Name: Adam Capdeville Lab 6

Database Systems October 23, 2018

**Part I. Get Ready (10pts)**

1. In your own words, describe the NoSQL approach and how it differs from the traditional database approach. What are the benefits and caveats to each? And when would one approach be preferred over the other? To support your answer, include one relevant example for both the NoSQL database and an RDBMS.  
   \*Complete answers should be around 1/2 page.  
   \*\*Don’t just copy paste from the slides, Wikipedia or online sources. If you use external sources cite them properly.  
   \*\*\*Use one unique and relevant example to support your claims.

To support your answer, include one relevant example for both the NoSQL database and an RDBMS.

* 1. **Describe the NoSQL approach and how it differs from the traditional database approach.**

SQL databases are defined as Relational Databases (RDBMS). NoSQL databases are non-relational or distributed databases. SQL databases are table based while NoSQL are document-based, key-value pairs, graph and column-family databases. SQL databases have defined schemas while NoSQL have dynamic schemas.

* 1. **What are the benefits and caveats to each?**

|  |  |
| --- | --- |
| **SQL** | **NoSQL** |
| Pros: | Pros: |
| 1. Predefined schema 2. Normalization 3. Well-defined standards    1. Like *ISI* and *ANSI* which are accepted worldwide. 4. Speed    1. Retrieving of database records with great ease 5. Single standardized language : SQL | 1. Capable of handling big data 2. Low-cost database 3. Easier and low-cost scalability. 4. Detailed database modeling is not required here. 5. Flexibility of schema |
| Cons: | Cons: |
| 1. The process of interfacing is complex. 2. As SQL is an object, it occupies space. 3. Handling Big data is very costly as you will have to increase the hardware for scaling. 4. When a table is dropped, the view becomes inactive. | 1. No: ACID properties. 2. Lacks standardization 3. Still relatively new, lacking a supporting community |

* 1. **When would one approach be preferred over the other?**

**Using SQL over NoSQL:**

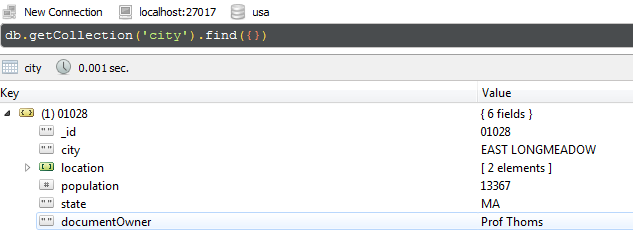
1. Complex queries: NoSQL doesn’t support relations between data types. Running queries in NoSQL is doable, but much slower.
2. If you need ACID compliance
3. High transaction application. SQL can ensure data integrity.
4. Small amount of data: NoSQL is overkill for small dbs

### Using NoSQL over SQL:

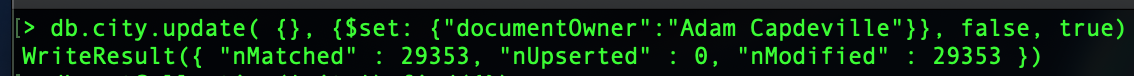
1. Constant change: If you are constantly adding new features, functions, data types, and it’s difficult to predict how the application will grow over time.
2. “Eh, Data consistency”: or data integrity is not the main goal.
3. Lots of data: NoSQL easily stores all different types of data without having to invest time into defining what type of data you’re storing in advance.
4. Scalability: Scale without the worry of hardware cost limitations.

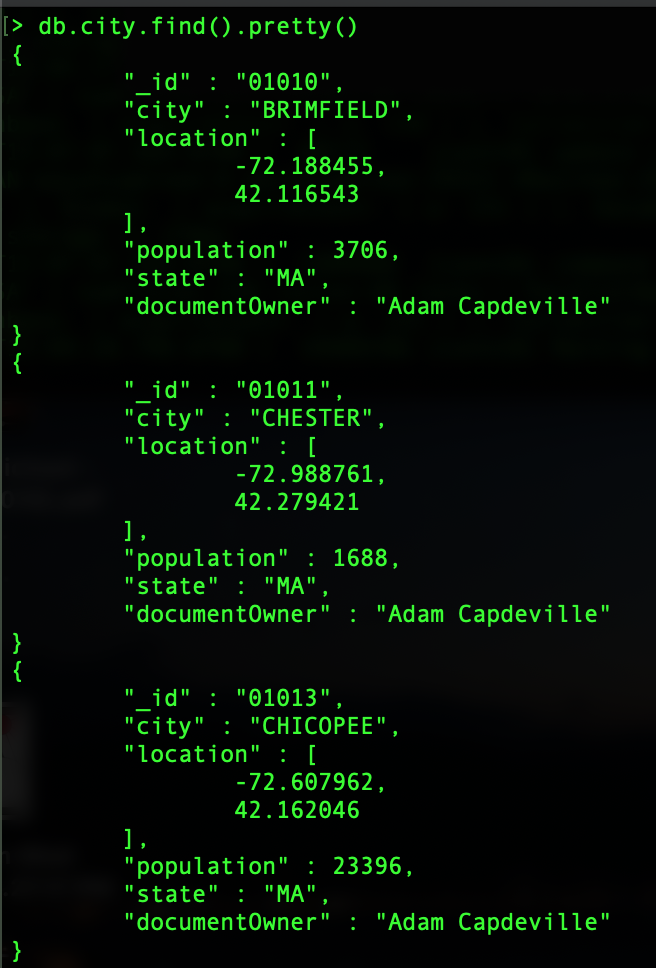
**Part II. Get Set (10pts)**

1. Create and run the query to import cities.json into a new database USA under the new collection, city. Update all documents to add the field documentOwner with the value *yourname.*

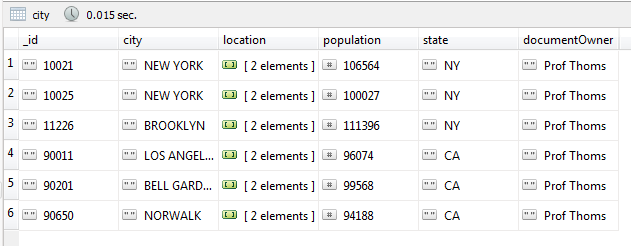


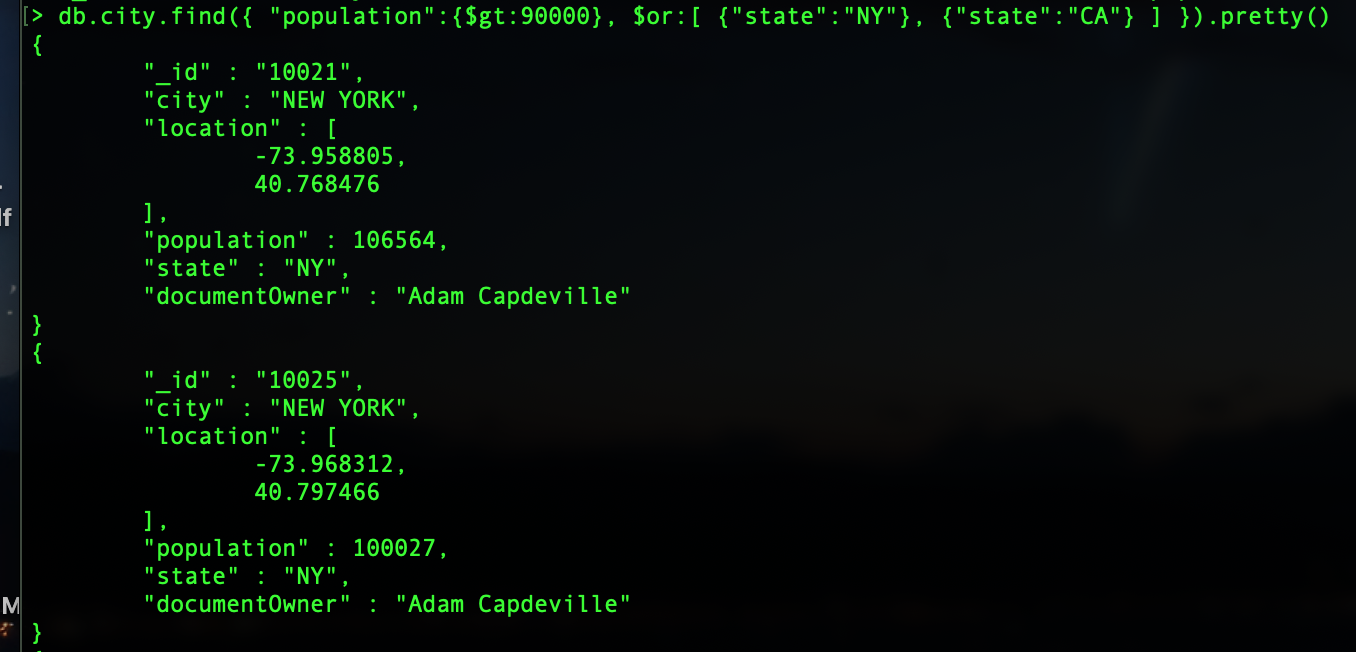
Mongo-Shell Command:



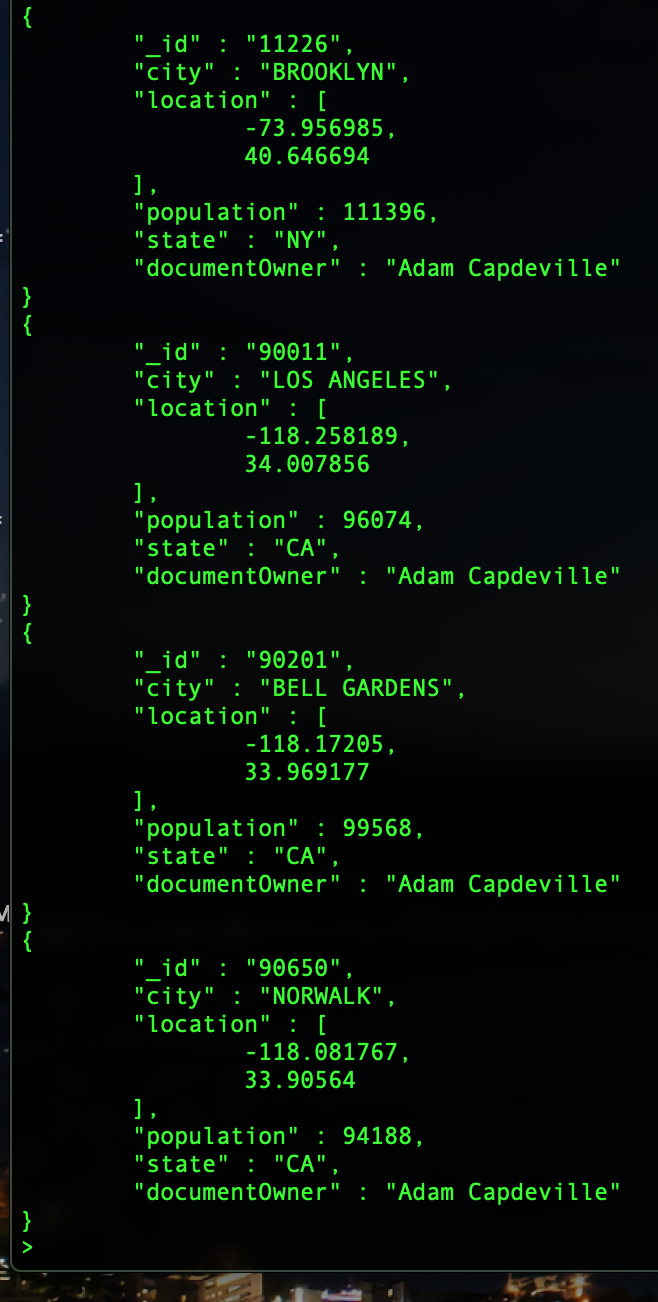


1. Create and run the query to return all documents with information related specifically to California, New York with populations greater than 90k.

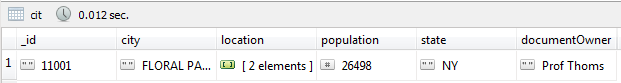


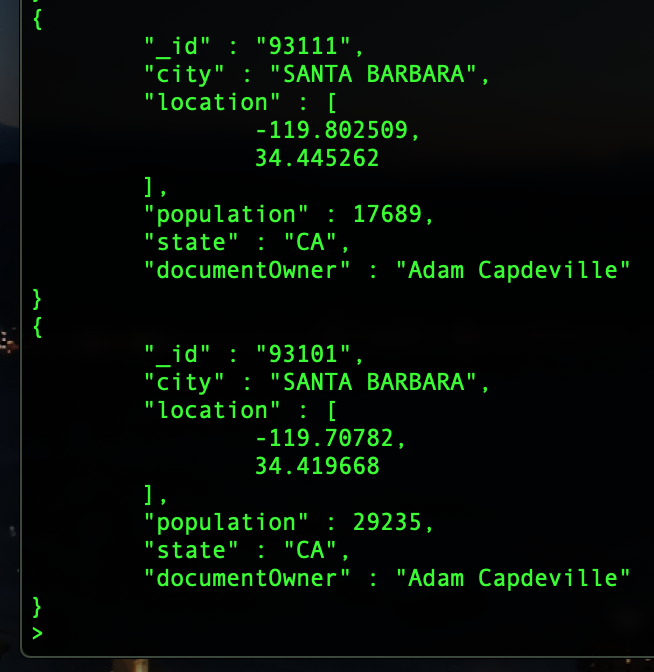
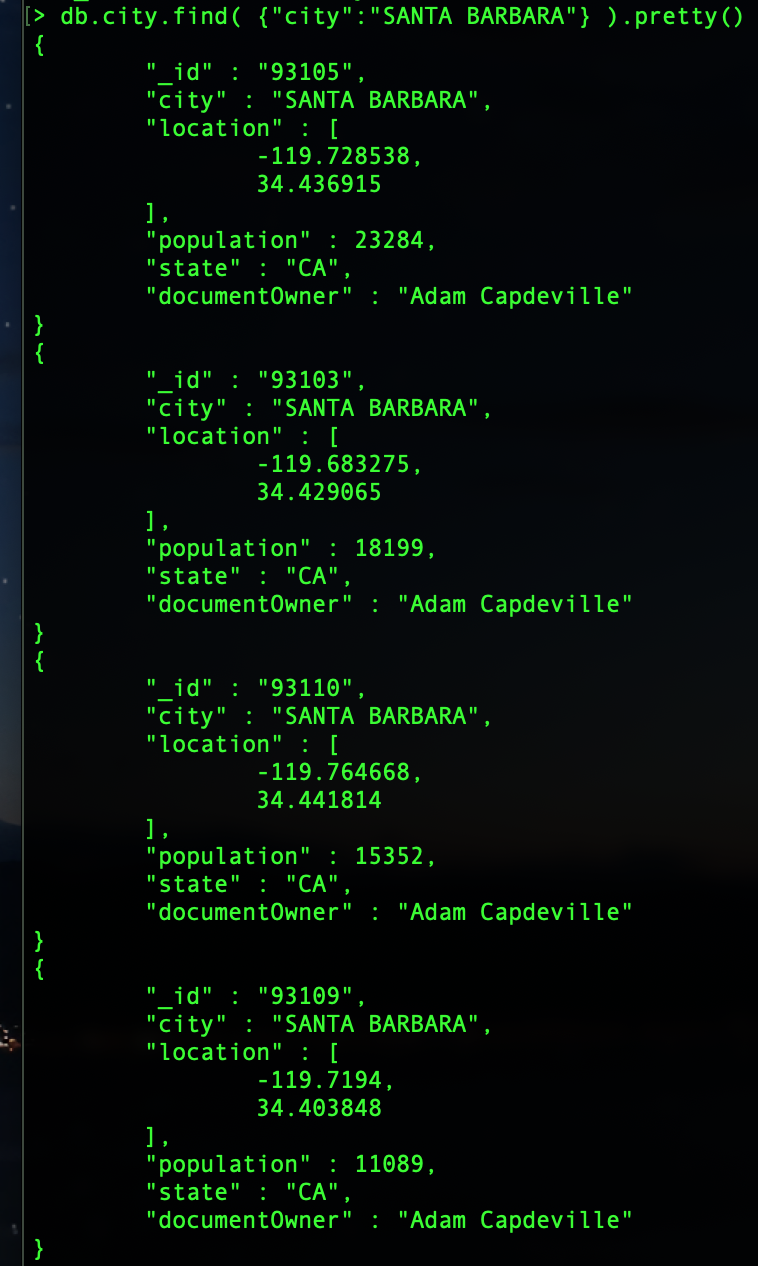


**( Rest of Query is on next page . . . )**

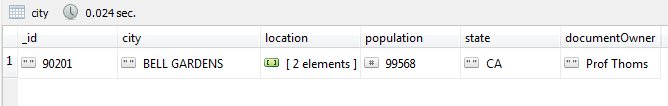
**( Question 3, query continued: )**  


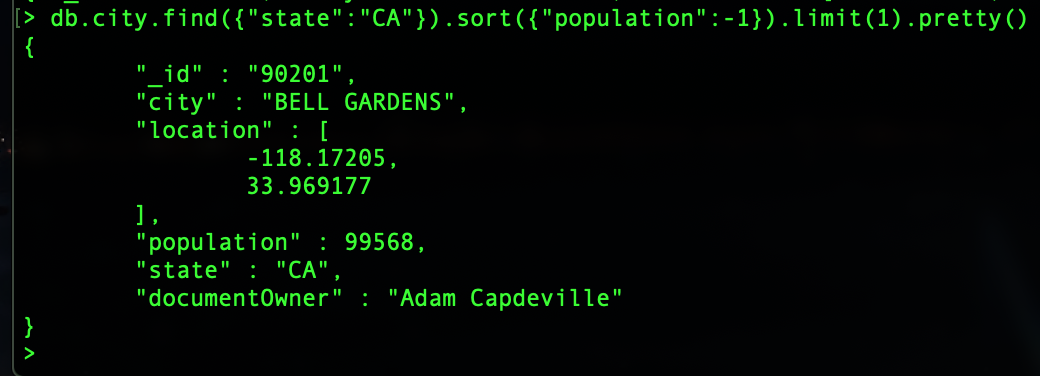
1. Create and run the query to return the document for the city where you grew up.

 **( Screenshot: Left: Query, Right: Query continued)**



1. Create and run the query to return the document with the largest population in California.



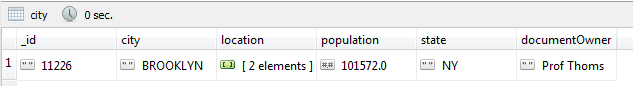


1. Use https://zipwho.com to find the correct population for one new city. Update the document for that city.

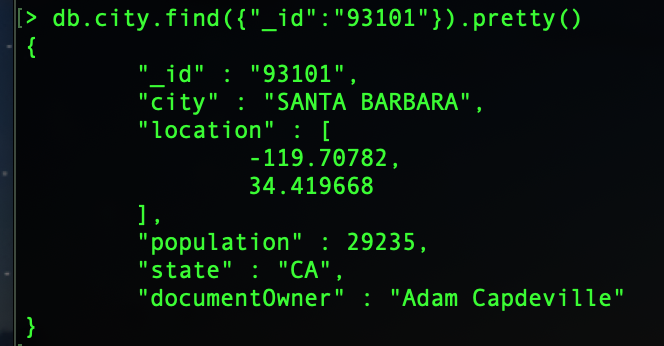
Before:



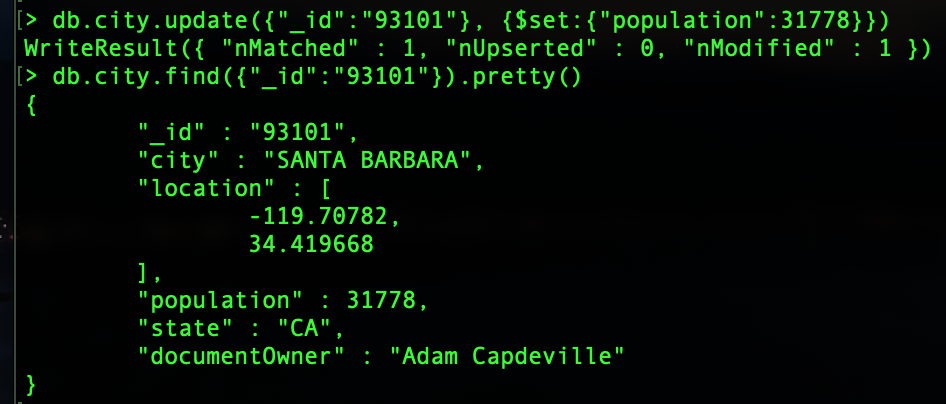
After:



Before:

****

After:



**Part III. Go (10pts)**

Within the USA database, create a new collection named, state, which will store data specific to states. This collection will be used to complete Q7 through Q11.

1. Write the insert statements to add documents for cities where you have lived, visited, or wish to visit in the future (up to 5). Include the dates you were there or when you hope to visit. If these documents already exist, update the documents to include the dates you last visited.

Lived: Santa Barbara

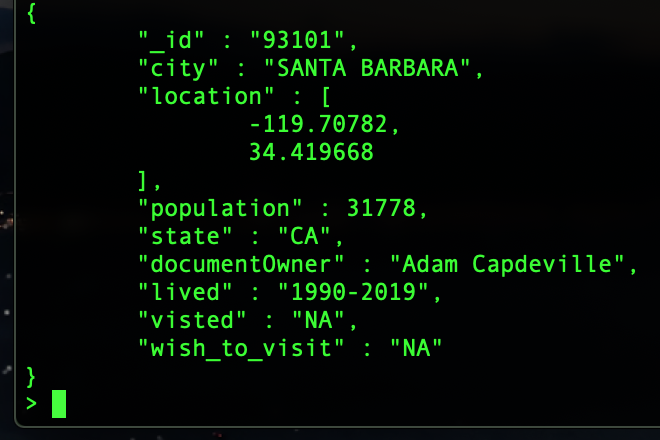
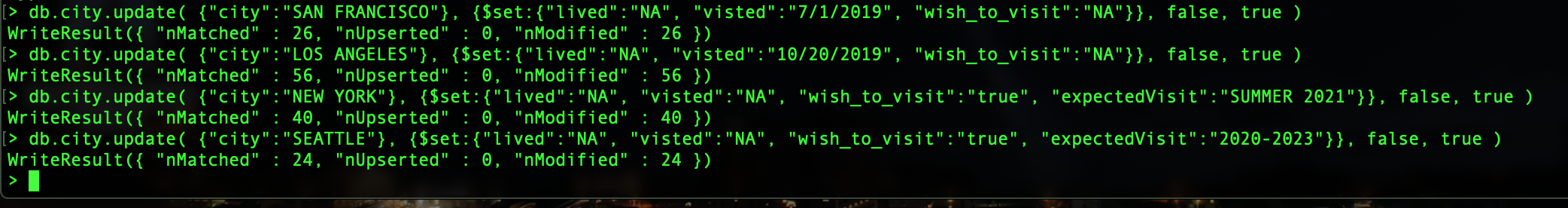
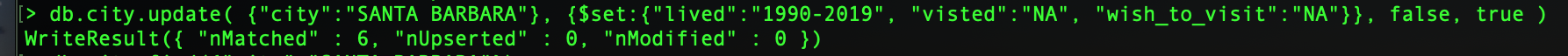
Visited: Los Angeles, San Francisco,

Wish-To-Visit: Seattle, New York

First update the collection to have these new fields:   
“city”, to contain a “lived”:date,”visited”:date & “toVisit”:boolean

Then, insert 5 new entries to newly created field.

//Add new fields ( lived, visited & wish\_to\_visit ) to the city collection

Update cities that I have lived in:  
  
  
Update cities that I have visited & cities that I want to visit with years that are my expected vist range.

1. Write the insert statements to add documents for 5 states. State data is up to you, but include at least three to five pieces of state-related data. Also include a field with your name as the governor.

**( “\_id”:”state”, stateName:, population, capital, avgIncome , governor)**

1. Write the update statements for 5 documents in the city collection based on the information found in the documents created in Q8 (**embed all state data**).

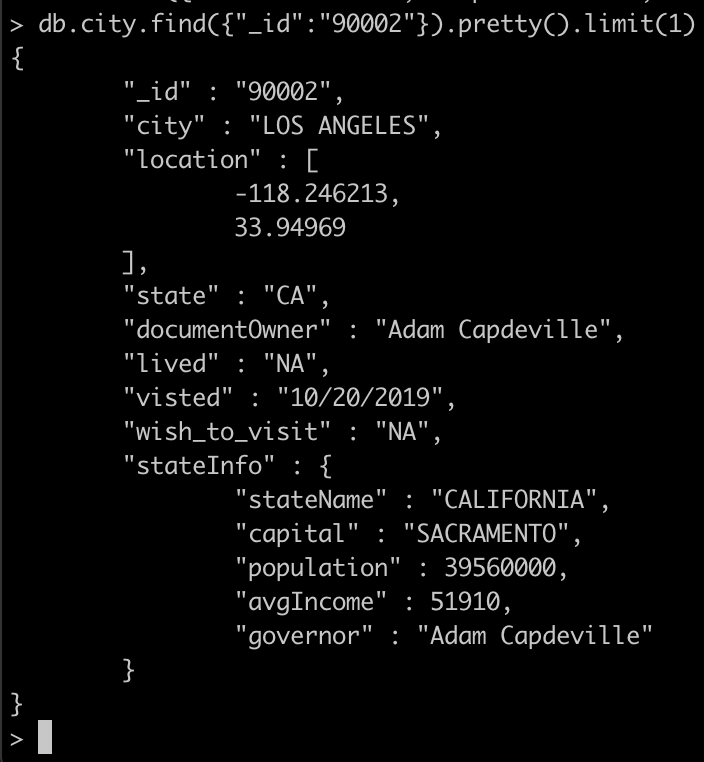
**// A better way for embedding:**

**> stateInfo = db.state.find( {"\_id":"<stateToRef>"},{ \_id:0, <fields…> } ).toArray()**

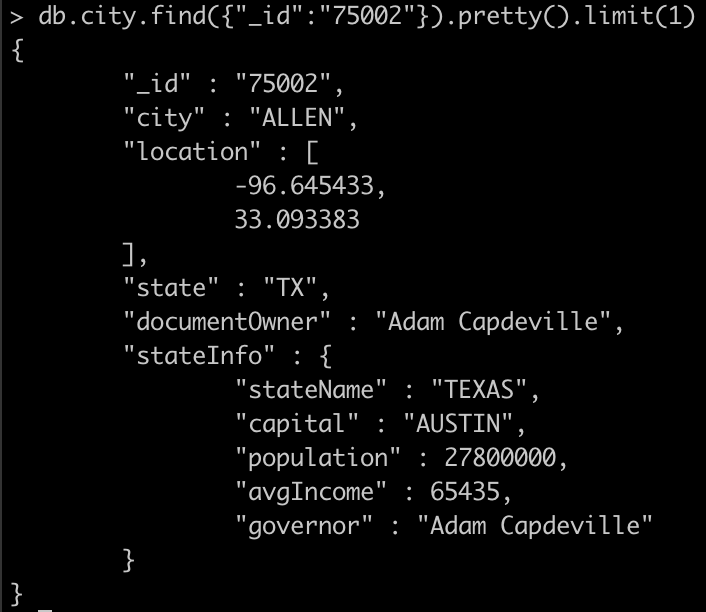
> db.city.update( {"\_id" : "99501"},{ $set: { stateInfo: { "stateName":"ALASKA", "capital":"JUENEAU", "population":737438, "avgIncome":73181, "governor":"Adam Capdeville"} } })



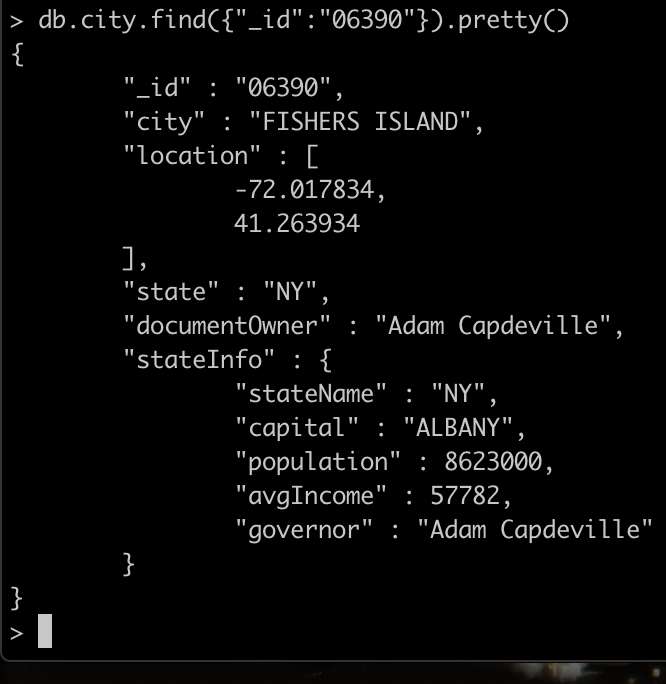
> db.city.update( {"\_id" : "90002"},{ $set: { stateInfo: { "stateName":"CALIFORNIA", "capital":"SACRAMENTO", "population":39560000, "avgIncome":51910, "governor":"Adam Capdeville"} } })



> db.city.update( {"\_id" : "75002"},{ $set: { stateInfo: { "stateName":"TEXAS", "capital":"AUSTIN", "population":27800000, "avgIncome":65435, "governor":"Adam Capdeville"} } })



> db.city.update( {"\_id" : "06390"},{ $set: { stateInfo: { "stateName":"NEWYORK", "capital":"ALBANY", "population":577737, "avgIncome":57782, "governor":"Adam Capdeville"} } })

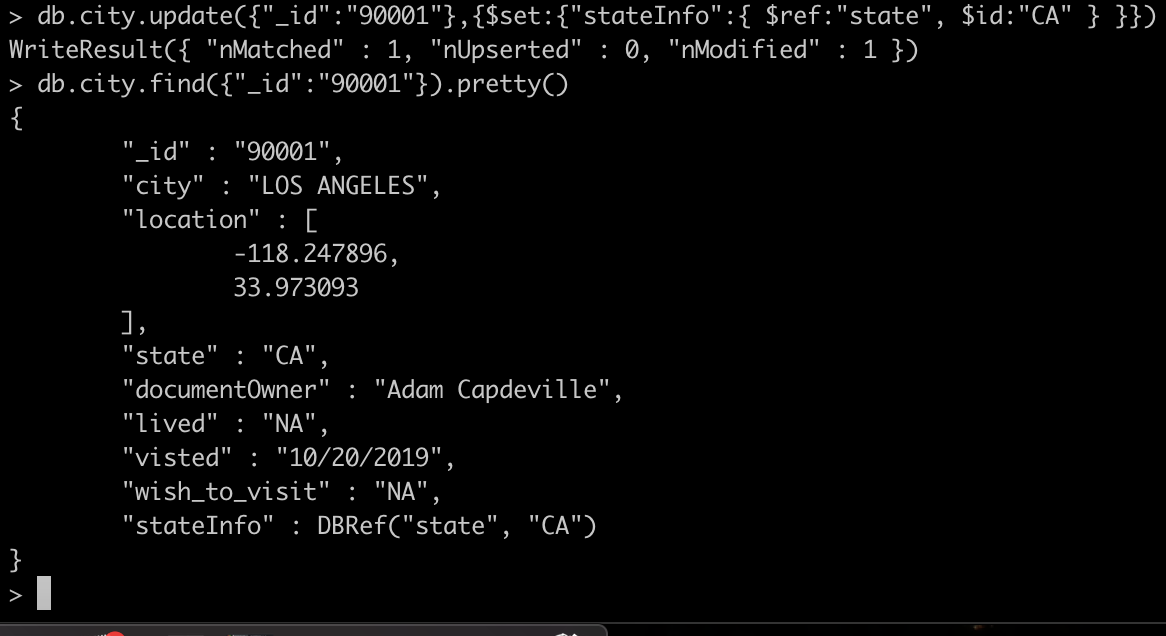
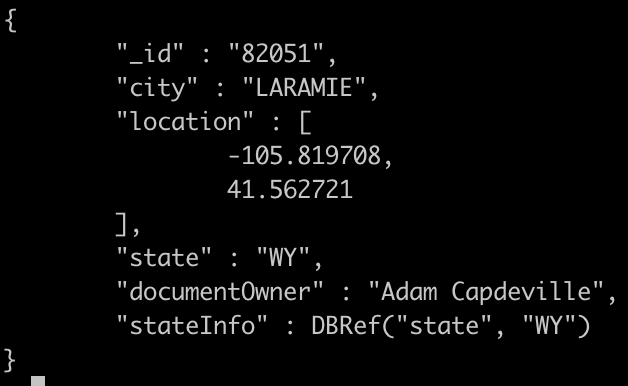
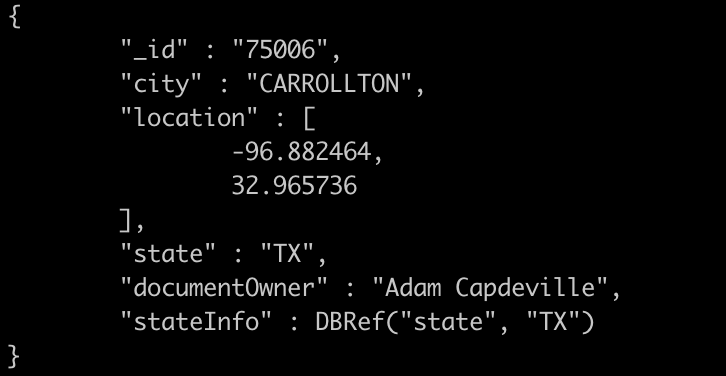


> db.city.update( {"\_id" : "82052"},{ $set: { stateInfo: { "stateName":"WYOMING", "capital":"CHEYENNE", "population":577737, "avgIncome":60434, "governor":"Adam Capdeville"} } })



1. Write the update statements for five more documents (not updated in Q9) in the city collection based on the information found in the documents created in Q8 (**reference state documents**).

> db.state.update({"\_id":""},{$set:{"<newFiledName>":{$ref:"city", $id:"<idName>"}}})

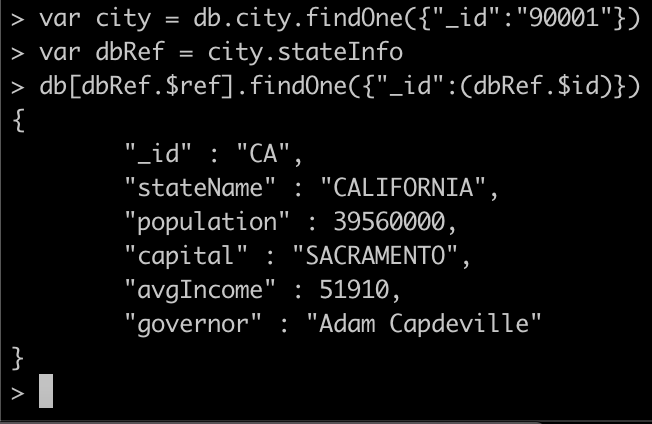
1. Create the Mongo query that will return state information based on Q10. What issues do you run into when querying MongoDB for referenced data? How can these issues be overcome? Provide 1 example.

**Two methods of Referencing:**

**Manuel Referencing:**

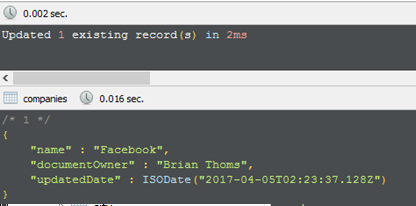
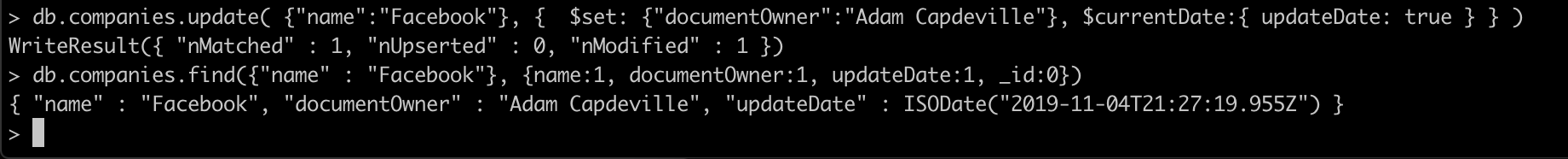
Setting the “collection name \_id” as the field name, and the field as the “\_id” as the point of reference in said collection.  
( EX: <collectionName\_id>:<\_id> )  
  
**DBRefs:**  
  
Similar to Manuel Referencing, the “collection name” & “\_id” are kept, but DBRef allows referencing collections from DB systems that exist outside of the current DB. The field name does not reference anything, but instead the contents of the DBRef is: $ref, $id, $db.

$ref : the collection,   
$id : the id in that collection,  
$db : the db name that the collection exists in.

DBRefs method:   


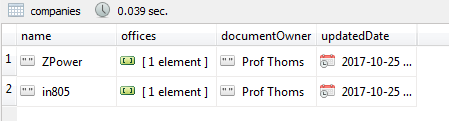
Query issues can occur when Manually Referencing due to the field name having to be explicitly <name> with “id” appended to it. So that the Administrator who knows nothing of the system, knows that that specific field is a reference. So by using DBRef, the field content is explicitly a reference.  
( Solution is the above DBRef )

**Part IV. Go… (10pts)**

For Part IV, import companies.json into a new database nasdaq, under the collection, companies. For each query, ensure that your name and date is populated as shown below. Do not include the \_id field for any results. The below example returns documents where the company name is Facebook will show two additional fields for documentOwner and updatedDate.   
SETUP:  


$set:{"documentOwner":"Adam Capdeville"}, $currentDate:{ updateDate: true }

1. Create and run the query to return all companies that have offices in Camarillo, CA. Include fields for the company name, category and address.



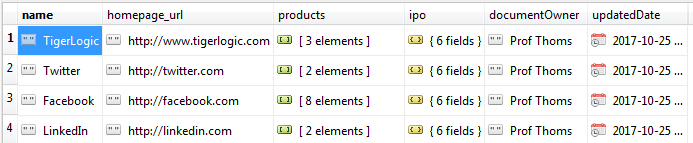
> db.companies.update({"offices.city":"Camarillo"},{ $set:{"documentOwner":"Adam Capdeville"}, $currentDate:{updateDate:true}},{multi:true} )

// WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })

> db.companies.find({"offices.city":"Camarillo"},{documentOwner:1, updateDate:1 ,name:1, offices:1, \_id:0}).pretty()



1. Create and run the query to return all documents for social-based companies. To determine social companies, use the category code. Include only companies that are publicly traded (IPO is not NULL) and sort your results by the year they were founded in descending order.

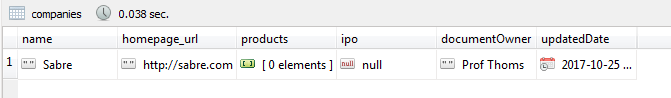


> db.companies.update({"category\_code":"social"},{ $set:{"documentOwner":"Adam Capdeville"}, $currentDate:{updateDate:true}},{multi:true} )

// WriteResult({ "nMatched" : 49, "nUpserted" : 0, "nModified" : 49 })

> db.companies.find({"category\_code" : "social", "ipo":{$ne:null}}, {"name":1, "homepage\_url":1, "products":1, "\_id":0, ipo:1, documentOwner:1, updateDate:1}).sort({"founded\_year":-1}).pretty()

1. Create and run the query to return the document for the oldest company where search is a core business function. To determine a company’s core business function, use the tag list.



> db.companies.update({"name":"Alstrasoft"},{$set:{"documentOwner":"Adam Capdeville"}, $currentDate:{ updateDate: true }})

> db.companies.find({founded\_year:{$ne:null}},{\_id:0, name:1, homepage\_url:1, products:1, ipo:1, founded\_year:1, documentOwner:1, updateDate:1}).sort({founded\_year:1}).pretty().limit(1)



*Use the mongodb function aggregate() to perform in-line query calculations for the following queries.*

1. Create and run the query to return the company with the most offices in New York.



db.companies.aggregate([

{$unwind: "$offices"},

{$project: {

\_id: { distinct:"companies", key:"$offices" },

state\_code: "$offices.state\_code",

name: "$name", documentOwner:1, updatedDate:1

}},

{$match:{ state\_code: "NY" }},

{$group:{ \_id: "$name", officesInNY:{$sum: 1} }},

{$sort:{officesInNY: -1}},

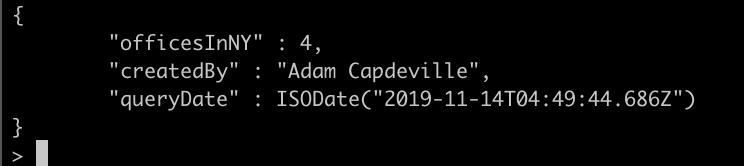
{$limit:1},

{$project: {\_id:0}},

{$addFields: {"createdBy":"Adam Capdeville"}},

{$addFields: {"queryDate":new Date()}}

]).pretty()



1. Create and run the query to return the average number of years until IPO for companies that had an initial public offering.



db.companies.aggregate([

{$match:{ipo:{$ne:null}}},

{$match:{"ipo.pub\_year":{$ne:null}}}, {$match:{"founded\_year":{$ne:null}}},

{$group:{\_id:"Average Values",

average: {$avg:{$subtract:["$ipo.pub\_year","$founded\_year"]}}}},

{$project: {\_id:0}},

{$addFields: {"createdBy":"Adam Capdeville"}},  
{$addFields: {"queryDate":new Date()}}

])