**Week 2:** **Crud Opetions**

**Coverage : CRUD operations from Mongo Shell, same operation with perspective of Java Driver.**

Crud stands for create, read , update and delete, is a common acronym for describing the kind of manipulation that can be performed on database Records.

In Mongo-DB terminalogy,

|  |  |  |
| --- | --- | --- |
| Operation | Sql Terminology | Mongo DB Terminoly |
| Create | Insert | Insert |
| Read | select | Find |
| Update | update | Update |
| Delete | delete | Remove |

-> One thing that is important to know about MongoDB is that CRUD operations and indeed all the operations, that manipulate data in the Database, exists as methods/functions in programming language API’s

-> Mongo DB does not have its own query language analogous to the SQL language that need to embedded as a String in a Program.

-> Instead, MongoDB has a wired protocol, which is relatively simple that encorporates a operation code and arguments to that operation, transmitted a squence of bytes over the network.

-> From a programmers perspective, we instead manipulate the data in MongoDB using methods on objects or function and those methods transfer data over the network according to wire protocol.

-> The actual coding for CRUD operation in MongoDB is relatively plesent for programmers, precisely because we don’t have to manipulate a Strange language i.e sql in coustomary DB.

1. **SECRETS OF THE MONGO SHELL**

Mongo Shell is a interactive javascript interpriter, that has built in functionality for connecting to and manupulating Mongo DB database.

Assuming ,we are already started the mongo Server, then to start a shell, we have just run “mongo” in the bin folder . When the shell starts, it prints out a banner that includes the shell version and database it connected.

Usually, any version of shell can support any version of mongo DB server. But every version introduce new features in shell or Db, sometimes to support these changes we may need hand shake between shell and server.

Mongo Shell is a inteactive Java scripts interpriter, which means we can write JavaScript code and shell will interpret or evaluate it immediately.

To get the previous input in Sheel, use Up arrow. Ctrl + a , moves to the begging of the line or simply use arrows to move. Ctrl+e for end of line.

**Helper :**  Shell has helper. If we type help, we can see a list of topics, we can visit.

For example, help keys then we see information abouts keys binding that are available.

The shell has the capability to complete the tokens of input i.e method of a JavaScript.

**Understanding JavaScript :**

In JavaScript, variable are assigned

🡪 x = 1 // this will assgine 1 to variable x.

🡪 y = “abc” // this will assgine abc to variable y

🡪 z = {“a” : “1”} // this will assigne a java script object with a property “a” and having a value “1”. This is something like a disctionary, hashtable in some other languages.

We can refer to properties in 2 different ways.

1. **dot notations:** z.a , will retrieve the property value of “a” of variable z.
2. **Alternate Notation :** **Z[“a”] :: z[string]**

Both syntaxes has similar purposes but have some differences. In particular, the dot notation z.a , does not permit “variable property Lookup”. “a” is treated as literal and z has a variable.

Where as in alrernate notation, w = “a” and z[w] will give the value of property a.

**BSON Introdution**

Before we go deep into, how to do things in Mongo DB using mongo shell, we have to talk, how mongo DB actually represents/stores data and how the mongo shell interprits the data, when it get back from the DB.

As we already know, mongo DB is a document oriented DB, what this actually mean is that MongoDB is fundamentally record type is a kind of nested dictionary of key-value associations. As a matter of programming language convience, we map the documents come out of Mongo DB to that of objects in the programming language, that can represent these kind of key-value associations.

In JavaScript, that represent these kind of associations is called as JavaScript Object.

For example: if I define

Obj = {“a” : 1 , “b” : “hello” , “c” : [“apples”,”tamatoes”] }

This object of javascipt is usable as mongo db document. Bu this, we use this object and keep in the DB. Though JavaScript used the syntax, that is much similar to JSON(which is inspired by JavaScript represetations) .

MongoDB internally doesn’t uses this string JSON syntax, for storing and retrieveing of Data. Instead, MongoDB uses a binary notations representation for the data inside the documents.

The specification for the binary representation that MongoDB use , is give at bsonspec.org . We call the binary format as bson, which stands for binary Json.

And as a fact, this is the serialisation format, that is designed to represent a superset, that can be transcribed in the Json syntax.

The basic types of JSON includes, double precision floating point number, string, embeded documents, embeded arrays(e.e nexted objects, whose keys are the integer 0, 1 etc. ), bollean value false and true, and the null value.

But BSON goes further, by adding binary, object\_id(used for unique identification of thing.), DateTIme(which represent time since Jan 01 1970), 32 bit integer, 64 bit integer type and few other types which are used for intenal purpose.

These data types, which goes beyond the JSON specifications, are correspond to types that available in various programing language or else have some sort of special handling in the programming languages. Below example explain the special handling mean.

For example : In the shell, we use the JavaScript, JS languae has support for numbers which are interpretted a double flaot, Strings which are represented as UTF-8 strings, arrays objects, true, falses and so on. To represent a integer value in the mongo shell(which is a JavaScript), the shell contains a constructor NumberInt(1), give a 32bit value. Where as NumberLong(1) gets a 64bit value and also these constructor supports arthematic on these values.

Same as above, ISODate contructor is for date, with date as a parameter.

Example: obj= {“a” : 1., b : ISODate(“2012-10-21T17:41:58.456Z”), “c” :NumberLong(42)}

In this way, the javascript shell, more or less fathfully can handle the all the data, that comes out of Database or inserts into DB. The arthematic operation that perfrom, may not preserve the type information, which may be changed to different type again.

Finally, MongoDB stores the Data as binary internally, which we called it as BSON. BSON representation supports the basic data types that MongoDB contains. BSON representation supports all the basic data types that JSON capable of and few more. Mongo Shell and other drivers that can talk to monfo DB supports the various datatypes that BSON supports, in some way or the other.

**Specification Version 1.0**

BSON is a binary format in which zero or more key/value pairs are stored as a single entity. We call this entity a *document*.

The following grammar specifies version 1.0 of the BSON standard. We've written the grammar using a pseudo-[BNF](http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form) syntax. Valid BSON data is represented by the document non-terminal.

**Basic Types**

The following basic types are used as terminals in the rest of the grammar. Each type must be serialized in little-endian format.

|  |  |
| --- | --- |
| byte | 1 byte (8-bits) |
| int32 | 4 bytes (32-bit signed integer, two's complement) |
| int64 | 8 bytes (64-bit signed integer, two's complement) |
| double | 8 bytes (64-bit IEEE 754-2008 binary floating point) |

**Non-terminals**

The following specifies the rest of the BSON grammar. Note that quoted strings represent terminals, and should be interpreted with C semantics (e.g. "\x01" represents the byte 0000 0001). Also note that we use the \* operator as shorthand for repetition (e.g. ("\x01"\*2) is "\x01\x01"). When used as a unary operator, \* means that the repetition can occur 0 or more times.

|  |  |  |  |
| --- | --- | --- | --- |
| document | ::= | int32 e\_list "\x00" | BSON Document. int32 is the total number of bytes comprising the document. |
| e\_list | ::= | element e\_list |  |
|  | | | "" |  |
| element | ::= | "\x01" e\_name double | 64-bit binary floating point |
|  | | | "\x02" e\_name string | UTF-8 string |
|  | | | "\x03" e\_name document | Embedded document |
|  | | | "\x04" e\_name document | Array |
|  | | | "\x05" e\_name binary | Binary data |
|  | | | "\x06" e\_name | Undefined (value) — *Deprecated* |
|  | | | "\x07" e\_name (byte\*12) | [ObjectId](http://dochub.mongodb.org/core/objectids) |
|  | | | "\x08" e\_name "\x00" | Boolean "false" |
|  | | | "\x08" e\_name "\x01" | Boolean "true" |
|  | | | "\x09" e\_name int64 | UTC datetime |
|  | | | "\x0A" e\_name | Null value |
|  | | | "\x0B" e\_name cstring cstring | Regular expression - The first cstring is the regex pattern, the second is the regex options string. Options are identified by characters, which must be stored in alphabetical order. Valid options are 'i' for case insensitive matching, 'm' for multiline matching, 'x' for verbose mode, 'l' to make \w, \W, etc. locale dependent, 's' for dotall mode ('.' matches everything), and 'u' to make \w, \W, etc. match unicode. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | "\x0C" e\_name string (byte\*12) | DBPointer — *Deprecated* |
|  | | | "\x0D" e\_name string | JavaScript code |
|  | | | "\x0E" e\_name string | *Deprecated* |
|  | | | "\x0F" e\_name code\_w\_s | JavaScript code w/ scope |
|  | | | "\x10" e\_name int32 | 32-bit integer |
|  | | | "\x11" e\_name int64 | Timestamp |
|  | | | "\x12" e\_name int64 | 64-bit integer |
|  | | | "\xFF" e\_name | Min key |
|  | | | "\x7F" e\_name | Max key |
| e\_name | ::= | cstring | Key name |
| string | ::= | int32 (byte\*) "\x00" | String - The int32 is the number bytes in the (byte\*) + 1 (for the trailing '\x00'). The (byte\*) is zero or more UTF-8 encoded characters. |
| cstring | ::= | (byte\*) "\x00" | Zero or more modified UTF-8 encoded characters followed by '\x00'. The (byte\*) MUST NOT contain '\x00', hence it is not full UTF-8. |
| binary | ::= | int32 subtype (byte\*) | Binary - The int32 is the number of bytes in the (byte\*). |
| subtype | ::= | "\x00" | Generic binary subtype |
|  | | | "\x01" | Function |
|  | | | "\x02" | Binary (Old) |
|  | | | "\x03" | UUID (Old) |
|  | | | "\x04" | UUID |
|  | | | "\x05" | MD5 |
|  | | | "\x80" | User defined |
| code\_w\_s | ::= | int32 string document | Code w/ scope |

**Notes**

* Array - The document for an array is a normal BSON document with integer values for the keys, starting with 0 and continuing sequentially. For example, the array ['red', 'blue'] would be encoded as the document {'0': 'red', '1': 'blue'}. The keys must be in ascending numerical order.
* UTC datetime - The int64 is UTC milliseconds since the Unix epoch.
* Timestamp - Special internal type used by MongoDB replication and sharding. First 4 bytes are an increment, second 4 are a timestamp.
* Min key - Special type which compares lower than all other possible BSON element values.
* Max key - Special type which compares higher than all other possible BSON element values.
* Generic binary subtype - This is the most commonly used binary subtype and should be the 'default' for drivers and tools.
* Code w/ scope - The int32 is the length in bytes of the entire code\_w\_s value. The string is JavaScript code. The document is a mapping from identifiers to values, representing the scope in which the string should be evaluated.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

* The BSON "binary" or "BinData" datatype is used to represent arrays of bytes. It is somewhat analogous to the Java notion of a ByteArray. BSON binary values have a *subtype*. This is used to indicate what kind of data is in the byte array. Subtypes from zero to 127 are predefined or reserved. Subtypes from 128-255 are user-defined.
  + \x02 Binary (Old) - This used to be the default subtype, but was deprecated in favor of \x00. Drivers and tools should be sure to handle \x02 appropriately. The structure of the binary data (the byte\* array in the binary non-terminal) must be an int32 followed by a (byte\*). The int32 is the number of bytes in the repetition.
  + \x03 UUID (Old) - This used to be the UUID subtype, but was deprecated in favor of \x04. Drivers and tools for languages with a native UUID type should handle \x03 appropriately.
  + \x80-\xFF "User defined" subtypes. The binary data can be anything.

**Inserting Documents**

So now that we know what a docuemt in mongoDB, lets find out how to actually put them into the database.

As we discussed, MongoDB objects are represents in JS language by JavaScript Objects. So we can create a variable in Javascript and call it doc

🡪 doc = {“name”:”pranith” , “age” : “30” , “profession”:”engineer” }

All this done is establish a binding between a variable in JS and a value.

To actually communicate with the database, we have to use a handle on the Database. As it happens, the shell has a varibale called “db”, whose value is a handle to the current database.

If you evalute the value of db, it prints the current database value.

🡪 db (enter)

Documents in a database, lives inside a collection, which is a simple sets of dcouments within a particular named DataBase.

From a programming language perspective in JavaScript shell, we represent collecion as a properties of a Database. So, if we type

🡪 db.people.insert

then the shell will interprit “prople” as name of a collections, inside the current DB. Insert is mentod on collection and it takes an argument, a JavaScript Object such variable doc(created above).

🡪 db.people.insert(doc)

Insert method puts the document into DB.

To Query a collection to fetch the data, we have to use find method. Find method without arguments returns all the documents in a collection.

🡪 db.people.find()

{“\_id”:ObjectID(“adsfasdfalk897658asd”),“name”:”pranith” , “age” : “30” , “profession”:”engineer” }

The output document has a extra field \_ID, whose value seems to be something called an ObjectID, consisting of sequence of Something. As it happens, when we insert the document into MongoDB, the server requires all the documents has a unique identifying field withing a collection. In face, we use the \_ID for this.

\_id in a documents, is a primary key field, that is the value of the field requies to be unique for that documents within a collection, and the primary is immutable i.e we cant change the value of \_ID field within a document

we could simulate the changing the ID by removing the document and putting the another documents in with a different value got the \_ID, which is not a atomic operation in withing the database.

All normal collections in MongoDB require primary key requirement as \_id. We will see abnormal collections later.

ObjectID --: In MongoDb, we need \_id to be unique within a collection. To facilitate this, Mongo shell provide a type i.e ObjectID, the contruction of which takes into account the time, identifier of the machine of the contructor, processID of the contructor, and a counter that’s global to the process that contructing objectID. All these value to gether has a high probability of being unique.

If the application is not filling a field \_id, then the objectID will be contructed for us, that will be used as \_id for the document. We can use any type of the value that we want for \_id i.e string or number, but if we don’t insert, it will take it up automatically .

**Introdution to findOne**

The basic operation to query object out of database is find. However, it is simpler to look into one document out of the database, for which we have findOne.

findOne when used without aguments, it will return one document in random, this will be helpful for examing the schema of the documents in our collections.

We can give arguments to findOne.

The first argument for findOne, specifies what criteria we are looking for to consider a document for the match. This is a alias to where clause in SQL. The argument which goes here is a documents.

🡪 db.people.findOne({“name”:”pranith”}) // This will return the document with name as jones.

This is the first example to query into mongoDB actually has any kind of restriction on it. As you can see, what documents count as a match is given itself as a documents. This is quite a variation to the string done in SQL. Instead of having String Based query language, we actually send the BSON encoding of this arguement to the findOne over the wire over the wire, wrapped up with some message headers as an operation that db has to execute i.e the query which is executed in the DB is also in structured document.

The second argument, we can specify the field that we want from the database. This is similar to column list in the selct query.

🡪 db.people.findOne({“name”:”pranith” } , {“name”:true ,“\_id”:false }) // this will give obect with name as pranith and output only name field.

🡪 db.people.findOne({“name”:”pranith” } , {“name”:true })

for this output, “\_id” field defaults to be present in the reponse from findOne, even if we don’t ask it. This is feature in MongoDB, that is added to support the Object Store, for which it is common to want to know the uniqure identifier of the object that you re retrieveing from the BD. But we can avoid by keeping it as false.

**Introduction into find.**

Lets learn some more interesting thing about querying.

🡪 Simple find method, will return all the documents in the collection.

🡪 in the shell, if the query is returning very many documents, they are retrieved in batches. There’s actually 2 layers of batching going on. But we can just think the batch size as 20.

🡪 type “it” to get next set of documents.

🡪 there is a cursor open untill we exhaust all the documents that will be iterated. In fact, the server will close the cursor after ten mn by default their would not be problem like exhauting server side resources.

**Argument for Find:**

🡪 first argument in the document that holds the condition.

db.people.find({“name”:”krishna”, “marks” : “50”}) // this will return the documents which have name value as krishna and marks as 50. Both condition should satisfy. We will have many type of condition that can be added to this like greater than e.t.c

🡪 second argument is also a document, which specify what fields should be retrieved from the database.

db.people.find({“name”:”krishna”, “marks” : “50”} , {“name” : true, “\_id”: false , “marks” : true})

**More Into Query Facilities (Query Operators).**

**$gt:** Greater Than

**$gte:** Greater Than equal

**$lt:** lesser Than

**$lte:** lesser Than equal

**single conditions :**

db.scores.find({“score” : { $gt : 95 }}); // subdocument with the operator and the value for the field “score”.

**Multiple condition on multiple Fileds:**

db.scores.find({“score” : { $gt : 95 } , “type” : “essay”}); // score greater than 95 and type is essay.

**Multiple condition on single Fileds:**

db.scores.find({“score” : { $gt : 95 , $lte : 99 } , “type” : “essay”}); // score great than 95 and less than equal to 99 and type is essay. It is logical AND.

**Inequality Comparision on Strings**

The inqueality operator that we have used on numeric value, also can be used on String valued comparisions.

🡪 db.people.find( { “name” : { $lt : “D” } } )

The query above will return the documents with name field is lexicographycally (**lexicographic** or **lexicographical order** (also known as **lexical order**, **dictionary order**, **alphabetical order** or **lexicographic(al) product**) is a generalization of the way the [alphabetical order](https://en.wikipedia.org/wiki/Alphabetical_order) of words is based on the alphabetical order of their component letters. )less than the capital letter D.

We can add some more condition like

🡪 db.people.find( { “name” : { $lt : “D” , $gt : “B”} } ) // this will return all documents which are less than “D” and greater than “B”. this is dictionary order. So reslut will include BA to cz.

A couple of notes on locale and locale-aware sorting. Right now, mongoDb has exactly zero knowledge of locales. In effect, the comparision we perform for $lt($gt and so on) are going to sort acording to the total ordering of the UTF-8 code units. i.e le **lexicographical** sorting of the bytes in UTF-8 representation of Strings. This happens to be correct only in the POSIX or C locales. That is mongo DB compares and sort things in a Ascii-batically correct fasion. It is so happens that there’s a location for which sorting things in UTF-8 byte order happens to be correct. Then mongoDb is also correct.

Future release of mongo will be supporting locale-aware sorting.

🡪 Mongo DB is schemaless, that means same field in the different documents can hold different types of value. i.e name in people collection can be inserted as a numberic value. The Db will not throw exception. This pattern is not recommended but nothing wrong in doing so, if needed.

All the operation in mongoDB are strongly typed and dynamicaly typed too. If we try to fetch the value using

db.people.find( { “name” : { $lt : “D” , $gt : “B”} } )

then the mongoDb will check only document, whose field “name” value is string and does not consider the documents whose value of the field “name” is numeric.

🡪 if we have to write a query to consider the both alphabets and numerics then we write a more complex query which we will see in future. As of now, the comparision are strongly typed.

🡪 MongoDb’s sorting and range operations are case-sensitive, in effect, and only compare in an Ascii sort.

**$exists :** if we want to fetch the document in which a particulat field exists or not, then we have $exists operator.

🡪 db.people.find( { “profession” : { $exists : true } } ) // this will return all documents that have a profession field.

🡪 db.people.find( { “profession” : { $exists : false } } ) // this will return all documents that does not have a profession field.

**$type :** this is used specify a condition on data type of the field.

🡪 db.people.find( { “name” : { $type : 2 } } ) // name filed is only string.

The types are specified as numbers out of BSON specification. Side to the each time, we have the numeric encodeing in the bsonspec.org. Use this to fetch.

**$regex :**  This is used to search for patterns. MongoDB support for pattern over string using a regular expression library, in particular, Perl compatible regular expression library (libpcre).

🡪 db.people.find( { “name” : { $regex : “a” } } ) // with letter a in the word.

🡪 db.people.find( { “name” : { $regex : “e$” } } ) //ends with letter a in the word.

🡪 db.people.find( { “name” : { $regex : “^A” } } ) //ends with letter A in the word.

Reqular expression not much to optimised,but we will check later this chapter.

**$or :** $or is a logical operator. It is a prefix operator and contain array of sub-documents which are each individual query.

db.people.find( { $or : [ { name : { $regex : “e$” } } , { age : { $exists : true } } ] } ) // this query will return the documents where name ends with e or it contains a age field .

If the systax of the json input for the query is wrong, then the shell will give (…) , which mean it is wrong. We have to enter twice to exit this.

**$and :**  $and is a logical operator. It is a prefix operator and contain array of sub-documents which are each individual query.

db.people.find( { $and : [ { name : {$gt : “C”}} , [ { name : {$regex : “a”}} ] } )

**is similar to**

db.people.find( { name : { $gt : “c” , $regex : “a”}} } } )

both are same, but they differ in the optimisation. Second is more optimised then the other.

What will the following query do?

db.scores.find( { score : { $gt : 50 }, score : { $lt : 60 } } );

Top of Form

Bottom of Form

1. Find all documents with score between 50 and 60
2. Find all documents with score greater than 50
3. Find all documents with score less than 60
4. Explode like the Death Star
5. None of the above

**Answer : C**

Reason : when parser parse will construct a JS Object, initially having a score with corresponding value the embeded document with $gt : 50 . later the second occurance of the score in the line will replace the first one.

In order to achieve an and effect, either use $and or condition in single object.

**Querying Inside Array**

MongoDb, in its query operations is Polymorphic. i.e

db.persons.find({ “favorites” : “iceCream”})

this query search for documents and if the document has a key favorites as simple string value, it check the string comparision and gives the output. But if favorites is a array then it will check the elements of the array for the match.

There is no recursion. This will a string patten for the whole array.

**$all (All the value ) :**  This will fetch all the records that have a array that include all the values sent as parameter.

db.persons.find( { “favorites” : { $all : [“beer” , “ice”] } } ) ;

the parameter for $all is a array. And the result of this query is a document that has a array with both the value.

This operator is used only for Arrays.

The order in which the data stored, doesn’t matter. Only the elements should have both beer and ice.

**$in (Atleast one of the values) :** If this parameter is used on string value then we will check for or conditions.

db.persons.find( { “name” : { $in : [ “pranith” , “Krishna” ] } } )

This query will return the documents whose name value is one of these two.

But if name is an Array, then it will fetch the document that as a array an element equal to any one of the passed parameters of $in. This is more like a or conditon.

**Queries with DOT notations**

As part of this we are working on a nexted documents. i.e

{

“name” : “richard”,

“email” : {

“work”:”pp@gmail.com”,

“professional” : “jj@gmail.com”

}

}

we can query this document as

db.employees.find( {

“email” : {

“work”:”pp@gmail.com”,

“professional” : “jj@gmail.com”

}

} )

this will fetch the document, but if we change the order of work and professional in the query, then it will return the document.

**If the right hand side of the colon does not contains “$”, then find compares byte by byte for the contents of the 2 docs(queyr and db-doc).**

If the order is differ, then the representation of data in BSON differs and so it will not consider the document.

The below query will not search of the documents described above. This will try to exact match in the document but we have a extra other prop in the email sub doc.

db.employees.find( {

“email” : {

“work”:”[pp@gmail.com](mailto:pp@gmail.com)”

}

} )

To fetch the document, based on the condition on the sub-document property, which is irrespective of the other, can be achieved by using Dot Notation.

db.employees.find(

{

“email.work”:”[pp@gmail.com](mailto:pp@gmail.com)”

}

)

this query will the document, with sub-document “email” and which has entry with key “work” and value as [pp@gmail.com](mailto:pp@gmail.com). This is irrespective of the other entries in the sub document, so we can fetch the result.

**Querying and cursors**

Let us understand, what is going behind the screen while fetching the documents. We execute the quries and shell will print the documents.

But, what actualy going on , when we execute a query. A cursor is constructed everytime when we try to execute a query and returned in the shell. The shell is configured to iterating to the cursors and printing those elements.

Cur = db.people.find() ; null; // we use null at last, to avoid early printing of the documents.

“Cur” is a variable holding a cursor object . cursor object has a varity of methods.

**hasNext();** returns true if there is any document.

**next():** returns the next document.

**limit(number):** As long as you haven’t actually stepped through the cursor or checked to see if there is any document left, we can impose limits.. This will limit the server to sent only these many documents.

Ex : cur.limit(5);

limit also returns cursor.

Untill, I actually retrieve some documents or check to see if there are any documents left on this curosr, no transmission is made to the DB or no query been executed on database.

**sort():** sort method instructs the db to return the documents, in an order that’s specified by the arguments to the sort. This also return the cursor.

Cur.sort( { “name” : -1 } ) /// this will give the output in the reverse order of name.

Cur.sort( { “name” : -1 } ) .limit(5);

Sort or limit cannot be applied after we started retrieving or check the documents exists. These conditions are executed on server but not by the shell. So we have to use it, before the query is actually executed.

**Skip(number):** this methods is used to skip the first n number of results avaiable.

i.e Cur.sort( { “name” : -1 } ) .limit(5).skip(5);

this will return a documents in reverse order of name, with final results size of 5 but skips first 2 results.

While executing the query, Sort will execute first then skip followed by limit. They are not processed in the client, though it looks like we are modifing a client side object, but actually modifing the query sent to DB.

**Counting Results**

db.peoples.count(arguments);

Arugment are documents that have the search criteria.

db.peoples.count({ “type” : “wages”});

**Updating of Documents**

**Updating an Array:**

**Update methods do four type of operation**

1. **update with replace.**
2. **Update a particular field**
3. **Insert a document if the document is not avaibale (Upserts)**
4. **Multiple update**

**Let look into one after the other**

**Update():** this is used to change the documents.

db.people.update( {name : “pranith”} , { name: “kuamr” , salary : “50000”});

the first arguments is a search criteria, i.e like a where clause in sql udate.

And the 2nd column is the document that has to be updated with the original except the primary key. i.e all the field in the current document in the database are removed and only include , { name: “kuamr” , salary : “50000”} along with older primary key.

This use of update operator is somewhat vestigial. MongoDB was originally designed as a object store and in Object store ,especially for dynamic typed language like javascript and python, it sort of convient to be able to say, I have a document in the app , manipulate the content of the document and replace it, disregarding what is actually in the DB and storing only those things that documents has now.

This use is however dangerour, so we use different methods which are explained below. However, this can be useful in sometimes.

**$set :** By using the update methods, we can’t modify or add a single field. However, we can do this with update by fetching the data from db and modify the same data and update it. This will replace the content of the document but this is not efficient all the time.

So to set only one particular field, we can use $set operator on update.

db.people.update( {name : “pranith”} , {$set : { father : “someshwarRao” } });

if field “father” is avaliable for pranith record, then father field is updated with “someshwarRao”. But if the field is not avaiable, then the new field “father” is added to the document.

$inc : this is used to increament a field value by a particualar limit and update.

db.people.update( {name : “pranith”} , {$inc : { salary : 10000 } });

this will increment the value of salary for record of pranith by 10000 i.e if the value in DB is 20000 then it will be update to 30000. But, if the field salary is not avaiable in the document then it simply adds the field to the document and set value as 10000.

**$unset:**  this is used to remove some fields from the documents. This is used in schemachanges .

db.people.update( {name : “pranith”} , {$unset : { salary : 1 } });

db.simple.insert(

{

\_id : 0 ,

a : [ 1, 2, 3, 4]

}

) ;

To update the arrays in a document we have to use $set and index location.

db.simple.update({\_id:0} , { $set : { “a.2” : 7 } } );

this will update the 3 element of the array to 7 from 3. The indexing of arrays starts with 0.

**Adding/removing an element to an arrays:** Sometimes you want to add elements to an array or remove elements from an array.

**$push:** To add an element to the right-hand side of the array.

db.simple.update({\_id:0} , { $push : { “a” : 10 } } ); // this will add the element to the array.

$pushAll operator has been deprecated since version 2.4: Use the [$push](http://docs.mongodb.org/manual/reference/operator/update/push/" \t "_blank) operator with [$each](http://docs.mongodb.org/manual/reference/operator/update/each/" \t "_blank)instead.

**$pop :** to remove the element from the right-hand side of the array.

db.simple.update({\_id:0} , { $pop : { “a” : 1} } ); // this will remove the right most element to the array.

db.simple.update({\_id:0} , { $pop : { “a” : -1} } ); // this will remove the left most element to the array.

**$pushAll:** To add many element to the array. This will have array as elements that has to be added..

db.simple.update({\_id:0} , { $pushAll : { “a” : [11,12,13] } } ); // this will add the element to the array.

**$pull:** Pull operator simply removes the element irespetive of the position. This will remove any accurance of any of those values in the arrays

db.simple.update({\_id:0} , { $pull : { “a” : 1 } } ); // this will remove the element 1 from the array.

**$pullAll:** Pull operator simply removes list of the element irespetive of the position. .

**$addToSet:** some time we may not have to treat arrays as a sequence of items, instead of set of items. In such case, we can add duplicate to the arrays. It will be difficult to check the content from our side and add to the array. Instead, we can use $addToSet. This will not add the duplicate entries. If the value is already avaiable it does nothing but if it is not available then element is added.

db.simple.update({\_id:0} , { $addToSet : { “a” : 1 } } );

**Upserts**

If the update command is not able to find the document with the condition specified then with upsert option to true, then the update command inserts a new document into the collection with the available information.

Let consider a empty collection ias.

db.ias.update (

{name : “pranith”},

{age : 50},

{upsert : true }

)

this is condition does not fetch any doc from a empty collection. So update will insert the docs with the avaliable information like

{\_id : “asdas” , “name” : “pranith” , “age” : 50}.

The name and age is from the above update query.

However, if the query is underdetermise(not a equal condition where we can’t determine the exact value) the information that needs to go into the resulting docs, then those fields will be left out of the result of upsert

db.ias.update (

{age : {$gt : 50 } }

{name : “pranith” },

{upsert : true }

)

age greater then 50, will not give the exact age so it will only insert the name field but not the age field.

{\_id : “asdas” , “name” : “pranith” }.

**Update Multiple Documents;**

🡪 Update operator can effect more than one document at a time.

🡪 To every document a new field can be added using empty condition. But with simple update, it will update the first document only. If I want to update all the documents then 3rd argument has to be passed.

db.people.update (

{name : “pranith”},

{age : 50},

{multi : true }

)

🡪 The default beahviour in SQL if where clause is not avaiable then it will update all the fields. Where as in MongoDB, it will update the only one document i.e randomly the first one. We have to specify the extra argument to update all.

🡪 Some of the drivers actually split this into two methods as update and multiUpdate.

**🡪 Processes for MultiUpdate :**  inside MongoDB, a single thread is ran for every query we execute. Every write operation that effects more then one doc is carefully coded in a co-operative multitasking fashion, to accosionally yeild control to thread to effect the same data set. This is called as Pausing and yeilding because , in fact theres a shared resource that differen write operation share.

This is basically a kind of new text lock.

Write operation that effect multiple documents are not Isolated transactions instead they might effect a arbitary number of docs , then yeild allow other readers or writers to operate and then pick up again to effect other remaing doc.

In General, it is not possible to have a isolated write operation that effect multiple docs inside a mongo Db. But single docs writign is atomic i.e if a write operation has to update 10 docs and it is in process of updating one of these docs, then at this point, the doc(which is updating now) is not shown to the world untill the update is done completely where as others are shown.

**Deleting Docs from DatabAes**

Remove() : Every colleciton has a method remove to delete the docs in a collection. Remove method will take the document as a argument which is similar to first argument in find. This argument is the condition to specify which docs has to be removed.

db.persons.remove({name : {$gt : “m” }});

this will remove all docs which are greater than “M”.

🡪if we pass a empty docs then all the docs are removed one after the other.

db.persons.remove({});

Remove same as multi update, can effect multiple docs i.e when ever the condition is matched. Same a update, this is not atomic, isolated operation. Instead, while removing it give access to other thread to the docs, but each individual docs removal is atomic in nature.

**drop() :** This will removed the complete collection along with indexes . this is more efficient way while deleting all the docs.

Remove is one-by-one where as drop will simply delete the internal represenation. So drop is very fast.

We have to take of isolation of operation in MongoDB for multip update and delete.

**Java MongoDb Driver**

With MongoDb Java driver, the entry point into MongoDB is class MongoClient. MongoClient has a no-args argument that by defaults will connect to the localhost and port 27017.

MongoClient client = new MongoClient();

But, we can change the default behavior using a number of different constructor, so that I can be explicit about the server that I’m connecting to as well as port.

MongoClient client = new MongoClient("localhost",27017);

Or else we can use a class ServerAddress, which is similar, which also takes a server name and optionally a port.

MongoClient client = new MongoClient(new ServerAddress("localhost",27017));

And also sometime we have to pass list of server address, which is used when we are using mongoDb over a clusters.

MongoClient client = new MongoClient(Arrays.asList(new ServerAddress("localhost",27017)));

We can also use connection string like below.

MongoClient client = new MongoClient(new MongoClientURI("mongodb://localhost:27017"));

We can also provide number of options to mongoClient with MongoClientOptions .

MongoClientOptions options = MongoClientOptions.*builder*().connectionsPerHost(1000).build();

MongoClient client = **new** MongoClient(**new** ServerAddress(), options);

MongoClientOptions is a immutable object, so we have use builder() method that will build it. We can set the required option to MongoClientOptions.

connectionsPerHost(): the default properties with the system is 100. We can manage change this options and build() it.

**So, what is MongoClient?** : it’s a heavy weight object, in that, it represents a pool of sockets or connections to all the servers it is connected to. In the case of standalone, it will be a single server. And if connections for host is 100, then this mongoclient will be a pool of at most 100 connections. As it is heavy, we have to take care that we are not using it un-necessary or as minimum as possible.

So it’s typical to store MongoClient in a static variable for simple application or instance variable if you have some sort of singleton class or you might want to configure it in an IOC Containers like spring and juice.

There are many methods in MongoClient but the most common is:

**getDatabase():** this method will get the MongoDatabase object that acts a interface to the MongoDB.

MongoDatabase database = client.getDatabase("test");

Unlike to MongoClient, MongoDatabase is are very lightweight and all they have in them is a few configurations properties and database name. So you can create these as much as we can and we don’t need to cache them.

**Databases are immutable, so once you create a instance of MongoDatabase, and if you want to reconfigure the it, then we to create a new instance of it.**

Configuration on database can be done with the helper methods provided with it.

MongoDatabase database = client.getDatabase("test").withReadPreference(ReadPreference.secondary());

Note.: As MongoDatabase is immutable. All Whenever we call a configuration method, new object will to created, so we have to care that it is assigned to object reference. If else it will be neglected.

MongoDatabase database = client.getDatabase("test");

database.withReadPreference(ReadPreference.secondary());

**MongoCollection :** MongoDatabase has a number of methods, but the most common getCollection(string collectionName). This will return MongoCollection. MongoCollection is generic(MongoCollection<TDocument>) .

Mongo package with a set of Documents type.

MongoCollection is a gateway to all the CRUD operations. MongoCollection is lightweight and **immutable. And if you want to reconfigure them, you should use the with methods.**

Ex1: MongoCollection<Document> collection = database.getCollection("test");

Ex2: MongoCollection<BsonDocument> collection1 = database.getCollection("test",BsonDocument.class);

**Java Driver: Representing Documents**

Earlier we have a session on BSON and how BSON are represented in a mongo shell. As part of this session, we have to understand, how BSON are represented using the MongoDB Java Driver.

We have many ways to do it, but the main class for doing this is Document, and we can create a new instance with no-args constructor.

Document class implements Map<String, Object>, because BSON is a key value pair, where key is a String and value can have many datatypes, so value is Object.

Document document = new Document();

document.append("key", "Hello World")

.append("key2", "pranith");

String value = (String)document.get("key");

In the code example, we have create the instance of new document().

We can use append method of document to set the key and its value. Append method returns document, so we can use the method continuously .

Get: the get method of the document, to fetch the value of the key. Get method return object, so we have to cast the objects.

getString(string key): this will directly give in string for a paricular key.

document.append("key", "Hello World")

.append("int", 1)

.append("l", 1l)

.append("double", 1.1)

.append("b", false)

.append("date", new Date())

.append("objectId", new ObjectId())

.append("null", null)

.append("embededDOC", new Document("x","doc"))

.append("arrays", Arrays.asList(1,2,3))

;

ObjectId class will create a objectID for the object. Other are self explained. For date, we can use java.util.Date .

🡪 Document class also implement serializable and BSON. Bson is a interface that has single method toBsonDocument which returns BSONDocument. BsonDocument is similar to document except it implement map<string, bsonValue>. BsonValue is a parent class for many other classes, to reproesent every possible type defiend in BSON Spec. so this is type safe.

**Please Read JsonWritter.**

**Inserting Docs Using Java Driver:**

**MongoCollections.insertOne(document):** This method will insert only one document at a time to the database connected. Same in the shell, While inserting the document and if the objectID is not avaiable then the java driver will create a objectID while inserting. If ObjectID is available, then the same is inserted to Database.

The documents is serialized immedialy after the insert i.e after the insert command is executed then document reference in the code also have \_id field to access.

**MongoCollections.insertMany(list< document > listOfDocuments):** This method will insert many document at a time to the database connected. Same in the shell, While inserting the document and if the objectID is not avaiable then the java driver will create a objectID while inserting. If ObjectID is available, then the same is inserted to Database.

The documents is serialized immedialy after the insert i.e after the insert command is executed then document reference in the code also have \_id field to access.

public static void main(String[] args) {

MongoClient client = new MongoClient();

MongoDatabase database = client.getDatabase("course");

MongoCollection<Document> collection = database

.getCollection("courseCollection");

collection.drop();

Document document = new Document().append("name", "Pranith")

.append("age", 30).append("profession", "Programmer");

collection.insertOne(document);

Document document1 = new Document().append("name", "Krishna")

.append("age", 27).append("profession", "Programmer");

Document document2 = new Document().append("name", "rao")

.append("age", 25).append("profession", "Programmer");

collection.insertMany(Arrays.asList(document1, document2));

}

**Fetching Docs Using Java Driver:**

Finding a single document in a collection.

Document firstDoc = collection.find().first();

System.out.println(firstDoc.get("\_id"));

Find methods actually returns a FindIterable, which extends MongoIterable and this has the method first, which will return a single result.

If we know the number of documents which we gonna get is small, then we can put them all into an Array and avoid the code to iterate.

List<Document> list = collection.find().into(new ArrayList<Document>());

for (Document docs : list) {

System.out.println(docs.get("\_id"));

}

the into method takes a arrays as a parameter.

If the number of documents are more then we have to use mongoCursors. But we have to take care the cursor is closed. So use try –fianaly

MongoCursor<Document> cursor = collection.find().iterator();

try {

// using try finally, becuae we have to close the cursor resource in

// all the cases

while (cursor.hasNext()) {

Document docs = cursor.next();

System.out.println(docs.get("\_id"));

}

} finally {

cursor.close();

}

**Count:**

**Long count = collection.count();**

**System.out.println(" count " + count);**

**public** **static** **void** main(String[] args) {

MongoClient client = **new** MongoClient();

MongoDatabase database = client.getDatabase("course");

MongoCollection<Document> collection = database

.getCollection("courseCollection");

collection.drop();

Document document = **new** Document().append("name", "Pranith")

.append("age", 30).append("profession", "Programmer");

Document document1 = **new** Document().append("name", "Krishna")

.append("age", 27).append("profession", "Programmer");

Document document2 = **new** Document().append("name", "rao")

.append("age", 25).append("profession", "Programmer");

collection.insertMany(Arrays.*asList*(document, document1, document2));

// To get the first Document

Document firstDoc = collection.find().first();

System.***out***.println(firstDoc.get("\_id"));

// To get the multiple Document, if the size of the documents is less

List<Document> list = collection.find().into(**new** ArrayList<Document>());

**for** (Document docs : list) {

System.***out***.println(docs.get("\_id"));

}

// To get the multiple Document, if the size of the documents is very

// more

MongoCursor<Document> cursor = collection.find().iterator();

**try** {

// using try finally, becuae we have to close the cursor resource in

// all the cases

**while** (cursor.hasNext()) {

Document docs = cursor.next();

System.***out***.println(docs.get("\_id"));

}

} **finally** {

cursor.close();

}

// to fetch the count of the collection

Long count = collection.count();

System.***out***.println(" count " + count);

}

**Querying with Filter**

We can do this with a couple of ways

-> Query filters using raw Documents : Creating docs and adding to find or count method. Below is the example which shows docs as filter.

Bson filter = new Document().append("age", 25);

Document firstDoc = collection.find(filter).first();

List<Document> list = collection.find(filter).into(new ArrayList<Document>());

MongoCursor<Document> cursor = collection.find(filter).iterator();

Long count = collection.count(filter);

Query with multiple parameters: age is 25 and x is less then 99 and greater than 55.

Bson filter = new Document()

.append("age", 25)

.append("x", new Document().append("$gt", 55).append("$lt", 99));

**Query filter Builders:** The builders are in a class called Filters and we have a set of static methods that are used to build the quries.

Bson filter = Filters.and(Filters.eq("age", 25), Filters.gt("x", 55) , Filters.lt("x", 99));

**Quering with Projections**

This includes exclude and include fields in the result sets. This can be done in 2 ways.

Raw Documents: 0 means false. Remember \_id is special case, we have to specically exclude it. If not, even if it is not plesented in the included list it show up.

Bson filter = Filters.and(Filters.eq("age", 25), Filters.gt("x", 55) , Filters.lt("x", 99));

Bson projection = new Document("x",0);

MongoCursor<Document> cursor = collection.find(filter).projection(projection).iterator();

**Builders :**

exclude methods to exclude values.

Include methods to include values

Fields to add both includes and excludes.

excludeId is method to avoid ID in the result.

Bson projection1 = Projections.*exclude*("x","\_id");

Bson projection2 = Projections.*include*("y","j");

// to some both include and exclude use fileds method of projections.

Bson projection3 = Projections.fields(Projections.include("y","j"),Projections.exclude("x","\_id")) ;

Sorting: Sort method on collecion will help to sort. 1 means accending and -1 means decending . This can be done as documents.

Bson sorting = new Document().append("x", 1).append("y", -1);

MongoCursor<Document> cursor = collection.find(filter).projection(projection1).sort(sorting).iterator();

Builders. Ascending and decending methods of Sorts class help to sort the query result. Both these can be joined by groupBy(BSON .. ) methods.

Bson sorting1 = Sorts.ascending("x","z");

Bson sorting2 = Sorts.descending("y");

Bson sorting = Sorts.orderBy(sorting1,sorting2);

Limit and Skip:

limit() and skip() methods in collection are used for this.

MongoCursor<Document> cursor = collection.find(filter)

.projection(projection1)

.sort(sorting)

.skip(20)

.limit(50)

.iterator();

**Updating Docs**

replaceOne(): this method of collection will helps us to replace the whole content of the document. It takes 2 arguments: 1 st argument is the document for which condition has to matched. And 2nd argument is the document that are to be replaced.

Note: As we have to completely replace the docs, so make sure that \_id is same.

collection.replaceOne(new Document().append("x", 5), new Document().append("\_id", 5).append("x", 25).append("y", 15) );

UpdateOne(); it is used to update a particular filed of a documents. This will also take 3 arguments, 1st condition and 2nd the field, 3rd is option for upsert.

collection. updateOne (new Document().append("x", 5), new Document().append("$set", new Document().append("x", 25)));

Upsert(update and insert): new docs creates, if the field is not matching.

collection.updateOne (new Document().append("x", 5), new Document().append("$set", new Document().append("x", 25), new UpdateOptions().upsert(true))) ;

Update many :

collection.updateOne(Filters.gt("x", 5) , new Document().append("$inc", new Document().append("x", 1)));

**Deleting :**

collection.deleteOne(filter);

collection.deleteMany(filter);