Introduction to MongoDB

Background

Increasingly, databases which do not use SQL are becoming popular. These are collectively known as "NoSQL" databases (see here for an explanation and discussion of the advantages). NoSQL databases typically store structured data (using JSON for instance) or whole objects. One of the commonly-used NoSQL databases is MongoDB.

MongoDB stores data in a JSON-based format which makes it very easy to use with JavaScript - and it works well with node.js. Furthermore, a JavaScript-like syntax is used to perform queries. Like MySQL, MongoDB operates as a ***database server*** running continuously in the background, waiting for requests to come in from clients. It operates on port 27017 by default. Also like MySQL, you need to use a ***client*** to send queries to the MongoDB server. A NodeJS script would be one example of such a client; another is the default client called ***mongo***, which is used from the command-line.

Using MongoDB

Mongo introductory exercise

Logon to Neptune with PuTTY and enter the following at the command-line to run the database client:

```
mongo
```

You will see a prompt like this:

MongoDB

Enter the following at the Mongo prompt:

```
db.getCollectionNames()
```

You should see the following appear:

```
[ "hits", "system.indexes" ]
```

A ***collection*** is equivalent to a table in a relational database. It is a collection of individual records; you can see that in the MongoDB on Neptune, there is a collection called "hits". Now type in:

```
db.hits.find()
```

You should now see all the individual records in the "hits" collection, in JavaScript object syntax. Note how each object has a series of properties, "song", "artist" and so on, similar to the column names in a relational database.

Hopefully you can see how MongoDB syntax is very closely related to that of JavaScript. ***db*** can be thought of as similar to a JavaScript object representing the database as a whole, and ***getCollectionNames()*** is a method which returns an array of all the collections in the database.

Likewise, ***find()*** can be considered a method of a collection which returns all the records in that collection. So ***db.hits.find()*** will return all the records in the ***hits*** collection.

Retrieving specific records

We probably want to retrieve only those records which match particular criteria. For example, we might want to search for all songs by a particular artist. Here is how to do this. Enter the following at the mongo prompt:

```
db.hits.find({artist : "Oasis"})
```

This will return all songs by Oasis. Note how we use syntax similar to a JavaScript object as a parameter to ***find()*** to describe the search.

Next enter the following:

```
db.hits.find({artist:"Oasis", song:"Some Might Say"})
```

This will return Some Might Say by Oasis. Note how we search on artist AND song by combining the two criteria in a single JavaScript-like "object".

You can also do "less than" and "greater than" queries. Here is an example:

```
db.hits.find({year: { $lt:1962}});
```

This will find all songs released before 1962. Note how the query is now a further object (***{$lt:2000}***) rather than a single value such as "Oasis". "$lt" is an operator which means "less than". Note how MongoDB makes use of the syntax of JavaScript objects to perform more complex queries. Another example:

```
db.hits.find({year: { $lt: 1990, $gt: 1980}})
```

will find songs released from 1981-1989 inclusive. The similar

```
db.hits.find({year: { $lte: 1990, $gte: 1980}})
```

will find songs released from 1980-1990 inclusive ($lte meaning "less than or equal to" and $gte meaning "greater than or equal to"). Again, note the similar approach to the simple searches above: we combine conditions by putting them in a single JavaScript-like "object".

"or" queries

"or" queries are a bit more complex than "and". The simplest way to do them, when you are searching for one of a range of discrete values for a particular property, is to use the $in operator, e.g:

```
db.hits.find({artist: {$in: ["Oasis", "Madonna"]}});
```

will find all hits by either Oasis or Madonna. For more complex queries you have to use the "$or" operator, e.g:

```
db.hits.find({ $or: [ {artist: "Madonna"}, {song:"Some
```

This will find records where EITHER the artist is Madonna OR the song is Some Might Say. Note how both $in and $or take an array: for $in this is an array of possible values whereas for $or it is an array of conditions, at least one of which must be matched.

Pattern matching - the equivalent of SQL "LIKE"

Pattern matching can also be done. This also follows JavaScript syntax: in JavaScript pattern matching (using regular expressions) can be done using the slash /. So for example:

```
db.hits.find({song: /The/})
```

will find all songs with "The" in the name, while

```
db.hits.find({song: /^The/})
```

will find all songs beginning with "The" (remember from the REST topic that ^ indicates the start of a string in regular expressions).

Updating the database

Setting up your own individual MongoDB database

We will now look at how to update a MongoDB database, for example add new records or modify existing ones. To do this, you should first set up your own database, so that any changes made by you don't interfere with other students. Quit mongo by entering

```
exit
```

to return you to the Linux shell prompt. You now need to import the database. Enter:

```
mongoimport --db yourusername --collection hits --drop
```

replacing "yourusername" by your actual username. This imports the hits collection into your own database.

Now re-enter the mongo client:

```
mongo
```

and enter:

```
use yourusername
```

(again replace "yourusername" by your actual username). This will switch to your own database from the default one, "test".

To test it worked, enter:

```
db.hits.find()
```

and all the hits should appear.

Adding records

Inserting new records is quite intuitive in MongoDB. You use the ***insert()*** method, which takes a JavaScript-like "object" representing the new song. For example:

```
db.hits.insert({artist:"Oasis", song: "Wonderwall", yea
```

Hopefully this should be obvious. The only thing that needs commenting on is the "_id" property. This is equivalent to the primary key in a relational database and ***must have a unique value***. By default, it is given a complex value by MongoDB if you do not specify an explicit value for it. However, giving it a simple value (such as an integer) will simplify things in some cases, such as giving each song a simple unique numeric ID which can be used to reference it in a web application. See the documentation.

Also note that unlike a relational database, the properties can be completely freeform. For example, you could do

```
db.hits.insert({author:"Owen Jones"});
```

and even though the record has nothing to do with music, it would be accepted.

Numeric values

It is important that any numeric values are inserted WITHOUT quotes. If quotes are used, they will be stored as a string which means that you will be unable to perform arithmetic operations on them, such as increase them by one or use less-than and greater-than operators. So for example it should be:

```
db.hits.insert({song:"Wonderwall", quantity:100})
```

and not

```
db.hits.insert({"song":"Wonderwall", quantity:"100"})
```

Exercise

Add a song of your own choice to the database, then search for it to prove that it has been added.

Updating records

You can also update records with ***update()***. Try this first (the find commands are to illustrate how the update happens):

```
db.hits.find({_id:1})
db.hits.update({_id: 1}, {quantity: 200})
db.hits.find({_id:1})
```

What do you see? Is this what you want?

Now try this:

```
db.hits.find({_id:2})
db.hits.update({_id: 2}, {$set:{quantity: 200}})
db.hits.find({_id:2})
```

This will update the song with the _id of 2 so that its quantity is now 200. Note:

- ***update()*** takes two parameters. The first is the set of search criteria to find a given record (here, we are searching for the record with the _id of 2) and the second is the update instructions.

- The update instructions (once again) have JavaScript object-like syntax. They consist of the instruction ***$set*** (i.e. we're setting given attributes to given values) followed by a list of new values to set.

Another example:

```
db.hits.update({_id: 2}, {$set:{quantity: 200, price:0
```

This will reset the quantity ***and*** the price of the song with the _id of 2.

```
db.hits.update({song: "Wonderwall", artist:"Oasis"}, {$
```

This will set the quantity of Wonderwall by Oasis to 200.

Be careful to use $set!

A possible mistake, illustrated above, is to leave out the ***$set*** instruction when updating records. For example, this is valid syntax but it will have an unexpected effect:

```
db.hits.update({_id: 1}, {quantity: 200})
```

Without the ***$set*** instruction, it will reset the details of the song with the _id of 1 so that it has a quantity of 200 and ***no other properties!*** In other words, update() without $set will ***clear out all the existing properties of a record!***

Multiple matches

By default update() will only update the first record. To tell it to match more than one, you have to pass 'true' as the fourth parameter e.g this one which might be to correct a typo:

```
db.hits.update({artist: "Betles"}, {$set:{artist: "Beat
```

(The third parameter, false here, is 'upsert', which if true will insert the record if it does not exist, i.e. an 'upsert' operation)

Another useful instruction: $inc

Another useful instruction in update queries is ***$inc***. This increases numerical properties by the specified amount. For example:

```
db.hits.update({_id:2}, {$inc:{quantity:1}})
```

will increase the quantity by 1, while

```
db.hits.update({_id:2}, {$inc:{"quantity":-1}})
```

will decrease it by 1. (Note that there is no ***$dec*** instruction!)

Deleting records

You can also ***delete*** records. This is done with ***remove***, e.g:

```
db.hits.remove({_id:2});
```

will remove the song with the _id of 2, or

```
db.hits.remove({song:"testing", artist:"test"});
```

will remove the song "testing" by the artist "test" from the hits collection.

Exercise

Here is some data about students:

_id	name	course	mark
1	Tom	Software Engineering	90
2	Jeremy	Medicine	5
3	Laura	Computing	60
4	Nigel	Computing	30
5	Sarah	Web Development	80

- Add the records above to a collection called ***students***.
- Find all students studying Computing.
- Find all students studying Computing with a mark of 60.
- Find all students studying Computing OR Software Engineering.
- Find all students who passed (got a mark of 40 or more).
- Update Tom's mark to 95.
- The Computing course is changing its name to Computer Studies. Update all Computing students so that they are now on Computer Studies.
- Remove all students with a mark of 10 or less.

Note that you do NOT have to create a collection (using the equivalent of CREATE TABLE): you can just start adding the records to the collection straight away, and if the collection does not exist, it will be created.