Workshop 3

MongoDB

MongoDB CreateCollection

If a collection does not exist, MongoDB will automatically create the collection when you first store data for that collection.

When you first use MongoDB you will find you have no collections:

```
show collections
```

When you insert data in the next step, a collection will automatically be created for you.

Inserting data

The format for the INSERT command is:

```
 db.collectionName.insert(
  {
    key_1: 'value1',
    key_n: 'valueN'
  }
 )
```

Note:

- character and date values must be enclosed in matching single (') or double quotes (").
- numeric values do not need quotes.
- key/value pairs are comma-separated (no comma needed after the last pair)
- each record is enclosed in curly brackets: {}

Department Data

The following examples will create a dept collection that will represent the DEPT table seen in the Oracle Sample Data. We will also include an object id (_id) for each record.

Department 10

Add department 10:

```
db.dept.insert(
{
   _id:    10,
   deptno: 10,
   dname:  "ACCOUNTING",
   loc:    "NEW YORK"
})
```

Assuming you have no error messages, the system should respond with:

```
 WriteResult({ "nInserted" : 1 })
```

Department 20

Next add department 20:

```
db.dept.insert(
{
   _id:    20,
   deptno: 20,
   dname:  "RESEARCH",
   loc:    "DALLAS"
})
```

Department 40

Next department 40:

```
db.dept.insert(
{
   _id:    40,
   deptno: 40,
   dname:  "OPERATIONS",
   loc:    "BOSTON"
})
```

Object IDs

Object ids can be used to provide a unique value in a collection, that is equivalent to a primary key field in relational databases. They are either system generated and are guaranteed to be unique, or can be created by the user as seen above. If user defined, the user must provide unique values within the collection.

For example, try and add another document with the id of 10:

```
db.dept.insert(
 {
   _id:    10,
   deptno: 50,
   dname:  "TEST",
   loc:    "WOLVERHAMPTON"
 })
```

You should get an error message along the lines of: *...duplicate key error collection: dbYourStudentNumber.dept....'*

Whereas if you add a new document with an existing deptno, but different object ID, it will quite happily accept it:

```
db.dept.insert(
{
  _id:    50,
  deptno: 40,
  dname:  "OPERATIONS V2",
  loc:    "BOSTON"
})
```

Things to note

If you are getting errors, check carefully that:

- every opening bracket has an appropriate closing bracket:
  - the insert statement uses round brackets: ()
  - a collection uses curly brackets: {}
- each key:value pair are separated by commas, except for the last item
- strings are enclosed in single or double quotes, e.g., 'myString', or "myString"

Exercise 2.1

- Compare how you added the above data and how it differs from INSERT records in a relational database
- Add Department 30, which has the following key/values: _id: 30, deptno: 30, dname: SALES and loc: CHICAGO

Employee data

The following examples will create a emp collection that will represent the EMP table seen in the Oracle Sample Data.

No object id (_id) is included this time, so you can compare the previous examples against the object id generated automatically in the documents below.

More than one record can be added at a time. The following examples will add several employees for each department.

Department 10 Employees

```
db.emp.insert( [
    {
       empno: 7782,
       ename: 'CLARK',
       job: 'MANAGER',
       mgr: 7839,
       hiredate: new Date('1989-06-09'),
       sal: 2450,
       deptno: 10
    },
    {
       empno:7839,
       ename: 'KING',
       job: 'PRESIDENT',
       hiredate: new Date('1980-11-17'),
       sal: 5000,
       deptno: 10
     },
     {
       empno: 7934,
       ename: 'MILLER',
       job: 'CLERK',
       mgr: 7782,
       hiredate: new Date('1985-01-23'),
       sal: 1300,
       deptno: 10
     }
   ]
 )
```

This time the results returned will be along the lines of:

```
 BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 3,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
```

Department 20 Employees

```
db.emp.insert( [
  {
     empno: 7876,
     ename: 'ADAMS',
     job: 'CLERK',
     mgr: 7788,
     hiredate: new Date(),
     sal: 1100,
     deptno: 20
  },
  {
     empno: 7902,
     ename: 'FORD',
     job: 'ANALYST',
     mgr: 7566,
     hiredate: new Date('1991-12-03'),
     sal: 3000,
     deptno: 20
  },
  {
     empno: 7066,
     ename: 'JONES',
     job: 'MANAGER',
     mgr: 7839,
     hiredate: new Date('1991-04-02'),
     sal: 2975,
     deptno: 20
```

```
   },
   {
      empno: 7788,
      ename: 'SCOTT',
      job: 'ANALYST',
      mgr: 7566,
      hiredate: new Date('2015-10-16'),
      sal: 3000,
      deptno: 20
   }
  ]
)
```

Date data types

Dates have been included in the above:

- date strings are enclosed in single/double quotes and the format is 'yyyy-mm-dd' e.g., Date('2016-10-10')
- use the Date() constructor to create a date datatype
- date can also be a datetime, e.g., new Date("<yyyy-mm-ddThh:mm:ss>")
- new Date() will return the current date

Exercise 2.2

- Add the employees for Department 30.

Remember this time, some employees will have a commission (COMM):

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-95	1600	300	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7654	MARTIN	SALESMAN	7698	28-SEP-93	1250	1400	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7521	WARD	SALESMAN	7698	22-FEB-94	1250	500	30

If you add all the documents in one insert statement, you will get a message along the lines of:

```
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 6,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
```

Querying a collection

The find() function can be used to query the documents.

The format is:

```
db.collectionName.find(optional_query_criteria)
```

Where the query_criteria follows a pattern:

```
db.collectionName.find({keyField: "value"})
```

Note:

- the criteria is enclosed in curly brackets: {}
- the value needs quotes if it is a string or date value
- all names and values are case sensitive
- quotes are optional for the fieldName, so long as they do not contain spaces

Find all documents

For example, show all the data so far in the *dept* collection:

```
db.dept.find()
```

The results should look like:

```
{ "_id" : 10, "deptno" : 10, "dname" : "ACCOUNTING", "loc" : "NEW YORK" }
{ "_id" : 20, "deptno" : 20, "dname" : "RESEARCH", "loc" : "DALLAS" }
{ "_id" : 40, "deptno" : 40, "dname" : "OPERATIONS", "loc" : "BOSTON" }
{ "_id" : 50, "deptno" : 40, "dname" : "OPERATIONS V2", "loc" : "BOSTON" }
{ "_id" : 30, "deptno" : 30, "dname" : "SALES", "loc" : "CHICAGO" }
```

To show the documents in the *emp* collection:

```
db.emp.find()
```

The data comes back messy. The pretty() function can be used to improve the layout:

```
db.emp.find().pretty()
```

A subset of the *emp* collection is shown below:

```
> db.emp.find().pretty()
{
        "_id" : ObjectId("5a09e79ac536e890d5a7a666"),
        "empno" : 7782,
        "ename" : "CLARK",
        "job" : "MANAGER",
        "mgr" : 7839,
        "hiredate" : ISODate("1989-06-09T00:00:00Z"),
        "sal" : 2450,
        "deptno" : 10
}
{
        "_id" : ObjectId("5a09e79ac536e890d5a7a667"),
        "empno" : 7839,
        "ename" : "KING",
        "job" : "PRESIDENT",
        "hiredate" : ISODate("1980-11-17T00:00:00Z"),
        "sal" : 5000,
        "deptno" : 10
}
....
```

Note the object ids are now system generated (and will be different values in your own data).

Find with query criteria

If working with a large collection, you will not want all the documents returned.

Find all the employees are are clerks:

```
 db.emp.find({job:"CLERK"})
```

For numerical data, the greater than (>) and less than (<) operators are represented by *$gt* and *$lt* respectively. Note, for these operators, the search criteria must be enclosed in {} brackets.

Find all employees who earn more than 2400:

```
db.emp.find({sal: {$gt:2400}})
```

Find all employees whose commission is less than 1000:

```
db.emp.find({comm: {$lt:1000}})
```

Working with the date field (hiredate) is more complex, since you have to create a new date for the comparison.

For example, find all employees who start after the 1st January 2000:

```
db.emp.find({hiredate: {$lt: new Date("2000-01-01")}})
```

Find employees who started on the 16th October 2015:

```
db.emp.find({hiredate: new Date("2015-10-16")})
```

Find One document

To find just one document requires the use of the equivalent of a primary key field. This can be a field that the user takes responsibility to keep unique, such as the *deptno*:

```
db.dept.find({deptno:10})
```

Or the object id can be used, which will be unique:

```
db.dept.find({_id:10})
```

In the *emp* collection, the _ids are system generated and generally along the lines of: '5a0727e99ba81dee9b1cc6a3', so less easy to use!

List all the records in emp:

```
db.emp.find().pretty()
```

and pick an _id from the collection and then try and find one record.

For example (note, your object id will be different):

```
db.emp.find( {_id : ObjectId("5a0727e99ba81dee9b1cc6a3")}).pretty()
```

The function *ObjectId()* must be used to convert the value into an object id.

Updating a Collection

The format of the update command is:

```
 db.collectionName.update({'keyField': 'value' },
  {$set: field: 'newValue' }
 )
```

The **update()** function can be used to update one or more documents. If the change should only apply to one document, the keyField needs to be a field with unique values, to ensure the correct document is updated.

This is similar in SQL to providng the WHERE clause of an UPDATE command.

Alternatively since version 3.2, MongoDB also supports the following functions:

```
db.collectionName.updateOne(); /* updates a single document that matches a
specified filter
                                   (even if several documents match the
filter) */
db.collectionName.updateMany(); /* updates all documents that matches a
specified filter */
db.collectionName.replaceOne(); /* replaces the first document that
matches a specified filter
                                     (even if several documents match the
filter) */
```

Updating the dept collection

Update department 40 to change the location to Wolverhampton:

```
db.dept.update({'deptno':40},
 {$set:  {'loc': 'WOLVERHAMPTON'}})
```

Check the changes have been made:

```
db.dept.find({"deptno":40}).pretty()
```

Updating the emp collection

Carter has now become an analyst:

```
db.emp.update({'empno':7782},
{$set:  {job: 'ANALYST'}})
```

This is equivalent to the SQL statement:

```
UPDATE emp SET job = 'ANALYST'
WHERE empno = 7782;
```

There are some limitations currently if you want to use a value in a field to update the value in another field, or even in the same field, such as increase existing salaries by 10%.

There are some field update operators that can be used, such as *$inc*, which increments a field by a specified value. The *$inc* operator accepts positive and negative values.

For example, observe what the following does:

```
db.emp.update({},
{$inc:  {sal: 100}})
```

The empty curly brackets this time means there is no equivalent of an SQL WHERE clause:

```
UPDATE emp SET sal = sal +100;
```

So you might think this should update all the salaries in all the documents by £100. Check the collection - is this the case?

```
db.emp.find().pretty()
```

You should find only the first document has been updated.

$inc can be used to decrease values too by using an negative number. To reduce the sal by £100:

```
db.emp.update({},
 {$inc:  {sal: -100}})
```

This should reverse the previous increase.

To update all the documents, *updateMany()* must be used instead:

```
db.emp.updateMany({},
 {$inc:  {sal: 100}})
```

Check the documents again and see if the salaries have now been increased by £100.

Exercise 2.4

- 2.4.1 Update the name of department 40 to: COMPUTING
- 2.4.1 Update the salary of employee number 7788 in department 20 to 3500

Deleting a Document

At some stage you may want to delete a document from a collection. The format is:

```
db.collectionName.deleteOne(query_criteria) /* deletes the first document
found that matches the query criteria */
db.collectionName.remove(query_criteria) /* deletes all documents found
that matches the query criteria */
db.collectionName.remove(query_criteria) /* deletes all documents in a
collection (use with care!) */
```

Deleting a Collection

If you want to drop a collection completely, including any data held in it:

```
db.collectionName.drop() /* removes a collection completely (use with
care!) */
```

This is equivalent to the SQL command:

```
DROP TABLE tableName;
```

Delete Department 40

Department 40 has closed down completely and should be removed. Note, if you added the duplicate department 40 previously you will have 2 documents:

```
db.dept.find({deptno:40})
```

Which should return:

```
{ "_id" : 40, "deptno" : 40, "dname" : "OPERATIONS", "loc" : "BOSTON" }
{ "_id" : 50, "deptno" : 40, "dname" : "OPERATIONS V2", "loc" : "BOSTON" }
```

If you use the *deleteOne()* function, it will likely delete the first record. To delete both documents *remove()* must be used instead:

```
db.dept.remove({deptno: 40})
```

Note, if the query_criteria only returns one document, **remove()** and **deleteOne()** will have the same effect.

This is equivalent to the SQL command:

```
DELETE FROM dept WHERE deptno = 40;
```

Check that the collection has gone:

```
db.dept.find({deptno:40})
```

No documents should be returned.

Aggregation Pipeline

So far **find()** either returns all the elements of an array, if one element matches the search criteria, or **$elematch** returns the first one found only. The latter is fine if there is only one to be found, but not so good if several items in the array should match the search criteria. The aggregation pipeline can help with this.

$filter

In the aggregation pipeline, array has various operators and the one we are interested in is $filter

This returns a subset of the array with only the elements that match the filter condition.

$filter has the following syntax:

```
{ $filter: {
       input: <array>,     /* expression for the array */
       as: <string>,     /* variable name for the element */
       cond: <expression> /* filter condition */
} }
```

$filter is one of the stages of the pipeline and can not be used by itself. It is used with an aggregation framework operator, such as **$project**.

We can use the pipeline to *gather* elements of our employees array to get the employees matching the query criteria only, rather than one, or everyone. This example has only one stage:

```
db.deptCollection.aggregate([ {
     $project: {
        empSet: {
           $filter: {
              input: "$employees",
              as: "employee",
              cond: { $gte: [ "$$employee.sal", 2000 ] }
           }
        }
     }
  }
]).pretty()
```

Now the system should **only** retrieve the employees with salary > 2000.

Count

The power of the aggregation pipeline is to do processing on the data.

Lets count how many employees each department has:

```
db.deptCollection.aggregate({
 "$project": {
    "deptno": 1,
    "Count": { "$size": { "$ifNull": [ "$employees", [] ] }
     }
   }})
```

The *$ifNull* operator is needed, since department 40 has no employees - you will get an error message if left out!

References

https://mi-linux.wlv.ac.uk/wiki/index.php/MongoDB_Workbook