Name: Anushka M Mulay

Class: TE IT (A)

Roll No: 35040

Batch: B

Group: C

**Assignment No. 3**

**Title:** $where, cursors, database commands.

**Problem Statement:** Execute at least 10 queries on any suitable MongoDB database that demonstrates following:

* $ where queries
* Cursors (Limits, skips, sorts, advanced query options)
* Database commands

**Requirements:** MongoDB

**Prerequisites:** Basic Knowledge about MongoDB.

**Theory:**

1. **$ where queries**

Key/value pairs are a fairly expressive way to query, but there are some queries that they

cannot represent. For queries that cannot be done any other way, there are "$where"

clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows

you to do (almost) anything within a query. For security, use of "$where" clauses should

be highly restricted or eliminated. End users should never be allowed to execute arbi‐

trary "$where" clauses.

The most common case for using "$where" is to compare the values for two keys in a

document.

1. **Cursors**

The database returns results from find using a cursor. The client-side implementations

of cursors generally allow you to control a great deal about the eventual output of a

query. You can limit the number of results, skip over some number of results, sort results

by any combination of keys in any direction, and perform a number of other powerful

operations.

To create a cursor with the shell, put some documents into a collection, do a query on

them, and assign the results to a local variable (variables defined with "var" are local).

**Limits, Skips, and Sorts :**The most common query options are limiting the number of results returned, skipping

a number of results, and sorting. All these options must be added before a query is sent

to the database.

**Limit:**

To set a limit, chain the limit function onto your call to find. For example, to only

return three results, use this:

> db.c.find().limit(3)

If there are fewer than three documents matching your query in the collection, only the

number of matching documents will be returned; limit sets an upper limit, not a lower

limit.

**Skip**:

skip works similarly to limit:

> db.c.find().skip(3)

This will skip the first three matching documents and return the rest of the matches. If

there are fewer than three documents in your collection, it will not return any

documents.

**Sort:**

sort takes an object: a set of key/value pairs where the keys are key names and the values

are the sort directions. Sort direction can be 1 (ascending) or −1 (descending). If multiple

keys are given, the results will be sorted in that order. For instance, to sort the results

by "username" ascending and "age" descending, we do the following:

> db.c.find().sort({username : 1, age : -1})

These three methods can be combined. This is often handy for pagination. For example,

suppose that you are running an online store and someone searches for mp3. If you

want 50 results per page sorted by price from high to low, you can do the following:

> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})

If that person clicks Next Page to see more results, you can simply add a skip to the

query, which will skip over the first 50 matches (which the user already saw on page 1):

> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})

However, large skips are not very performant; there are suggestions for how to avoid

them in the next section.

**Advanced Query Options**

There are two types of queries: wrapped and plain. A plain query is something like this:

> var cursor = db.foo.find({"foo" : "bar"})

There are a couple options that “wrap” the query. For example, suppose we perform a

sort:

> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})

Instead of sending {"foo" : "bar"} to the database as the query, the query gets wrapped

in a larger document. The shell converts the query from {"foo" : "bar"} to

{"$query" : {"foo" : "bar"}, "$orderby" : {"x" : 1}}.

1. **Database commands**

There is one very special type of query called a database command. We’ve covered

creating, updating, deleting, and finding documents. Database commands do “every‐

thing else,” from administrative tasks like shutting down the server and cloning data‐

bases to counting documents in a collection and performing aggregations.

Commands are mentioned throughout this text, as they are useful for data manipula‐

tion, administration, and monitoring. For example, dropping a collection is done via

the "drop" database command:

> db.runCommand({"drop" : "test"});

{

"nIndexesWas" : 1,

"msg" : "indexes dropped for collection",

"ns" : "test.test",

"ok" : true

}

You might be more familiar with the shell helper, which wraps the command and pro‐

vides a simpler interface:

> db.test.drop()

Often you can just use the shell helpers, but knowing the underlying commands can be

helpful if you’re stuck on a box with an old version of the shell and connected to a new

version of the database: the shell might not have the wrappers for new database com‐

mands, but you can still run them with runCommand().

We’ve already seen a couple of commands in the previous chapters; for instance, we

used the getLastError command in Chapter 3 to check the number of documents

affected by an update:

> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)

> db.runCommand({getLastError : 1})

{

"err" : null,

"updatedExisting" : true,

"n" : 5,

"ok" : true

}

**Conclusion:** Thus, in this assignment we studied about $where queries, cursors and various database commands.