Big Data Systems: MongoDB Homework Assignment

Instructions

The objective of this assignment is to hone your skills with MongoDB. In particular, a very common task with MongoDB that you might encounter in practice is to load up a dataset and carry out a series of data transformations before extracting the results. Consequently, in this assignment you will be given a few datasets from the web and asked to perform such operations for them. This is an individual assignment; you are not allowed to work in groups. Please only submit text files containing your code for this assignment, there is no need for an accompanying write-up. Do, however, be certain to use comments effectively in your code to indicate which question you are solving. Do not include the dataset import statements that I give you in this assignment – only include commands that you write inside the Mongo shell.

A. Football, Glorious Football [20 points]

In this section, we are going to analyze match data from the English Premier League Football Season 2015/2016. To do this, import the JSON file *football.json* into a collection called *football* using the following command:

```
mongoimport --db aueb --collection football --file
football.json
```

This JSON file is actually only contains a (quite large) single JSON document and if you import it into a collection you will see that the collection will contain just one document. The document contains a complete history of all the football matches played in the English Premier League for the period 2015/2016.

For those of you who are not familiar with how Football in the English Premier League works, I will give you everything you need to know to answer the questions that follow. For the period 2015/2016 there were 20 teams competing. In the Premier League, over the course of the year, every team plays every other team once at their home stadium and once at the other team's home stadium. This means that every team will have played 38 matches by the end of the year. These matches are grouped together into Rounds (also called Matchdays) so that usually after each Round, every team will have played one other team. Typically, a round corresponds to a few consecutive days in a particular week so that at the end of each week, a league table can be computed showing how many points each team has collected and their respective position in the league. Teams collect points as follows. A draw occurs when both teams score the same number of goals (or nobody scored at all, i.e. 0 - 0) and each team gets 1 point. When one team scores more goals than the other team, the winner gets 3 points and the loser gets 0 points. When all teams have played each other the teams are ranked based on their number of points. To resolve ties, a quantity known as goal difference is used. This is the difference between the total number of goals that a team has scored over the entire season and the total number of goals that a team has conceded over that season. If two teams have a the same number of points but team A has a higher goal difference than team B, team A will appear on top of team B in the final rankings table. There are further rules for resolving ties that still remain but we won't need them here.

Now let's look at the JSON document. The structure of the top level document is as follows:

```
{
 "name": "English Premier League 2015/16",
 "rounds": [<Array of ROUNDS>]
}
```

The structure of the JSON documents corresponding to ROUNDS are:

```
{
 "name": <Name of the Round>,
 "matches": [<Array of MATCHES>]
}
```

An example MATCHES JSON document is given here:

```
 {
        "date": "2015-08-08",
        "team1": {
          "key": "manutd",
          "name": "Manchester United",
          "code": "MUN"
        },
        "team2": {
          "key": "tottenham",
          "name": "Tottenham Hotspur",
          "code": "TOT"
        },
        "score1": 1,
        "score2": 0
  },
```

This format should be self explanatory barring one clarification: Team 1 refers to the home team and Team 2 refers to the away team. So the example document shows that Manchester United won Tottenham Hotspur 1 - 0 at home.

Ok now for the questions. We want you to use the aggegate() pipeline that you learned in class in order to write queries that will answer the following questions. Bear in mind that for each question, we should see a document or documents that clearly have the answer to the question using appropriately named fields. ***NOTE CAREFULLY***: If we ask you something like "who won the league" it is not appropriate to return an ordered array of documents and assume we will pick out the document at the top. A question asking for one answer must get back one document.

1) What was the score when Leicester City played Norwich at home? **[2 points]**
2) On what dates did Manchester City and Liverpool play against each other? **[2 point]**
3) Which team(s) won the most away games? HINT 1: Your query needs to work correctly in case more than 1 team have won the most away games. HINT 2: Lookup the $cond aggregation operator in the online documentation – this should be useful. **[5 points]**
4) Transform this document into a single document containing the final league table. The document should look like this:

```
{
     results: [<Array of TEAM JSON>]
}
```

An example TEAM JSON to show the format (the values are random):

```
{
     name: Manchester United
     points: 59
     goals_scored : 90
     goals_conceded: 34
     goal_difference: 56

}
```

The order of the TEAMS in the results array should be from most points to fewer points and should respect the tie breaking condition on goal_difference as described earlier. HINT: Yes, this part is hard. The only extra thing you need from the notes though, is $cond that you saw in the previous question. This is perfectly doable in one aggregate query. **[10 points]**

5) Repeat Question 4 but this time, calculate the league table as it stood at the end of 2015. **[1 point]**

B. Magic: The Gathering [10 points]

Magic: The Gathering (MTG) is a popular trading card game with a theme of magic and fantasy. A game sees two or more players simulating a battle among wizards by drawing and playing cards representing a wide variety of spells and creatures. The rules are complex but we don't need to know anything more to take a peak at a database consisting of different MTG cards and start running some queries on it to learn more about this crazy and wondrous universe.

First, let's import our collection of cards using the file mtg.json (this is a modified version of a file available on *mtgjson.com*:

```
mongoimport --db aueb --collection mtg --file mtg.json
```

Here's what a typical entry looks like:

```
{
    "_id" : ObjectId("588c72004c805c6bcf9fc356"),
    "layout" : "normal",
    "name" : "Air Elemental",
    "manaCost" : "{3}{U}{U}",
    "cmc" : NumberInt(5),
    "colors" : [
        "Blue"
    ],
    "type" : "Creature — Elemental",
    "types" : [
        "Creature"
    ],
    "subtypes" : [
        "Elemental"
    ],
    "text" : "Flying",
    "power" : "4",
    "toughness" : "4",
    "imageName" : "air elemental",
    "colorIdentity" : [
        "U"
    ]
}
```

No doubt the precise semantics of some of these entries will escape most of you unless you are familiar with the game. Nonetheless we can still play around with these data and have some fun.

1) Write a query to bring up just the names, types and subtypes of any cards that have the Elemental subtype but are not Creatures like the Air Elemental above. **[2 points]**
2) Aggregate all the type: "Creature – Elemental" cards by *power* and create documents in the following format, one for each different value of *power*:

```
{
      "power" : "4",
     "elementals": [{"name" : "Air Elemental",
                      "colors" : ["Blue"]
                      }, … ]
}
```

To further clarify, in the *elementals* field, we're expecting a JSON array whose documents are just the names and color arrays for the elementals that have a particular power rating indicated by the *power* field. Order these by the *power* field. **[3 points]**

3) Write the command to create an index on your collection first by color descending then by name. **[1 points]**
4) Write a query to return the first three cards (sorted by name) with either red or green in their color array, whose names begin with "Z". **[2 points]**
5) Write a query that first filters out cards that don't have both *colors* and *subtypes* and then computes the number of cards for each possible color and subtype combination. Some cards will be counted more than once as they can have more than one color or more than one subtype in their corresponding arrays **[2 points]**

C. Earth Meteorite Landings [10 points]

Our final dataset is going to help us learn a few more extra commands in MongoDB. Concretely, we will learn just a few things about working with geospatial data with MongoDB. Our dataset consists of a series of meteorite landings on Earth. Let's load this up first.

```
mongoimport --db aueb --collection meteorites --file
meteorites.json --jsonArray
```

Here is an example document:

```
{
    "_id" : ObjectId("588c4dcd4c805c6bcf9f6d73"),
    "fall" : "Fell",
    "geolocation" : {
        "type" : "Point",
        "coordinates" : [
            6.08333,
            50.775
        ]
    },
    "id" : "1",
    "mass" : "21",
    "name" : "Aachen",
    "nametype" : "Valid",
    "recclass" : "L5",
    "reclat" : "50.775000",
    "reclong" : "6.083330",
    "year" : "1880-01-01T00:00:00.000"
}
```

As we can see, the *geolocation* field has coordinates (longitude, latitude) for the point of impact. MongoDB has native capabilities of handling geospatial queries, which can prove very useful with data like these. Before we can work with these data however we discover that the mass field is written as a string instead of a number. There is no way to change this within an update() query so we will have to use some JavaScript instead. Execute the following command on your Mongo shell before attempting the following questions.

```
db.meteorites.find({}).forEach(function(theCollection) {
         theCollection.mass =
parseInt(theCollection.mass);
        db.meteorites.save(theCollection);
     });
```

1) Let's start off by building a geospatial index. This is a necessary step in MongoDB before we can run some of the queries we need. To learn how to do this, study the page on 2dsphere Indexes and then write down the command needed to create such an index on the *geolocation* field **[2 points]**
2) It turns out that if you have such an index on your collection, you have access to another type of stage in the aggregate() pipeline that we have not seen, the *$geonear* stage. Read the online documentation for this

stage, and then construct a query that will find the 5 meteorite landings that were nearest to Athens, Greece **[4 points]**

3) Write a query that computes the average mass, the number of meteorites, the most recent and the oldest meteorite landing for each value of *recclass*, sorted by *recclass*, but for only those values of *recclass* that appear in at least 5 meteorites in the collection **[2 points]**
4) Write a statement that will update all documents in the collection so that if their mass exceeds 10000, they will have acquire an extra field called *"big_one"* with the value *true* **[2 points]**