Introduction

This exercise will cover document stores. As a representative of document stores, MongoDB was chosen for the practical exercises. You can install it on your local machine or use Azure Portal.

1. Document stores

A record in document store is a document. Document encoding schemes include XML, YAML, JSON, and BSON, as well as binary forms like PDF and Microsoft Office documents (MS Word, Excel, and so on). MongoDB documents are similar to JSON objects. Documents are composed of field-value pairs and have the following structure:

```json
{
    name: "sue",
    age: 26,
    status: "A",
    groups: [ "news", "sports" ]
}
```

The values of fields may include other documents, arrays, and arrays of documents. Data in MongoDB has a flexible schema in the same collection. All documents do not need to have the same set of fields or structure, and common fields in a collection’s documents may hold different types of data.

Questions

1. What are advantages of document stores over relational databases?
2. Can the data in document stores be normalized?
3. How does denormalization affect performance?
4. How does a large number of small documents affect performance?
5. What makes document stores different from key-value stores?
1. Flexibility. Not every record needs to store the same properties. New properties can be added on the fly (Flexible schema).
2. Yes. References can be used for data normalization.
3. All data for an object is stored in a single record. In general, it provides better performance for read operations (since expensive joins can be omitted), as well as the ability to request and retrieve related data in a single database operation. In addition, embedded data models make it possible to update related data in a single atomic write operation.
4. It degrades performance since document stores are basically key-value stores. You should consider embedding for performance reasons if you have a collection with a large number of small documents. If you can group these small documents by some logical relationship and you frequently retrieve the documents by this grouping, you might consider "rolling-up" the small documents into larger documents that contain an array of embedded documents.

Rolling up these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses. Additionally, rolling up documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields and there would be fewer associated key entries in the corresponding index. See Indexes for more information on indexes.

However, if you often only need to retrieve a subset of the documents within the group, then rolling up the documents may not provide better performance. Furthermore, if small, separate documents represent the natural model for the data, you should maintain that model.
5. Document-oriented databases are inherently a subclass of the key-value store. The difference lies in the way the data is processed: in a key-value store, the data is considered to be inherently opaque to the database, whereas a document-oriented system relies on an internal structure of the documents in order to extract metadata that the database engine uses for further optimization. Although the difference is often mostly in tools of the systems, conceptually the document-store is designed to offer a richer experience with modern programming techniques.

2. Data models in document stores
Create a mapping between the following data models and MongoDB documents. Data models may be chosen twice.

1. One-to-one relationship with embedded documents
2. One-to-one relationship with normalized data model
3. One-to-many relationship with embedded documents
4. One-to-many relationship with normalized documents (at least in 1NF)
5. One-to-many relationship with document references
6. Tree structure with parent references
7. Tree structure with children references
8. Tree structure with an array of ancestors
9. Tree structure with materialized paths
10. Tree structure with nested sets

a.

```json
{
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA",
    books: [12346789, 234567890, ...]
}
```

```json
{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: ["Kristina Chodorow", "Mike Dirolf"],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}
```

```json
{
    _id: 234567890,
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English"
}
```
b.

```json
{
    _id: "MongoDB",
    parent: "Databases"
}
{
    _id: "dbm",
    parent: "Databases"
}
{
    _id: "Databases",
    parent: "Programming"
}
{
    _id: "Languages",
    parent: "Programming"
}
{
    _id: "Programming",
    parent: "Books"
}
{
    _id: "Books",
    parent: null
}
```

c.

```json
{
    id: "joe",
    name: "Joe Bookreader"
}
{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
```

d.

```json
{
    _id: "MongoDB",
    children: []
}
{
    _id: "dbm",
    children: []
}
{
    _id: "Databases",
    children: ["MongoDB", "dbm"]
}
{
    _id: "Languages",
    children: []
}
{
    _id: "Programming",
    children: ["Databases", "Languages"]
}
{
    _id: "Books",
    children: ["Programming"]
}
```

e.

```json
{
    id: "joe",
    name: "Joe Bookreader"
}
{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
{
    patron_id: "joe",
    street: "1 Some Other Street",
    city: "Boston",
    state: "MA",
    zip: "12345"
}
```
f. 

```json
{
   id: "joe",
   name: "Joe Bookreader",
   address: {
      street: "123 Fake Street",
      city: "Faketon",
      state: "MA",
      zip: "12345"
   }
}
```

g. 

```json
{
   id: "Books", path: null
   {
   }
   {
      id: "Databases", path: ",Books,Programming,"
   }
   {
      id: "Languages", path: ",Books,Programming,"
   }
   {
   }
   {
      id: "dbm", path: ",Books,Programming,Databases,"
   }
}
```

h. 

```json
{
   id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}
```

```json
{
   id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: ["Kristina Chodorow"],
   published date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher id: "oreilly"
}
```

```json
{
   id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: ["Kristina Chodorow", "Mike Dirolf"],
   published date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher id: "oreilly"
}
```
i.

```json
{
  _id: "MongoDB",
  ancestors: ["Books", "Programming", "Databases"],
  parent: "Databases"
}
{
  _id: "dbm",
  ancestors: ["Books", "Programming", "Databases"],
  parent: "Databases"
}
{
  _id: "Databases",
  ancestors: ["Books", "Programming"],
  parent: "Programming"
}
{
  _id: "Languages",
  ancestors: ["Books", "Programming"],
  parent: "Programming"
}
{
  _id: "Programming",
  ancestors: ["Books"],
  parent: "Books"
}
{
  _id: "Books",
  ancestors: [],
  parent: null
}
```

j.

```json
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: []
    
    
    street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
    
    
    street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

k.

```json
{
  _id: "Books",
  parent: 0,
  left: 1,
  right: 12
}
{
  _id: "Programming",
  parent: "Books",
  left: 2,
  right: 11
}
{
  _id: "Languages",
  parent: "Programming",
  left: 3,
  right: 4
}
{
  _id: "Databases",
  parent: "Programming",
  left: 5,
  right: 10
}
{
  _id: "MongoDB",
  parent: "Databases",
  left: 6,
  right: 7
}
{
  _id: "dbm",
  parent: "Databases",
  left: 8,
  right: 9
}
```

**Solution**

1-f
2-c
3-j
4-e,h
5-a,e,h
6-b
7-d
8-i
9-g
10-k
3. MongoDB

3.1 Install MongoDB

MongoDB is an open-source document database. The next step is to install it on your local machine. For that, you can follow general instruction on MongoDB web page [install MongoDB](https://docs.mongodb.com/manual/administration/install-community/)

Here the short instruction for Ubuntu is outlined:

1. Import the public key used by the package management system.
   ```
   sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
   ```

2. Create the `/etc/apt/sources.list.d/mongodb-org-3.2.list` list file using the command appropriate for your version of Ubuntu:
   - Ubuntu 12.04
     ```
     echo "deb http://repo.mongodb.org/apt/ubuntu precise/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
     ```
   - Ubuntu 14.04
     ```
     echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
     ```
   - Ubuntu 16.04
     ```
     echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
     ```

3. Install the MongoDB packages.
   ```
   sudo apt-get update
   sudo apt-get install -y mongodb-org
   ```

To run MongoDB, execute the following command:

   ```
   sudo service mongod start
   ```

To stop MongoDB, execute the following command:

   ```
   sudo service mongod stop
   ```
3.2 Import the dataset

Retrieve the "restaurants" dataset from

https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json

Use mongoimport to insert the documents into the restaurants collection in the test database. If the collection already exists in the test database, the operation will drop the restaurants collection first.

    mongoimport --db test --collection restaurants --drop --file ./primer-dataset.json

3.3 Mongo shell

The mongo shell is an interactive JavaScript interface to MongoDB. You can use the mongo shell to query and update data as well as to perform administrative operations.

To start mongo use:

    `mongo --shell`

In the mongo shell connected to a running MongoDB instance, switch to the test database.

    test

Try to insert a document into the restaurants collection. In addition, you can see the structure of documents the in the collection.

    db.restaurants.insert(
        {
            "address" : {
                "street" : "2 Avenue",
                "zipcode" : "10075",
                "building" : "1480",
                "coord" : [-73.9557413, 40.7720266]
            },
            "borough" : "Manhattan",
            "cuisine" : "Italian",
            "grades" : [
                {
                    "date" : ISODate("2014-10-01T00:00:00Z"),
                    "grade" : "A",
                    "score" : 11
                },
                {
                    "date" : ISODate("2014-01-16T00:00:00Z"),
                    "grade" : "A",
                    "score" : 17
                }
            ],
            "name" : "Vella",
            "restaurant_id" : "41704620"
        }
    )

Query all documents in a collection:

    db.restaurants.find()

Query one document in a collection:

    db.restaurants.findOne()

To format the printed result, you can add .pretty() to the operation, as in the following:

    db.restaurants.find().limit(1).pretty()
Query Documents

For the `db.collection.find()` method, you can specify the following optional fields:

- a query filter to specify which documents to return,
- a query projection to specify which fields from the matching documents to return (the projection limits the amount of data that MongoDB returns to the client over the network),
- optionally, a cursor modifier to impose limits, skips, and sort orders.

```
    db.users.find(
        { age: { $gt: 18 } },
        { name: 1, address: 1 }
    ).limit(5)
```

3.4 Questions

Write queries that return the following:

1. All restaurants in borough (a town) "Brooklyn" and cuisine (a style of cooking) "Hamburgers".
2. The number of restaurants in the borough "Brooklyn" and cuisine "Hamburgers".
3. All restaurants with zipcode 11225.
4. Names of restaurants with zipcode 11225 that have at least one grade "C".
5. Names of restaurants with zipcode 11225 that have as first grade "C" and as second grade "A".
6. Names and streets of restaurants that don't have an "A" grade.
7. All restaurants with a grade C and a score greater than 50.
8. All restaurants with a grade C or a score greater than 50.
9. All restaurants that have only A grades.
10. A table with zipcode and number of restaurants that are in the borough "Queens" and have "Brazilian" cuisine.

You can read more about it here:

https://docs.mongodb.com/getting-started/shell/query/
https://docs.mongodb.com/getting-started/shell/aggregation/
https://docs.mongodb.com/manual/aggregation/

3.4 Solution

```
    db.restaurants.find({"borough" : "Brooklyn", "cuisine" : "Hamburgers" })
    db.restaurants.find({"borough" : "Brooklyn", "cuisine" : "Hamburgers" }).count()
    db.restaurants.find({"address.zipcode" : "11225" })
    db.restaurants.find({"address.zipcode" : "11225", "grades.grade" : "C" } {"name" : 1 })
    db.restaurants.find({"address.zipcode" : "11225", "grades.0.grade" : "C", "grades.1.grade" : "A" })
    db.restaurants.find({"grades.grade" : { $ne : "A"}} {"name" : 1 , "address.street": 1})
    db.restaurants.find({"grades.score" : {$gt : 50}, "grades.grade" : "C"})
    db.restaurants.find( { $or: [ { grades.score : {$gt : 50} }, { grades.grade : "C"}]} )
    db.restaurants.find( { "grades": { "$not": { "$elemMatch": { "grade" :{$ne : "A" }}}} })
    db.restaurants.aggregate(  
        [  
            { $match: { "borough": "Queens", "cuisine": "Brazilian" } },  
            { $group: { "_id": "$address.zipcode", "count": { $sum: 1 } } }  
        ]  
    )
```
4. Indexing in MongoDB

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. Scan can be highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

MongoDB supports indexes that contain either a single field or multiple fields depending on the operations that this index type supports.

By default, MongoDB creates the _id index, which is an ascending unique index on the _id field, for all collections when the collection is created. You cannot remove the index on the _id field.
Managing indexes in MongoDB

An explain operator provides information on the query plan. It returns a document that describes the process and indexes used to return the query. This may provide useful insight when attempting to optimize a query.

```javascript
db.restaurants.find({"borough": "Brooklyn"}).explain()
```

In the mongo shell, you can create an index by calling the createIndex() method.

```javascript
db.restaurants.createIndex( { borough: 1 } )
```

Now, you retrieve a new query plan for indexed data.

```javascript
db.restaurants.find({"borough": "Brooklyn"}).explain()
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

To remove all indexes, you can use `db.collection.dropIndexes()`. To remove a specific index you can use `db.collection.dropIndex()`, such as `db.restaurants.dropIndex({ borough: 1 })`.

Questions

1. Write a query that can benefit from the following index:

   ```javascript
   db.restaurants.createIndex( { borough: 1 } )
   ```

2. We have an index on address field as follows:

   ```javascript
   db.restaurants.createIndex( { address: -1 } )
   ```

   Will the query

   ```javascript
   db.restaurants.find( {"address.zipcode": "11225" } )
   ```

   use that index?

3. Write a command for creating an index on the zipcode field.

4. Let us have the compound index:

   ```javascript
   db.restaurants.createIndex( { borough: 1, cuisine: -1 } )
   ```

   Which of the following queries use the index above?

   A. `db.restaurants.find({"borough": "Brooklyn"})`
   B. `db.restaurants.find({"cuisine": "Hamburgers"})`
   C. `db.restaurants.find({"borough": "Brooklyn", "cuisine": "Hamburgers"})`
   D. `db.restaurants.find().sort( {borough: -1} )`
   E. `db.restaurants.find().sort( {borough: 1, cuisine: 1} )`
   F. `db.restaurants.find().sort( {borough: -1, cuisine: 1} )`
   G. `db.restaurants.find().sort( {cuisine: 1, borough: -1} )`

5. Answer Question 4, but for the following index:

   ```javascript
   db.restaurants.createIndex( { "borough": 1, "cuisine": -1, "name": -1 } )
   ```

6. Is it possible to create the index below? Why? Why not?

   ```javascript
   db.restaurants.createIndex( { "address.coord": 1, "grades": -1 } )
   ```

7. Write an index to speed up the following query:

   ```javascript
   db.restaurants.find( {"grades.grade": { $ne: "A" } } )
   ```

8. Write an index to speed up the following query:

   ```javascript
   db.restaurants.find( {"grades.score": { $gt: 50 }, "grades.grade": "C"} )
   ```

9. What are the differences between two index strategies below

   A. `db.restaurants.createIndex( { "borough": 1, "cuisine": -1 } )`
   B. `db.restaurants.createIndex( { "borough": 1 } )`
   C. `db.restaurants.createIndex( { "cuisine": -1 } )`

10. How are sparse indexes different from normal ones?
Solution

1. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index. For example:

```javascript
db.restaurants.find({"borough" : "Brooklyn"})
```

```javascript
db.restaurants.find().sort("borough" : 1)
```

2. No, indexes on embedded documents support only equality matches.
3. `db.restaurants.createIndex({"address.zipcode" : 1 })`
4. The following queries benefit from the index:
   - A (since `borough` is a index prefix of the compound index),
   - C (equality match for compound index),
   - D (since `borough` is a index prefix of the compound index),
   - F (MongoDB can traverse the index in either direction).
5. The same answer, since `{ "borough" : 1, "cuisine": -1, "name" : -1} has prefix { "borough": 1, "cuisine": -1 }
6. No, for a compound multikey index, each indexed document can have at most one indexed field whose value is an array. As such, you cannot create a compound multikey index if more than one to-be-indexed field of a document is an array.
7. Just `db.restaurants.createIndex( "grades.grade": 1)`, since "{"name" : 1 , "address.street": 1}" is a projection.
8. `db.restaurants.createIndex( {"grades.score": 1, "grades.grade": 1 })`
    However it won't work for
    ```javascript
    db.restaurants.find({"grades.grade" : "C", grades.score" : {$gt : 50 } })
    ```
9. B covers more cases since MongoDB will use index intersection for queries on two fields. Thus, B includes A. However the compound index is faster than two single-field indexes when you querying for two fields.
10. Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

5. SQL to MongoDB Mapping

Create a one-to-one mapping between the following SQL and MongoDB queries.

```sql
INSERT INTO users(user_id, age, status)
VALUES ("bcd001", 45, "A")

SELECT * FROM users

SELECT user_id, status FROM users

SELECT * FROM users
WHERE age > 25 AND age <= 50

SELECT * FROM users
WHERE status = "A" OR age = 50

CREATE TABLE users (  
    id MEDIUMINT NOT NULL AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)

SELECT COUNT(user_id)
FROM users
```
a.  
```javascript
db.users.find(
    { age: { $gt: 25, $lte: 50 } }
)
```

b.  
```javascript
db.users.find()
    { }],
    { user_id: 1, status: 1, id: 0 }
```

c.  
```javascript
db.createCollection("users")
```

d.  
```javascript
db.users.insert(
    { user_id: "bcd001", age: 45, status: "A" }
)
```

e.  
```javascript
db.users.count({ user_id: { $exists: true}})
```

f.  
```javascript
db.users.find(
    { $or: [ { status: "A" }, { age: 50 } ] }
)
```

g.  
```javascript
db.users.find()
```

**Solution**

1-d  
2-g  
3-b  
4-a  
5-f  
6-c  
7-e

### 6. True or False

Say if the following statements are true or false.

1. In document stores, you must determine and declare a table's schema before inserting data.
2. Documents stores are not subject to data modeling and support only one denormalized data model.
3. Different relationships between data can be represented by references and embedded documents.
4. MongoDB provides the capability to validate documents during updates and insertions.
5. There are no joins in MongoDB.

**Solution**

1. False. Document stores have a flexible scheme, which does not require specifying the schema as in relational databases.
2. False. Document stores can be used with a wide range of data models. See Section 2.
3. True. These two tools allow applications to represent different data models. See Section 2.
4. True. Starting in version 3.2, MongoDB provides the capability to validate documents during updates and insertions.
5. True. Nonetheless, starting in version 3.2, MongoDB supports aggregations with “lookup” operator, which can perform a LEFT OUTER JOIN.