

## Homework 2: MongoDB and Redis

**Instructions:** There are several tasks in this homework (not all will result in a deliverable). 3 tasks are explained in this part. The last tasks are explained in part 2 of the homework which will be uploaded soon.

- The first task is to either install MongoDB or get it to run on one of the PKI student computers.
- The second task is to go through inserting, deleting, updating databases, collections, and documents.
- The third task covers many different query operations. I do expect that you have queried data before. If you are still familiar with SQL and how to manipulate data in a relational database, you will find many similarities here and you should have no problems in understanding the queries. Writing the queries, of course, is more difficult as the syntax is different. I do advise you to try out all the examples as you go through the different tasks. It will help you with the deliverable.

**Deliverables:** Explained in part 2.

## Introduction to MongoDB

MongoDB is an open source document-oriented DBMS database. It is written in C++ and focuses on high performance, high availability, and easy scalability. (MongoDB Inc., 2008-2015) MongoDB has its own a rich, ad-hoc query language and does not support SQL. It is also used to store data for desired high performance applications and during cases of load increases for example when adding more computers to the system, the performance can still be retained. (www.w3resource.com, 2015 )

### History

Mongodb was first developed by MongoDB Inc. in October 2007. In 2009, MongoDB started offering commercial support and other services and shifted to open source development model. Since then, MongoDB has been adopted as backend software by a number of major websites and services, including Craigslist, eBay, and Foursquare among others (Wikipedia, 2015). As of July 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores. (Db-engines.com, 2015)

### Key Features

The key features of Mongo Db are as follows:

1. **Flexibility:** MongoDB stores data in JSON documents which is then serialized to BSON. JSON allows for rich data models that can be seamlessly mapped to native programming language types. The dynamic schema also allows for easier evolution of data models than with a system that enforces strict schemas like RDBMS. (MongoDB Inc., 2008-2015). BSON basically stands for Binary JSON. It is a binary-encoded serialization of JSON-like documents.
2. **Power:** MongoDB provides feature that gives a breadth of functionalities similar to that used in RDBMS and couples it with flexibility and scaling capabilities in that of a non- relational model. Some of feature similar to RDBMS are dynamic queries, rich updates, upserts (updating if documents exists and inserting if it does not exist), secondary indexes and easy aggregation (MongoDB Inc., 2008-2015).
3. **Speed/Scaling:** Scaling a database is easier in MongoDB. If there is increase in the load in terms of more processing power, more storage space, the load can be distributed. This can be achieved by distributing the load to other nodes. (This means adding more machines handle the load.) (www.w3resource.com, 2015 ). This ability is called autosharding. This ability allows increase in capacity without any downtime. This feature becomes very important on the web especially when load and spike suddenly and bring down the website resulting in extended maintenance cost (MongoDB Inc., 2008-2015). MongoDB keeps related data together in documents. This allows for faster querying than in the relational databases. In relational database related data is separated into multiple tables which is then joined later for querying purpose (MongoDB Inc., 2008-2015).
4. **Ease of use** “MongoDB is very easy to install, configure, maintain and use. It tries to automatically do the “Right thing” whenever possible and provides very few configuration options. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.” (MongoDB Inc., 2008-2015)

### MongoDB: Databases, Schemas and Tables

- **Databases:** The database of MongoDB consists of document oriented DBMS with JSON like objects. It does not support joins nor transaction like in RDBMS but does feature secondary indexes, query language, atomic writes on per document level and fully consistent read (www.w3resource.com, 2015 ).
- **Schemas:** MongoDB uses dynamic schemas which allows users to create collections (Collection: A grouping of MongoDB documents. A collection is the equivalent of an RDBMS table) (MongoDB Inc., 2008-2015). The creation of collection can be done without defining the structure (the fields or the types of their values, of the documents). The structure of the documents can be simply changed by addition of new fields or deleting existing ones. It is important to note that the documents contained in a collection need a unique set of fields (www.w3resource.com, 2015 ).

- Tables: MongoDB stores its data in collections which is equivalent to tables in RDBMS. A collection holds one or more documents. These documents are equivalent to a record or a row in a relational database table. Each document has one or more fields which is equivalent to a column in a relational database table (www.w3resource.com, 2015 ).

The document below should further shows the difference:

RDBMS	MongoDB
<b>Database</b>	Database
<b>Table</b>	Collection
<b>Tuple/Row</b>	Document
<b>column</b>	Field
<b>Table Join</b>	Embedded Documents
<b>Primary Key</b>	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
<b>Mysqld/Oracle</b>	mongod
<b>mysql/sqlplus</b>	mongo

- (www.tutorialspoint.com, 2015)

### Basics of MongoDB (Terminologies)

- Database: The physical container for collections of documents are called database. A MongoDB server may typically have multiple databases each with its own set of files on the file system.
- Collection: As explained above, collection is a group of MongoDB documents which can be considered as equivalent of an RDBMS table. Collections in a database do not enforce any schema.
- Document: A document is a set of key-value pairs. Key-value pairs can be explained by the following:
  - FName: Hansen,
  - LName: Olaf
  - Age: 23
  - Here FName, LName, Age are keys and Hansen, Olaf and 23 are respective values.

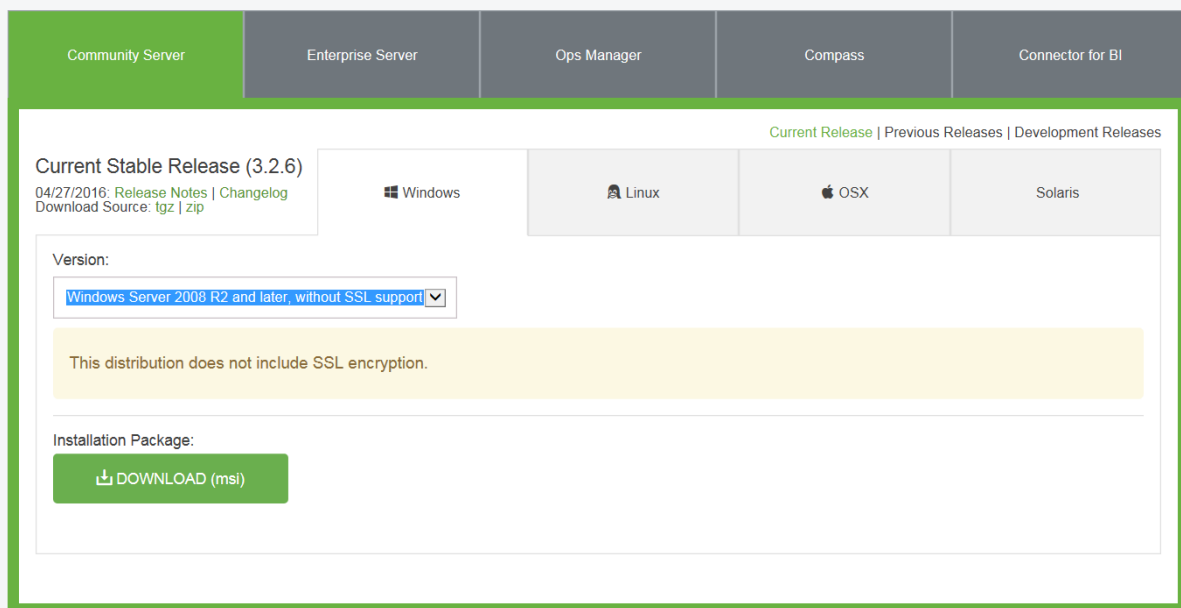
A documents within a single collection may have different fields. A collection usually contains documents which are similar or related. They are also dynamic in nature, meaning their structure can vary. So, the structures of documents residing in the same collection do not necessarily require to be same (www.tutorialspoint.com, 2015).

## Task 1: Installing and Starting MongoDB

### Installation of the MongoDB Community Edition (Optional, if you use PKI Student Computers)

The following describes the installation of MongoDB on a Windows 7 machine, 64-bit (release 3.2.6). The instructions for download are also available at <https://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/>. If you install on another operating system or another version of Windows, please follow the installation instructions for your operating system.

1. Go to <https://www.mongodb.org/downloads#production> and download the current stable release for your Windows system. For me this is: Windows 64-bit 2008 R2. I choose without SSL support (encryption), but I don't think that matters for our purposes.

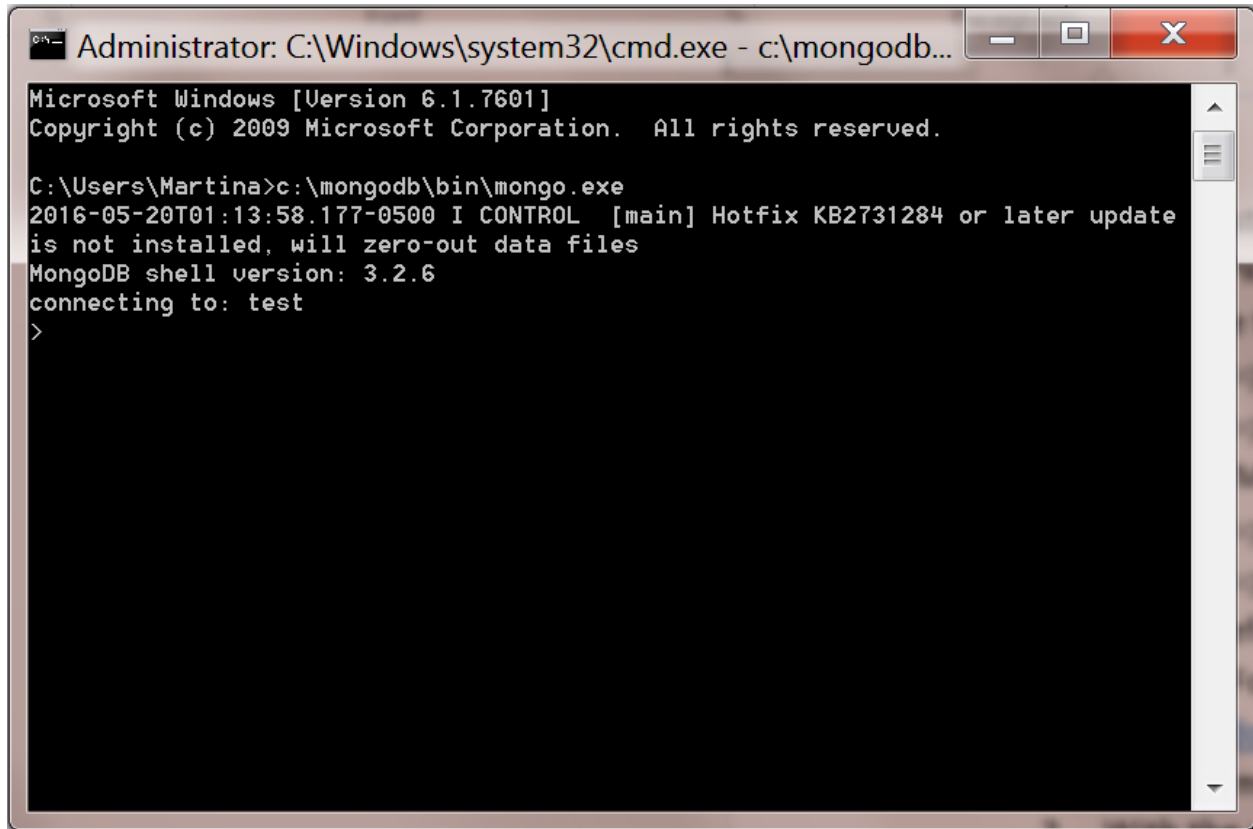


2. Double-click on the .msi file and follow the installation process. The changes I made to the default was to choose a Customer install, all components, and installation location `c:\mongodb`.
3. In your windows explorer, you should now see a new directory "mongodb" under `c:/`. You can of course install to another location.

### Starting MongoDB

4. If you installed MongoDB on your computer, navigate to the installation directory. If you are using a PKI Student computer, open App2 -> MongoDB in a Windows Explorer. Make sure that you have the data directory installed. If not, make a new data directory `c:\mongodb\data\db` (if you installed MongoDB yourself that would be `c:\mongodb\data\db`).
5. To start MongoDB server, navigate to the folder you installed mongodb to. For me, that is `C:\mongodb\bin`. Run `mongod.exe`. You will see a bunch of output and once you see `waiting for connections` you should be ready. If you see a Security Alert dialog box, please refer to the installation guide for more information (Run MongoDB Community Edition, step 2). For more information about security, refer to <https://docs.mongodb.org/manual/security/>. Btw. You can stop the server by pressing Ctrl+C in the command prompt, but for now, please don't. **Leave the window open.**

6. With the server running (previous step)m we need to connect to MongoDB through the mongo.exe shell. Open another Command prompt cmd.exe and execute mongo.exe by entering `c:\mongodb\bin\mongo.exe`.



```
Administrator: C:\Windows\system32\cmd.exe - c:\mongodb...
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Martina>c:\mongodb\bin\mongo.exe
2016-05-20T01:13:58.177-0500 I CONTROL [main] Hotfix KB2731284 or later update
is not installed, will zero-out data files
MongoDB shell version: 3.2.6
connecting to: test
>
```

## Task 2: Creating and dropping databases, creating collections, inserting, updating, and deleting

### Creating or switching to a database

You can have multiple databases. For this task, we use a test database. In your mongo shell (step 6 in Task 1), type in `use users` to switch to the users database. “use” will create a new database if you do not already have one. The syntax for creating or switching to a database is `use <name of database>`.

The query and the resulting message should be as follows:

```
>use users
switched to db users
```

If you want to check the current database being used we can use the command `<db>`

```
>db
users
```

If you want to check your databases list, then use the command `show dbs`.

```
>show dbs
local          0.000GB
```

Your created database (users) is may not be present in the list. To do so you have to insert at least one document into it. The following inserts a document into the profiles collection in the users database.

```
>db.profiles.insert({"_id" : 1, "name" : "Joe"})
WriteResult({ "nInserted" : 1 })
>show dbs
local      0.000GB
users      0.000GB
```

In mongodb default database is test. If you didn't create any database then collections will be stored in test database.

## Dropping a Database

MongoDB **db.dropDatabase()** command is used to drop a existing database.

```
db.dropDatabase()
```

The above code will delete the database that has currently active/ currently in use. If no database has been selected then default 'test' database will be deleted. It is always good practice to check the list of available database and using **<show dbs>**, or to see which database is currently being used before using **<db>** command, before dropping the database. (tutorialspoint, 2015)

If you want to delete new database users, then **dropDatabase()** command and message would be as follows:

```
>use users
Switched to db users
>db.dropDatabase()
{ "dropped" : "users", "ok" : 1 }
```

## MongoDB- Creating a Collection

MongoDB creates collection automatically, when you insert some document into a database (tutorialspoint, 2015). You can create a collection explicitly with the **createCollection()** command is as follows

```
db.createCollection(name, options)
```

In the syntax, **name** would be name of collection to be created. **Options** is a document and used to specify configuration such as max size or documents of the collection. The options parameter is optional.

To create a collection using basic syntax of **createCollection()** is as follows:

```
>use users
switched to db users

>db.createCollection("profiles")
{"ok" : 1 }
```

To check the collection created we can use **<show collections>** command

```
>show collections
profiles
```

We usually, don't need to create a collection in MongoDB. Collections will automatically be created when inserting a document.

```
>db.anotherCollection.insert({"test" : "test"})
WriteResult({ "nInserted" : 1 })
>show collections
anotherCollection
profiles
```

You can remove a collection with **db.<name of collection>.drop()**.

```
> db.anotherCollection.drop()
true
> show collections
Profiles
```

## Inserting, updating, and deleting a Document

MongoDB's write operations include inserting, updating, and deleting a document. The operations only affect one document, the write operations are atomic in nature.

The syntax for inserting a document is `db.<name of collection>.insert ( <document > )`. The document has to be in JSON format. Every document will have a unique identifier, a primary key, in order to identify a document. The unique identifier is either specified explicitly with `"_id" : <objectID>` or will be added by MongoDB if not `"_id"` field is defined. The `_id` value needs to be unique for the collection. Have a look at the following code: we first insert a new user with the unique identifier 1. Then, we try to insert a new user with the same id which does not work. For the third user, we do not specify an `_ID` field and MongoDB adds a unique identifier. The next command inserts two documents at the same time.<sup>1</sup> The last command `find()` displays all documents in the `profiles` collection.

```
> db.profiles.insert ( { "_id" : 1, "name" : "Joe" } )
WriteResult({ "nInserted" : 1 })
> db.profiles.insert ( { "_id" : 1, "name" : "Sue" } )
WriteResult({
  "nInserted" : 0,
  "writeError" : {
    "code" : 11000,
    "errmsg" : "E11000 duplicate key error collection: users.profile
s index: _id_ dup key: { : 1.0 }"
  }
})
> db.profiles.insert( { "name" : "Time" } )
WriteResult({ "nInserted" : 1 })
> db.profiles.insert ( [ { "_id" : 2, "name" : "Rob" }, { "_id" : 3, "name" : "Bob" } ] )
> db.profiles.find()
{ "_id" : 1, "name" : "Joe" }
{ "_id" : ObjectId("56b106a6e785d43f950c7ae5"), "name" : "Time" }
{ "_id" : 2, "name" : "Rob" }
{ "_id" : 3, "name" : "Bob" }
```

---

<sup>1</sup> With release 3.2, MongoDB has two other insert commands `insertOne()` which adds one document and `insertMany()` which add multiple documents.

A document can be updated with the command `db.<name of collection>.update(<update criteria>, <update action>, <update option>)`.

```
db.users.update(           ← collection
  { age: { $gt: 18 } },    ← update criteria
  { $set: { status: "A" } }, ← update action
  { multi: true }          ← update option
)
```

Image: (<https://docs.mongodb.org/manual/core/write-operations-introduction/#update>)

The above update operation finds all the documents in users collection with age greater than 18. Then based on the result sets the `status` of those documents to A. The update option `multi` allows to update multiple documents

For our collection, the following code corrects the name Time to Tim. Without the `multi:true` specified, it will update the first entry in the collection.

```
> db.profiles.update( { name: "Time" }, { $set: {name: "Tim" } })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

You can also use update to add or replace a whole document with the option `upsert:true`.

```
> db.profiles.update( { "name" : "Joe" }, { "name" : "Bob", "age" : 23 }, { upsert:
true })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The command above will replace the document matching Joe with the specified document name Bob and age 23. If no document matches the name Joe, then a new document will be entered.<sup>2</sup>

A document can be removed with the command `db.<name of collection>.remove ( {<remove criteria>} )`. The following removes all documents if the name contains an o. To remove only the first matching document, add `,1` or `,true` after the remove criteria.

```
> db.profiles.remove ( {name:/o/i} )
WriteResult({ "nRemoved" : 2 })
```

If you do not specify a criteria, all documents from the collection will be removed. Any indexes in the collection will not be removed. To remove everything, including indexes, use the drop method.

---

<sup>2</sup> With release 3.2 there are other options such as `updateOne()`, `updateMany()`, and `replaceOne()`. Please see <https://docs.mongodb.org/manual/core/write-operations-introduction/>.



## Task 3: Querying documents

Note: If you forget to close braces or brackets correctly, you may see ... as a result. This indicates a multi-line operation. You can use Ctrl + D to exit the query. More information about the Mongo Shell here:

<http://docs.mongodb.org/manual/mongo/> -> The mongo Shell.

The basic read operation is `db.<name of collection>.find (<query criteria>, <projection>) .<modifier>`. The query criteria specifies which documents to find. The projection specifies which fields to return. The find method returns a cursor. The modifier also allows to limit, skip, and sort the order.

Before we continue, let's load some data into MongoDB that we can query. Please download from Blackboard and then copy the file `productDataForHomework2.txt` into your directory `c : /mongodb/bin/`. Open a new command prompt `cmd.exe` and navigate to the `c : /mongodb/bin/` folder. We can use the important tool from MongoDB `mongoimport.exe` to import csv, tsv, and json files. The `.txt` file contains 52 documents in JSON format. You can open it and have a look if you want.

Type in `mongoimport -d store -c products --file productDataForHomework2.txt` . The output should indicate that 52 documents have been uploaded. In your mongo shell type in `use store`, and `show collections`, to see whether you have the products collection.

Try the basic method `db.products.find()` . You will see that only a few documents are displayed at the time, and you are prompted to enter `it` to continue to iterate through the collection. If you append `.pretty()` to the find method, the output will be formatted in a nicer way `db.products.find().pretty()` . The `find()` in this basic form is the equivalent to `SELECT * FROM products`. If you want to only display the first document `db.products.findOne()` . Do this and have a look at some basic fields. We have an item with a name and other attributes such as category and price. We have an objects called details that gives us more details. We also have an array called stock that shows us how many items are in stock, divided by color.

Let's go through some queries together.

### Projection – Select fields (SELECT item\_name FROM products;)

We can limit the fields to display. Try the following `db.products.find( {}, {item_name: 1} )` . This will display only the `item_name` and the `_id` field. The `_id` field is included by default.

If you want to hide the `_id` field, you have to set `_id: 0` `db.products.find( {}, {item_name: 1, _id:0} )` . The query criteria `{}` is left empty and this will retrieve all records.

### Sort result (SELECT item\_name FROM products ORDER BY item\_name ASC;)

We can sort the `item_name` list by including `.sort()` at the end of the query `db.products.find( {}, {item_name: 1, _id:0} ).sort( {item_name: 1} )` . This sorts by `item_name` ascending. `-1` sorts descending.

### Top 5 (SELECT TOP 5 item\_name FROM products ORDER BY item\_name ASC;)

In order to only display the top X documents from a collection, you can use `limit(X)`. `db.products.find( {}, {item_name: 1, _id:0} ).sort( {item_name: 1} ).limit(5)` will only show the first 5 documents.

### Selection – Select documents matching a criteria (SELECT item\_name FROM products WHERE category = "Laptop";)

Let's display item name for all documents of one category Laptop with `db.products.find ( { category: "Laptop" } , {item_name: 1, _id:0 } )` . Only documents where the value of category matches the text (case-sensitive) will be displayed.

## NOT or <>

We can invert that selection, by including `$ne` before the value. This is like the NOT in SQL `db.products.find( { category: {$ne:"Laptop"} } , {item_name: 1, _id:0 } )`. This will display all documents excluding the documents with category Laptop.

## LIKE (WHERE item\_name LIKE '%phone%')

It is possible to use regular expression. For example, `db.products.find( { item_name : {$regex:/phone/i} } , {item_name:1, _id:0 } )` displays all documents where the item\_name contains the text phone, i means that we ignore the case.

- `/phone/i` -> `%phone%`
- `/^phone/i` -> `phone%` (no documents in our example)
- `/phone$/i` -> `%phone` (will not show Windows Phones)

There is another modifier skip. It can skip the first X documents. For example, try first `db.products.find( { item_name : {$regex:/phone$/i} } , {item_name:1, _id:0 } )` and then `db.products.find( { item_name : {$regex:/phone$/i} } , {item_name:1, _id:0 } ).skip(3)`.

`<, <=, >, >=`

`$lt` displays documents that are less than a value, and `$lte` displays documents less than or equal to a value. Try out `db.products.find({ price:{$lt:500} } , {item_name:1, _id:0, price: 1 } )`. Substitute `$lt` with `$lte`, `$gt`, and `$gte` to see what it does.

You can combine them to search for ranges, `db.products.find({ price:{$gt: 200, $lt:500} } , {item_name:1, _id:0, price: 1} )` searches for prices between 200 and 500 (excluding 200 and 500).

## DISTINCT

If you want to get an array of unique values in a field, you can use `db.<name of collection>.distinct(field,query)`. `db.products.distinct("details.manufacturer")` gives you all manufacturer. `db.products.distinct("details.manufacturer", {category: "Laptop"})` returns all manufacturer for the Laptop category.

## IN (SELECT item\_name FROM products WHERE manufacturer IN ("Dell", "Sony");)

`IN` displays documents where the value of a field matches the values in a list. Try out `db.products.find( { "details.manufacturer" : {$in:["Dell", "Sony"]}} , {item_name: 1, _id: 0} )`. Please pay attention that details.manufacturer (a field in an object) needs to be set in double quotes. Also, after `$in` MongoDB expects an array, so don't forget the `[ ]`. Btw. `$nin` is the equivalent to NOT IN.

## Checking for the existence of a field

Since documents do not have all the same field, `$exists` is an operator that checks whether a document has a field. For example, `db.products.find( {"details.screen": {$exists:false}} , {item_name: 1, _id:0 } )` displays all documents that do not have a field called screen. Be careful that you get the name of the field right otherwise all documents will be displayed. And you may not even find out that you got the wrong result. For example, using screen instead of "details.screen" will display all documents. Screen is a field inside an objects and needs to be referenced with the document path "details.screen".

**And, or (SELECT item\_name, price FROM products WHERE manufacturer = 'Dell' AND price >= 300;)**

The \$and operator allows us to search for documents that match multiple values, e.g., `db.products.find( { $and: [ { "details.manufacturer": "Dell" }, { price: { $gte: 300 } } ] }, {"detail.manufacturer":1, item_name:1, price:1 _id:0})` displays 2 documents that match the value Dell for manufacturer with a price greater than 300. The basic syntax for \$and is `{ $and: [ {expression1}, {expression2}, ... {expression} ] }`.

\$or is like the OR operator in SQL (try it out by replacing the \$and in the query above).

**Counting documents (SELECT COUNT(\*) FROM products;)**

In order to count documents returned in a query, append `count()` to the method, e.g., `db.products.find().count()` simply counts all documents in a product. To count all products with manufacturer Dell, use `db.products.find( {"details.manufacturer" : "Dell"} ).count()`.

**Accessing nested fields**

We already saw that "details.manufacturer" gives us the field in an object. But what about arrays? To see whether a value in an array of strings is present, you can use `db.<name of collection>.find( { <name of array>: "string" } )`. We don't have such an example in our product list. But we do have an array called stocks that contains objects. In order to check an array of documents (or objects), you can use \$elemMatch. For example, the following gives you all documents that have a stock level of 10 in the color white `db.products.find( { stock: { $elemMatch: { color: "white", qty:10 } } }, { item_name:1, _id:0 } )`. If you leave of color: "white", it would give you all products that are in stock with quantity 10 disregarding the color. You can combine other operators, e.g., you can replace `qty:10` with `qty: {gt:10}` to get quantities larger than 10.

Okay, enough for now. Please refer to part 2 of the homework.