THE UNIVERSITY OF SYDNEY

School of Computer Science
Dr. Ying Zhou

COMP5338: Advanced Data Models **Sem. 2/2020**

Week 3: MongoDB Aggregation

08.09.2020

Learning Objectives

This week's lab focuses on a additional MongoDB features including:

- Basic shell scripting
- Basic aggregation
- Querying and Aggregation with Array type of Data
- Join through aggregation

Question 1: JavaScript shell scripting

The `mongo` shell is written by `JavaScript` and simple Javascript statements/functions can be used to query or manipulate data. In particular, a read query `db.collection.find()` always returns a `cursor` object as the result. The interactive JavaScript shell by default iterates through the cursor for up to 20 items by calling the `next()` method and prints the documents out. In addition to the `next()` method, a lot of other methods can be used to traverse or manipulate the result set. The reference document for all `cursor` methods can be found from Mongo Shell Methods: Cursor Method
The following example uses simple script to update the data type of existing documents in `revisions`.

```
db.revisions.find().forEach(function(doc){
    doc.timestamp = new ISODate(doc.timestamp);
    db.revisions.save(doc)
});
```

The script starts with a read query `db.revisions.find()` to find all documents in the `revisions` collection. This returns a `cursor` object. It then uses the `forEach()` method of the `cursor` to apply a JavaScript function to each document returned . The function (`.forEach(function(doc){...})`) is an anonymous function defined right at the spot with two statements: the first statement `doc.timestamp = new ISODate(doc.timestamp);`

converts the `timestamp` field to type `ISODate`; the second statement `db.revisions.save(doc)` saves the updated document back in the collection. Because both documents have the same ID, the new one will override the old one. (https://docs.mongodb.com/manual/reference/method/db.collection.save/)

The `mongo` shell can also load and execute script written in JavaScript. A JavaScript script can be loaded at the command line as `mongo test.js` or from within the shell as `load("test.js")`. When a script is loaded at the command line, it needs to make an connection to the server and to specify the database before running any `mongo` shell command. The following example shows how to initiate a connection to a locally run MongoDB and to set the global `db` variable pointing to database "wikipedia". It also includes statements for printing the result. More details on `mongo` shell scripting can be found from Write Scripts for the mongo Shell

```
// initiate connections to a local MongoDB instance
conn = new Mongo();
// specify the database as "wikipedia"
db = conn.getDB("wikipedia");
//run a query and get the returned cursor object
cursor = db.users.find({gender:female}) //run a query
//print the results
while ( cursor.hasNext() ) {
   printjson( cursor.next() );
}
```

Listing 1: Example mongo shell script

Question 2: Aggregation

Aggregation framework allows us to run complex queries on the collection by grouping documents based on certain criteria and summarizing the results in different ways. The example and exercises in this question use the Wikipedia collection `revisions` and `users` we created in week 2.

Run the following aggregation find out which five editors made the most revisions on Donald Trump page:

```
db.revisions.aggregate([
    {$match:{title:"Donald_Trump"}},
    {$group:{_id:"$user", numOfEdits: {$sum:1}}},
    {$sort:{numOfEdits:-1}},
    {$limit:5}
])
```

The aggregation has four stages: the first stage ($match) filters out all revision documents belong to Donald_Trump; the second stage groups ($group) the documents based on the "user" field value, and create a new field called "numOfEdits", the value of which is set to the number of document with that particular "user" value; the third stage ($sort) sorts the resulting documents by the filed "numOfEdits" in descending order, the last stage ($limit) limits the output to the first five documents.

```
db.users.aggregate
(
[
{$group:{_id:"$gender",
numOfEditors:{$sum:1}}},
{$sort:{numOfEdits:-1}},
]
)
```

Now write your own aggregation to find out:

1. The number of editors in each gender: "female", "male" and "unknown"
2. The number of minor revisions belonging to each page: Donald Trump and Hillary Clinton

Question 3: Query and aggregating with Array type data

The capability to model array and embedded object is a key advantage of document storage over traditional RDBMS. MongoDB provides rich set of operators to handle array type in simple query and in aggregation. We use another data set to illustrate a few typical array operators. Download and extract the json data file pagecat.json.gz from Canvas and import it to a collection pagecat in the same database.

The newly created collection contains data about page categories. Each document represent a page with two fields:title and categories, in addition to the default _id field. Field Categories contains an array of string, each representing a category. Below is an example document:

```
{
    "_id" : ObjectId("57e35483fb9adeae53738e9d"),
    "categories" : [
        "Category:2016-related lists",
        "Category:2016 American television seasons",
        "Category:Lists of American comedy television series episodes"
    ],
    "title" : "List of Full Frontal with Samantha Bee episodes"
}
```

a) Querying a single array item has similar syntax as querying simple value field. For instance, the following query find all pages in category "Category:Good articles", we only want to show the title of each page.

```
db.pagecat.find({categories: "Category:Good articles"},
    {title:1, _id:0})
```

b) We can also query combination of items in an array field. For instance, the following query find all pages belonging to both "Category:Good articles" and "Category:Football clubs in England".

```
db.pagecat.find({categories:
    {$all:["Category:Good articles","Category:Football clubs in England"]}})
```

The $all operator selects the documents where the value of a field is an array that contains all the specified elements.

c) We can simulate the `or` predicate with operator `$in` followed by an array value. This will return all documents matching any of the items in the array.

```
db.pagecat.find({categories:
{$in:["Category:English-language films", "Category:American films"]}})
```

d) We can find all pages belonging to exactly three different categories.

```
db.pagecat.find({categories: {$size:3}})
```

In the above command, the $size operator matches any array with the number of elements specified by the argument.

e) We can use aggregation to find out the top 10 categories with most pages.

```
db.pagecat.aggregate(
    [{$match:{categories: {$exists: true}}},
    {$unwind: "$categories" },
    {$group:{_id: '$categories', cat_number: {$sum:1}}},
    {$sort:{cat_number:-1}},
    {$limit: 10}
    ])
```

Here, $unwind stage flattens an array field from the input documents to output a document for each element. For instance, the following document:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] }
```

will be deconstructed to the following results:

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

f) Now write your own query to find out

1. The pages with no category information
2. The page with the largest number of categories
3. All categories with “film” in the name and the number of pages in each, sort the results in descending order by the number of pages in each category.
4. The editors that have made revisions on both `Donald Trump` and `Hillary Clinton` pages.
5. The common categories of page “Ben Affleck” and “Warren Beatty”

Question 4: Join in Aggregation

Traditional join is implemented as `$lookup` stage in aggregation pipeline. It performs a left outer join from the current pipeline result to a given collection.

The following example joins the collection `revisions` to the collection `users`. The two join fields are: "name" from `users`, specified as `localField`, and "user" from `revisions`, specified as `foreignField`. The join result will be stored as an array field called "revisions" in the original `users` document. In another words, the aggregation embeds all revision documents to the user's document who made those revisions. Each use document is augmented with a new array field to store revisions made by this user. You may save the joined result to a new collection using the `$out` stage. For instance, adding a new stage:`{$out: "usersExt"}`in the aggregation will save the join results in a new collection `userExt`

```
db.users.aggregate(
[{$lookup: {
    from: "revisions",
    localField: "name" ,
    foreignField: "user",
    as: "revisions"}
  }
])
```

Run the aggregation and inspect the results.

Run also the following aggregation to inspect the results. This aggregation joins the `users` collection to the `revisions` collection. Different to the previous join, it augment each revision document with a new array field to store details of the editor of this revision. Since a revision can be made by only one user, the array contains at most one element. You may save the result to a new collection called `revisionsExt` using the `$out` stage.

```
db.revisions.aggregate(
[{$lookup: {
    from: "users",
    localField: "user" ,
    foreignField: "name",
    as: "user_detail"}
  }
])
```

Collection `usersExt` and `collectionsExt` represent alternative schema to store similar data in the original `users` and `revisions` collections.

Now write your own aggregation to

1. compute the editor gender distribution for each page. In other words, we want to find out how many female,male,unknown editors have edited Donald Trump and Hillary Clinton page.
2. compute the average time between revisions. The revision document contains a `parentid` field pointing to the revision it is based on. The average time between revisions is the average of the interval between a revision and its parent revision in the collection.

Alternative MongoDB Client

As queries get more complex with many hierarchies of brackets, you may find it is a bit hard to keep track of the matching closing brackets. If you are familiar with programming IDE and would like to use an editor with syntax highlighting and intelligent code completion, you may try the MongoDB extension on Visual Studio Code, an alternative client tool built on source code editor. We have updated week 1's MongoDB set up document to include brief installation and usage guide of VS code and MongoDB extension. Note that the official client in tutorial is still Robo3T. Commands used in this lab are released in JavaScript format and in plain text format.