MONGODB

LAB GUIDE

MongoDB documentation: https://docs.mongodb.com/manual/tutorial/getting-started/

IMPORTANT NOTE: It is beneficial to save your queries in a file along with exercise numbers. Many exercises will build upon queries created in other exercises. It will thus save a lot of your time and effort.

1. **DROPPING COLLECTIONS AND DBS** : Find out how to delete all documents of a collection, drop a collection, and drop a database. Answer: Explore drop() method of collection, and dropDatabase() of database.

2. **DATA TYPES**: Create a moviesDB with a movies collection. The movie lists out year of release (int32), box office collection (this has to be stores as a long int as it can exceed maximum value of int32), a release date (a date object), and a timestamp when document as added.

   Use db.stats() to check average document size. Change data types for some value using an update query. See how it affects the size.

   Reference: https://docs.mongodb.com/manual/core/shell-types/. Apart from what's mentioned here, you can use new Timestamp() for Timestamp.

3. **CRUD OPERATIONS**: Create a moviesDB with an albums collection. The albums lists out year of release, music director, lyricist and a list of songs. Every song has details of singers (names), length of song (string).
   a) Insert 3 documents with at least 1 song entry each
   b) Update album data for an album. Change the name of the lyricist, and also an entirely replaced history entry.
   c) Find all albums released before 2019
   d) Remove all documents that have a particular music director / lyricist as value

4. **DB DESIGN**: We shall create a database for a blogging site. There are many blog posts (blog articles), each created by a user registered with the website. An article can be commented upon by registered users. Design a data model for this application.
   a) What are the entities in this application?
   b) What are the collections you would create?

c) What fields would be part of each collection?

d) Would you duplicate any fields from one collection in another (i.e. perform denormalization)? What advantages would that have?

e) Imagine how the DB would be used by the application. Would that determine any fields you would store in the collections?

Requirements:

a) User's name, age, gender, description about self, profilePic URL and interests must be captured.

b) Post title, text, creation date, author information must be captured.

c) Comments will have a title and text, creation date

Define the DB and define at least one collections with a schema (use db.createCollection() on users collection, say). Insert a few documents for each collection. Try both insertOne() and insertMany().

References:
https://docs.mongodb.com/manual/reference/bson-types/
https://docs.mongodb.com/manual/core/schema-validation/
https://www.mongodb.com/blog/post/json-schema-validation--checking-your-arrays

Write queries to do the following.

a) Retrieve all posts by a given author (assume user id is known)

b) Retrieve all users who have a particular interest, say "sports"

c) Retrieve some user document (say user with _id = 1). Retrieve all posts whose tags match some tag belonging to the user's interests.

d) Retrieve all comments for a post (assume post id is known)

e) Retrieve all comments for a post (assume slug is known). Try using both findOne(), and also find() + next()

f) Explore the toArray() method of the cursor returned by find(). Use it to convert the results to an array

g) Explore the forEach() method of the cursor returned by find(). Use it along with print() / printjson() to print all results.

h) Retrieve all comments by a user

i) Add a comment for a blog post

j) Update a post by setting a new field called votes to a post. It stores an array of users who upvoted the post.

k) Add a new interest for a user. How will you add a set of interests to the user in one go? Explore the docs for $addToSet.

l) Remove an interest for a user. How will you remove a set of interests from the user in one go? Explore the docs for $pull.

How would your answers change for a DB design that stored comments within Posts collection, and for one that stored Comments as a separate collection?

5. OPTIONS OBJECT IN INSERT

ORDERED INSERTS:

Syntax: { ordered: true | false }

Default value: { ordered: false}

Use insertMany() to insert the following 2 documents whose _id is specified.

[{ _id: 1, x: 1}, { _id: 2, x : 2 }]

Now use insertMany() to insert the following

[{ _id: 3, x: 3}, { _id: 2, x : 2 }, { _id: 4, x: 4}, { _id: 5, x: 5 }]

Do documents get inserted? If yes, which one?

While inserting use the options object - pass { ordered: false } as second argument. How does the behavior change?

WRITE CONCERN:

Syntax: { writeConcern: { w: number_of_acknowledgments, j: undefined | false } }

Default value: { writeConcern: {w: 1, j: undefined } }

NOTE: Here undefined means j is not specified as undefined, or not specified at all

Insert a few documents. Make sure it is acknowledged and written to journal. What does the response from server look like?

Insert another set of document. Do not wait for acknowledgment from server (how will you do this?). Do you see the insert id in the response?

6. **READ OPERATIONS**: find() and findOne()

   We shall understand the query operators here (those used in filter operation).

   Reference: https://docs.mongodb.com/manual/reference/operator/query/

   For this exercise, you will need to import the shows collection into some DB first. Then try the following.

   a) Comparison (i.e. relational) operators (including equality, i.e. exact match)
      i. Find shows that are less than one hour duration
      ii. Find shows with runtime between 30 and 60 minutes
      iii. Find shows with average rating at least 8
      iv. Find shows with rating between 8 and 9
      v. Find shows in horror genre
      vi. Find shows which have ONLY horror as their genre
   b) Special comparison operators that match within a list of values - $in, $nin (use these operators in your solution)
      i) Find shows that have one of Drama or Horror as genres in them
      ii) Find shows that are of type Animation or Reality
      iii) Find shows that have neither Drama nor Horror as genres in them
      iv) Find shows that are neither of type Animation, nor Reality
      v) Find shows that are not running on the network HBO, nor FOX
   c) Logical operators
      i) Find shows that have one of Drama or Horror as genres in them (try this without using $in)
      ii) Find shows that have neither Drama nor Horror as genres in them (try this with $not and $in
      iii) Find shows that have BOTH Drama and Horror as their genre
      iv) Find shows that have genre Drama but not Horror. Again, find shows that have genre Horror but not Horror.
      v) Find shows that **do not have at least one of** Drama or Horror as their genre (i.e. show could have Drama but not Horror, Horror but not Drama, or neither Drama not Horror)
   d) Element operators - $exists and $type
      i) Find shows that have a webChannel property

    ii) Find shows that have a webChannel with a country field within
    iii) Find shows which do not have a web channel (null)
    iv) Find shows which have a web channel (not null, but an object)
    v) Find shows which have a web channel (not null, but an object), but country for webChannel is null

e) Evaluation operators - $regex, $each
    i) Find all shows whose name contains the word "Last"
    ii) Repeat the above but with a "last" (all lowercase)
    iii) Repeat again with a case-sensitive match
    iv) Find all shows whose weight is more than 10 times their average rating
    v) Find all shows whose weight is less than or equal to 10 times their average rating value. Use $not rather than $lte.
    vi) Find all shows where 8% of the weight (0.08 * weight) is less than the average rating value. Make sure the average rating is not null.

f) Array query operators
    i) Find shows that have BOTH Drama and Horror as their genre. Use $all.
    ii) Find shows that are scheduled on both "Monday" and "Tuesday"

    Create a collection to store scores of students in various subjects. Include name of students in one field, and scores in another (between 0 – 100). Scores is an array of document, each with the subject name and score. Insert some documents so that some subjects are shared among students and some subjects are specific to a student (elective subjects). Now do the following.

    i) Find all students who have a score of more 90% in some subject and have taken up history
    ii) Find all students who have a score of more than 90% in history
    iii) Find all students who have taken up exactly 2 subjects

g) Projection operator - $, $slice

    NOTE: This operator is used on the projection object (second argument) and not the filter object (first argument) – it transforms arrays values that are projected.

    i) Find all students who have taken up history and project the matching subject details (i..e history)
    ii) Find all students who have taken up history and project the first 2 subjects they have in the document.

7. **SORT, SKIP & LIMIT (methods on cursor)**

sort(), skip() and limit() are called on cursors and also return cursors. The shell iterates through the cursor that is finally returned from the chain of calls to these methods – The order of the calls hence DOES NOT affect the results.

i) Retrieve all shows and sort by rating – first ascending and then descending

ii) Retrieve all shows and sort by rating first, and runtime when ratings are the same

iii) Retrieve all shows and sort by rating first, and runtime when ratings are the same**.** This time skip 20 documents and retrieve only 10 documents.

8. **UPDATE OPERATIONS:** updateOne() and updateMany()

We shall understand the update operators here (those used in update object – the second argument of update methods).

Reference: https://docs.mongodb.com/manual/reference/operator/update/
For this exercise too we will use the shows collection.

a) Field update operators - $inc, $min, $max, $mul, $rename, $set, $unset

i. Find all shows that are in English and have network -> country code as US, and set the language as English (US) instead. Also add a new field locale and set it to "en-US"

ii. Find the first show that has a weight of less than 40 and rating more than 7 and increase weight by 10. Also set a new field "criticsChoice" to true.

iii. Find the first show that has a weight of more than 80 and rating less than 6 and decrease weight by 10. Also set a new field "criticsChoice" to false.

iv. Find all shows that have a weight of less than 50 and rating more 7 and increase weight to maximum(50, current value).

v. Find all shows that have a weight of less than 60 and rating more 8 and multiply the weight by 1.333333

vi. Rename criticsChoice field as cc in all documents

vii. Remove field cc (criticsChoice) from all documents

viii. Try finding a document with a show name that does not exist (also use language : "English" while finding). Set the rating and genres for it. Use the upsert option and check that the upserted documented has fields that are part of the filter clause, as well as the update clause.

b) Array update operators - $, $push, $each, $sort, $slice, $pull, $pop, $addToSet

   i) Update all shows that have a scheduled screening on "Monday", and replace the item "Monday" with "monday" (lowercase). **Hint**: Use $ operator.
   ii) Update all shows with genre "Horror" by adding another genre "Supernatural"
   iii) Update all shows with genre "Horror" by adding 2 other genres "Supernatural" and "Spook" (you will need to use $each). Also explore how $sort and $slice can be used in this case.
   iv) Remove the genre Supernatural from the first matching document
   v) Remove the last genre from every document
   vi) Add genre Supernatural to all documents of genre Horror. However the Supernatural genre should not be added if it already exists as a genre in the document.

9. THE AGGREGATION FRAMEWORK

We shall understand the aggregate() method, aggregation stage operators, and other aggregation operators here.

References:
Aggregation pipelines stages:
https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/
Aggregation pipelines operators:
https://docs.mongodb.com/manual/reference/operator/aggregation/

For this exercise too we will use the shows collection.

a) Using $match to get a collection with a filtered set of documents
   i) Find all shows that have Drama as a genre
   ii) Find all shows on HBO
b) Using $group to group documents by field values and produce a new collection representing groups
   i) Group shows by the name of network they are running on, and also find the number of shows in each network

ii) Group shows by name of network and country they are running in, and also find the number of shows, and average runtime of shows in each group (network+country combination)

iii) Repeat the above query but create a new field called "stats" in the output documents. The "stats" field should have number of shows, and average runtime of shows for the group.

iv) Just like we can transform document to form new fields with subdocuments while projecting, we can also create a new array. Using the $push operator in $group stage, we can create a new array with one item per document in the group! This is a special feature of MongoDB with no equivalent in SQL (you can calculate only aggregate values like sum, average etc. there). Repeat the above exercise, and create an additional field "names" that is an array of names of all shows in the group.

v) Select all shows that are in English ("language" value), and then group them by their type. The output should have the names of the group in the type field, along with the number of shows in each group.

c) Using $match to filter grouped documents

SQL has a WHERE clause to filter records ***before*** grouping, and a HAVING clause ***after*** grouping (which filters the grouped records based on some aggregate value usually). In MongoDB, $match fulfils both these requirements.

i) Repeat the exercise grouping shows by network name and country. The final results should show only the grouped documents of networks that have at least 5 shows.

ii) Repeat the same but show only groups with average runtime at less than 50.

d) Using $sort to sort documents

i) Group shows by name of network and country they are running in, and also find the number of shows, and average runtime of shows in each group (network+country combination). Now sort them by the number of shows (group with highest number of shows appears first). If 2 networks are tied on number of shows, the one with the lower average runtime appears first.

ii) Repeat the above exercise, but make sure groups are formed only on shows that are "Running"

e) Using $project to create a new collection with only selected fields.

NOTE: Some useful operators would be $concat, $toDate, $year, $convert

i) Find the name, network name, schedule and runtime of all shows

ii) Modify the above query so that the network name is reported along with the network's country code like so – "name (code)", i.e. like "HBO (US)"

iii) Repeat the above query, but add premiered to the list of fields. However it should be converted to a Date object. Use $toDate.

iv) Repeat the above query using $convert instead of $toDate.

v) Repeat the above query, but premiered should now be an object with fields year, month and date when the show was premiered (use $year, $month, $dayOfMonth – you may also use $dateToParts).

   NOTE: You can make use of the fact that there can be multiple stages of the same kind, for example you can use 2 project stages in the pipeline.

vi) Just like we can transform document to form new fields with subdocuments while projecting, we can also create a new array. Repeat the above query but set premiered as an array with the 3 parts of the date as items within.

vii) We can use $size to get the size of any array. Use this to find name and number of days on which a show is aired for each show.

viii) We can use $slice to project only a portion of an array. Modify the above query, to use $slice to additionally find the first 2 days on which a show is aired.

   NOTE: You may need to add a $match stage to filter out documents that may not have the schedule days, or it exists but is not an array

ix) We can use $filter to choose items of an array based on a condition. Modify the above query to include the schedule days in the projected array, only if it airs on a weekend. The $in operator is used differently in the pipeline stage (not like comparison operator in queries/$match). Check https://docs.mongodb.com/manual/reference/operator/aggregation/in/ for how to use it

   NOTE: The above query can definitely be simplified using a $match to obtain shows that air only on weekends (using $in: ['Sunday', 'Saturday'] in $match instead)

f) Using $unwind creates a set of documents in place of a single document, using the different items in an array-valued field of the original document.

   i) Group shows by network name and country and create a new field "genres" that has all the genres of all the shows in the group (you will need to $unwind, then $group). Try both $push, as well as $addToSet when defining the genres array in $group stage.

ii) Sort the above results in descending order of number of genres in a network. This is not straight-forward in MongoDB.

    Hint: One way you can do this is to add a $project phase after $group, that adds a "numGenres" field, setting it to the length of genres array from $group stage. Results can then be sorted based on numGenres.

g) Using $out creates a new collection
   i) Use $out to create a new collection called "networkGenres" that has all genres of a TV network (result of $unwind section exercise i).
   ii) Again use $out to create a new collection called "networkStats" that has some statistics of a TV network (result of $group section exercise ii/iii/iv or v).

h) Use $lookup to join the networkGenres and networkStats tables.

   NOTE: Here each network has only one document in each collection. In general, there can be multiple documents from one collection that match another collection.

10. INDEXING

1. Find all shows with weight = 90. Use explain() to find
   a. execution plan
   b. rejected plans (if any)
   c. execution time
   d. number of documents examined
   e. number of keys examined
   f. number of results as a percentage of documents examined
2. Repeat the above exercise for query finding shows with weight > 90.
3. Find all indexes on shows collection using getIndexes(). Are there any present? What are their names?
4. Add an index on weight (use the collection's createIndex() method). Repeat the above queries and note the results.

   NOTE: The execution time may actually increase as this is a small data set. For a real world data set with thousands, if not millions of documents, the query will run faster when indexed.
5. Create a **compound index** on weight and average rating (weight comes first, and average rating next). Use the name option to specify name of the index. Repeat the above exercise. What difference do you observe? Note especially if there are any rejected plans.

6. Query for all shows that have average rating at least 9. What is the winning plan? Did the compound index help? Why, or why not?
7. Given we have 2 indexes (apart from the one automatically created on _id), is any of these indexes redundant? Why? Drop the one that is redundant. Use the collection's dropIndex() method.
8. A **unique index** on a field prevents creation of 2 documents that have the same value for the field. This functions like the SQL equivalent of a unique constraint on a column. Try creating a unique index on the name field for the shows collection (you need to specify { unique: true } in the second arguments which specify the index options). Are you able to create it? Why, or why not? Try creating another index on (name, type). Try inserting a show with an existing name, and type combination and verify the operations fails.
9. A partial index is an index created only on a subset of documents. It is useful when a particular subset of documents is fetched repeatedly (eg. weight > 90 / network.name = 'HBO'). You can use partialFilterExpression option to specify the filter criteria for choosing documents to be indexed. Create a compound partial index for weight > 90 and network.name = 'HBO'. What is the query plan when you query for shows with weight between 95 – 100, and on HBO network?
10. Suppose a unique index is set on a field. Then 2 documents cannot even have the field missing in them (they are considered having the same "value" on that field). We can overcome this problem of unique indexes by making the index a partial index using partial filter expression – { field : { $exists: true } }. Create a unique index on the url that allows multiple documents with the url field missing.
11. If the data returned by a query (i.e. projection) is fully part of the index (i.e. covered by the index), then the data from the index is sufficient, and the documents are not examined. This makes such queries very fast. Create a **compound index** on name, weight and average rating (name comes first, weight comes second, and average rating last). Find all shows with name having "Last" as a substring, and weight > 80. Select only any of the 3 fields in the index – set _id: 0 too. How does this query execute? Does it scan the shows collection at all? What is totalDocsExamined?
12. $regex is not an efficient way to search for text. We can create text index (only one allowed per collection) - many fields can be part of the text index though. The stopwords like 'a', 'in' 'the' are removed and the words are stored in an array in the index. Try to create a text index on the description field for shows.

Since the documents have a language field, this is assumed to be the language of the text and the index will not be created if you try as usual. You can add these options to overcome the problem

default_language: "en", language_override: "en"

Reference: https://stackoverflow.com/questions/23815024/mongodb-text-index-error-language-override-not-supported/24365657

Do a text search for the word "Last" ({ $text: { $search: "Last" } }). Make it a case-sensitive search using $caseSensitive: true ({ $text: { $search: "Last", $caseSensitive: true } })

13. Searching for phrases – Phrases have to be included in quotes. Search for "last man"and "\"last man\"", and see the difference.
14. Searching for absence of value – You can search for absence of a word by adding a '-' before it. Search for shows with *last* but without *man*.
15. MongoDB ranks the match based on weights assigned to fields that are part of the index, and also the degree of match (whether 1 word matched or more, how many times the match was found etc.). It calculates a textScore (meta information) – we can then sort based on textScore.

    We can add score to the projection this way and set it to the textScore calculated this way.

    { score : { $meta: "textScore" } }

    Go ahead and sort based on this score now.

11. Using MongoDB in a Node.js Application

Make sure Node.js has been installed – else download and install from https://nodejs.org/en/

1. Create a folder for the app and open the command prompt / terminal in this folder.
2. Create the package.json file

   npm init -y
3. Next install MongoDB driver for Node.js and Express.js (web application framework)

   npm i mongodb express

Reference: http://mongodb.github.io/node-mongodb-native/3.5/installation-guide/installation-guide/

4. Application shall be created and discussed in class.

   Here is a tutorial for using MongoDB in a Node.js app –

   References:

   http://mongodb.github.io/node-mongodb-native/3.5/tutorials/main/

   http://expressjs.com/

5. Connecting to MongoDB server

   Reference: http://mongodb.github.io/node-mongodb-native/3.5/tutorials/connect/

   Changes:

```
const dbName = 'showsDB';


const client = new MongoClient(url, {
   useUnifiedTopology: true
});
```

   Note: General form of the connections string is
   mongodb://[username:password@]host1[:port1][,...hostN[:portN]][/[database][?options]]
   Reference: https://docs.mongodb.com/manual/reference/connection-string/

6. For creating collections and specifying schema through the application you can refer this page. We shall however only access the existing shows collection.

   Reference:

   http://mongodb.github.io/node-mongodb-native/3.5/tutorials/collections/

7. CRUD operations

   Reference: http://mongodb.github.io/node-mongodb-native/3.5/tutorials/crud/

   Import necessary JSON data

```
const oneShow = require( '../data/sample-one.json' );
const manyShows = require( '../data/sample-many.json' );
```

   Change the name of the collection and insert oneShow and manyShows

```
// Insert a single document
db.collection('shows').insertOne( oneShow, function(err, r) {
```

```
assert.equal(null, err);
        assert.equal(1, r.insertedCount);

        // Insert multiple documents
        db.collection('shows').insertMany( manyShows, function(err, r) {
assert.equal(null, err);
assert.equal(2, r.insertedCount);

client.close();
});
});
```

8. Run the file using Node and verify that a new document has been inserted.