Lab

Topic: Working with MongoDB.

Purpose: Setting up a database. Documents. Installation and administration of the database. Adding data. Sampling and filtering. Pagination and sorting. Indexes. Aggregate functions. Sampling operators. Updating data. Deleting data. Setting links in the database. Collection management.

Progress

Setting up the database. Documents

If in relational databases the content consists of tables, then the mongodb database consists of collections.

Each collection has its own unique name - an arbitrary identifier consisting of no more than 128 different alphanumeric characters and underscores.

Unlike relational databases, MongoDB does not use a table device with a clearly defined number of columns and data types. MongoDB is a document-oriented system in which the central concept is a document.

A document can be submitted as an object that stores certain information. In a sense, it is similar to strings in relational subds, where strings store information about a single element. For example, a typical document:

```
1   {
2   "name": "Tom",
3   "surname": "Smith",
4   "age": "37",
5   "company": {
6   "name" : "Microsoft",
7   "salary" : "100"
8   }
9   }
```

A document represents a set of key-value pairs. For example, in the expression "name": "Tom" "name" is the key and "Tom" is the value.

Keys are strings. Values may vary by data type. In this case, almost all values are also of string type, and only one key (company) refers to a single object. In total, there are the following types of values:

- **String**: string data type (UTF-8 encoding is used for strings)
- **Array**: data type for storing arrays of elements
- **Binary data**: type for storing data in binary format
- **Boolean**: a Boolean data type that stores boolean valuesTRUEorFALSE, example,{"married": FALSE}
- **Date**: stores the date in Unix time format
- **Double**: a numeric data type for storing floating-point numbers
- **Integer**: used to store 32-bit integer values, e.g.{"age": 29}
- **Long**: used to store 64-bit integer values
- **JavaScript**: data type to store javascript code
- **Min key/Max key**: are used to compare values with the smallest/largest BSON elements
- **Null**: data type to store the valueNull
- **Object**: an object containing a set of properties
- **ObjectId**: data type for storing document id
- **Regular expression**: used to store regular expressions
- **Decimal128**: a data type for storing 128-bit decimal fractional numbers, which solves problems with the precision problem of calculations when using fractional numbers that represent type Double.
- **Timestamp**: used to store time

Unlike strings, documents can contain heterogeneous information. So, next to the document described above, there can be another object in the same collection, for example:

```
1    {
2    "name": "Bob",
3    "birthday": "1985.06.28",
4    "place": "Berlin",
5    "languages":[
```

```
6     "english",
7     "German",
8     "Spanish"
9     ]
10    }
```

It would seem that different objects except for individual properties, but they can only be in a collection.

A couple more important notes: in MongoDB, queries are case-sensitive and strictly typed. That is, such two documents will not be identical:

```
1     {"age": "28"}
2     {"age": 28}
```

If in the first case a string value is defined for the age key, then in the second case the value is a number.

Document identifier

Each MongoDB document is defined by a unique identifier called _id. When a document is added to a collection, this ID is automatically generated. However, the developer can explicitly specify the identifier himself, rather than relying on automatically generated ones, by specifying the corresponding key and its value in the document.

This field must have a unique value within the collection. And if we try to add two documents with the same ID to the collection, only one of them will be added, and when we add the second one, we will get an error.

If the identifier is not explicitly specified, MongoDB creates a custom 12-byte binary value. This value consists of several segments: a 4-byte timestamp value (representing the number of seconds since the start of the Unix epoch), a 5-byte random number, and a 3-byte counter that is initialized to a random number. This identifier construction model guarantees with a high probability that it will have a unique value.

Installation and administration of the database

In this and subsequent articles, we'll cover basic MongoDB data operations using both the mongosh console shell and the MongoDB Compass graphical client.

However, in any case, at the beginning of working with the server, you should not forget to start the server itself - that is, the mongod application.

When starting to work with MongoDB in the mongosh console shell, first of all we need to set the database we need as the current one, in order to use it later. To do this, use the use command followed by the name of the database. At the same time, it is not important whether such a database exists or not. If it doesn't exist, MongoDB will automatically create it when you add data to it.

So, let's start the mongosh console shell and enter the following command there:

```
1    use usersdb
```


Now the usersdb database will be set as the current one. At the same time, it is not important that such a database may not exist: if it does not exist, it is created during the first operation.

If you are suddenly not sure whether a database with this name already exists, you can use the show dbs command to display the names of all existing databases on the console:

```
usersdb> show dbs
admin 40.00 KiB
config 72.00 KiB
local 72.00 KiB
test 40.00 KiB
usersdb>
```

Note that there is no usersdb in the list of databases yet, because I haven't done anything with it yet.

You can specify any name for the database, but there are some restrictions. For example, the name should not contain the characters /, \, ., ", *, <, >, :, |, ?, $. In addition, database names are limited to 64 bytes.

There are also reserved names that can be used: local, admin, config. These names represent databases that are already on the server by default and are intended for service purposes.
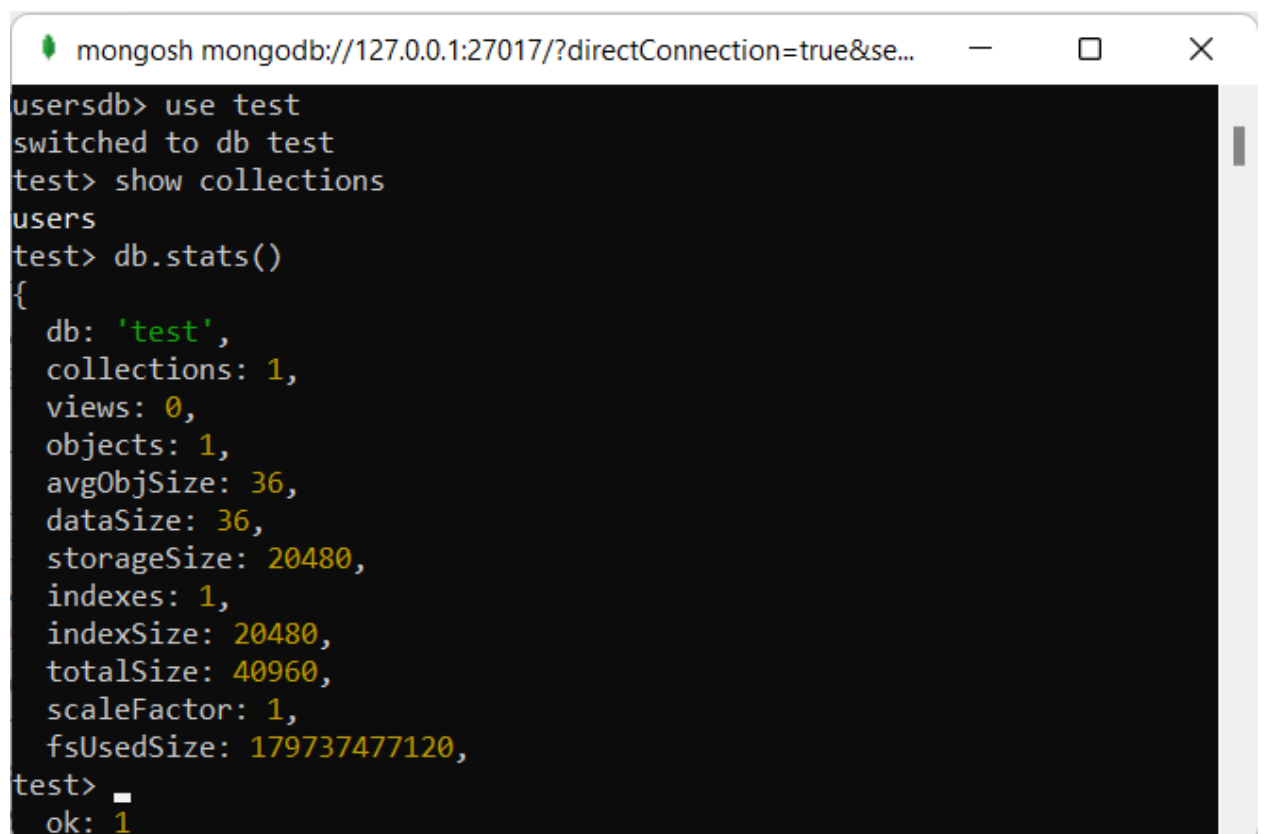
Moreover, as you can see, there is no test database in this list, because I have not yet added data to it.

In addition to databases, we can view a list of all collections in the current database using the command

```
1    show collections
```

Getting statistics

Using the db.stats() command, you can get the statistics of the current database. For example, we currently have the test database installed:

Similarly, we can find out all the statistics about an individual collection. For example, we learn statistics from the collection users:db.users.stats()

Adding data

Having installed the database, we can now add data to it. All data is stored in the database in BSON format, which is close to JSON, so we also need to enter data in this format. And although we may not have any collection at the moment, when we add data to it, one is automatically created.

As mentioned earlier, the collection name is an arbitrary identifier consisting of no more than 128 different alphanumeric characters and an underscore. At the same time, the collection name should not start with the system. prefix, because it is reserved for internal collections (for example, the system.users collection contains all database users). Also, the name should not contain a dollar sign - $.

Three of its methods can be used to add to the collection:

- **insertOne()**: adds one document
- **insertMany()**: adds multiple documents

Let's say we use the test database. Let's add one document to it:

```
1   test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",
    "spanish"]})
```

A document represents a set of key-value pairs. In this case, the attached document has three keys: name, age, languages, and assigns a certain value to each of them. For example, an array is mapped to the languages key as a value.

Key names can be used with or without quotes.

Some limitations when using key names:

- The $ character cannot be the first character in a key name
- The key name cannot contain a dot character.

When adding data, if we have not explicitly provided a value for the "_id" field (that is, the unique identifier of the document), it is automatically generated. Yes, after performing the addition operation, the console will display the identifier generated for the added document:

```
test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",
"spanish"]})
{
acknowledged: true,
insertedId: ObjectId("62e27b1b06adfcddf4619fc1")
```

```
}
test>
```

In the server's response, we will receive an object whose insertedId parameter will contain an identifier.

But, in principle, we can set this identifier ourselves when adding data:

```
1   test> db.users.insertOne({"_id": 123457, "name": "Tom", "age": 28,
    languages: ["english", "spanish"]})
```

or use the ObjectId type for the identifier

```
1   test> db.users.insertOne({"_id": ObjectId("62e27b1b06adfcddf4619fc6"),
    "name": "Tom", "age": 28, languages: ["english", "spanish"]})
```
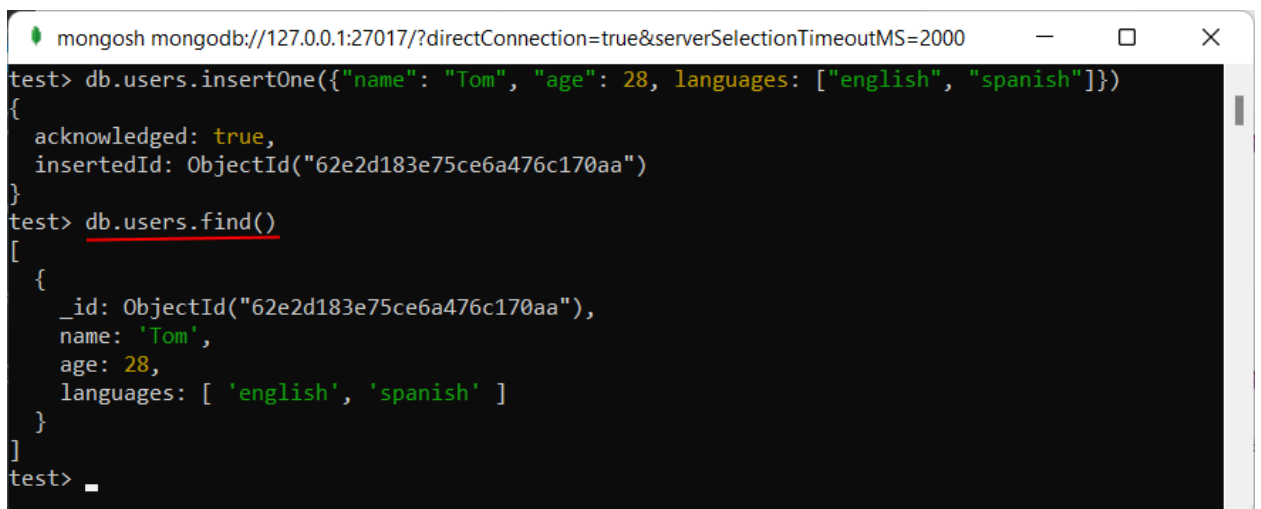
It is worth considering that if the identifier definition uses the ObjectId type, it must contain a string of 12 bytes or a string of 24 characters.

If the addition is successful, the ID of the added document will be displayed on the console.

And to make sure that the document is in the database, we output it with the find function.

```
1   test> db.users.find()
```

By default, it displays all documents in the collection:

If we need to add a number of documents, we can use the insertMany() method, which accepts an array of objects:

```
1   db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english",
    "french"]},
2   {"name": "Alice", "age": 31, languages:["german", "english"]}])
```

After adding, the console displays the identifiers of the added documents:

There is another way to add a document to the database, which includes two stages: defining the document (document = ({ ... })) and actually adding it:

```
1    document=({"name": "Bill", "age": 32, languages: ["english", "french"]})
2    db.users.insertOne(document)
```

It may not be convenient for everyone to enter all pairs of keys and properties in one line. But MongoDB's intelligent JavaScript-based interpreter allows you to enter multi-line commands as well. If the expression is not finished (from the point of view of the JavaScript language) and you press Enter, the input of the next part of the expression is automatically moved to the next line:

Loading data from a file

The mongodb database data can be defined in a plain text file, which is quite convenient because we can transfer or forward this file independently of the mongodb database. For example, let's define somewhere on the hard disk the file users.js with the following content:

```
1    db.users.insertMany([
2    {"name": "Alice", "age": 31, languages: ["english", "french"]},
3    {"name": "Lene", "age": 29, languages: ["english", "spanish"]},
```

```
4    {"name": "Kate", "age": 30, languages: ["german", "russian"]}
5    ])
```

That is, three documents are added to the users collection using the insertMany method.

To load a file into the current database, the load() function is used, to which the path to the file is passed as a parameter:

```
1    load("D:/users.js")
```

In this case, it is assumed that the file is located in the path "D:/users.js".

Sampling and filtering

The easiest way to retrieve documents from a collection is to use the find() function. The action of this function is in many respects similar to the usual query SELECT * FROM Table, which is used in SQL and which extracts all rows. For example, to get all the documents from the users collection created in the previous topic, we can use the command:

```
1    db.users.find()
```

```
test> db.users.find()
[
  {
    _id: ObjectId("62e2d183e75ce6a476c170aa"),
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ab"),
    name: 'Bob',
    age: 26,
    languages: [ 'english', 'french' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ac"),
    name: 'Alice',
    age: 31,
```

Data filtering

However, if we need to receive not all documents, but only those that meet a certain requirement. For example, we previously added the following documents to the database:

```
1    db.users.insertOne({"name":  "Tom",  "age":  28, languages: ["english",
     "spanish"]})
```

```
2   db.users.insertOne({"name":  "Bill",  "age": 32, languages: ["english",
    "french"]})
3   db.users.insertOne({"name":  "Tom",   "age": 32, languages: ["english",
    "german"]})
```

Let's display all documents in which name=Tom:

```
1   db.users.find({name: "Tom"})
```

Such a query will give us two documents in which name=Tom.

```
test> db.users.find({name: "Tom"})
[
{
_id: ObjectId("62e2d6a5e75ce6a476c170b3"),
name: "Tom"
age: 28,
languages: [ 'english', 'spanish' ]
},
{ _id: ObjectId("62e2d348e75ce6a476c170ae"),
_id: ObjectId("62e2d6a5e75ce6a476c170b5"),
name: "Tom"
age: 32,s: [ 'english', 'spanish' ]
languages: [ 'english', 'german' ]
}
] _id: ObjectId("62e2d3d5e75ce6a476c170af"),
test>
```

Now a more complicated request: we need to display those objects that have name=Tom and at the same time age=32. That is, in the SQL language it could look like this: SELECT * FROM Table WHERE Name='Tom' AND Age=32. The last added object meets this criterion. Then we can write the following query:

```
1   db.users.find({name: "Tom", age: 32})
```

Filtering by missing properties

Some documents may have a certain property, others may not. What if we want to retrieve documents that do not have a certain property? In this case, the property is passed null . For example, we will find all documents where the languages property is missing:

```
1   db.users.find({languages: null})
```

Or we will find all documents where name="Tom", but the languages property is not defined.

```
1   db.users.find({name: "Tom", languages: null})
```

Filtering by array elements

It is also easy to search by element in an array. For example, the following query returns all documents that have english in the languages array:

```
1   db.users.find({languages: "english"})
```

Let's complicate the query and get those documents that have two languages in the languages array at the same time: "english" and "german":

```
1   db.users.find({languages: ["english", "german"]})
```

And it is in this order, where "english" is defined first, and "german" - second.

Now let's display all documents in which "english" is in the first place in the languages array:

```
1   db.users.find({"languages.0": "english"})
```

Note that "languages.0" provides a complex property and is therefore enclosed in quotes. Accordingly, if we need to display documents where English is in the second place (for example, ["german", "english"]), then instead of zero we put one: "languages.1".

Consider a more complex example, where an array element represents a complex object. Suppose we have the following documents in the database:

```
1   db.users.insertOne({"name": "Bob", "age": 28, friends: [{"name": "Tim"},
    {"name": "Tom"}]})
2   db.users.insertOne({"name": "Tim", "age": 29, friends: [{"name": "Bob"},
    {"name": "Tom"}]})
3   db.users.insertOne({"name": "Sam", "age": 31, friends: [{"name": "Tom"}]})
    db.users.insertOne({"name": "Tom", "age": 32, friends: [{"name": "Bob"},
4   {"name": "Tim"}, {"name": " Sam"}]})
```

Let's select all documents where the name property of the first element in the friends array is equal to "Bob":

```
1   test> db.users.find({"friends.0.name": "Bob"})
```

Console output:

```
test> db.users.find({"friends.0.name": "Bob"})
[
{
_id: ObjectId("62e39da1c881653067e87901"),
name: "Tim"
age: 29,
friends: [ { name: 'Bob' }, { name: 'Tom' } ]
},
```

```
{
_id: ObjectId("62e39da1c881653067e87903"),
name: "Tom"
age: 32,
friends: [ { name: 'Bob' }, { name: 'Tim' }, { name: 'Sam' } ]
}
]
test>
```

Projection

A document may have many fields, but not all of these fields may be necessary and important to us when making a request. And in this case, we can include only the required fields in the sample using projection. For example, let's display only a portion of information, for example, the value of the "age" fields in all documents in which name=Tom:

```
1    db.users.find({name: "Tom"}, {age: 1})
```

Using one as the parameter {age: 1} indicates that the query should return only the content of the age quality.

```
test> db.users.find({name: "Tom"}, {age: 1})
[
{ _id: ObjectId("62e2d6a5e75ce6a476c170b3"), age: 28 },
{ _id: ObjectId("62e2d6a5e75ce6a476c170b5"), age: 32},
{ _id: ObjectId("62e2d799e75ce6a476c170b7"), age: 28 },
{ _id: ObjectId("62e39da1c881653067e87903"), age: 32 }
]
test>
```

And the opposite situation: we want to find all fields of the document, except for the age property. In this case, we specify 0 as a parameter:

```
1    db.persons.find({name: "Tom"}, {age: 0})
```

At the same time, it should be taken into account that even if we note that we want to get only the name field, the _id field will also be included in the resulting selection. Therefore, if we do not want to see this field in the sample, we must explicitly specify: {"_id":0}

Alternatively, instead of 1 and 0, you can use true and false:

```
1    db.users.find({name: "Tom"}, {age: true, _id: false})
```

If we do not want to specify the sample, but want to display all documents, we can leave the first curly brackets empty:

```
1    db.users.find({}, {age: 1, _id: 0})
```

Query nested objects

Previous queries were applied to simple objects. But documents can be very complex in structure. For example, let's add the following document to the users collection:

```
1    db.users.insertOne({"name":      "Alex",       "age":       28,      company:
     {"name":"Microsoft", "country":"USA"}})
```

A nested object with the company key is defined here. And to find all documents in which the name=microsoft property is nested in the company key, we need to use the dot operator:

```
1    db.users.find({"company.name": "Microsoft"})
```

Using JavaScript

In addition to executing database queries, we can execute JavaScript expressions. For example, we can create a function and apply it:

```
1    function sqrt(n) { return n*n; }
2    sqrt(5)
```

Console output:

```
test> function sqrt(n) { return n*n; }
[Function: sqrt]
test> sqrt(5)
25
test>
```

And we can use similar JavaScript functions and expressions in database queries. For example, we will find all documents where the age field is equal to sqrt(5)+3:

```
test> db.users.find({age: sqrt(5)+3})
[
{
_id: ObjectId("62e2d6a5e75ce6a476c170b3"),
name: "Tom"
age: 28,
languages: [ 'english', 'spanish' ]
},
{ _id: ObjectId("62e2d76ae75ce6a476c170b6"), name: 'Thomas', age: 28 },
{ _id: ObjectId("62e2d799e75ce6a476c170b7"), name: 'Tom', age: 28 },
{
_id: ObjectId("62e39da1c881653067e87900"),
name: "Bob"
age: 28
}
```

```
]
test>
```

Using regular expressions

Another great feature when building queries is using regular expressions. For example, we will find all documents in which the value of the name key begins with the letter B:

```
db.users.find({name:/^B\w+/i})
```

Sample console output:

```
test> db.users.find({name:/^B\w+/i})
[
{
_id: ObjectId("62e2d6a5e75ce6a476c170b4"),
name: "Bill"
age: 32,
languages: [ 'english', 'french' ]
},
{
_id: ObjectId("62e39da1c881653067e87900"),
name: "Bob"
age: 28
}
]
test>
```

Single document search

If all documents are retrieved by the find function, then a single document is retrieved by the findOne function

For example, let's select one element with name="Tom":

```
1    test> db.users.findOne({name: "Tom"})
2    {
3    _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
4    name: 'Tom',
5    age: 28,
6    languages: [ 'english', 'spanish' ]
7    }
8    test>
```

Cursors

The result of the sample obtained using the find function is called a cursor. If necessary, we can transfer the cursor to a separate variable:

```
1    var cursor = db.users.find()
```

Cursors encapsulate sets of objects obtained from the database. Using JavaScript language syntax and cursor methods, we can display the received documents on the screen and somehow process them. Example:

```
1    var cursor = db.users.find()
2
3    while(cursor.hasNext()){
4    obj = cursor.next();
5    print(obj["name"]);
6    }
```

The cursor has a hasNext method, which shows when iterating whether there is another document in the set. And the next method extracts the current document and moves the cursor to the next document of the set. The result of the obj variable is a document whose fields we can access.

```
test> var cursor = db.users.find()


test> while(cursor.hasNext()){
... obj = cursor.next();
... print(obj["name"]);
...}
volume
Bob
Sam


test>
```

Also, to sort through documents in the cursor, as an alternative, we can use the javascript iterator construct - forEach :

```
1    var cursor = db.users.find()
2    cursor.forEach(function(obj){
3    print(obj.name);
4    })
```

Pagination and sorting

MongoDB provides a number of features that help manage database retrieval. One of them is the limit function. It sets the maximum allowable number of received

documents. The quantity is passed as a numeric parameter. For example, let's limit the sample to three documents:

```
1    db.users.find().limit(3)
```

In this case, we will receive the first three documents (if there are 3 or more documents in the collection). But what if we want to make a sample not from the beginning, but by omitting some number of documents? The skip function will help us in this. For example, let's skip the first three entries:

```
1    db.users.find().skip(3)
```

By combining both functions, we can retrieve a certain number of documents starting from a certain document. For example, let's select documents from 4 to 6:

```
1    db.users.find().skip(3).limit(3)
```

MongoDB provides the ability to sort the data set received from the database using the sort function. By passing the value 1 or -1 to this function, we can specify in which order to sort: ascending (1) or descending (-1). In many ways, this function is similar to the ORDER BYSQL expression. For example, sorting in ascending order by the name field:

```
1    db.users.find().sort({name: 1})
```

For example, let's extract from the database only the values of the "name" field, sorting them by growth:

```
test> db.users.find({}, {name:1, _id: 0}).sort({name: 1})
[
{ name: 'Bill'},
{ name: 'Bob' },
{ name: 'Sam' },
{ name: 'Tim' },
{ name: 'Tom' },
{ name: 'Tom' },
{ name: 'Tom' },
{ name: 'Tom' },
{name: 'Tomas'}
]
test>
```

$ slice operator

$slice is to some extent a combination of limit and skip functions. But, unlike them, $slice can work with arrays.

The $slice operator has two forms:

```
1    $slice: limit
2    $slice: skip, limit
```

The limit parameter indicates the total number of returned documents. The skip parameter indicates elimination relative to the start (like the skip function).

For example, each document defines an array of languages to store the languages spoken by a person. There can be 1, 2, 3 or more of them. And suppose we previously added the following object:

```
1    db.users.insertOne({"name":  "Tom",  "age":  32, languages: ["english",
     "german", "spanish"]})
```

And when displaying documents, we want to make sure that only one language from the languages array is included in the sample, and not the entire array:

```
1    db.users.find ({name: "Tom"}, {languages: {$slice : 1}})
```

This request will leave only the first language from the languages array as a result when extracting the document, that is, in this case, English.

```
test> db.users.find ({name: "Tom"}, {languages: {$slice : 1}})
[
{
_id: ObjectId("62e3c70079a0a7792a9de20c"),
name: 'Tom',
age: 32,
languages: [ 'english' ]
}
]
test>
```

Back: we also need to leave one element in the array, but not from the beginning, but from the end. In this case, it is necessary to pass a negative value to the parameter:

```
1    db.users.find ({name: "Tom"}, {languages: {$slice : -1}});
```

Now the array will show "spanish" because it is the first from the end in the added element.

We use two parameters at once:

```
1    db.users.find ({name: "Tom"}, {languages: {$slice : [-2, 1]}});
```

The first parameter says to start sampling elements from the end (since the value is negative), that is, sampling starts from the second element from the end, and the

second parameter indicates the number of elements of the returned array. As a result, "german" will appear in the language array

Indexes

When searching for documents in small collections, we do not experience any particular problems. However, when collections contain millions of documents, and we need to make a selection by a certain field, then searching for the necessary data can take some time, which can be critical for our task. And here indexes can help us. Indexes allow you to organize data about a certain field, which will subsequently speed up the search. For example, if in our application or task we usually search by the name field, then we can index the collection by this field.

Creating an index

To create an index, the createIndex() function is used, to which an object is passed specifying the fields for which the index is created. For example, creating an index on the "name" field:

```
1    db.users.createIndex({"name" : 1})
```

When creating an index, the console will return the name of the index:

```
test> db.users.createIndex({"name" : 1})
name_1
test>
```

That is, in the example above, an index with the name "name_1" was created on the name field. MongoDB allows for up to 64 indexes per collection.

To create several indexes, the createIndexes() function is used - an array of objects that set fields for indexes is passed to it:

```
1    db.users.createIndexes([{"name" : 1}, {"age": 1}])
```

In this case, two indexes are created - one for the name field, the other for the age field.

Deleting indexes

To remove indexes, the dropIndex() function is used, to which the name of the index is passed. For example, let's delete a certain index "name_1" above:

```
1    db.users.dropIndex("name_1")
```

Index settings

If we simply define an index for the collection, for example db.users.createIndex({"name" : 1}) we can still add documents with the same name key value to the collection. However, if we want a document with the same key value to be added to the collection only once, we can set the unique flag:

```
1    db.users.createIndex({"name" : 1}, {"unique" :true})
```

At the same time, when adding a unique index, there should be no documents in the collection that have the same values for a certain field.

Now, if we try to add two documents with the same name value to the collection, we will get an error.

At the same time, there are subtleties here. Yes, a document can have a name key. In this case, a null name key is automatically created for the attached document. Therefore, when adding a second document in which the name key is not defined, an exception will be thrown because the name key with the null value is already present in the collection.

You can also set one index for two fields at once:

```
1    db.users.createIndex({"name" : 1, "age" : 1})
```

However, in this case, all attached documents must have unique values for both fields.

In addition, there are limitations. For example, the value of the indexed field must be greater than 1024 bytes.

Management of indices

All database indexes are stored in the system collection indexes . To access it, we can use the getIndexes function, for example, to display all information about the indexes for a specific collection:

```
1    db.users.getIndexes()
```

This command will return output like the following:

```
test> db.users.getIndexes()
[
{ v: 2, key: { _id: 1 }, name: '_id_' },
{ v: 2, key: { name: 1 }, name: 'name_1' }
]
```

```
test>
```

As you can see, 2 indexes are defined here for the users collection (from the test database): id and name. The key field is used to find the maximum and minimum values, for various operations where this index should be used. The name field is used as an identifier for administration operations, such as deleting an index.

Aggregate functions

The number of items in the collection

Using the countDocuments() function, you can get the total number of documents in the collection:

```
1    db.users.countDocuments()
```

If we need to know not the total number of documents in the collection, but only the number of documents in a specific sample, we can use the count() function. For example, let's count the number of documents that have name=Tom:

```
1    db.users.find({name: "Tom"}).count()
```

Moreover, we can create function chains to specify the counting conditions:

```
1    db.users.find({name: "Tom"}).skip(2).count(true)
```

It should be noted here that by default the count function is not used with the limit and skip functions. To use them, as in the example above, you need to pass the Boolean value true to the count function

The distinct function

A collection can contain documents that contain the same values for one or more fields. For example, name: "Tom" is defined in several documents. And we need to find only unique values for one of the fields of the document. For this we can use the distinct function. For example, let the following documents be added to the database:

```
1    db.users.insertOne({"name":  "Tom",  "age":  38,  languages:  ["english",
     "spanish"]})
2    db.users.insertOne({"name":  "Bob",  "age":  41,  languages:  ["english",
     "french"]})
3    db.users.insertOne({"name": "Sam", "age": 28, languages: ["english"]})
4    db.users.insertOne({"name":  "Tom",  "age":  22,  languages:  ["english",
     "german"]})
```

Let's display all unique values for the "name" field:

```
1   test> db.users.distinct("name")
2   [ 'Bob', 'Sam', 'Tom' ]
3   test>
```

Min and max functions

The min and max functions set the minimum value for a certain field to be included in the sample. At the same time, these functions can use only those fields for which indexes are set. For example, let's take the db.users collection advanced above and define an index for the age field in it:

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

When executing the function, you must also use the hint() function, to which the index is passed. For example, let's select all documents in which the age field is greater than 30:

```
test> db.users.find().min({age:30}).hint({age:1})
[
{
_id: ObjectId("62e3d63a79a0a7792a9de210"),
name: 'Tom',
age: 38,
languages: [ 'english', 'spanish' ]
},
{
_id: ObjectId("62e3d63a79a0a7792a9de211"),
name: 'Bob',
age: 41,
languages: [ 'english', 'french' ]
}
]
test>
```

The max() function, which sets the maximum value, works similarly. For example, let's select documents where age is less than 30:

```
test> db.users.find().max({age:30}).hint({age:1})
[
{
_id: ObjectId("62e3d64079a0a7792a9de213"),
name: 'Tom',
age: 22,
languages: [ 'english', 'german' ]
```

```
},
{
_id: ObjectId("62e3d63a79a0a7792a9de212"),
name: 'Sam',
age: 28,
languages: [ 'english' ]
}
]
test>
```

Sampling operators

Conditional operators

Conditional operators specify a condition to which the value of a document field must comply:

- **$eq**(exactly)
- **$no**(not exactly)
- **$gt**(more than)
- **$lt**(less than)
- **$gte**(more or one)
- **$lte**(less than one)
- **$in**defines an array of values, one of which must be a document field
- **$nin**defines an array of values that a document field should have

For example, we will find all documents in which the value of the age key is less than 30:

```
1   db.users.find ({age: {$lt : 30}})
```

The use of other comparison operators will be similar. For example, the same key, only greater than 30:

```
1   db.users.find ({age: {$gt : 30}})
```

Note that the comparison here is on whole types, not strings. If the age key is a string value, then a comparison should be made over the lines: db.users.find ({age: {$gt : "30"}}), but the result will be the same.

But let's imagine a situation when we need to find all objects with the value of the age field greater than 30, but less than 50. In this case, we can combine two operators:

```
1    db.users.find ({age: {$gt : 30, $lt: 50}})
```

Let's find users whose age is 22:

```
1    db.users.find ({age: {$eq : 22}})
```

In essence, this is an analogy to the following query:

```
1    db.users.find ({age: 22})
```

The reverse operation - we will find users whose age is not equal to 22:

```
1    db.users.find ({age: {$ne : 22}})
```

The $in operator defines an array of possible expressions and searches for keys whose values are in the array:

```
1    db.users.find ({age: {$in : [22, 32]}})
```

The $nin operator works in the opposite way - it defines an array of possible expressions and looks for keys whose values are missing from this array:

```
1    db.users.find ({age: {$nin : [22, 32]}})
```

Logical operators

Logical operators are executed according to the sampling conditions:

- **$or**: connects two conditions, and the document must match one of these conditions
- **$and**: joins two conditions, and the document must match both conditions
- **$not**: the document must NOT meet the condition
- **$nor**: joins two conditions, and the document must not match both conditions

The $or operator

The $or operator represents a logical OR operation and defines a set of key-value pairs that should be in the document. And if the document has at least one such key-value pair, it corresponds to this request and is extracted from the database:

```
1    db.users.find ({$or : [{name: "Tom"}, {age: 22}]})
```

This expression will return us all documents that have either name=Tom or age=22. Another example will return us all documents in which name=Tom, and age is either 22, or among the languages values is "german":

```
1    db.users.find ({name: "Tom", $or : [{age: 22}, {languages: "german"}]})
```

Conditional operators can be used in or subexpressions:

```
1   db.users.find ({$or : [{name: "Tom"}, {age: {$gte:30}}]})
```

In this case, we select all documents where name="Tom" or the age field has a value between 30 and from.

The $and operator

The $and operator represents a logical AND operation (logical multiplication) and defines a set of criteria that the document must meet. Unlike the $or operator, the document must meet all of the specified criteria. Example:

```
1   db.users.find ({$and : [{name: "Tom"}, {age: 22}]})
```

Here, the selected documents must have the name Tom and the age 22 - both of these characteristics.

Array search

A number of operators are designed to work with arrays:

- **$all**: specifies the set of values to be in the array
- **$size**: specifies the number of elements to be in the array
- **$elemMatch**: specifies the condition that the elements in the array must meet

$all

The $all operator defines an array of possible expressions and requires documents to have the entire set of expressions defined. Accordingly, it is used for array searches. For example, in documents there is an array languages that stores foreign languages spoken by the user. And to find all the people who speak both English and French at the same time, we can use the following expression:

```
1   db.users.find ({languages: {$all : ["english", "french"]}})
```

$ elemMatch operator

The $elemMatch operator allows you to select documents in which arrays contain elements that meet certain conditions. For example, suppose a database contains a collection that contains user ratings for certain courses. Let's add some documents:

```
1   db.grades.insertMany([{student: "Tom", courses:[{name: "Java", grade: 5},
    {name: "MongoDB", grade: 4}]},
2   {student: "Alice", courses:[{name: "C++", grade: 3}, {name: "MongoDB",
    grade: 5}]}])
```

Each document has an array of courses, which in turn consists of nested documents.

Now let's find students who have a score higher than 4 for the MongoDB course:

```
1   db.grades.find({courses: {$elemMatch: {name: "MongoDB", grade: {$gt: 4}}}})
```

$ size operator

The $size operator is used to find documents in which arrays have the number of elements equal to the $size value. For example, let's extract all documents in which there are two elements in the laguages array:

```
1   db.users.find ({languages: {$size:2}})
```

Such a request will correspond, for example, to the following document:

```
1   {"name": "Tom", "age": 32, languages: ["english", "german"]}
```

The $exists operator

The $exists operator allows you to retrieve only documents in which a certain key is present or absent. For example, let's return all documents that have the company key:

```
1   db.users.find ({company: {$exists:true}})
```

If we specify $exists as a false parameter in the operator, the query will return us only documents in which the company key is not defined.

The $type operator

The $type operator retrieves only documents in which a given key has a value of a given type, such as string or number:

```
1   db.users.find ({age: {$type:"string"}})
2   db.users.find ({age: {$type:"number"}})
```

The $regex operator

The $regex operator specifies a regular expression that the field value must match. For example, let the name field necessarily have the letter "b":

```
1   db.users.find ({name: {$regex:"b"}})
```

It is important to understand that $regex accepts not simple strings, but regular expressions, for example: name: {$regex:"om$"}- the name value must end with "om".

Updating data

Like other database management systems, MongoDB provides the ability to update data. There are a number of functions for this.

replaceOne

If we need to completely replace one document with another, the replaceOne function can also be used:

```
1   db.collection.replaceOne(filter, update, options)
```

- filter: accepts a request to fetch the document to be updated
- update: represents a new document that will replace the old one when updated
- options: defines additional parameters when updating documents, the main one of which is the upsert parameter.

    If the upsert parameter is true, mongodb will update the document if it is found and create a new one if there is no such document. If it is false, then mongodb will not create a new document if the fetch query does not find any documents.

Example:

```
1   db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
```

In this case, we find the document in which name = "Bob" and replace it with the document {name: "Bob", age: 25}

After the operation, the console will return the result of the update:

```
test> db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
{
acknowledged: true,
insertedId: null,
matchedCount: 1,
modifiedCount: 1,
upsertedCount: 0
}
```

In the obtained result, the matchedCount parameter indicates the number of documents that match the query. And the modifiedCount parameter indicates the number of modified documents. That is, in this case, 1 document corresponds to the request and it has been changed.

updateOne and updateMany

Often it is not necessary to update the entire document, but only the value of one or more of its properties. For this, the functions updateOne() (updates only one document) and updateMany() (updates many documents) are used.

To update individual fields in these functions, the $set operator is used. If the document does not contain an updateable field, one is created.

```
1    db.users.updateOne({name : "Tom", age: 22}, {$set: {age : 28}})
```

Here we search for a document with name="Tom" and age=22 and set its age property to 28

If it is necessary to update all documents that meet a certain criterion, the updateMany() function is used:

```
1    db.users.updateMany({name : "Tom"}, {$set: {name : "Tomas"}})
```

If there is no updateable field in the document, it is added:

```
1    db.users.updateOne({name : "Tom", age: 28}, {$set: {salary : 300}})
```

If you need to update the values of several fields, they are passed to the $set operator through a comma:

```
1    db.users.updateOne({name : "Tom"}, {$set: {name: "Tomas", age : 25}})
```

To simply increase the value of a numeric field by a certain number of units, the $inc operator is used. If the document does not contain an updateable field, one is created. This operator applies only to numeric values.

```
1    db.users.updateOne({name : "Tom"}, {$inc: {age:2}})
```

Delete a field

To delete an individual key, the $unset operator is used:

```
1    db.users.updateOne({name : "Tom"}, {$unset: {salary: 1}})
```

If suddenly such a key does not exist in the document, then the operator has no influence. You can also delete several fields at once:

```
1    db.users.updateOne({name : "Tom"}, {$unset: {salary: 1, age: 1}})
```

Update arrays

The $push operator

The $push operator allows you to add another value to an existing one. For example, if the key as value stores an array:

```
1    db.users.updateOne({name : "Tom"}, {$push: {languages: "russian"}})
```

The updateOne function was used above, but this operator also applies to the updateMany function

```
1    db.users.updateMany({name : "Tom"}, {$push: {languages: "russian"}})
```

If the key for which we want to add a value is not an array, then we will get the error Cannot apply $push/$pushAll modifier to non-array.

Using the $each operator, you can add several values at once:

```
1   db.users.updateOne({name : "Tom"}, {$push: {languages: {$each: ["russian",
    "spanish", "italian"]}}})
```

A few more operators allow you to customize the insert. The $position operator specifies the position of the array to insert elements, and the $slice operator specifies how many elements to leave in the array after the insertion.

```
1   db.users.updateOne({name : "Tom"}, {$push: {languages: {$each: ["german",
    "spanish", "italian"], $position:1, $slice:5}} })
```

In this case, the elements ["german", "spanish", "italian"] will be inserted into the languages array from the 1st index, and after the insertion, only the first 5 elements will remain in the array.

$addToSet statement

The $addToSet operator, like the $push operator, adds objects to the array. The difference is that $addToSet adds data if it is not already in the $push array.

```
1   db.users.updateOne({name : "Tom"}, {$addToSet: {languages: "russian"}})
```

Removing an element from an array

The $pop operator allows you to remove an element from an array:

```
db.users.updateOne({name : "Tom"}, {$pop: {languages: 1}})
```

By setting the languages key to 1, we remove the first element from the end. To remove the first element from the beginning of the array, you need to pass a negative value:

```
1   db.users.updateOne({name : "Tom"}, {$pop: {languages: -1}})
```

A slightly different action is provided by the $pull operator. It removes each occurrence of an element in the array. For example, through the $push operator, we can add the same value to the array multiple times. And now with the help of $pull we will remove it:

```
1   db.users.updateOne({name : "Tom"}, {$pull: {languages: "english"}})
```

And if we want to remove not one value, but several at once, then we can apply the $pullAll operator:

```
1   db.users.updateOne({name : "Tom"}, {$pullAll: {languages: ["english",
    "german", "french"]}})
```

Deleting data

To delete documents in MongoDB, functions deleteOne() - deletes one document and deleteMany() - allows you to delete several documents are provided. The filter of the documents to be deleted is passed as a parameter to these functions.

For example, let's delete a document in which name="Tom":

```
1    db.users.deleteOne({name : "Tom"})
```

As a result, the first document found with name=Tom will be deleted. After deletion, the console displays an object in which the deletedCount parameter indicates the number of deleted documents:

```
test> db.users.deleteOne({name: "Tom"})
{ acknowledged: true, deletedCount: 1 }
```

To delete all documents that match the filter, the deleteMany() function is used:

```
1    db.users.deleteMany({name : "Tom"})
```

Moreover, as in the case of find, we can specify the selection conditions for deletion in different ways (in the form of regular expressions, in the form of conditional constructions, etc.):

```
1    db.users.deleteOne({name : /^T\w+/i})
2    db.users.deleteOne({age: {$lt : 30}})
```

To delete all documents from the collection together, leave the query parameter empty:

```
1    db.users.deleteMany({})
```

Deleting collections and databases

We can delete not only documents, but also collections and databases. The drop function is used to delete collections:

```
1    db.users.drop()
```

And if the removal of the collection is successful, the console will output:

```
1    true
```

To drop the entire database, use the dropDatabase() function:

```
1    db.dropDatabase()
```

Setting links in the database

In relational databases, you can set foreign keys when fields from one table refer to fields in another table. You can also set links in MongoDB. Next, we'll look at

establishing links between the users collection, which stores users, and the companies collection, which stores a list of companies where users work.

Manual installation of links

Manual setting of links is reduced to determining the value of the _id field of one document to the field of another document. Let's say we might have collections that represent companies and the employees who work for those companies. So, first, let's add a document representing the company to the collection of companies:

```
1    db.companies.insertOne({"_id" : "microsoft", year: 1974})
```

Now let's add a document that represents an employee to the user collection. This document will have a company field representing the company where the employee works. And it is very important that we set the value for this field not to the company object, but to the value of the _id key of the document added above:

```
1    db.users.insertOne({name: "Tom", age: 28, company: "microsoft"})
```

Now let's get a document from the users collection:

```
1    user = db.users.findOne()
```

In this case, it is implied that the added element will be the only one in the collection. Otherwise, it is necessary to select the _id of the last added document.

After that, the console displays the received document. And now we will find a link to his company in companies:

```
1    db.companies.findOne({_id: user.company})
```

And if a document with this ID is found, it is displayed on the console:

```
test> db.companies.insertOne({"_id" : "microsoft", year: 1974})
{ acknowledged: true, insertedId: 'microsoft' }
test> db.users.insertOne({name: "Tom", age: 28, company: "microsoft"})
{
acknowledged: true,
insertedId: ObjectId("62e427069fcbf14521ea012a")
}
test> user = db.users.findOne()
{
_id: ObjectId("62e427069fcbf14521ea012a"),
name: 'Tom',
age: 28,
company: 'microsoft'
}
test> db.companies.findOne({_id: user.company})
```

```
{ _id: 'microsoft', year: 1974}
test>
```

Automatic binding

Using DBRef functionality, we can establish automatic linking between documents. Let's look at an example of using this functionality. First, let's add a new document to the company collection:

```
1   google = db.companies.insertOne({name : "google", year: 1998})
```

Note that we get the result of adding to the variable google . When a new document is added, it generates an _id, which we can get using the insertedId property of the function result.

Now let's create a new document for the users collection, in which we will associate the company key with the apple document we just added:

```
1   sam = ({name: "Sam", age: 25, company: { "$ref" : "companies", "$id" :
    google.insertedId}})
2   db.users.insertOne(sam)
```

And we can protest:

```
1   db.companies.findOne({_id: sam.company.$id})
```

```
test> google = db.companies.insertOne({name : "google", year: 1998})
{
acknowledged: true,
insertedId: ObjectId("62e427c29fcbf14521ea012b")
}
test> sam = ({name: "Sam", age: 25, company: { "$ref" : "companies", "$id" :
google.insertedId}})
{
name: 'Sam',
age: 25,
company: { '$ref': 'companies', '$id': ObjectId("62e427c29fcbf14521ea012b") }
}
test> db.users.insertOne(sam)
{
acknowledged: true,
insertedId: ObjectId("62e427dc9fcbf14521ea012c")
}
test> db.companies.findOne({_id: sam.company.$id})
{
_id: ObjectId("62e427c29fcbf14521ea012b"),
name: 'google',
```

```
Year: 1998
}
test>
```

Having looked at an example, now we will analyze the organization of links between documents. The following expression { "$ref" : "companies", "$id" : google.insertedId} was used to link to the google document. The formal DBRef syntax is as follows:

```
1    { "$ref" : collection_name, "$id": value [, "$db" : database_name ]}
```

The first parameter $ref points to the collection where the linked document is stored. The second parameter specifies a value that will represent something like a foreign key. The third optional parameter specifies the database.

When testing, the expression _id: sam.company.$id is specified as a sampling request. Since sam.company now represents an object { "$ref" : "companies", "$id" : google.insertedId}, we need to specify the parameter sam.company.$id

Collection management

Explicit collection creation

In previous themes, a collection was implicitly created automatically when the first data was added to it. But we can also create it explicitly by using the db.createCollection(name, options) method, where name is the name of the collection and options is an optional object with additional initialization settings. Example:

```
1    db.createCollection("accounts")
2    {"ok" : 1}
```

Thus, a collection of accounts is created.

Renaming the collection

You may need to change the name of the collection as you work. For example, if there was an error in its name when you first added data. And in order not to delete and then transform the collection, you should use the renameCollection function:

```
1    db.users.renameCollection("new_name")
```

And if the renaming is successful, the console will display:

```
{"ok" : 1}
```

Limited collections

When we query the database for a sample, MongoDB returns the documents to us, usually in the order in which they were added. However, this order is not always guaranteed, as data can be deleted, moved, changed. Therefore, MongoDB has the concept of a limited collection (capped collection). A similar collection ensures that the documents will be arranged in the same order in which they were added to the collection. Limited collections have a fixed size. And when there is no more space in the collection, the oldest documents are deleted, and new data is added at the end. Unlike regular collections, we can supply limited ones explicitly. For example, let's create a limited collection named profiles and set its size to 9500 bytes:

```
1   db.createCollection("profiles", {capped:true, size:9500})
```

And after the collection is successfully created, the console will output:

```
{"ok":1}
```

You can also limit the number of documents in the collection by specifying it in the max parameter:

```
1   > db.createCollection("profiles", {capped:true, size:9500, max: 150})
```

However, with this method of creating a collection, it should be taken into account that if the entire space for the collection is full (for example, we allocated 9500 bytes), and the number of documents has not yet reached the maximum, in this case 150, then when adding a new document, the oldest document will be deleted, and on a new document will be inserted in its place.

When updating documents in such collections, you should remember that the documents should not grow in size, otherwise the update will not be possible.

You also cannot delete documents from such collections, you can only delete the entire collection.

Subcollections

To simplify the organization of data in collections, we can use subcollections. For example, data from the users collection should be separated into profiles and credentials. And we can use to create db.users.profiles and db.users.accounts collection. At the same time, they will not be associated with the users collection.

That is, as a result, there will be three different collections, but in terms of logical organization of data storage, it may be easier for someone.