Project 1: MongoDB

*This project is developed by **Team 1**:*

- Sarah Cooney
- Mingyuan Li
- Jason Qiao Meng

Table of Content

Introduction

This project is a course project of **CSC643 - Bigdata**. It is about a MongoDB client script and a Python program which runs experimental `NOSQL` queries.

As required, This project implements the following queries on top of `zipcodes` database:

- *Query A*: Find the total number of cities in the database.
- *Query B*: Create the list of states, cities, and city populations.
- *Query C*: List the cities in the state of Massachusetts with populations between 1000 and 2000.
- *Query D*: A mapReducer to compute the total number of cities and total population in each state.
- *Query E*: A mapReducer to find the average city population for each state.
- *Query F*: A mapReducer to find the least densely-populated state(s).

Implementation

The sources contains two parts, one is MongoDB client scripts, the other is the equivalents in Python.

The MongoDB client scripts is included by `src/queries.txt`; The Python equivalents are included by `src/queries.py`.

The Python implementation consists of a MongoDB connector module, a global environmental settings module, and the query runner module.

Database Connection

MongoDB connection is managed by `mongodb_connector.py`. This module uses the `pymongo` library to create or destroy a connection to a MongoDB instance. The variables defined in `settings.py` are utilized in this module.

The file contains a class called `MongoDB` which acts as a MongoDB client which gets an instance of a database, gets an instance of a collection.

The member method `simple_connction_string` constructs a MongoDB connection string used by `pymongo` to connect to the MongoDB instance, and incorporates the protocol, host, and port variables defined in `settings.py`.

```
def simple_connction_string(self):
    """Get the connection string.

    :return: A legal MongoDB connection string from the settings.
    """
    self.__conn_str = "%s%s:%s/" % (DB_PROTOCOL, DB_HOST, DB_PORT)
    return self.__conn_str
```

Figure 1: `simple_connction_string` *to constructs a MongoDB connection string.*

The next method in the class is `get_database`. This method is used to access a specific Mongo database. The method takes an optional `name` parameter. If no database name is specified, the method defaults to the `DB_NAME` variable defined in `settings.py`. An error is thrown if no database of the specified name exists, whether by parameter or from the `settings.py`.

```
def get_database(self, name=None):
    """Get the database object with the specified name.

    :param name: The name of the database. If given None or omitted,
                 this method uses the name set in the settings file.
    :return: An instance of Database.
    """
    if not self.__client:
        self.get_client()

    dbname = name if name else DB_NAME
    try:
        self.__db = self.__client[dbname]
    except InvalidName as ine:
        self.__db = None
        print 'No such database: %s. %s' % (dbname, ine)

    return self.__db
```

Figure 2: `get_database` to get a database instance.

To use the class `MongoDB`, simply import the class from `mongodb_connector` module. Figure 3 demonstrates the usage.

```
from mongodb_connector import MongoDB

if __name__ == '__main__':
    mongo = MongoDB()
    cli = mongo.get_client()
    if cli and cli.database_names():
        print 'connect successful'
        print 'databases: ',
        for n in cli.database_names():
            print '%s, ' % n,
        print ''
    db = mongo.get_database()
    if db:
        print 'database connected'
        print 'database test collections: ',
        for n in db.collection_names():
            print '%s, ' % n,
        print ''
        print 'database test get document count: ',
        collection = db[db.collection_names()[0]]
        print collection.count()
    mongo.close();
```

Figure 3: Class `MongoDB` usage

Global Settings

To avoid modifications to the database connector class and concrete query functions, a `settings` module, implemented in `settings.py`, to manage the global shared environmental variables.

Both the module `mongodb_connector.py` and `queries.py` incorporate this settings module. Figure 4 shows the detail of `settings.py`.

```
# database related
DB_NAME = "zipcodes"
DB_PROTOCOL = "mongodb://"
DB_HOST = "localhost"
DB_PORT = "27017"
COLLECTION = "zipcodes"
```

Figure 4: The global variables

NOSQL Queries

This section will describe the setup of the queries.py file and give details about each of the six queries. Screenshots of the output for each query will also be shown.

The `queries.py` file contains a function for each of the six queries and a main section that runs all six queries in succession.

The queries are:

- ☑ *Query A*: Find the total number of cities in the database.

- ☑ *Query C*: List the cities in the state of Massachusetts with populations between 1000 and 2000.
- ☑ *Query D*: A mapReducer to compute the total number of cities and total population in each state.
- ☑ *Query E*: A mapReducer to find the average city population for each state.
- ☑ *Query F*: A mapReducer to find the least densely-populated state(s).

Query A

Query A: Find the total number of cities in the database.

This query, shown in Figure 5, is a simple `pymongo` query. The `distinct` command is used with the `city` parameter to find all of the distinct cities in the database, since some have multiple zipcodes; for instance, *Boston*. The Python `len` method was called on the collection that was returned to find the number. Figure 6 shows the output from this query.

```
def total_cities(mongodb):
    """This query function returns the total number of cities in the database."""
    db = mongodb.get_database()
    return len(db[COLLECTION].distinct('city'))
```

Figure 5: Query A implementation

Figure 6: Query A output

Shown by Figure 7, the actual MongoDB client command and output are:

```
> use zipcodes;
> z = db.zipcodes;
> z.distinct("city").length;
    16584
```

Figure 7: Query A by MongoDB client command

Query B

Query B: Create the list of states, cities, and city populations.

This function uses MongoDB's aggregate framework; it constructs a pipeline for the aggregation. An array is created, and for each entry in the database, the state, city, and population information are appended and then added to the array. The method returns this array, which is printed in the output. Figure 8 shows the code for this query, and Figure 9 contains a sample of the output produced.

```
def list_states_cities_populations(mongodb):
    """This query function returns the list of states, cities, populations in the database."""
    db = mongodb.get_database()
    collection = db[COLLECTION]
    pipline = [
        {"$project": {"state": 1, "city": 1,  "pop": 1}},
        {"$group": {
            "_id": SON([("state","$state"), ("city", "$city")]),
            "popTotal": {"$sum": "$pop"}
        }},
        {"$sort": {"_id.state": 1}}
    ]
    return collection.aggregate(pipline)
```

Figure 8: Query B implementation

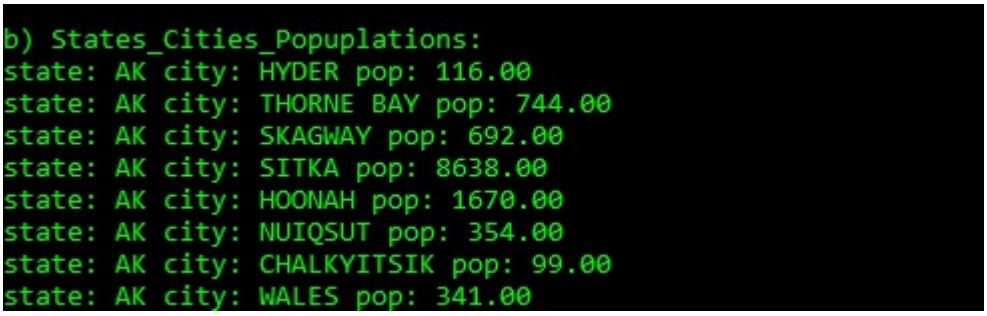

Figure 9: Query B output

Shown by Figure 10, the actual MongoDB client command and sample output are:

```
> db.zipcodes.aggregate([
... {$project: {state: 1, city: 1, pop: 1}},
... {$group: {
...     _id: {"state": "$state", "city": "$city"},
...     popTotal: {$sum: "$pop"}, }},
... {$sort: {state: 1}} ]);
{ "_id" : { "state" : "AK", "city" : "HYDER" }, "popTotal" : 116 }
{ "_id" : { "state" : "AK", "city" : "THORNE BAY" }, "popTotal" : 744 }
{ "_id" : { "state" : "AK", "city" : "SKAGWAY" }, "popTotal" : 692 }
{ "_id" : { "state" : "AK", "city" : "SITKA" }, "popTotal" : 8638 }
{ "_id" : { "state" : "AK", "city" : "HOONAH" }, "popTotal" : 1670 }
{ "_id" : { "state" : "AK", "city" : "NUIQSUT" }, "popTotal" : 354 }
{ "_id" : { "state" : "AK", "city" : "CHALKYITSIK" }, "popTotal" : 99 }
{ "_id" : { "state" : "AK", "city" : "WALES" }, "popTotal" : 341 }
{ "_id" : { "state" : "AK", "city" : "WAINWRIGHT" }, "popTotal" : 492 }
{ "_id" : { "state" : "AK", "city" : "VENETIE" }, "popTotal" : 184 }
{ "_id" : { "state" : "AK", "city" : "TELLER" }, "popTotal" : 260 }
{ "_id" : { "state" : "AK", "city" : "TANANA" }, "popTotal" : 345 }
{ "_id" : { "state" : "AK", "city" : "SELAWIK" }, "popTotal" : 0 }
{ "_id" : { "state" : "AK", "city" : "POINT HOPE" }, "popTotal" : 640 }
{ "_id" : { "state" : "AK", "city" : "GOLOVIN" }, "popTotal" : 3706 }
{ "_id" : { "state" : "AK", "city" : "NENANA" }, "popTotal" : 393 }
{ "_id" : { "state" : "AK", "city" : "POINT LAY" }, "popTotal" : 139 }
{ "_id" : { "state" : "AK", "city" : "MINTO" }, "popTotal" : 228 }
{ "_id" : { "state" : "AK", "city" : "LAKE MINCHUMINA" }, "popTotal" : 32 }
{ "_id" : { "state" : "AK", "city" : "MANLEY HOT SPRIN" }, "popTotal" : 122 }
Type "it" for more
```

Figure 10: Query B by MongoDB client command

Query C

Query C: List the cities in the state of Massachusetts with populations between 1000 and 2000.

Similar to the query B, This function uses MongoDB's aggregate framework; it constructs a pipeline for the aggregation. In addition, a `$match` filter is applied to filter and return just cities in the state of Massachusetts and then just those with populations between 1000 and 2000.

The results of the find are returned as an array. The code for query C is shown in Figure 11, and the output is shown in Figure 12.

```
def list_massachusetts_populations(mongodb):
    """This query function returns the list the cities in the state of Massachusetts with populations between 1000 and 2000."""
    db = mongodb.get_database()
    collection = db[COLLECTION]
    pipline = [
        {"$match": {"state": "MA"}},
        {"$group": {
            "_id": SON([("state","$state"), ("city", "$city")]),
            "popTotal": {"$sum": "$pop"}
        }},
        {"$match": {"popTotal": {"$gte":1000, "$lte": 2000}}}
    ]

    return collection.aggregate(pipline)
```

Figure 11: Query C implementation

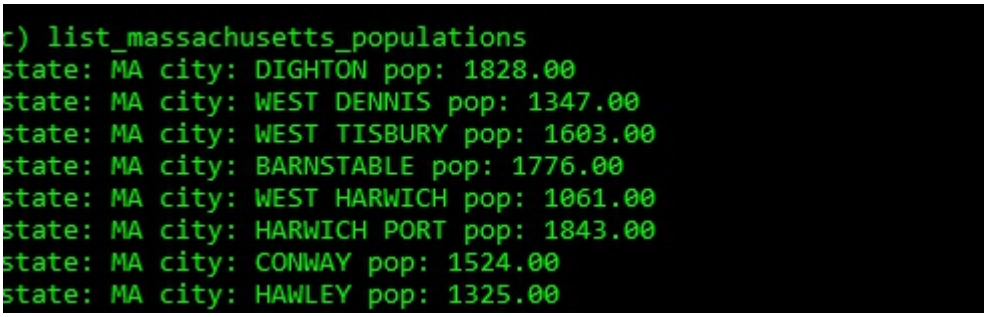

Figure 12: Query C output

Shown by Figure 13, the actual MongoDB client command and sample output are:

```
> db.zipcodes.aggregate([
... {$match: {$and: [{state: "MA"}, {pop: {$gte: 1000, $lte: 2000}}]}},
... {$group: {
        _id: {"state": "$state", "city": "$city"},
...     popTotal: {$sum: "$pop"}}},
... ]);
{ "_id" : { "state" : "MA", "city" : "WEST TISBURY" }, "popTotal" : 1603 }
```

```
{ "_id" : { "state" : "MA", "city" : "HARWICH PORT" }, "popTotal" : 1843 }
{ "_id" : { "state" : "MA", "city" : "CAMBRIDGE" }, "popTotal" : 1336 }
{ "_id" : { "state" : "MA", "city" : "LYNN" }, "popTotal" : 1187 }
{ "_id" : { "state" : "MA", "city" : "ROCHDALE" }, "popTotal" : 1154 }
{ "_id" : { "state" : "MA", "city" : "WEST DENNIS" }, "popTotal" : 1347 }
{ "_id" : { "state" : "MA", "city" : "PETERSHAM" }, "popTotal" : 1131 }
{ "_id" : { "state" : "MA", "city" : "MONTAGUE" }, "popTotal" : 1699 }
{ "_id" : { "state" : "MA", "city" : "CONWAY" }, "popTotal" : 1524 }
{ "_id" : { "state" : "MA", "city" : "DIGHTON" }, "popTotal" : 1828 }
{ "_id" : { "state" : "MA", "city" : "HAWLEY" }, "popTotal" : 1325 }
{ "_id" : { "state" : "MA", "city" : "NEWTONVILLE" }, "popTotal" : 1427 }
{ "_id" : { "state" : "MA", "city" : "ASHFIELD" }, "popTotal" : 1535 }
{ "_id" : { "state" : "MA", "city" : "RICHMOND" }, "popTotal" : 1134 }
{ "_id" : { "state" : "MA", "city" : "WEST STOCKBRIDGE" }, "popTotal" : 1173 }
{ "_id" : { "state" : "MA", "city" : "WEST HARWICH" }, "popTotal" : 1061 }
{ "_id" : { "state" : "MA", "city" : "BECKET" }, "popTotal" : 1070 }
{ "_id" : { "state" : "MA", "city" : "DEERFIELD" }, "popTotal" : 1281 }
{ "_id" : { "state" : "MA", "city" : "OAKHAM" }, "popTotal" : 1503 }
{ "_id" : { "state" : "MA", "city" : "SHUTESBURY" }, "popTotal" : 1533 }
Type "it" for more
```

Figure 13: Query C by MongoDB client command

Query D

Query D: Write a mapReducer to compute the total number of cities and total population in each state.

Because the documents are identified by the zipcodes, the city names can be duplicate in the database. This query must be able to filter out the duplicated city names in a state, and count the distinct cities.

The mapper for this function emits a list of city for the reducer to merge different cities to get a list of distinct cities. The reducer looped through the emitted values and aggregated the population on the state key, formed a list of distinct cities. The finalizer did the last step - got the length of the city list in the reduced value, such length is the count of cities for each state.

Figure 14 contains the code for query D, and Figure 15 a sample of the output.

```
def state_pop_city_count_map_reduce(mongodb):
    """A mapReducer to compute the total number of cities and total population in each state."""
    db = mongodb.get_database()
    collection = db[COLLECTION]
    mapper = Code("""
        function() {
            emit(this.state, {city: [this.city], pop: this.pop});
        };""")
    reducer = Code("""
        function(key, values) {
            var res = {city: [], pop: 0};
            for (var i = 0; i < values.length; ++i) {
                var val = values[i];
                res.pop += val.pop;
                res.city = res.city.concat(val.city);
            }
            // remove duplicates
            res.city = res.city.filter((elem, index) =>  res.city.indexOf(elem) === index);
            return res;
        }""")
    finalizer = Code("""
        function(key, reducedValue) {
            reducedValue.cityCount = reducedValue.city.length;
            return reducedValue;
        }""")
    result = collection.map_reduce(
        mapper, reducer, 'state_counts', finalize=finalizer)
    return result.find()
```

Figure 14: Query D implementation

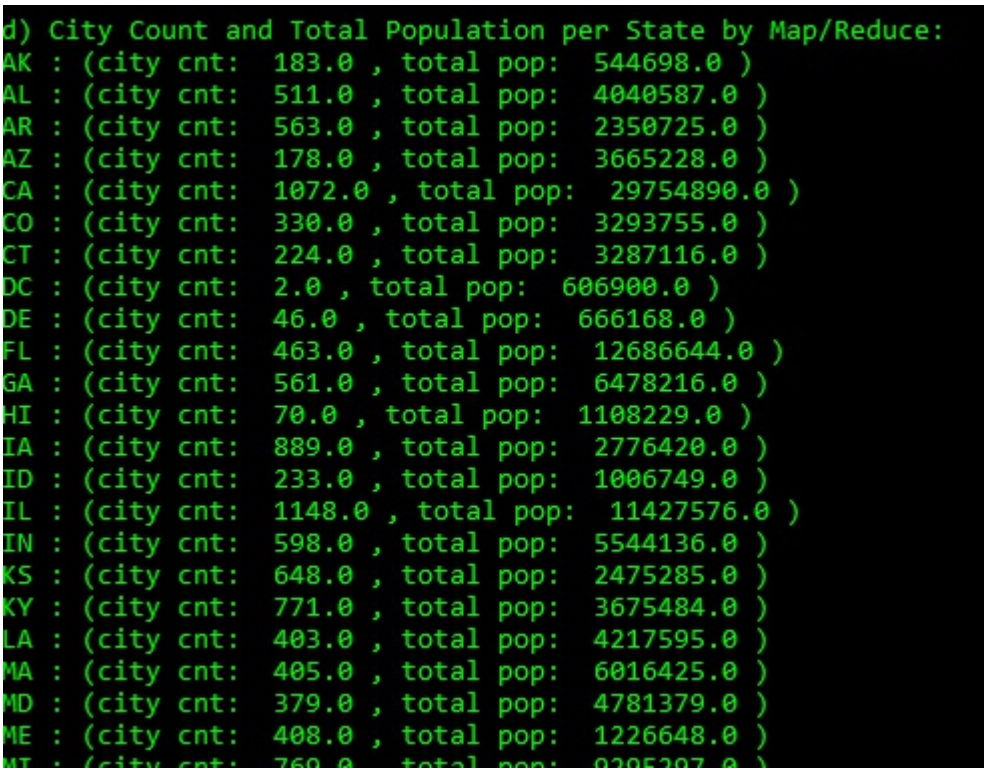

Figure 15: Query D output

Shown by Figure 16, the actual MongoDB client command and sample output are:

```
> var map = function() {
...     emit(this.state, {city: [this.city], pop: this.pop});
... };
>
> var reducer = function(key, values) {
...     var res = {city: [], pop: 0};
...     for (var i = 0; i < values.length; ++i) {
...         var val = values[i];
...         res.pop += val.pop;
...         res.city = res.city.concat(val.city);
...     }
...     res.city = res.city.filter((elem, index) =>  res.city.indexOf(elem) === index);
...     return res;
... };
>
>
> var finalizer = function(key, reducedValue) {
...     reducedValue.cityCount = reducedValue.city.length;
...     return reducedValue;
... };
>
> var r = z.mapReduce(map, reducer, {out: "state_pop_city_count", finalize:finalizer});
> var cur = r.find();
> while (cur.hasNext()) {
... var doc = cur.next();
... print(doc._id, ": (pop:", doc.value.pop, ", cityCount:", doc.value.cityCount, ")");
... };
AK : (pop: 544698 , cityCount: 183 )
AL : (pop: 4040587 , cityCount: 511 )
AR : (pop: 2350725 , cityCount: 563 )
AZ : (pop: 3665228 , cityCount: 178 )
CA : (pop: 29754890 , cityCount: 1072 )
CO : (pop: 3293755 , cityCount: 330 )
CT : (pop: 3287116 , cityCount: 224 )
DC : (pop: 606900 , cityCount: 2 )
DE : (pop: 666168 , cityCount: 46 )
FL : (pop: 12686644 , cityCount: 463 )
GA : (pop: 6478216 , cityCount: 561 )
HI : (pop: 1108229 , cityCount: 70 )
IA : (pop: 2776420 , cityCount: 889 )
ID : (pop: 1006749 , cityCount: 233 )
IL : (pop: 11427576 , cityCount: 1148 )
IN : (pop: 5544136 , cityCount: 598 )
KS : (pop: 2475285 , cityCount: 648 )
KY : (pop: 3675484 , cityCount: 771 )
LA : (pop: 4217595 , cityCount: 403 )
MA : (pop: 6016425 , cityCount: 405 )
```

```
MD : (pop: 4781379 , cityCount: 379 )
ME : (pop: 1226648 , cityCount: 408 )
MI : (pop: 9295297 , cityCount: 769 )
MN : (pop: 4372982 , cityCount: 814 )
MO : (pop: 5110648 , cityCount: 901 )
```

Figure 16: Query D by MongoDB client command

Query E

Query E: Write a mapReducer to find the average city population for each state.

This query is built from the code for Query D. It uses the same mapper and reducer. However, a finalize method is added to compute the average from the number of cities and the total population for each state.

A sample of the code for query E is shown in Figure 17, and sample output is shown in Figure 18.

```
def average_state_population_with_map_reduce(mongodb):
    """A mapReducer to compute the average population in each state."""
    db = mongodb.get_database()
    collection = db[COLLECTION]
    mapper = Code("""
        function() {
            emit(this.state, {city: [this.city], pop: this.pop});
        }""")
    reducer = Code("""
        function(key, values) {
            var res = {city: [], pop: 0};
            for (var i = 0; i < values.length; ++i) {
                var val = values[i];
                res.pop += val.pop;
                res.city = res.city.concat(val.city);
            }
            res.city = res.city.filter((elem, index) =>  res.city.indexOf(elem) === index);
            return res;
        }""")
    finalizer = Code("""
        function(key, reducedValue) {
            reducedValue.cityCount = reducedValue.city.length;
            reducedValue.avgPop = reducedValue.pop / reducedValue.cityCount;
            return reducedValue;
        }""")
    result = collection.map_reduce(
        mapper, reducer, 'state_avgs', finalize=finalizer)
    return result.find()
```

Figure 17: Query E implementation

```
e) Average Population for Each State with MapReduce:
AK: (city cnt: 183, total pop: 544698, avg pop: 2976.49)
AL: (city cnt: 511, total pop: 4040587, avg pop: 7907.22)
AR: (city cnt: 563, total pop: 2350725, avg pop: 4175.36)
AZ: (city cnt: 178, total pop: 3665228, avg pop: 20591.17)
CA: (city cnt: 1072, total pop: 29754890, avg pop: 27756.43)
CO: (city cnt: 330, total pop: 3293755, avg pop: 9981.08)
CT: (city cnt: 224, total pop: 3287116, avg pop: 14674.62)
DC: (city cnt: 2, total pop: 606900, avg pop: 303450.00)
DE: (city cnt: 46, total pop: 666168, avg pop: 14481.91)
FL: (city cnt: 463, total pop: 12686644, avg pop: 27400.96)
GA: (city cnt: 561, total pop: 6478216, avg pop: 11547.62)
HI: (city cnt: 70, total pop: 1108229, avg pop: 15831.84)
IA: (city cnt: 889, total pop: 2776420, avg pop: 3123.08)
ID: (city cnt: 233, total pop: 1006749, avg pop: 4320.81)
IL: (city cnt: 1148, total pop: 11427576, avg pop: 9954.33)
IN: (city cnt: 598, total pop: 5544136, avg pop: 9271.13)
KS: (city cnt: 648, total pop: 2475285, avg pop: 3819.88)
KY: (city cnt: 771, total pop: 3675484, avg pop: 4767.16)
LA: (city cnt: 403, total pop: 4217595, avg pop: 10465.50)
MA: (city cnt: 405, total pop: 6016425, avg pop: 14855.37)
MD: (city cnt: 379, total pop: 4781379, avg pop: 12615.78)
```

Figure 18: Query E output

Shown by Figure 19, the actual MongoDB client command and sample output are:

```
> var map = function() {
...     emit(this.state, {city: [this.city], pop: this.pop});
... };
>
> var reducer = function(key, values) {
...     var res = {city: [], pop: 0};
...     for (var i = 0; i < values.length; ++i) {
...         var val = values[i];
...         res.pop += val.pop;
...         res.city = res.city.concat(val.city);
...     }
...     res.city = res.city.filter((elem, index) =>  res.city.indexOf(elem) === index);
...     return res;
... };
>
>
> var finalizer = function(key, reducedValue) {
...     reducedValue.cityCount = reducedValue.city.length;
...     reducedValue.avgPop = reducedValue.pop / reducedValue.cityCount;
...     return reducedValue;
... };
> var r = z.mapReduce(map, reducer, {out: "state_avg_pop", finalize:finalizer});
> var cur = r.find();
> while (cur.hasNext()) {
... var doc = cur.next();
... print(doc._id, ": (pop:", doc.value.pop, ", cityCount:", doc.value.cityCount, ", avgPop:", doc.value.avgPop, ")");
... };
AK : (pop: 544698 , cityCount: 183 , avgPop: 2976.4918032786886 )
AL : (pop: 4040587 , cityCount: 511 , avgPop: 7907.2152641878665 )
AR : (pop: 2350725 , cityCount: 563 , avgPop: 4175.355239786856 )
AZ : (pop: 3665228 , cityCount: 178 , avgPop: 20591.16853932584 )
CA : (pop: 29754890 , cityCount: 1072 , avgPop: 27756.42723880597 )
CO : (pop: 3293755 , cityCount: 330 , avgPop: 9981.075757575758 )
CT : (pop: 3287116 , cityCount: 224 , avgPop: 14674.625 )
DC : (pop: 606900 , cityCount: 2 , avgPop: 303450 )
DE : (pop: 666168 , cityCount: 46 , avgPop: 14481.91304347826 )
FL : (pop: 12686644 , cityCount: 463 , avgPop: 27400.958963282937 )
GA : (pop: 6478216 , cityCount: 561 , avgPop: 11547.62210338681 )
HI : (pop: 1108229 , cityCount: 70 , avgPop: 15831.842857142858 )
IA : (pop: 2776420 , cityCount: 889 , avgPop: 3123.0821147356583 )
ID : (pop: 1006749 , cityCount: 233 , avgPop: 4320.811158798283 )
```

Figure 19: Query E by MongoDB client command

Query F

Query F: Write a mapReducer to find the least densely-populated state(s).

The mapper for this query emits the state as a key and population as the value for each piece of data.

The reducer then sums the population for each key (state). The reducer returns a collection containing the fields state and total population as key and value, respectively.

The find command is then used on this collection in conjunction with the sort command, which is run on the population totals, and returned in ascending order. The results of the find are then limited to one, which is the state with the smallest population.

The code for query F is in Figure 20, and Figure 21 displays the output.

```
def least_populated_state(mongodb):
    """A mapReducer to find the least densely-populated state(s)."""
    db = mongodb.get_database()
    collection = db[COLLECTION]
    mapper = Code("""
                function() { emit(this.state, this.pop); };
              """)
    reducer = Code("""
                  function(state, pop) { return Array.sum(pop); };
                """)
    result = collection.map_reduce(mapper, reducer, "theResult")
    rs = result.find().sort('value', 1).limit(1)
    return {rs[0]['_id']: rs[0]['value']}
```

Figure 20: Query F implementation

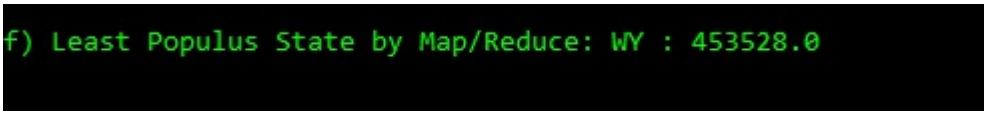

Figure 21: Query F output

Shown by Figure 22, the actual MongoDB client command and sample output are:

```
> var map = function() { emit(this.state, this.pop); };
> var reducer = function(state, pop) { return Array.sum(pop); };
> var r = z.mapReduce(map, reducer, {out: "state_pop"});
> r.find().sort({value: 1}).limit(1);
{ "_id" : "WY", "value" : 453528 }
```

Figure 22: Query F by MongoDB client command

Run the Queries

Both the Python implementations and the MongoDB client script are not platform dependant. They are compatible with POSIX/Darwin and WIN32 platforms.

However, there are some prerequisites are needed before the run.

Prerequisites

To be able to run the queries, the following steps must be done beforehand:

- ☑ Install and run MongoDB server instance properly
- ☑ Install Python2.7 runtime
- ☑ Install pymongo and its dependancies
- ☑ MongoDB command-line client is running and its interactive environment is launched and ready

Set Up

There are some configuration steps also needs to be done.

- ☑ Import the data from `zipcodes.json` to your MongoDB server instance
- ☑ Use any text editor open up `settings.py` and set appropriate values to the below variables:

```
# database related
DB_NAME = "zipcodes"
DB_PROTOCOL = "mongodb://"
DB_HOST = "localhost"
DB_PORT = "27017"
COLLECTION = "zipcodes"
```

Begin Running the Queries

Launch a terminal or a CMD, change the current directory to the project's `src` subdirectory.

Type in the following command:

```
# python queries.py
```

References

- MongoDB CRUD Operations
- MongoDB Aggregation
- Pymongo - Collection level operations
- Aggregation Examples

About Team 1

Team 1 consists of three students, who are:

- Jason Qiao Meng *(Team Lead)*
- Sarah Cooney *(Developer)*
- Mingyuan Li *(Developer)*