The University of Cape Town

CSC4023Z: Big Data Assignment 1

Group: MatGuys
SWNJON004, CHTTIN007, RMRSUV002

Part A: Data

Fortune Top 1000 Companies by Revenue:
https://www.kaggle.com/datasets/surajjha101/fortune-top-1000-companies-by-revenue-2022

rank: Rank of the company as per their revenue in 2021.

name: Name of the company.

revenues: Revenue they generated in 2021 (in US$ Million)**.**

revenuepercentchange: Revenue changed in 2021 over the previous year i.e., 2020.

profits: Profit they generated in the year.

profitspercentchange: Profit changed in 2021 over the previous year i.e., 2020.

assets: Worth of assets the company has in US$ Millions.

marketvalue: Market value of the company

changein_rank: Change in company's rank in 2021 from the last year.

employees: Number of people employed in their company.

Part B: 4 types of NoSQL stores

1. Key-value Model:

The key value store follows a directory model. Every object inserted into the database is given a unique key and this key is used to subsequently fetch or delete that object and its respective value.

The primary key, in the instance of our dataset, will be the company name, such as ‘amazon’. Despite our dataset ranking companies by position in total revenue, there would be no interest or value in fetching data by a primary key referenced as the revenue rank position. The goal of most queries would be to find information regarding a single company referenced by name. It would not be based on finding information per their specific rank. All of this information would be stored within the primary key’s value, also known as a “blob”. This information in the values are not queryable - a major disadvantage of the speedy key-value model. This is the trade-off needed to ensure high speed gets, puts or deletes. As there are 1 000 entries, the speed that key-value provides would suit our dataset. We could even expand to the top 10 000 companies with minimal risk to the query performance.

As mentioned above, a disadvantage of using this system is that we cannot query anything within the value of a specific key. The values returned are schema-less and there is therefore not much structure. We can not do any analysis or effective queries with this set-up, largely rendering it ineffective for analytical processing. Should we be wanting to compare more than one company, this model would not be effective as we can’t divulge into these different values.

Due to these disadvantages, to create a more effective model to ensure we can have some aspect of informed queries, our key’s should be more specific. This would result in the addition of partially duplicated values, which would be housed in new key pairings with added suffixes. A more specific key, allows a more specific and detailed value, which results in better information per query.

As opposed to only setting a single key of ‘amazon’, and all the other 9 fields and their attributes are stored within the value, we would design duplicated keys over and above this “summarised” key. This would allow for more efficient, informed and valuable queries maximising the speed that key-value store provides.

For example, fetching the primary key ‘amazon’ would return a summary of all available information. However, by additionally creating primary keys such as amazon_revenue, amazon_size, and amazon_profit, we can target more specific information, respective to the targeted suffix, at a very fast speed. This would be suitable when we are only interested in a certain part of a business as opposed to fetching all information on that specific business.

1.1 Inserting into the designed redis database:

set **company:amazon** ‘{
“name”: “Amazon”, “rank”: “2”, “changeInRank:” “0”,
“revenues”: “469822”, “revenuePercentChage:” “21.70”,
“profits”: “13673”, “profitPercentChage:” “56.40”,
“marketvalue”: “1658807.30”, “employees:” “1,608,000”, “assets”: “420549”
}’
set **company:amazon_revenue** ‘{
“revenues”: “469822”, “revenuePercentChage:” “21.70”
}’
set **company:amazon_profit** ‘{
“profits”: “13673”, “profitPercentChage:” “56.40”
}’
set **company: amazon_size** ‘{
“marketvalue”: “1658807.30”,
“employees:” “1608000”,
“assets”: “420549”
}’

1.2 Accessing and deleting in the designed redis database:

These gets would be quick, and return specific information based on the attached suffix (available revenue and profit information from amazon, respectively):

get **company:amazon_revenue**
get **company:amazon_profit**

Deleting all duplicate/extra tables:

del **company:amazon_profit**
del **company:amazon_size**
del **company:amazon_revenue**

2. Document-oriented Model:

This model has the advantages of efficient writing and updating, as opposed to the key-value which is only write. It is also a query-based language which provides analysis value in our case. This advantage would allow analysis of companies, their revenues/profits/size, and allow the analyser to draw comprehensive conclusions.

Another advantage of the document store as opposed to key-value is the ability of nested objects within documents. These nested objects can also be directly referenced and quiered. These nested objects are a much more effective method to query within documents and retrieve information, as opposed to a standard relational database join - which is a lot more time-consuming and expensive to implement.

The design of our document model would be highly structured, as we know how many fields are within every object. Our objects would contain a range of named values, paired with nested “child” documents which provide

similar information to a certain theme (i.e. revenue-focused information). Despite designing with a structured approach, the flexibility that document store provides by being schema-less creates less friction for later down the line should an additional column, such as the company's liabilities, need to be added to the dataset.

Seeing as there are no relationships within the dataset, having nested objects in each document entry such as companyRevenue, or companyProfit, increases the efficiency of any query.

2.1 Inserting into the designed mongoDB database:

```
db.company.insert_one({
        name:'Amazon',
        rank: 2,
        changeInRank: 0,
        companyRevenue:
        {
                revenues: 469822,
                revenuePercentChange: 21.70
        }
        companyProfit:
        {
                profits: 13673,
                profitsPercentChange: 56.40
        }
        companySize:
        {
                marketvalue: 1658807.30,
                employees: 1608000,
                assets: 420549
})
```

3. Column-oriented Model:

Designing a column-oriented database refers to having a single table as the instance. This respective table has multiple rows, where values of columns (or column-families) are identified by the primary key (in this case, the company).

Every row can have the same, or differ, in the amount of columns and data per row entry. In our case, every company has the same number of values. However, should additional information, such as listed stock price, be available for only a select number of companies, this information could easily be inserted into a respective column (or column-family) without affecting the columns of other companies, nor needing null values inserted.

The advantages of using a column-oriented model must be noted in their efficient ability of processing columns to achieve some form of analytics processing. The lookup is extremely fast and thus makes it a great option in our case for finding trends or patterns within the 1000 companies in our dataset. For example, we could query every company's "revenuePercentageChange" column efficiently, and gain the average change of all the companies revenue Year on Year in milliseconds. However, this query syntax is limited as opposed to the document model. The queries are based on columns and not along rows or specified companies. We therefore cannot perform nested, or comparable queries, such as those possible within the document store.

Columns stored within column-families reduce time wasting when writing to certain values. This is because the db targets only the respective column-family opposed to the full row affected (thus dealing with less data point entries). Our data does not contain much variety, the advantage of the flexibility of the column-families won't be maximised in our case.

When designing the database, we should bear the analytical processing and speed advantages in mind. The following column families should be defined before data is loaded: companyRevenue, companyProfit, companySize. We would be able to quickly process and analyse the entries of each row specific to these columns.

3.1 Inserting into the designed Cassandra column-oriented database:

INSERT into **companies** (name, rank, changeInRank) VALUES (‘amazon’, ‘2’, ‘0’);

CREATE table **companySize** (name *text,* marketvalue *decimal*, employees *decimal*, assets *decimal*, PRIMARY KEY (name));
CREATE table **companyProfit** (name *text,* profits *decimal*, profitsPercentChange *decimal*, PRIMARY KEY (name));
CREATE table **companyRevenue** (name *text,* revenues *decimal*, revenuesPercentChange *decimal*, PRIMARY KEY (name));

INSERT into **companySize**(name, marketvalue, employees, assets) VALUES (‘amazon’, ‘1658807.30’, ‘420549’);
INSERT into **companyProfit**(name, profits, profitsPercentChange) VALUES (‘amazon’, profits: ‘13673’, profitsPercentChange: ‘56.40’);
INSERT into **companyRevenue**(name, revenues, revenuesPercentChange) VALUES (‘amazon’, revenues: ‘469822’, revenuePercentChange: ‘21.70’);

4. Graph database model:

The graph database model would not work for this data set. The graph database model is mainly used to store objects and their relationships with other objects. Example use cases are e-commerce recommendation engines, social media sites, and identity and access management. Given this, this database model would not work for the given data set, as there are no relevant relationships between companies we would need to depict and store. Every company row is independent of others, and only related to the others by their rank in revenue. The columns of the database are also independent of each other. Each column simply stores particular information about a given company. Therefore, there would be no use in attempting to extract relationships between either rows or columns of the dataset.

Part C: Relational vs NoSQL vs Polyglot

In order to choose a database appropriately we would need to consider the following:

- Do we value availability over consistency
- How often will your database schemas change?
- Is the data structured or unstructured
- How complex the queries and analysis of the data is needed

Relational databases are best utilised when dealing with highly structured data with complex relationships between records across multiple relations. Relational databases require the entire schema to be modelled up front, and do not respond/handle changes to the schema in an effective manner. Changes in schema will require downtime of the database. They provide comprehensive querying which often require complex joins.

NoSQL Databases are best utilised when storing semi-structured/unstructured data since they don’t enforce a concrete schema for tables. This allows for updates without needing to change the structure of the entire table or

adding redundant fields to records. The data stored in NoSQL databases are generally stored in a format in which it will be accessed. Due to the unstructured nature of NoSQL DBs queries have limited complexity.

Relational databases comply strictly with ACID properties and ensure that the data is consistent across the various partitions that may exist. Consistency of the data is prioritised over performance. NoSQL Databases support only eventual consistency but have increased availability when compared to Relational databases.

Polyglot persistence allows us to split the data into multiple databases. This allows you to leverage the most appropriate store and query framework for specific sections of your data. Polyglot persistence would maximise performance and scalability of your system. The use of multiple databases does introduce complexity as the development team are now required to maintain multiple databases. Maintenance of polyglot systems is often time consuming.

The dataset chosen for this assignment consists of the ranks of the top performing companies, each record in the relation is unique and self-contained. Our dataset is a single relation and therefore does not contain relationships. Availability of the data is prioritised and eventual consistency meets requirements.
The general analysis of our dataset would most likely entail generating summary key performance indicators of the various companies. Filtering would consist of simplistic tasks such as selecting companies(rows) that fit the specified criteria. The robust querying framework of relational databases would not be necessary to derive meaningful insights and has the caveat of potentially increasing the complexity of database operations.

The structure of our data and nature of database operations mentioned above supports the choice of utilising NoSQL as it will aid in meeting these requirements whilst allowing us to build a maintainable/scalable database. A polyglot system would allow the use of a relational database. This would allow us to increase the complexity of queries but has a caveat of negatively impacting the development team maintaining the system.

Part E: Work Allocation

Jonathan: 3x types of NoSQL store (15 marks)
Mukundi: 1x types of NoSQL store and 1x Documentation (17 marks)
Suvanth: Relational vs NoSQl vs Polyglot and 1x Documentation (16 marks)

Part D: x2 Sample Code

NoSQL MongoDB Guide and Sample Code Manual

Introduction

This is a guide to aid developers in the initial stages of using MongoDB as a document store. We present Python sample code and the key information required to make effective meaningful use of MongoDB.

Sample Code Repository Link

https://github.com/Suvanth/Fortune-Top-1000-Companies-NoSQL

About Dataset chosen

The dataset was obtained from Kaggle.com and contains data regarding the financial performance of top performing public and privately held United States corporations.

Data Preparation

The data, however, contained inconsistencies with regards to data types and format. This necessitated data cleaning and preparation so that meaningful analysis could be carried out. The data cleaning process is detailed in the processing.py script. The Pandas module was used to aid in processing the raw csv data into structured JSON objects. The resulting CSV is detailed below:

Column Title	Details	Type
rank	Rank of the company as per their revenue in 2021	Integer
name	Name of the company	String
revenues	Revenue they generated in 2021 (in US$ Million)	Float - Currency
revenue_percent_change	Revenue changed in 2021 over the previous year i.e., 2020	Float - Percentage
profits	Profit they generated in 2021 (in US$ Million)	Float - Currency
profits_percent_change	Profit changed in 2021 over the previous year i.e., 2020	Float - Percentage
assets	Worth of assets the company have in US$ Millions	Float - Currency
market_value	Market value of the company	Float - Currency
change_in_rank	Change in company's rank in 2021 from the last year	Integer
employees	Number of people employed in their company	Integer

JSON Usage

MongoDB stores the data in a JSON-like format (binary JSON), which is the binary encoded version of JSON, and is optimised for performance and space. This motivated the processing of the CSV data into a JSON document with multiple objects with built in hierarchies. General object structure:

```
{
   "name": "Walmart",
   "rank": 1,
   "change_in_rank": 0,
   "companyRevenue": {
     "revenues": 572754.0,
     "revenuePercentChange": 2.4
   },
   "companyProfit": {
     "profits": 13673.0,
     "profitPercentChange": 1.2
   },
   "companySize": {
     "assets": 244860.0,
     "marketValue": 409795.0,
     "employeeCount": 2300000
   }
 }
```

MongoDB Setup

MongoDB Instance

Note that the mongod process must be running in order for this sample Python code to connect with your local instance of the database.
Start the server and shell using the following commands:

```
brew services start mongodb-community
```

If you don't want a background service:

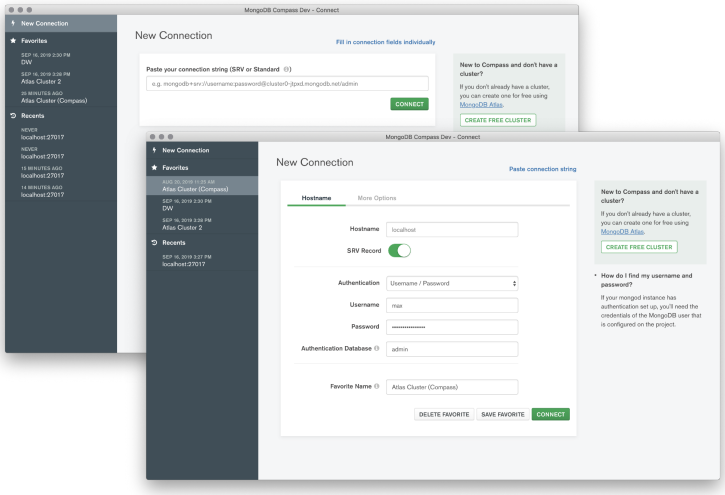
```
mongod
```

End services by entering:

```
brew services stop mongodb-community
```

MongoDB Compass

MongoDB Compass is a convenient tool for browsing through the data stored in a MongoDB database through a graphical interface. It removes the burden of having to remember the names of obscure databases or collections, and you can navigate to any database or collection on your MongoDB server with just a few clicks.



mongoGuide.py as a Learning Tool

The mongoGuide.py script consists of various functions that display valuable information required when starting to use MongoDB.

Python Modules Required for MongoDB operations in mongoGuide.py

- Pymongo {pip install Pymongo}
- JSON {pip install JSON}

Basic Steps to using MongoDB in Python

1. Create MongoDB service instance via terminal using:
   `mongod` or `brew services start mongodb-community`
   The port number is set to 27017 by default.

2. Connect to the localhost at port 27017 in MongoDB Compass to visually monitor the local databases and collections present. (OPTIONAL)

3. In Python import the Pymongo library, then create a connection to the locally hosted database. The following code extract may prove useful. It is important to instruct the Mongo database to close a client connection and free the associated database resources when database operations are completed.

```
from pymongo import MongoClient
client = MongoClient( )
client = MongoClient("mongodb://localhost:27017/")
# database operations here
client.close( )
```

4. Creating a database

```
client_user = pymongo.MongoClient() # creates mongo client
db_name='FortuneCompanies' # name of DB to be created
db = client_user[db_name] # creating DB
```

5. Accessing Database Objects:
   To create a database or switch to an already created database we use:

```
mydatabase = client['name_of_the_database'] #Dictionary Style
mydatabase = client.name_of_the_database    #Dot approach
```

6. Insert Data into a database collection
   The insert_many or insert_one methods may be used. Below is an extract of inserting the `Fortune1000.json` dataset into the `CompanyRanks` Collection.

```
def load_mass_json_document(self, db, data_json_file):
    with open(data_json_file) as file:
        file_data = json.load(file)
    db.CompanyRanks.insert_many(file_data)
```

7. Querying and Sorting in MongoDB
   There are certain query and sorting functions which are used to filter the data in the database. The most commonly used approaches are find, count and aggregate.
   The mongoGuide.py script contains the following functions `sort_documents()`, `rank_calculations()`, `kpi_calculations()` and `logical_operators()`. The functions mentioned above make use of the various filtering approaches commonly used to perform analysis on MongoDB collections.

8. Additional useful functions presenting CRUD operations in mongoGuide.py
   `delete_documents()` – This function shows how to delete documents in a collection based on criteria defined by the author.
   `update_documents()` - This function shows how to update documents and commit changes
   `drop_collections_db()` – Demonstrates how to programmatically drop Mongo collections and databases.

Guide to writing queries for MongoDB Collections

We can query a MongoDB database using PyMongo with the find function to get all the results satisfying the specified conditions. The find_one function may also be used in order to return only one result satisfying the specified condition.

The following is the syntax of the find and find_one:

```
<name_of_collection>.find( {<< query >>} , { << fields>>}
```

Filter based on fields

For instance, you have hundreds of fields and you want to see only a few of them. You can do that by just putting all the required field names with value 1. For example:

```
company_rank_collection.find_one( {}, {"name" : “Walmart”})
```

```
{
   "name": "Walmart",
   "rank": 1,
   "change_in_rank": 0,
   "companyRevenue": {
     "revenues": 572754.0,
     "revenuePercentChange": 2.4
   },
   "companyProfit": {
     "profits": 13673.0,
     "profitPercentChange": 1.2
   },
   "companySize": {
     "assets": 244860.0,
     "marketValue": 409795.0,
     "employeeCount": 2300000
   }
 }
```

Filter with a condition

We will provide a condition in the first braces and fields to discard in the second. Consequently, it will return the first document with rank is equal to 55 and change_in_rank is equal to -6 and will also discard all documents where the field employeeCount is equal to 0.

```
company_rank_collection.find_one( {"rank" : 55, "change_in_rank" : -6}, {"employeeCount" : 0} )
```

```
{
    "name": "Lockheed Martin",
    "rank": 55,
    "change_in_rank": -6,
    "employeeCount": 1000
}
```

Filter based on Comparison and Logical Operators

The following are the nine comparison operators in MongoDB

Comparators	Functionality
$eq	It will match the values that are equal to a specified value.
$gt	It will match the values that are greater than a specified value.
$gte	It will match all the values that are greater than or equal to a specified value.
$in	It will match any of the values specified in an array.
$lt	It will match all the values that are less than a specified value.
$lte	It will match all the values that are less than or equal to a specified value.
$ne	It will match all the values that are not equal to a specified value.
$nin	It will match none of the values specified in an array.

Logic	Functionality
$and	It will join query clauses with a logical AND and returns all documents that match both the conditions.
$not	It will invert the effect of a query and returns documents that do not match the query expression.
$nor	It will join the query clauses with a logical NOR and return all documents that fail to match the clauses.
$or	It will join the query clauses with a logical OR and return all documents that match the conditions of either clause.

Table Source: https://www.analyticsvidhya.com

Examples of query using these comparison operators to find companies with a `change_in_rank <=3 and rank > 5`

```
def logical_operators(self, db):
    collection = db['CompanyRanks']
    results = collection.find({"$and": [
        {
            "change_in_rank": {"$lte": 3}
        },
        {
            "rank": {"$gt": 5}
        }
    ]
    })
    for doc in results:
        print(doc)
```

Filter with Regular Expressions

Regular Expressions are of great use when you have text fields and you want to search for documents with a specific pattern. Regular expression cheat sheet : https://www.pcwdld.com/python-regex-cheat-sheet
It can be used with the operator $regex and we can provide value to the operator for the regex pattern to match. In the query below we use regex to delete all documents where the name field starts with the character A.

```
collectionDelete = db['CompanyRanksDelete']
# delete all documents which have a name starting with A
manyDeleteQuery = { "name": {"$regex": "^A"} }
x = collectionDelete.delete_many(manyDeleteQuery)
```

Aggregation Framework

MongoDB's aggregation pipeline provides a framework to perform a series of data transformations on a dataset. The following is its syntax:

```
your_collection.aggregate( [ { <stage1> }, { <stage2> },.. ] )
```

The first stage takes the complete set of documents as input, and from there each subsequent stage takes the previous transformation's result set as input to the next stage and produces the output.

Example of query using the aggregate framework:

```
def rank_calculations(self, db):
    cursor = db.CompanyRanks.aggregate(
        [
            {
                "$group":
                {
                    "_id": "RankCalcs",
                    "max_change_rank": {"$max": "$change_in_rank"},
                    "min_change_rank": {"$min": "$change_in_rank"},
                    "average_change_rank": {"$avg": "$change_in_rank"},
                }
            }
        ]
    )
    for item in cursor:
        print(item)
```

NoSQL Redis Guide and Sample Code Manual

Introduction

This part of the guide focuses on redis, a NoSQL database that uses key value pairs to store data. Redis has three basic functions, set, get, and delete. For this manual, we set up a docker container that runs a redis database. We then connect to our database from our local machine. The python file 'redis-demo.py' demonstrates how redis can be used to connect to the database, load data, and access the data.

Redis Database Setup

1. Run: `$ docker pull redis`
2. Run: `$ docker run --name redis-demo -d redis`

These two commands will pull the docker image for redis, and then start the server. We then open the docker container using docker desktop and copy the connection link in the top right corner of the screen.

Connecting to the redis database

We use python to connect to the database and run queries to insert, access, and delete the data.
In python, we use this line to connect to the database:

```
def connect(redis_url):
    '''
    Connects to the redis database
    ARGS: url
    Return: redis instance
    '''
    r = redis.StrictRedis.from_url(redis_url)
    return r
```

From here, we can then insert the data into the database using the set method.

```
def set_row(r,key,value):
    r.set(key,value)
    return
```

Accessing data

We can then access the data by key using the get function:

```
def get_row(r,key):
    '''
    Returns the corresponding row given a key
    ARGS: redis instance, key
    '''
    return r.get(key)
```

Deleting data

We can delete a row from the database by key using the delete function:

```
def delete_row(r,key):
    r.delete(key)
    return
```

Redis Pipelining

Redis also has pipelining, where multiple operations can be done at the same time. This improves efficiency, as operations that do not have to wait for each other, can be executed concurrently. The code block below demonstrates how pipelining can be used to speed up operations. Here, we first add all the set operations to the pipeline, then run the execute command for the pipeline to start. We compared pipelining to a none pipelined way of adding data, and for this dataset, adding all the data using pipelining took 0.3 seconds, while a none pipelined method took 5.4 seconds.

```
def set_company_pipeline(r,df):
    '''
    Loads data from a dataframe into the redis database
    ARGS: redis instance, filename
    Return: Boolean
    '''
    pipeline = r.pipeline()
    for index in df.index:
        data = df.loc[index].to_dict()
        pipeline.set(data['name'],json.dumps(data))
    pipeline.execute()
    Return
```

Pipelining can also be used for other operations like delete and get, as illustrated in the code base.

Key selection

Given that in a key-value oriented database like redis, access to data is only through the primary key, key selection is an important aspect of the database design. The keys have to be distinct, such that we can get expected results. For example, in the example data set, the name of the company is a distinct feature of the data, hence we can use this as the primary key.

In a key-value database, we can't query the values, as they are just interpreted as a blob of data. To combat this, we can have key value pairs that have a subset of some data. For example, in this dataset, we can have a key for the company name with the value being all the company data. We can then have another key for company_revenue, with the key being the revenue only. We can group other elements of the data like this as well. More information on this is in Part B.

Need additional support?

The sample code provided contains additional useful examples and the documentation is extensive. The Doc Strings and in-line comments will support you in adjusting to a NoSQL data store.