**DATABASE SYSTEMS**
**SOEN 363 – WINTER 2020**
**Project Phase 2:**
**Firas Sawan ID#26487815**
**Kenza Boulisfane ID# 40043521**

**(a) Download a big real dataset; it is recommended to get a dataset of at least 0.5 GBs.**

The dataset we chose can be found using the following link:
https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1410014001

**(b) Provide the data model for your datasets, i.e., graph, document, key-value, or column-store.**

Our dataset was grouped into one collection called phase3. A typical document in this collection contains 16 fields. The following is a document data model:

```
{
    _id: ObjectId(''),
    REF_DATE: String,
    GEO: String,
    DGUID: String,
    Sex: String,
    'Age group': String,
    UOM: String,
    UOM_ID: Integer,
    SCALAR_FACTOR: String,
    SCALAR_ID: Integer,
    VECTOR: String,
    COORDINATE: String,
    VALUE: Long,
    STATUS: String,
    SYMBOL: String,
    TERMINATED: String,
    DECIMALS: Integer
}
```

An example from our database would be the following document:

```
{
    _id: ObjectId('5e854097337881432cd3cea8'),
    REF_DATE: '1997-01',
    GEO: 'Canada',
    DGUID: '',
    'Beneficiary detail': 'All types of income benefits',
    Sex: 'Both sexes',
    'Age group': '15 to 29 years',
    UOM: 'Persons',
    UOM_ID: 249,
    SCALAR_FACTOR: 'units ',
    SCALAR_ID: 0,
    VECTOR: 'v64579140',
    COORDINATE: '1.1.1.17',
    VALUE: 298140,
    STATUS: '',
    SYMBOL: '',
    TERMINATED: '',
    DECIMALS: 0
}
```

**(c) Create a NoSQL database for a real dataset of your choice AND (d) Load the dataset into your NoSQL system.**

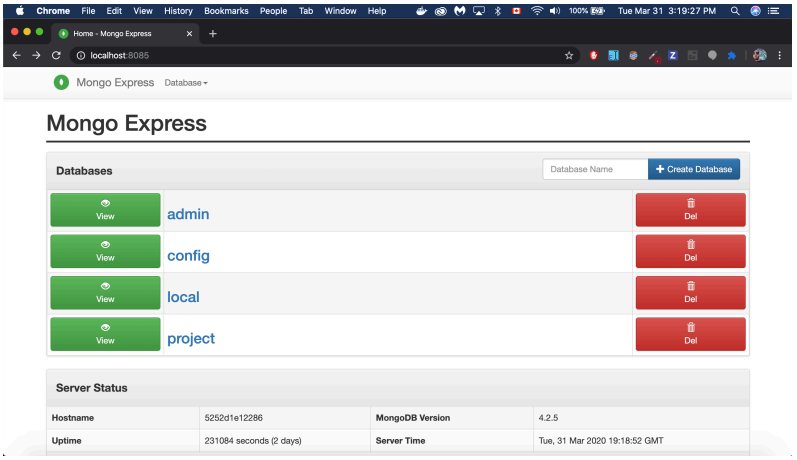
Please follow the below steps to create a NoSQL database and then load the dataset into the system.

Step 1:
Open a terminal window and start Docker using the command: *docker-compose up*

Step 2:
Get the docker Container ID (for port 27017) using the command: *docker ps*

Step 3:
Connect docker to MongoDB using the command: *docker exec -it <CONTAINER_ID> bash*

Step 4:
In your terminal window cd into the scripts folder of the project and import the dataset using the command: *./mongoseed.sh*
Note that if you encounter a "permission denied" message here, please use the following command:
*chmod +x ./mongoseed.sh*




Step 5:
Wait until the importation is done and then head to *http://localhost:8085/* The page should look as the screenshot to the right. Note the green "project" tab which is our imported dataset.

Step 6: In order to run the queries, we must connect to our database first using the command: *mongo --host mongo --port 27017 --username root --password root --authenticationDatabase=admin*

**(e) Write at least 10 different queries, that show some useful information about the dataset. This should include different aspects of your NoSQL.**

- First, we need to indicate which database is to be used. We can do this through terminal by running the command: *show dbs*

- The command above will give us a list of available databases. From there we can select the database we just created using the command: *use project*

**QUERY #1:**

// For each location in the province of Ontario, find the total value of all benefits ever recieved, the average value of all benefits ever recieved, the minimum value of all benefits ever recieved and the maximum value of all benefits ever recieved in that location. **(Sample output in black screenshot)**

```
db.phase3.aggregate([{
   $match: {
      "GEO": {$regex:".*Ontario.*"}
   }
}, {
   $group: {
      "_id": "$GEO",
      "totalVal": {
         $sum: "$VALUE"
      },
      "avgVal": {
         $avg: "$VALUE"
      },
      "minVal": {
         $min: "$VALUE"
      },
      "maxVal": {
         $max: "$VALUE"
      }
   }
}, {
   $project: {
      "_id": 0,
      "GEO":"$_id",
      "total_benefits_paid": "$totalVal",
      "average_benefits_paid": "$avgVal",
      "min_benefits_paid": "$minVal",
      "max_benefits_paid":"$maxVal"
   }
}]).pretty();
```

```
{
        "GEO" : "Elgin, Ontario",
        "total_benefits_paid" : 5012670,
        "average_benefits_paid" : 326.8564162754304,
        "min_benefits_paid" : 0,
        "max_benefits_paid" : 5280
}
{
        "GEO" : "Ontario",
        "total_benefits_paid" : 552571250,
        "average_benefits_paid" : 36030.98917579551,
        "min_benefits_paid" : 320,
        "max_benefits_paid" : 437550
}
{
        "GEO" : "Haliburton, Ontario",
        "total_benefits_paid" : 609880,
        "average_benefits_paid" : 39.767866458007305,
        "min_benefits_paid" : 0,
        "max_benefits_paid" : 530
}
```

**QUERY #2:**

// Find out the coordinates of all males that received benefits in January of 1997, we can do so via the following query: **(Sample output in black screenshot)**

```
db.phase3.find({
   $and: [{}, {
      "Sex": "Males"
   }, {
      "REF_DATE": "1997-01"
   }]
}, {
   "REF_DATE": 1,
   "Sex": 1,
   "COORDINATE": 1,
   "GEO": 1,
   "_id": 0
}).pretty();
```

```
{
        "REF_DATE" : "1997-01",
        "GEO" : "Division No. 1, Newfoundland and Labrador",
        "Sex" : "Males",
        "COORDINATE" : "17.16.2.17"
}
{
        "REF_DATE" : "1997-01",
        "GEO" : "Division No. 2, Newfoundland and Labrador",
        "Sex" : "Males",
        "COORDINATE" : "18.1.2.1"
}
{
        "REF_DATE" : "1997-01",
        "GEO" : "Division No. 2, Newfoundland and Labrador",
        "Sex" : "Males",
        "COORDINATE" : "18.1.2.5"
}
```

**QUERY #3:**

// Find the geolocation of all females aged between 15-24 years old that received benefits in the year 2014. We can do so with the help of $regex that can specify to include all the months of 2014. For example (2014-01, 2014-02 etc etc), we display the coordinate field to distinguish between documents in the database. **(Sample output in black screenshot)**

```
db.phase3.find({
   $and: [{}, {
      "Age group": "15 to 24 years"
   }, {
      "Sex": "Females"
   }, {
      "REF_DATE": {
         $regex: ".*2014.*"
      }
   }]
}, {
   "REF_DATE": 1,
   "Sex": 1,
   "COORDINATE":1,
   "GEO": 1,
   "Age group": 1,
   "_id": 0
}).pretty();
```

```
{
        "REF_DATE" : "2014-01",
        "GEO" : "Division No. 1, Newfoundland and Labrador",
        "Sex" : "Females",
        "Age group" : "15 to 24 years",
        "COORDINATE" : "17.16.3.2"
}
{
        "REF_DATE" : "2014-01",
        "GEO" : "Division No. 2, Newfoundland and Labrador",
        "Sex" : "Females",
        "Age group" : "15 to 24 years",
        "COORDINATE" : "18.1.3.2"
}
{
        "REF_DATE" : "2014-01",
        "GEO" : "Division No. 2, Newfoundland and Labrador",
        "Sex" : "Females",
        "Age group" : "15 to 24 years",
        "COORDINATE" : "18.2.3.2"
}
```

**Query #4:**

// Find all income benefit values higher than 20000 sorted in **ascending order of value** and exclusively in the geolocations of either Quebec and Ontario. Then sort the result in descending order of the date. **(Sample output in black screenshot).**

```
db.phase3.find({
   $and: [{
      "VALUE": {
         $gt: 20000
      }
   }, {
      $or: [{
         "GEO": {
            $regex: ".*Quebec.*"
         }
      }, {
         "GEO": {
            $regex: ".*Ontario.*"
         }
      }]
   }]
}, {
   "GEO": 1,
   "_id": 0,
   "VALUE": 1,
   "REF_DATE": 1
}).sort({
   "VALUE": 1,
   "REF_DATE": -1,
}).pretty();
```

```
{ "REF_DATE" : "2014-09", "GEO" : "Quebec", "VALUE" : 20010 }
{ "REF_DATE" : "2013-03", "GEO" : "Montréal, Quebec", "VALUE" : 20010 }
{ "REF_DATE" : "2013-02", "GEO" : "Ontario", "VALUE" : 20010 }
{ "REF_DATE" : "2010-11", "GEO" : "Montréal, Quebec", "VALUE" : 20010 }
{ "REF_DATE" : "2010-07", "GEO" : "York, Ontario", "VALUE" : 20010 }
{ "REF_DATE" : "2008-08", "GEO" : "Peel, Ontario", "VALUE" : 20010 }
{ "REF_DATE" : "2007-01", "GEO" : "Montréal, Quebec", "VALUE" : 20010 }
{ "REF_DATE" : "1997-01", "GEO" : "Québec, Quebec", "VALUE" : 20010 }
{ "REF_DATE" : "2009-08", "GEO" : "Waterloo, Ontario", "VALUE" : 20020 }
{ "REF_DATE" : "2007-11", "GEO" : "Ontario", "VALUE" : 20020 }
{ "REF_DATE" : "2006-01", "GEO" : "Toronto, Ontario", "VALUE" : 20020 }
{ "REF_DATE" : "2004-12", "GEO" : "Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "2004-01", "GEO" : "Ontario", "VALUE" : 20020 }
{ "REF_DATE" : "2003-03", "GEO" : "Montréal, Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "2002-09", "GEO" : "Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "2002-08", "GEO" : "Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "1999-07", "GEO" : "Ontario", "VALUE" : 20020 }
{ "REF_DATE" : "1999-05", "GEO" : "Montréal, Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "1997-04", "GEO" : "Quebec", "VALUE" : 20020 }
{ "REF_DATE" : "2012-12", "GEO" : "Ontario", "VALUE" : 20030 }
Type "it" for more
```

**Query #5:**

// Find the geolocation with the highest value for income benefits in the year 2012 where the sex of the beneficiary is Males. **(Full output in black screenshot).**

```
db.phase3.find({
   $and: [{
      "REF_DATE": {
         $regex: ".*2012.*"
      },
      "GEO": {
         $not: {
            $regex:".*Canada.*"
         }
      }
   }, {
      "Sex": "Males"
   }]
}, {
   "_id": 0,
   "VALUE": 1,
   "Sex": 1,
   "GEO": 1
}).sort({
   VALUE: -1
}).limit(1).pretty();
```

```
{ "GEO" : "Quebec", "Sex" : "Males", "VALUE" : 179070 }
```

**Query #6:**

// Find the total value of all benefits collected in Ontario the year 2010, we can do this using an aggregate function. **(Full output in black screenshot).**

```
db.phase3.aggregate([{
   $match: {
      $and: [{
         "GEO": {
            $regex: ".*Ontario.*"
         }
      }, {
         "REF_DATE": {
            $regex: ".*2010.*"
         }
      }]
   }
}, {
   $group: {
      "_id": 0,
      "totalBenefitsValue": {
         $sum: "$VALUE"
      }
   }
}, {
   $project: {
      "_id": 0
   }
}]);
```

```
{ "totalBenefitsValue" : 88940860 }
```

**Query #7:**

For the first 3 months in the year 2010, find the location with the highest value of collected benefits. Display the location and the value collected. **(Full output in black screenshot).**

```
db.phase3.aggregate([{
      $match: {
         $and: [{
            "GEO": {
               $regex: ".* .*"
            }
         }, {
            "REF_DATE": {
               $in: [
                  "2010-01",
                  "2010-02",
                  "2010-03"]
            }
         }]
      }
   }, {
      $group: {
         "_id": "$REF_DATE",
         "maxVal": {
            $max: "$VALUE"
         },
         "GEO": {
            $first: "$GEO"
         }
      }
   }, {
      $sort: {
         _id: -1
      }
   }, {
      $project: {
         "_id": 0,
         "Location With Highest Earning": "$GEO",
         "Month Referenced": "$_id",
         "Value Earned": "$maxVal"
      }
   }
]).pretty();
```

```
{
        "Location With Highest Earning" : "Newfoundland and Labrador",
        "Month Referenced" : "2010-03",
        "Value Earned" : 140530
}
{
        "Location With Highest Earning" : "Newfoundland and Labrador",
        "Month Referenced" : "2010-02",
        "Value Earned" : 141750
}
{
        "Location With Highest Earning" : "Newfoundland and Labrador",
        "Month Referenced" : "2010-01",
        "Value Earned" : 145030
}
>
```

**Query #8:**

// Find the total value of all beneficiaries that were received per geolocation, this should cover all years available in the dataset and display the total value that each location has been granted. Sorted in ascending alphabetical order of Geolocation. **(Sample output in black screenshot).**

```
db.phase3.aggregate([{
   $group: {
      "_id": '$GEO',
      "TotalValue": {
         $sum: '$VALUE'
      }
   }
}, {
   $sort: {
      "_id": 1
   }
}, {
   $project: {
      "_id": 0,
      "category": '$_id',
      "Total Recieved": '$TotalValue'
   }
}]).pretty();
```

```
{ "category" : "Abitibi, Quebec", "Total Recieved" : 3405470 }
{ "category" : "Abitibi-Ouest, Quebec", "Total Recieved" : 3096110 }
{ "category" : "Acton, Quebec", "Total Recieved" : 1655730 }
{ "category" : "Alberni-Clayoquot, British Columbia", "Total Recieved" : 2697310 }
{ "category" : "Albert, New Brunswick", "Total Recieved" : 2390940 }
{ "category" : "Alberta", "Total Recieved" : 129176410 }
{ "category" : "Algoma, Ontario", "Total Recieved" : 8975550 }
{ "category" : "Annapolis, Nova Scotia", "Total Recieved" : 3005710 }
{ "category" : "Antigonish, Nova Scotia", "Total Recieved" : 3260120 }
{ "category" : "Antoine-Labelle, Quebec", "Total Recieved" : 5498500 }
{ "category" : "Argenteuil, Quebec", "Total Recieved" : 3007770 }
{ "category" : "Arthabaska, Quebec", "Total Recieved" : 5962610 }
{ "category" : "Asbestos, Quebec", "Total Recieved" : 1453570 }
{ "category" : "Avignon, Quebec", "Total Recieved" : 4424320 }
{ "category" : "Baffin, Nunavut", "Total Recieved" : 697740 }
{ "category" : "Beauce-Sartigan, Quebec", "Total Recieved" : 3716520 }
{ "category" : "Beauharnois-Salaberry, Quebec", "Total Recieved" : 4435100 }
{ "category" : "Bellechasse, Quebec", "Total Recieved" : 2359350 }
{ "category" : "Bonaventure, Quebec", "Total Recieved" : 5168480 }
{ "category" : "Brant, Ontario", "Total Recieved" : 7341320 }
Type "it" for more
```

**Query #9:**

// For each year, find the total sum of benefits paid in British Columbia. Then sort the years in ascending order of that total. **(Full output shown in black screenshot).**

```
db.phase3.aggregate([{
      $match: {
         "GEO": {$regex: ".*British Columbia.*"}
      }
   },
   {
      $group: {
         "_id": { $substr: ["$REF_DATE", 0, 4]},
         "VALUE": {$sum: "$VALUE"}
      }
   },
   {
      $sort: {
         "_id": 1
      }
   },
   {
      $project: {
         "Year": "$_id",
         "totalBenefits": "$VALUE",
         "_id": 0
      }
   }
]);
```

```
{ "Year" : "1997", "totalBenefits" : 26529380 }
{ "Year" : "1998", "totalBenefits" : 27366150 }
{ "Year" : "1999", "totalBenefits" : 25152450 }
{ "Year" : "2000", "totalBenefits" : 21822650 }
{ "Year" : "2001", "totalBenefits" : 23752020 }
{ "Year" : "2002", "totalBenefits" : 26472720 }
{ "Year" : "2003", "totalBenefits" : 26506950 }
{ "Year" : "2004", "totalBenefits" : 24451890 }
{ "Year" : "2005", "totalBenefits" : 21813090 }
{ "Year" : "2006", "totalBenefits" : 18910950 }
{ "Year" : "2007", "totalBenefits" : 17834310 }
{ "Year" : "2008", "totalBenefits" : 19921020 }
{ "Year" : "2009", "totalBenefits" : 35540580 }
{ "Year" : "2010", "totalBenefits" : 34091260 }
{ "Year" : "2011", "totalBenefits" : 27886150 }
{ "Year" : "2012", "totalBenefits" : 24726920 }
{ "Year" : "2013", "totalBenefits" : 22690500 }
{ "Year" : "2014", "totalBenefits" : 16846920 }
>
```

**Query #10:**

// Find the total value of benefits that was paid to both people with declared income and those without declared income. **(Full output shown in black screenshot).**

```
db.phase3.aggregate([{
   $match: {
      $and: [{
         "Beneficiary detail": {
            $regex: ".*with.*"
         }
      }, {
         "Beneficiary detail": {
            $regex: ".*without.*"
         }
      }]
   }
}, {
   $group: {
      "_id": 0,
      "totalBenefitsValue": {
         $sum: "$VALUE"
      }
   }
}, {
   $project: {
      "_id": 0
   }
}]);
```

```
{ "totalBenefitsValue" : 1523904420 }
```

**(f) Investigate the balance between the consistency and availability in your NoSQL system.**

To begin this investigation, I will start by briefly describing what consistency and availability actually mean and provide a brief about MongoDB:

- **Consistency** in the CAP theorem refers to the fact that when two users access the system at the same time they are expected to see the same data. It exclusively refers to data consistency across a cluster of nodes and not on a single server/node. Furthermore, consistency means that if some data is written to the distributed system, then the user should be able to read the same data at any point in time from any nodes of the system. If that is not possible, and if the data is in an inconsistent state then the system should simply return an error.

- When we speak about **Availability** we mainly refer to the fact that an available system is one that is up 24/7 and responds in a reasonable time. Furthermore, an available system is one that can always perform reads/write operations on any non-failing node of the cluster successfully and without any error.

For the purposes of this project we chose MongoDB, a cross-platform document-oriented database program. It stores data in flexible, JSON-like documents, which means that the fields can vary from document to document and the data structure can be changed over time. Furthermore, MongoDB is considered to be a single leader based system which means that it has one leader node to which read/write operations come to, and multiple node replicas that listen to this leader and update themselves asynchronously. Note that in this system, non-leader nodes also act as a backup in case the leader node stops functioning and each node listens to all other nodes' heartbeat to keep track on their status as "dead" or "Alive".

Figure 1 below shows a visual representation of the single leader based structure discussed above.

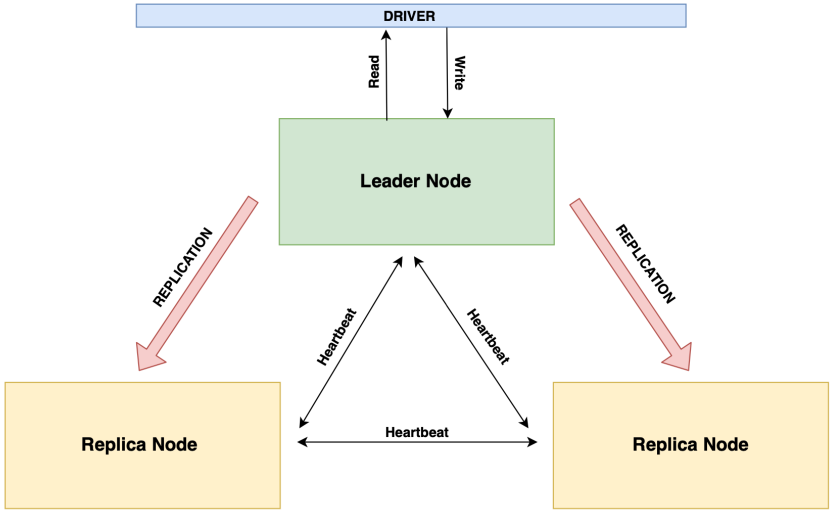



Figure 1: Single Leader Based Strucutre (MongoDB default behavior).

In terms of consistency and availability when it comes to the system described above, the default behaviour includes sending all reads/write operations to the leader node shown in the graph above, which allows for the system to be consistent. However, while this behavior allows our system to be consistent, it does not allow it to be available, i.e. up 24/7 without interruption. This is mainly because we can have scenarios where the lead node might "die" which means that the system would need to elect one of the replica nodes to respond, a process that consumes time. Hence, the time taken until a replica node is elected constitutes a period where our system is not available, i.e. we can not perform any reads or writes during this period.

One other scenario that we should take into account when discussing the default behavior of MongoDB is that when we have network issues or when a node disconnects from the cluster due to a partitioned network which makes it unavailable. What we conclude from this is that the default behavior of MongoDB is that which is characterized by being consistent, but not available.

Note that there are ways however to make our system both fully consistent and available. As mentioned above. This can be done by adopting two techniques:

1) The first is to make it possible for the MongoDB client to read from replica nodes. This can be done through the preferences of the client but it raises a new problem for us in the sense that we are breaking consistency. For clarification, consider the scenario where a read operation is being performed on a replica node that has not yet been updated. This would mean that we would have inconsistent results since the data in the replica node has not yet been updated with the latest information. In this scenario we are replacing consistency, with what is known as eventual consistency. To solve this issue we need to also adopt the second technique below.

2) The second technique involves employing MongoDB's write concerns. Write concerns are a way for us to specify the number of nodes to which the data should be written in order to constitute a successful write operation. We could pass a write-option which specifies that the data should be written to the majority of the nodes or to all of them. By doing this we can have the same data across all nodes and that ensures consistency.

Finally, although employing these 2 techniques ensures that we have a system that is fully consistent, we still experience some unavailability when it comes to write operations. This is because although we can solve the availability issue for reads via a few configurations in the client, there is no solution for the availability of the writes. This makes it possible for MongoDB system to be fully consistent (with some configuration) and fully available for read operations, but not always available for write operations.

Below I will try to take a different approach to demonstrate the consistency of our database specifically:
// Suppose we want to find a record that doesn't exist and modify it by incrementing the value and changing the sex. The client in this case should respond with null since the document we

```
db.phase3.findAndModify({
  query: {
    "GEO": "USA"
  },
  update: {
    $set: {
      SEX: "Females"
    },
    $inc: {
      "VALUE": 50
    }
  },
  new: false
})
```

```
> db.phase3.findAndModify({
...     query: {
...         "GEO": "USA"
...     },
...     update: {
...         $set: {
...             SEX: "Females"
...         },
...         $inc: {
...             "VALUE": 50
...         }
...     },
...     new: false
... })

null
```

// Now Suppose we make use of the upsert option of the findAndModify function and we set that option to true. Upsert basically indicates to MongoDB that if there is no match to our query, then it should create a new document with the info we passed. The result of execution is shown to the right.

```
db.phase3.findAndModify({
  query: {
    "GEO": "USA"
  },
  update: {
    $set: {
      SEX: "Females"
    },
    $inc: {
      "VALUE": 50
    }
  },
  new: true,
  upsert: true
})
```

```
> db.phase3.findAndModify({
...     query: {
...         "GEO": "USA"
...     },
...     update: {
...         $set: {
...             SEX: "Females"
...         },
...         $inc: {
...             "VALUE": 50
...         }
...     },
...     new: true,
...     upsert: true
... })
{
        "_id" : ObjectId("5e8919aa105a5dd47c986eb2"),
        "GEO" : "USA",
        "SEX" : "Females",
        "VALUE" : 100
}
```

In this way, I have tried to the best of my ability to demonstrate the consistency of my database by showing some of the expected responses from the MongoDB.

Please note that my answer relies on information from the following **reference**:

Katwal. B. (2019). What is the CAP Theorem? MongoDB vs Cassandra vs RDBMS, where do they stand in the CAP theorem? Retrieved From: https://medium.com/@bikas.katwal10/mongodb-vs-cassandra-vs-rdbms-where-do-they-stand-in-the-cap-theorem-1bae779a7a15

**(g) Investigate the indexing techniques available in your NoSQL system.**

As in any database system, indexes are essential to the efficient execution of queries. This is especially true for MongoDB as it would need to scan all existing documents within a collection to retrieve the ones that match the query at hand. Indexing facilitates query operations by reducing the latency over which data is retrieved by minimizing the number of documents that need to be scanned. It takes into consideration the storage available, the ratio of reads and writes in the queries, as well as the types of queries in the system. It is also worth noting that collections with high read-to-write ratio are better at utilizing indexes than those with a high write-to-read ratio.

Although indexes can be costly in terms of performance, they are very useful when it comes to performing recurrent queries on large datasets. In fact, when a document is created, MongoDB assigns an _id field by default if no index is specified and makes this a unique index for that document. Basically, this is to prevent duplicate documents that are inserted more than once in a particular collection.

Below I will demonstrate the types of indexing techniques available in MongoDB and attempt to provide examples by applying them to our chosen database.

**1. Single field indexes:** As the name suggests, this type of index involves assigning a single field of a document as an index by giving it either the value of 1 to signify ascending order of the data or the value of -1 to signify descending order.

Example of single field indexes:

```
Create the index:
db.phase3.createIndex( { Sex: 1 } )

Run queries using the created index:
db.phase3.find( { Sex: "Females" } )
```

**2. Compound indexes:** Compound indexes are often used to facilitate the sort operation within a query and support queries that match on multiple fields. The syntax for creating a compound index is:

```
db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, … } )
```

Just like single field indexes shown above, compound indexes can have a specification that determines the order or the queried data. This can be applied to all the created indexes.

Please note that with this type of indexing, there are a few limitations:

- There is a limit of maximum 32 fields that can be supported.
- Best practice indicates not to create compound indexes that have hashed index type.
- The order of the fields in this kind of index is important since sorting can only be done in accordance with the order of the fields.

Example of compound indexes:

```
Create the index:
db.phase3.createIndex( { "VALUE": 1, "Sex": 1 } )

Run queries using the created index:
db.phase3.find( { Sex: "Both sexes" } )
db.phase3.find( { Sex: "Both sexes", "VALUE: 249" } )
```

**3. Text indexes:** Text indexes are often used to improve the performance of search queries for a string in a particular collection. Note that with these types of indexes, a collection can have at most one text index.

Example of text indexes:

```
db.phase3.createIndex(
   {
     GEO: "Quebec",
     Sex: "Females"
   }
)
```

One key benefits of text indexes is that you can specify weights for each index to indicate which field matters the most when conducting a query search. Below is an example of the specifying weight indexes:

```
db.phase3.createIndex(
  {
    GEO: "Quebec",
    Sex: "Females"
  },
  {
    weights: {
      GEO: 8,
      Sex: 4
    },
    name: "LocationSexIndex"
  }
 )
```

Please note that with this type of indexing, there are a few limitations:

- A compound index can include a text index in combination with the ascending/descending index key.
- All text index keys must be adjacently in the index specification document when creating a compound text index.
- Cannot use multikey index fields in the compound text index.

- To perform a text search, the query predicate must include equality match conditions on preceding keys.

Please note that indexing types 4 and 5 below are difficult to work given the database we have chosen but I have included them anyway for thorough analysis of MongoDB's indexing techniques.

**4. Multikey Indexes:**

Multikey indexes are used to index fields that hold an array value. In this type of indexing, MongoDB creates separate index entries for every element in the array. The benefit of this type of indexing is through helping a query select documents that consist of arrays by matching a single element or many elements of the array.

Limitations of the Multikey indexes come from the fact that only one array field can be used in the multikey indexing for a document in the collection.

**4. Hashed indexes:** Hashed indexes are used to help with the sharding technique of MongoDB, Sharding helps improve horizontal scaling and involves using hashed indexes. Note that the way hash indexes typically work is by using a hashing function to compute the hash of the value of the index field.

Example of hashed indexes:

**Create the index:**

```
db.phase3.createIndex(
   {
      GEO: "hashed"
   }
)
```

Note that with this type of indexing, there are a few limitations:

- Hashed indexing does not support multi-key indexes
- Compound indexes cannot have hashed indexes fields.
- Using a hashed shard key to shard a collection results in a more random distribution of data.

**Please refer to the next page for a demonstration of performance enhancement through using single field indexes.**

Below we will demonstrate a small example of single field indexing that shows the benefits of indexing:

- First we begin by showing the execution time it takes to find a retrieve data from our database when there is no indexing at all.

```
db.phase3.find(
    { "GEO" : "USA" }
).explain("executionStats")
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 5369,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 4633201,
        "executionStages" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "GEO" : {
                                "$eq" : "USA"
                        }
                },
```

- Then we will create an index for GEO in our database using the following query:

```
db.phase3.createIndex({ "GEO" : 1})
```

```
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

- Finally we will test again our query from above to check the execution stats and verify that the retrieval time has in fact decreased post indexing:

```
db.phase3.find(
    { "GEO" : "USA" }
).explain("executionStats ")
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
                "stage" : "FETCH",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
```

Notice that adding the index has indeed resulted in a huge decrease in the amount time taken to search through our documents making it almost immediate and showing the benefits of indexing.