INFS3200/7907 Advanced Database System – Practical4

MongoDB

Submission:

For this practical there are a series of questions that are listed on the final page. These must be completed and shown to your tutor in one of the scheduled practical sessions.

Introduction

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling. MongoDB works on concept of collection and document.

Documents

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

```
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ]
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2014-01-16T00:00:00Z"),
         "grade" : "B",
         "score" : 17
      }
   ],
   "name" : "Vella",
   "restaurant_id" : "41704620"
}
```

Collection

MongoDB stores documents in collections. Collections are analogous to tables in relational databases. Unlike a table, however, a collection does not require its

documents to have the same schema.

In MongoDB, documents stored in a collection must have a unique **_id** field that acts as a primary key.

Connect to MongoDB

This time, we use the same way as in Practice 2 to access the ITEE moss server via putty (You cannot find it from the Start menu currently). You can find it in "C:\Program Files (x86)\PuTTY". When you open it, it should look like this (Figure 1):

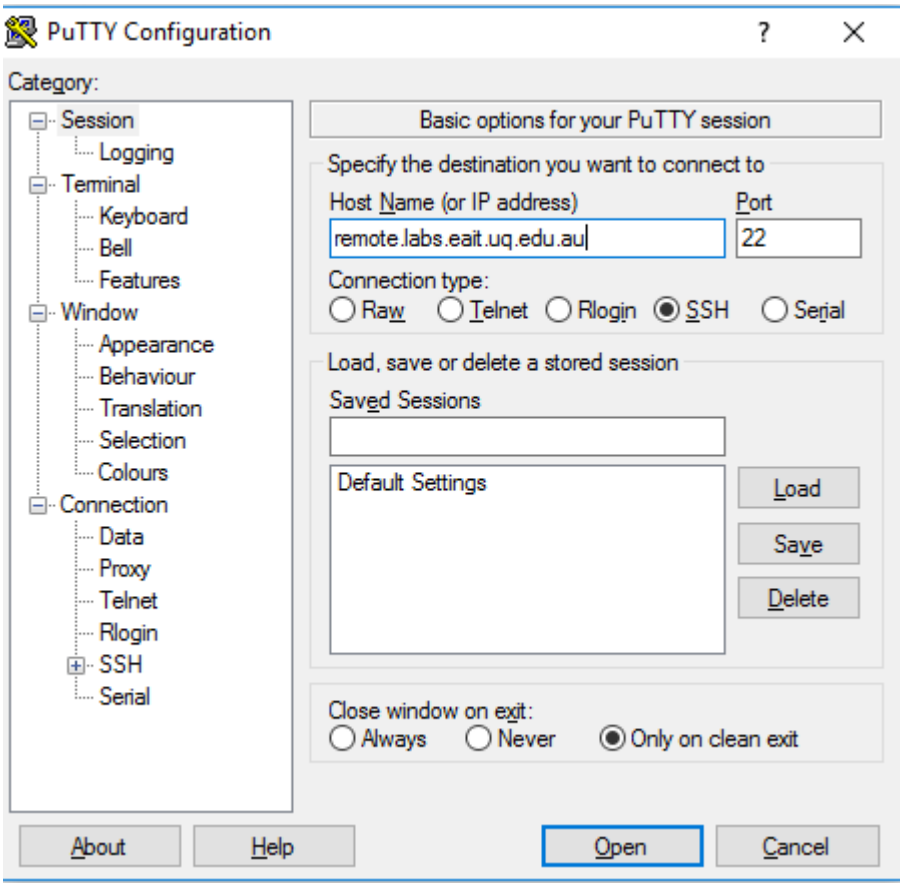


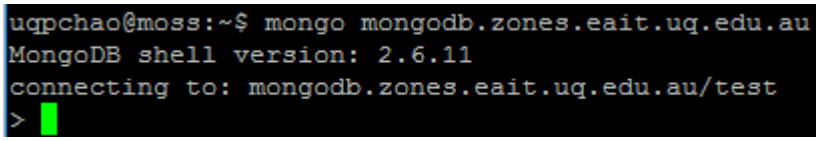
Fig 1

Type in the host name of Moss Server addr: remote.labs.eait.uq.edu.au. Then press **open**. Then type your student number and password, it may allow you to login. (student number should start with s, e.g. s4000888)

After login, the second step is to connect to remote MongoDB server through Moss (addr: mongodb.zones.eait.uq.edu.au). From a system prompt, start mongo connection by issuing the **mongo** command, as follows:

mongo mongodb.zones.eait.uq.edu.au

After a successful log in, the command line looks like:



After starting the mongo shell your session will use the **test** database by default. At any time, issue the following operation at the **mongo** to report the name of the current database:

```
db
```

Switch to a new database named in your own student name (s4000888 for example), with the following operation:

```
use s4000888
```

Use **db** to confirm that you are using your own database.

```
db
```

MongoDB will not permanently create a database until you insert data into that database. Now we need to insert data into our database.

Task 1 Data Import and Retrieval

Create a Collection and Insert Documents

MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses dynamic schemas, you also need not specify the structure of your documents before inserting them into the collection.

1. Create two documents named **j** and **k** by using the following sequence of JavaScript operations:

```
j = { name : "mongo" }
k = { x : 3 }
```

2. Insert the **j** and **k** documents into the **testData** collection with the following sequence of operations:

```
db.testData.insert( j )
db.testData.insert( k )
```

   When you insert the first document, the mongod will create both the s4000888 database and the **testData** collection.

3. Confirm that the **testData** collection exists. Issue the following operation:

```
show collections
```

The mongo shell will return the list of the collections in the current (i.e. s4000888) database. At this point, the only collection with user data is **testData**.

4. Confirm that the documents exist in the **testData** collection by issuing a query on the collection using the **find()** method:

   db.testData.**find**()

   This operation returns the following results.

```
> db.testData.find()
{ "_id" : ObjectId("573fec2e38af85cb9948083e"), "name" : "mongo" }
{ "_id" : ObjectId("573fec3438af85cb9948083f"), "x" : 3 }
```

   All MongoDB documents must have an **_id** field with a unique value. These operations do not explicitly specify a value for the _id field, so mongo creates a unique **ObjectId** value for the field before inserting it into the collection.

5. Use the **for** loop to insert more data:

```
for (var i = 1; i <= 25; i++) {
  db.testData.insert( { x : i } )
}
```

   And use **find()** to check the result:

```
> db.testData.find()
{ "_id" : ObjectId("573fec2e38af85cb9948083e"), "name" : "mongo" }
{ "_id" : ObjectId("573fec3438af85cb9948083f"), "x" : 3 }
{ "_id" : ObjectId("573ff01638af85cb99480840"), "x" : 1 }
{ "_id" : ObjectId("573ff01638af85cb99480841"), "x" : 2 }
{ "_id" : ObjectId("573ff01638af85cb99480842"), "x" : 3 }
{ "_id" : ObjectId("573ff01638af85cb99480843"), "x" : 4 }
{ "_id" : ObjectId("573ff01638af85cb99480844"), "x" : 5 }
{ "_id" : ObjectId("573ff01638af85cb99480845"), "x" : 6 }
{ "_id" : ObjectId("573ff01638af85cb99480846"), "x" : 7 }
{ "_id" : ObjectId("573ff01638af85cb99480847"), "x" : 8 }
{ "_id" : ObjectId("573ff01638af85cb99480848"), "x" : 9 }
{ "_id" : ObjectId("573ff01638af85cb99480849"), "x" : 10 }
{ "_id" : ObjectId("573ff01638af85cb9948084a"), "x" : 11 }
{ "_id" : ObjectId("573ff01638af85cb9948084b"), "x" : 12 }
{ "_id" : ObjectId("573ff01638af85cb9948084c"), "x" : 13 }
{ "_id" : ObjectId("573ff01638af85cb9948084d"), "x" : 14 }
{ "_id" : ObjectId("573ff01638af85cb9948084e"), "x" : 15 }
{ "_id" : ObjectId("573ff01638af85cb9948084f"), "x" : 16 }
{ "_id" : ObjectId("573ff01638af85cb99480850"), "x" : 17 }
{ "_id" : ObjectId("573ff01638af85cb99480851"), "x" : 18 }
Type "it" for more
>
```

   When you query a collection, MongoDB returns a "**cursor**" object that contains the results of the query. The mongo shell then iterates over the cursor to display the results. Rather than returning all results at once, the shell iterates over the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. In the shell, enter **it** to iterate over the next set of results.

Query for Specific Documents

1. In the mongo shell, query for all documents where the x field has a value of 18 by passing the { x: 18 } query document as a parameter to the find() method:

   db.testData.**find**({ x : 18 })

   MongoDB returns one document that fits this criteria:

```
> db.testData.find( { x : 18 } )
{ "_id" : ObjectId("573ff01638af85cb99480851"), "x" : 18 }
```

2. To increase performance, you can constrain the size of the result by limiting the amount of data your application must receive over the network. Call the **limit()** method on a cursor, as in the following command:

   db.testData.find().**limit**(3)

Task 2 Advanced Operations and Queries

Basic database operations

Insert documents

1. Insert a document into a collection named **inventory**. The operation will create the collection if the collection does not currently exist.

```
db.inventory.insert(
  {
    item: "ABC1",
    details: {
      model: "14Q3",
      manufacturer: "XYZ Company"
    },
    stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
    category: "clothing"
  }
)
```

   The operation returns a **WriteResult** object with the status of the operation. A successful insert of the document returns the following object:

   WriteResult({ "nInserted" : 1 })

2. Use **find()** to verify your insertion:

   db.inventory.**find**()

   The document you inserted should return:

```
> db.inventory.find()
{ "_id" : ObjectId("57400e7438af85cb99480872"), "item" : "ABC1", "details" : { "
model" : "14Q3", "manufacturer" : "XYZ Company" }, "stock" : [ { "size" : "S", "
qty" : 25 }, { "size" : "M", "qty" : 50 } ], "category" : "clothing" }
```

3. Insert an array of documents. First we create them by define a variable **mydocuments** that holds an array of documents to insert:

```
var mydocuments =
  [
   {
    item: "ABC2",
    details: { model: "14Q3", manufacturer: "M1 Corporation" },
    stock: [ { size: "M", qty: 50 } ],
    category: "clothing"
   },
   {
    item: "MNO2",
    details: { model: "14Q3", manufacturer: "ABC Company" },
    stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
    category: "clothing"
   },
   {
    item: "IJK2",
    details: { model: "14Q2", manufacturer: "M5 Corporation" },
    stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
    category: "houseware"
   }
];
```

4. Pass the **mydocuments** array to the **db.collection.insert()** to perform a bulk insert.

```
db.inventory.insert(mydocuments)
```

Update documents

1. For the document with item equal to "MNO2", use the **$set** operator to update the **category** field and the details field to the specified values and the **$currentDate** operator to update the field **lastModified** with the current date.

```
db.inventory.update(
  { item: "MNO2" },
  {
   $set: {
    category: "apparel",
    details: { model: "14Q3", manufacturer: "XYZ Company" }
   },
   $currentDate: { lastModified: true }
```

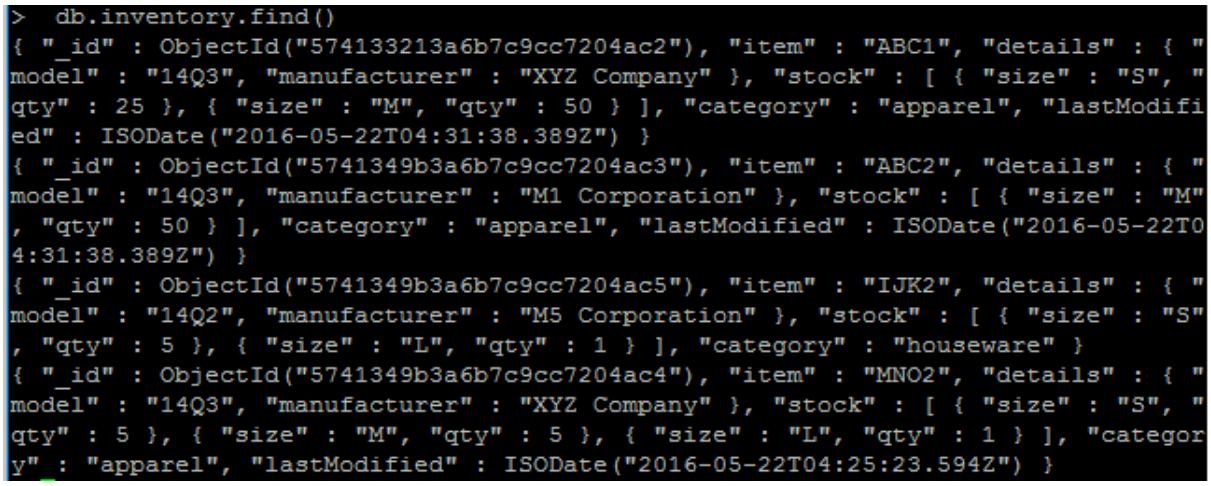
```
    }
)
```

2. By default, the **update()** method updates a single document. To update multiple documents, use the **multi** option in the **update()** method:

```
db.inventory.update(
  { category: "clothing" },
  {
    $set: { category: "apparel" },
    $currentDate: { lastModified: true }
  },
  { multi: true }
)
```

It updates the **category** field to "apparel" and update the **lastModified** field to the **current date** for all documents that have **category** field equal to "clothing".

Use **find()** to verify your modification:



Remove documents

1. In order not to affect the original data. We use a new collection **inventory1** and **inventory2** to perform remove operation.

```
db.inventory.find().forEach(function(x){db.inventory1.insert(x)})
```

```
db.inventory.find().forEach(function(x){db.inventory2.insert(x)})
```

2. To remove all documents from a collection, pass an empty query document {} to the **remove()** method. The **remove()** method does not remove the indexes:

```
db.inventory1.remove({})
```

To remove all documents from a collection, it may be more efficient to use the **drop()** method to drop the entire collection, including the indexes, and then

recreate the collection and rebuild the indexes.

3. Remove a **single** document that match a deletion criteria, call the **remove()** method with the **<query>** parameter, and the **justOne** parameter set to true or 1:

```
db.inventory2.remove( { category : "apparel" }, 1 )
```

4. Check the result for both collections using **find()**.

Query Documents

Specify Conditions Using Query Operators

1. Selects all documents in the **inventory** collection where the value of the **type** field is either 'food' or 'snacks':

```
db.inventory.find( { item: { $in: ['MNO2', 'ABC1'] } } )
```

Query on Embedded Documents

When the field holds an **embedded document**, a query can either specify an exact match on the embedded document or specify a match by individual fields in the embedded document using the **dot notation**.

1. Match all documents where the value of the field **details** is an embedded document that contains only the field **model** with the value '14Q2' and the field **manufacturer** with the value 'M5 Corporation', in the exact order:

```
db.inventory.find(
  {
   details:
     {
      model: '14Q2',
      manufacturer: 'M5 Corporation'
     }
  }
)
```

2. Uses the **dot notation** to match all documents where the value of the field **details** is an embedded document that contains a field **manufacturer** with the value 'M5 Corporation' and may contain other fields:

```
db.inventory.find( { 'details.manufacturer': 'M5 Corporation' } )
```

Analyze Query Performance

1. Prepare the dataset for query analysis by inserting the pre-defined variable

queryperf:

```
var queryperf =
[
{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 },
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 },
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 },
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 },
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 },
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 },
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 },
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 },
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 },
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }
];
```

and insert the document array:

```
db.queryanalysis.insert(queryperf)
```

Query with No Index

1. The following query retrieves documents where the quantity field has a value between 100 and 200, inclusive:

```
db.queryanalysis.find( { quantity: { $gte: 100, $lte: 200 } } )
```

2. The query returns the following documents:

```
> db.queryanalysis.find( { quantity: { $gte: 100, $lte: 200 } } )
{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 3, "item" : "p1", "type" : "paper", "quantity" : 200 }
{ "_id" : 4, "item" : "p2", "type" : "paper", "quantity" : 150 }
```

3. To view the query plan selected, use the **explain()** method:

```
db.queryanalysis.find( { quantity: { $gte: 100, $lte: 200 } } ).explain()
```

4. A part of the output includes:

```
"executionSuccess" : true,
"nReturned" : 3,
"totalDocsExamined" : 10
"winningPlan" : {
        "stage" : "COLLSCAN",
```

nReturned displays 3 to indicate that the query matches and returns three documents.

totalDocsExamined display 10 to indicate that MongoDB had to scan ten documents (i.e. all documents in the collection) to find the three matching documents.

winningPlan display COLLSCAN to indicate that MongoDB use collection scan to find the result.

Query with Index

5. To support the query on the quantity field, add an index on the quantity field:

```
db.queryanalysis.ensureIndex( { quantity: 1 } )
```

6. Use the explain() method to view the query plan statistics, and it returns this output:

```
"totalDocsExamined" : 3
"winningPlan" : {
        "stage" : "FETCH",
```

When run with an index, the query scanned 3 index entries and 3 documents to return 3 matching documents. Without the index, to return the 3 matching documents, the query had to scan the whole collection, scanning 10 documents.

Questions

Now having learned the basics of MongoDB it is time to try some for yourself.

The file `people.json` on Blackboard contains data about people who immigrated to Australia in the 1800's.

In a collection voyages, insert the data that is contained in `people.json`. (Pasting the data into the console is fine).

1. Write a query that gets all the people with an age less then 20.
2. Update the user with the `id` field of 592680 to have a `qsa_item_page` of 222.
3. Write a query that prints out ONLY the names and the `_id` values of each person and order them alphabetically with respect to name. For this question the documentation on these pages may be useful:
   https://docs.mongodb.com/v3.2/tutorial/project-fields-from-query-results/
   https://docs.mongodb.com/manual/reference/method/cursor.sort/
4. Create an index on the `age` field and rerun the query in Question 1 using the explain function to get the execution plan. Assuming that MongoDB created tree based index for the query, explain step by step the execution plan that was produced.