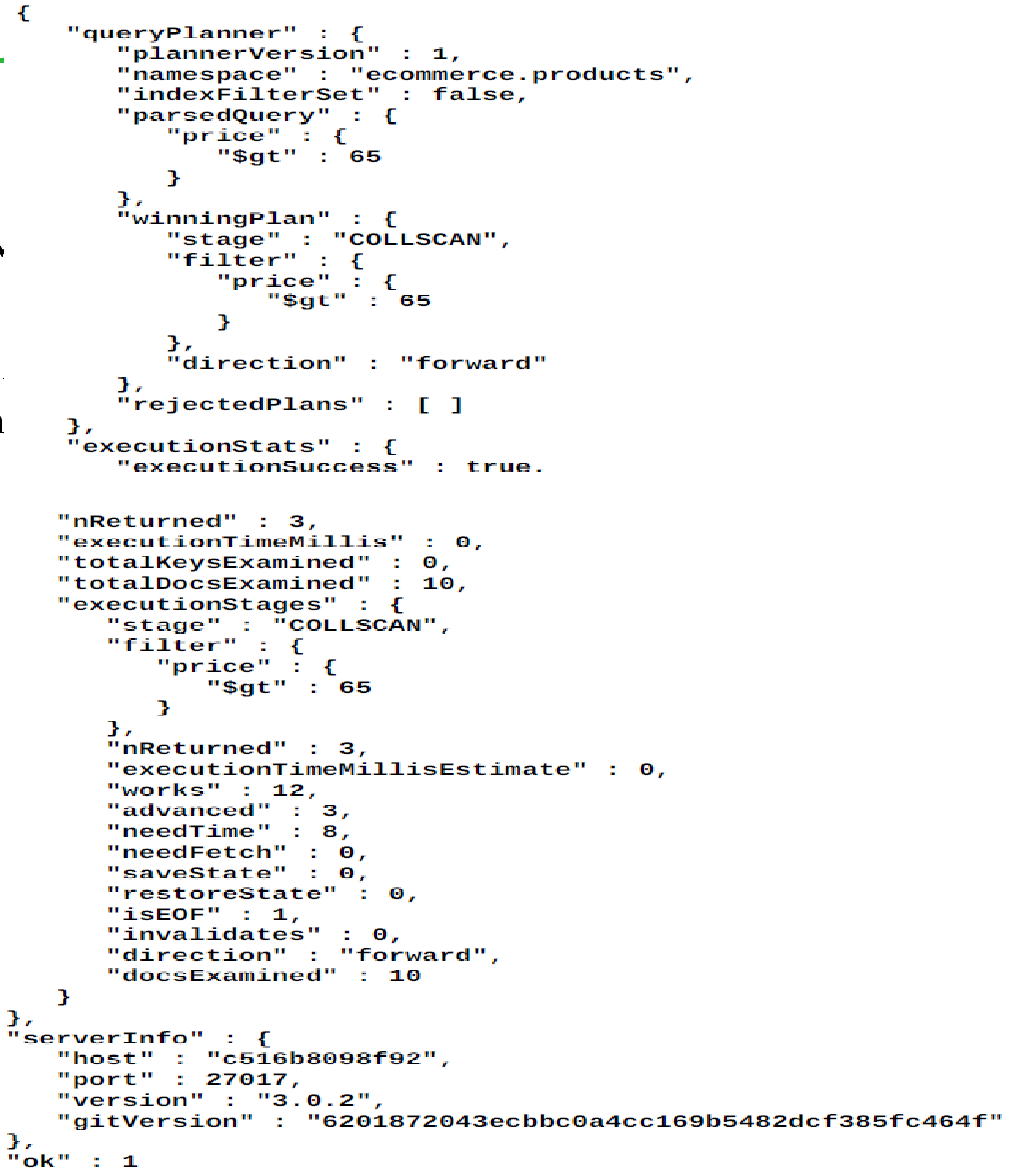
Optir

- **Evaluating queries**

- To help you understand how M
method on the cursor that was

    db.products.find({price:

- The result of this operation is a
plan:

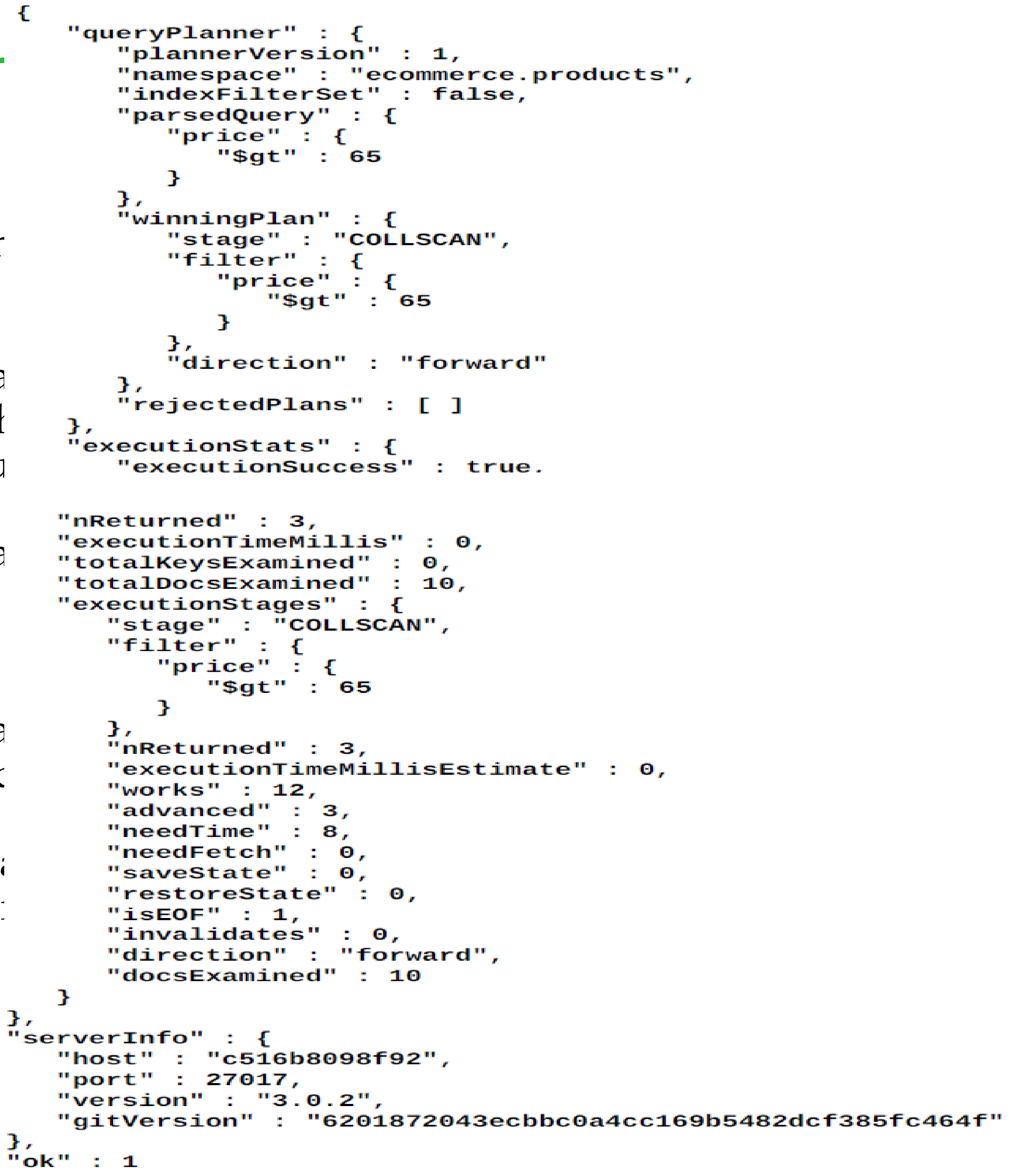


```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "ecommerce.products",
      "indexFilterSet" : false,
      "parsedQuery" : {
         "price" : {
            "$gt" : 65
         }
      },
      "winningPlan" : {
         "stage" : "COLLSCAN",
         "filter" : {
            "price" : {
               "$gt" : 65
            }
         },
         "direction" : "forward"
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true.

  "nReturned" : 3,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 10,
  "executionStages" : {
     "stage" : "COLLSCAN",
     "filter" : {
        "price" : {
           "$gt" : 65
        }
     },
     "nReturned" : 3,
     "executionTimeMillisEstimate" : 0,
     "works" : 12,
     "advanced" : 3,
     "needTime" : 8,
     "needFetch" : 0,
     "saveState" : 0,
     "restoreState" : 0,
     "isEOF" : 1,
     "invalidates" : 0,
     "direction" : "forward",
     "docsExamined" : 10
  }
},
"serverInfo" : {
   "host" : "c516b8098f92",
   "port" : 27017,
   "version" : "3.0.2",
   "gitVersion" : "6201872043ecbbc0a4cc169b5482dcf385fc464f"
},
"ok" : 1
```

Optir

- **Evaluating queries**
- The queryPlanner.winningPlar will be performed.
- The queryPlanner.executionSta match the query criteria. In otl returned from the query execu
- The queryPlanner.executionSta documents from the collection documents were scanned.
- The queryPlanner.executionSta index entries that were scannec
- When executing a collection sc represents the number of docu



```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "ecommerce.products",
      "indexFilterSet" : false,
      "parsedQuery" : {
         "price" : {
            "$gt" : 65
         }
      },
      "winningPlan" : {
         "stage" : "COLLSCAN",
         "filter" : {
            "price" : {
               "$gt" : 65
            }
         },
         "direction" : "forward"
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true.

   "nReturned" : 3,
   "executionTimeMillis" : 0,
   "totalKeysExamined" : 0,
   "totalDocsExamined" : 10,
   "executionStages" : {
      "stage" : "COLLSCAN",
      "filter" : {
         "price" : {
            "$gt" : 65
         }
      },
      "nReturned" : 3,
      "executionTimeMillisEstimate" : 0,
      "works" : 12,
      "advanced" : 3,
      "needTime" : 8,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 10
   }
},
"serverInfo" : {
   "host" : "c516b8098f92",
   "port" : 27017,
   "version" : "3.0.2",
   "gitVersion" : "6201872043ecbbc0a4cc169b5482dcf385fc464f"
},
"ok" : 1
```

Optimizi

- **Evaluating queries**

- What happens if we create an index o

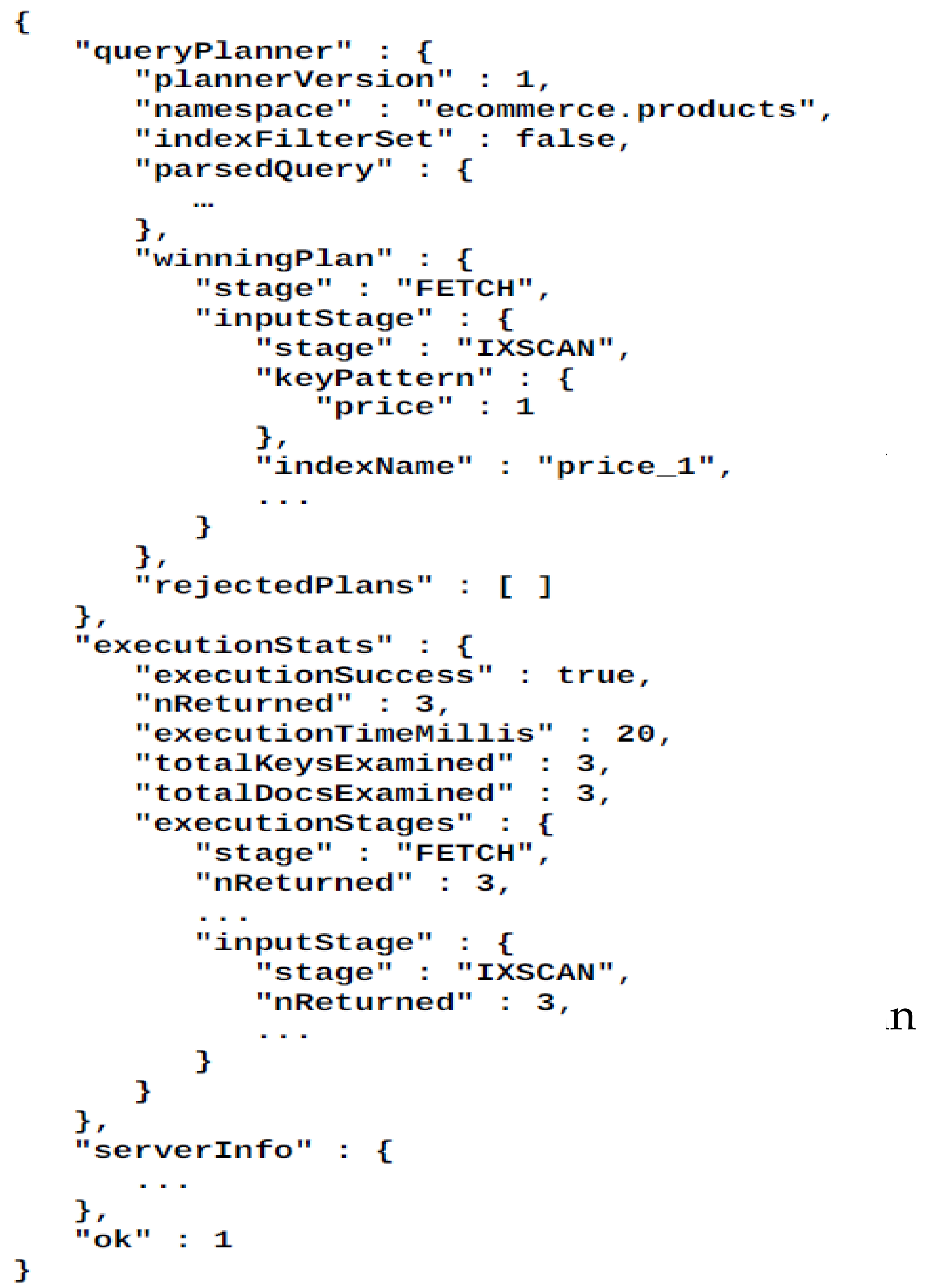
  db.products.createIndex({price:

- Obviously, the query result will be the the previous execution. However, the following:

- let’s focus on
- these four fields: queryPlanner.winnir
- queryPlanner.executionStats.nReturne
- queryPlanner.executionStats.totalKey
- queryPlanner.executionStats.totalDoc

- This time, we can see that we did not had a FETCH stage with a child IXSC queryPlanner.winningPlan.inputStage index. The name of the index can be fo queryPlanner.winningPlan.inputStage

n



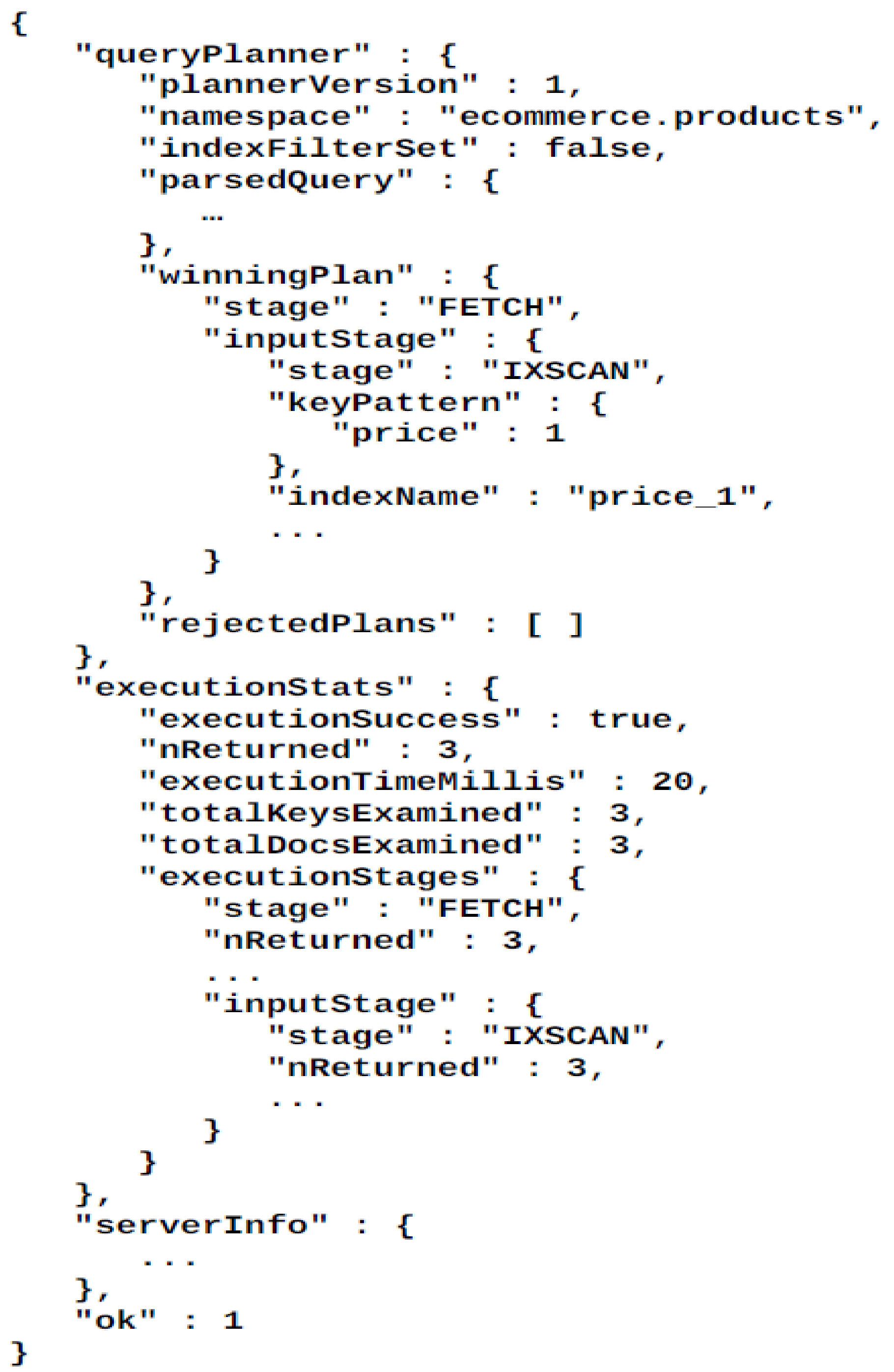
```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "ecommerce.products",
      "indexFilterSet" : false,
      "parsedQuery" : {
         …
      },
      "winningPlan" : {
         "stage" : "FETCH",
         "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
               "price" : 1
            },
            "indexName" : "price_1",
            ...
         }
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 3,
      "executionTimeMillis" : 20,
      "totalKeysExamined" : 3,
      "totalDocsExamined" : 3,
      "executionStages" : {
         "stage" : "FETCH",
         "nReturned" : 3,
         ...
         "inputStage" : {
            "stage" : "IXSCAN",
            "nReturned" : 3,
            ...
         }
      }
   },
   "serverInfo" : {
      ...
   },
   "ok" : 1
}
```

Optimizi

- **Evaluating queries**

- Furthermore, the mean difference in tl
queryPlanner.executionStats.totalDoc
queryPlanner.executionStats.totalKey
that three documents were scanned. It
we saw when executing the query wit

- One point we should make is that the e
same as we can see in queryPlanner.e
queryPlanner.executionStats.totalKey ot
covered by the index.



```
{
   "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "ecommerce.products",
      "indexFilterSet" : false,
      "parsedQuery" : {
         …
      },
      "winningPlan" : {
         "stage" : "FETCH",
         "inputStage" : {
            "stage" : "IXSCAN",
            "keyPattern" : {
               "price" : 1
            },
            "indexName" : "price_1",
            ...
         }
      },
      "rejectedPlans" : [ ]
   },
   "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 3,
      "executionTimeMillis" : 20,
      "totalKeysExamined" : 3,
      "totalDocsExamined" : 3,
      "executionStages" : {
         "stage" : "FETCH",
         "nReturned" : 3,
         ...
         "inputStage" : {
            "stage" : "IXSCAN",
            "nReturned" : 3,
            ...
         }
      }
   },
   "serverInfo" : {
      ...
   },
   "ok" : 1
}
```

Reference Book

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

The End

Thanks for listening ..

Any questions ?