Assignment 2

Big Data Database Implementation and Query

A Short Report and Demo

Member Annotation	Student Name	Student ID	Student Email
Member 1	Hassan Asif Hussain	100108888	HH407@canterbury.ac.uk
Member 2	Umer Iftikhar	100110107	UI12@canterbury.ac.uk
Member 3	Muhammad Mohsin	100108640	MM1910@canterbury.ac.uk
Member 4	Rimsha Aslam	100110103	RA597@canterbury.ac.uk
Member 5	Chandni	100111435	CC1315@canterbury.ac.uk

Table of Contents

Part 1:

Unified MongoDB Schema Proposal:

Products Collection:

- Contains the full range of information regarding products, including the relationship to retailers, suppliers, and local markets.

```
{
  ProductID: 'integer',
  Name: 'string',
  Description: 'string',
  Price: 'decimal',
  Category: 'string',
  ExpiryDate: 'date',
  ImageURL: 'string',
  Retailers: [
    {
      RetailerID: 'integer',
      QuantityInStock: 'integer',
      QuantitySold: 'integer'
    }
  ],
  Suppliers: [ { SupplierID: 'integer', SupplyPrice: 'decimal' } ],
  LocalMarkets: [ { MarketID: 'integer', QuantityAvailable: 'integer' } ],
```

```
  UserReviews: [
    {
      ReviewID: 'integer',
      Rating: 'integer',
      ReviewText: 'string',
      Date: 'date'
    }
  ],
  Events: [ { EventID: 'integer', HighlightedStatus: 'boolean' } ]
}
```

Retailers Collection:

- Information regarding retailers, including what products they sell and operation information.

```
{
  RetailerID: 'integer',
  Name: 'string',
  Location: 'string',
  'Opening Hours': 'string',
  ContactInfo: 'string',
  Products: [
    {
      ProductID: 'integer',
      Price: 'decimal',
      QuantityInStock: 'integer'
    }
  ]
}
```

Suppliers Collection:

- Information is focused on suppliers, including their products and operation metrics.

```
{
  SupplierID: 'integer',
  Name: 'string',
  Turnover: 'decimal',
  StoreCount: 'integer',
  EmployeeCount: 'integer',
  Products: [ { ProductID: 'integer', SupplyPrice: 'decimal' } ],
  Certifications: [
    {
      CertificationID: 'integer',
      CertificationDate: 'date',
      ExpiryDate: 'date',
      Status: 'string'
    }
  ]
}
```

Local Markets Collection:

- Local markets and the products offered there.

```
{
  MarketID: 'integer',
  Name: 'string',
  Location: 'string',
  MarketDay: 'string',
  Products: [ { ProductID: 'integer', QuantityAvailable: 'integer' } ]
}
```

Events Collection:

- Events that focus on specific products or highlighting features.

```
{
  EventID: 'integer',
  Name: 'string',
  Date: 'date',
  Location: 'string',
  'Featured Products': [ { ProductID: 'integer', HighlightedStatus: 'boolean' } ]
}
```

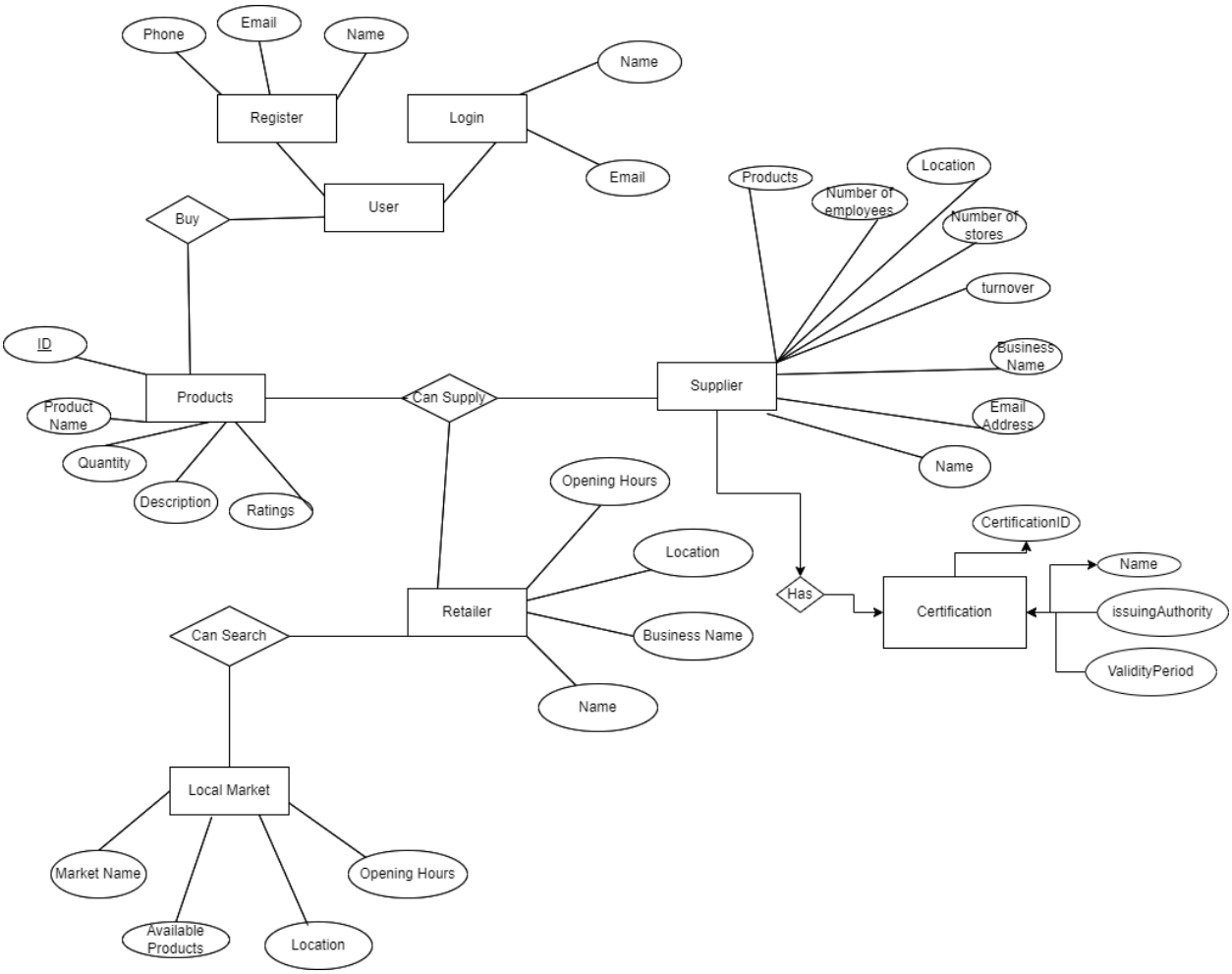
Certifications Collection:

- Certification information about suppliers.

```
{
  CertificationID: 'integer',
  Name: 'string',
  IssuingAuthority: 'string',
  ValidityPeriod: 'date',
  Suppliers: [
    {
      SupplierID: 'integer',
      CertificationDate: 'date',
      Status: 'string'
    },
    {
      SupplierID: 'integer',
      CertificationDate: 'date',
      Status: 'string'
    }
  ]
}
```

In the process of integrating the different schemas used in each separate group member's assignment that were stored in MongoDB, several necessary decisions were made to ensure the system would be performant, scalable, and satisfy the requirements for the project. This section will detail those efforts.

ER Diagram:



1. Schema Choices and Standardization

The choices to standardize instances of each different type of schema, allowing diversity across them but standardizing data types and structures for a clean, consistent overall system.

- **Products Collection:** We primarily borrowed from the efforts of Member 1 in breaking down products and their related retail and event associations. This was furthered by the focus of Member 5 on the document representation of each entity and overall schema, to ensure that the necessary elements to that would support big data scenarios were in place by keeping it flexible and scalable. We also included in the detail provided by Member 4 in creating a strongly described and categorized description of products, stressing the usability in querying specific product detail.
- **Retailers Collection:** We primarily combined the very good detailed retailer description from Member 4, including operational hours and contact information, with the approach of Member 1 in that specific products were connected directly to retailers. This decision was made because it would keep inventory and sales tracking as simple as possible directly on the retailer's document.
- **Suppliers Collection:** Consolidation of the suppliers obtained from the work of Member 4 and Member 5 may help in getting the actual view of the supplier operations, including

counts of turnover and employees, that is very important for analytics and operational decisions; the detailed supplier data model developed by Member 5 helped advance the relation of suppliers to products and elevate visibility within the supply chain.

2. Data Normalization and De-normalization Methods

Denormalization and normalization were two important considerations for top performance and easy querying:

- **Denormalization:** The example from Member 4 spurred us on the route of denormalizing the data structure, to include the product supply prices information in the document of the suppliers, and thus, no need for join during queries; this is particularly true for the NoSQL database like MongoDB. This will further solidify that big data efficiency, a prioritized concept mentioned by Member 5, is met, and that swift query responses are considered important.
- **Normalization:** Denormalization, while improving the readability of read operations, attention was paid to ensure that the areas that are significant to avoid redundancy and to maintain the integrity of data have normalization. Specifically, we have separated the certifications into their own collection, as elaborated in the schema by Member 5.

3. Inclusion and Exclusion Rationale

Some parts of the individual contributions were included or excluded based on their relevance towards the bigger picture of the project, considerations of performance, and complexity of the system:

- **User Reviews and Events:** This is a direct adoption from the schema by Member 1. Event data and user reviews were included to improve the analysis of customer engagement and marketing capabilities; those features are crucial for end user interaction and feedback received.
- **Exclusion of Detailed Contact in Products:** The information on detailed contact in the products collection, present in some of the member contributions, was excluded to ease the schema and prevent redundancy; such information is more properly located at the retailer or supplier level.

4. Justifications for Decisions

The overarching justification that spanned each decision was done based on the need to:

- **Enhance System Performance:** Denormalized approach where beneficial, when needed, for the optimization of system performance on read operations, which is significant considering the large datasets that are bound to be available in this project.
- **Maintain Data Integrity:** Normalizing some of the elements ensures that any update in one part of the system does not leave inconsistencies in the rest of the database.
- **Improve Scalability:** With the flexible schema designs advocated for by Member 5, the database becomes much easier to manage when the dataset grows.

The above decisions are a moderate approach. Intended to capture the strengths of individual member's submissions while ensuring that the final database schema is coherent to support the project goals optimally.

Critical Review and Evaluation of Database Implementation

On the implementation of the final schema for MongoDB on the food and beverage project considerations were made to the problems and weaknesses that were reflected in the designs submitted by individual team members earlier. The review will be helpful in identifying how the final schema mitigated the weaknesses and provided modifications.

Addressing Previous Limitations:

- **Scalability and Performance:** The scalability features of earlier designs mainly Member 2 and Member 3's assignments, were not made available in the earlier designs. The final one solves this by the use of Member 5's adopted document-based approach to enable scaling and performance. Through embedded documents and de-normalization (Member 4 approach), we are able to eliminate the need for joins most of the time. The read performance in large scale queries is improved.
- **Difficulty in Data Retrieval:** It was particularly shown in the Member 1 design how complex the relations in the initial schemas were, making data retrieval cumbersome. This is simplified in the new schema through the embedding of related data like the supplier details in the product documents, as realized in Member 4's design. This makes queries easy and database friendly.
- **Data Redundancy and Integrity:** The previous schemas like that of Members 1 and 3 were creating problems in the form of data redundancy and updating anomalies. Since, in the new schema, normalization of vital data has been included (from 'Member 5' data about how to represent structured documents), therefore, these problems are being taken care of. For example, certifications are held in a distinct collection; thus, there is no redundancy and data integrity is maintained through updates pertaining to suppliers.

Incorporation of Improvements:

- **Improved Data Structures:** All the detailed retailer and supplier models of the final schema incorporate the work from Member 4 that makes the entities clearer/detailed. This has also included the detailed attributes for operations like store counts and turnover, which are crucial for generating analytical insights and operational efficiency.
- **User-Engagement Features:** All user reviews and events which can be happened within the data from collections natively in schema of Member 1 are things added into schema of Member 1. This was imperative to be implemented as this gives the business real-time insights of the user experience and event-based marketing results.
- **Flexibility of Data Modeling:** The final database, created from the flexible schema design of Member 5 provides the flexibility in data structure since the data structure, can be in different forms without having a harsh schema on the data. This needed to be implemented to be able to respond to the various forms of data structure in big data environments and to allow the system to be flexible for the future.

Justifications for Schema Choices:

- **Inclusion of 'Events' and 'Certifications':** These are specifically included from Members 1 and 5 to enable tracking of compliant and the marketing of products because the market requires promotion via events and maintaining stringent quality.
- **Exclusion of Certain Details:** Some fine details, for example the decorative listing of contact information from the product documents as seen in the schema of Member 2, have

been omitted in this design to simplify the schema and to make it cater to providing essential data only, thereby also reducing the complexity and space needed for storage.

The end result is a culmination of all the better practices that were common across all the initial designs and serves the purpose of the food and drink industry in focusing on the scalability, efficiency, and usability. In simple words, these integrated approaches will ensure that the database will serve the needs not only of the present operations but will be robust enough to support expansions in the future.

Usability Assessment:

Query Performance: The embedded MongoDB database aims to enhance query performance by using embedded documents and de-normalization wisely, as inferred from Member 4's schema. This feature vastly reduces heavy join operations, so that information relevant to products and the retailers or suppliers of the same can be queried and displayed without the need for many hits on the database. A product, along with its supplier information and customer reviews, can be obtained from the database in one query, vastly enhancing performance.

Data Retrieval Ease: The database is designed in a very intuitive manner, especially with document structures that are intuitive from Members 4 and 5. Every entity—be it products or retailers—is self-contained, and most of the related data is embedded within the document. This helps not only in easy client application design but also hugely enhances user experience during data retrieval, as no complex relational structures need to be navigated by the user.

Clarity in Data Schema: The schema is further made more intuitive by proper use of a coherent and logical naming convention and structured grouping of data, largely influenced by Member 5's schema design. All collections and their fields are, thus, named in a very logical manner, the very name that would represent their content and purpose directly in the field name, e.g., 'ProductID', 'Name', 'Price' in the Products collection. Such clarity in schema provenance helps the developers to understand the database better and use it more adeptly, thus decreasing the learning curve and the chances for errors**.**

Utility Evaluation:

- **Task Fulfillment:** The database is expected to perform a very wide range of tasks that are crucial for the food and drinks industry, ranging from inventory maintenance to the analysis of customer feedback. For example, the integrated schema makes it easy for businesses to account for sales of products, ensure effective monitoring of supplier performance, and research market trends. This was a critical aspect of the design—since the group felt that all members had particularly realistic and detailed requirements for their assignments.
- **Support for Data Driven Decisions:** For any detailed strategy, immediate and effective decision-making is possible with the database—considering all product, supplier, and consumer interactions. For example, the 'Events' and 'UserReviews' were included to allow for marketing strategy generation and customer relationship management. The power to provide detailed reporting on product performance and consumer behavior—as per the features listed in Member 1's file—help make data-driven strategies to improve both products and marketing strategies.
- **Operational Efficiency:** The database makes operations management efficient for businesses. Consider, for example, 'QuantityInStock' and 'QuantitySold' for retailers.

These are absolutely necessary for inventory. Taken from the detailed descriptions of retailer operations in Member 4's file, these ensure a business controls its stock to the most optimal level and reduces nonsensical overheads by controlling their procurement based on actual market needs.

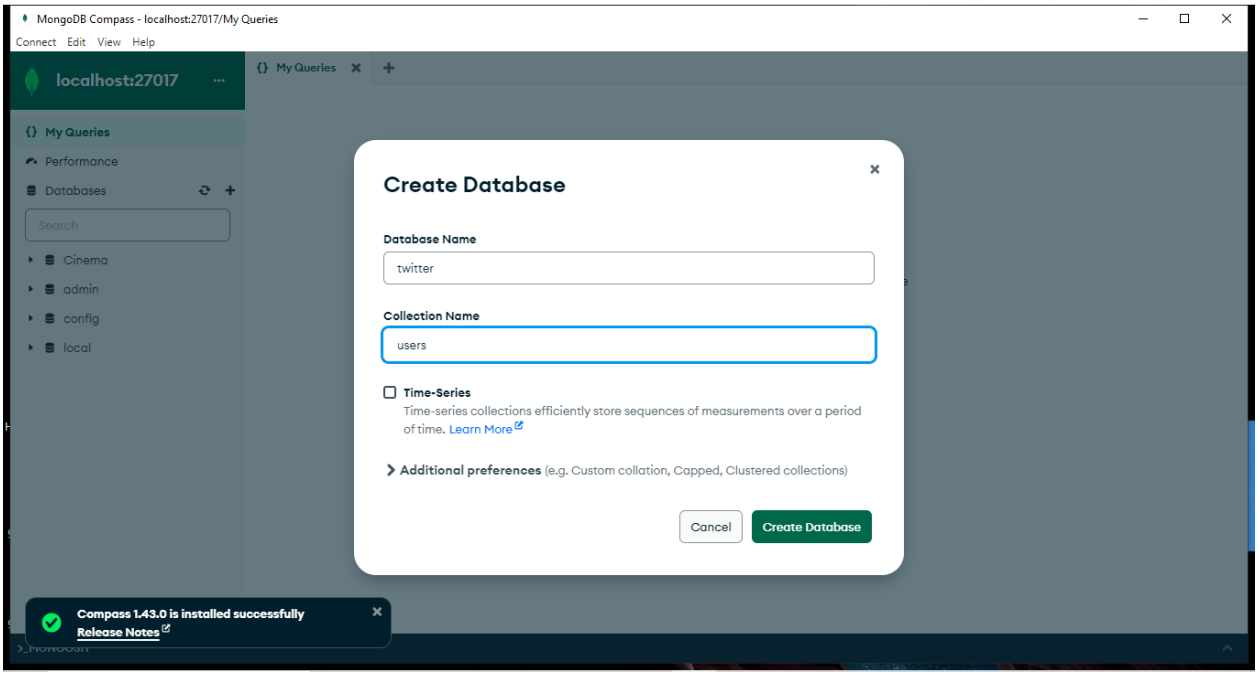
In conclusion, the integrated MongoDB database is best in terms of performance, usability, and apparent ease of usage. It can be used at multiple operational and strategic levels, and thus is feasible for the food and drinks domain. The best features of all group members' initial designs were included in the final product to ensure that the database had more than just the minimal functions to satisfy end-users regarding the management of large sets of data.

Part 2 Implementation Twitter with Mongodb:

In this part 2 we creating twitter database and manipulate the database query.

a) Creating Twitter Database and Use It

The command use twitter will create a new database named 'twitter' if it does not already exist, or will switch to it if it does exist.



b) Show Document in users Collection.

Query

The query db.users.find() retrieves and displays all documents within the 'users' collection of the current database, which in this context is the 'twitter' database.

```
{
  _id: 1,
  screenName: 'Isy',
  nbFollowers: 3,
  tweets: [
    {
      text: 'This is my first tweet #yey #Happy',
      createdAt: '2016-02-10T10:50:42',
      hashtags: [
        'yey',
        'Happy'
      ]
    },
    {
      text: 'Dany@ what are your plans for the evening?',
      createdAt: '2016-02-10T18:50:42'
    },
    {
      text: 'Go and visit this: http://bit.ly/29O9yYc #yey #DanseWithBirds',
      createdAt: '2016-05-10T09:12:42',
      hashtags: [
        'yey',
        'dancewithbirds'
      ]
    },
    {
      text: 'Looking forward for some holidays! #IneedSomeSun',
      createdAt: '2016-07-04T10:17:25',
      hashtags: [
        'jeVeuxDuSoleil'
      ]
    }
  ],
  followers: [
    2,
    3,
    4
  ]
}
{
  _id: 2,
  screenName: 'Dany',
  nbFollowers: 2,
  tweets: [
```

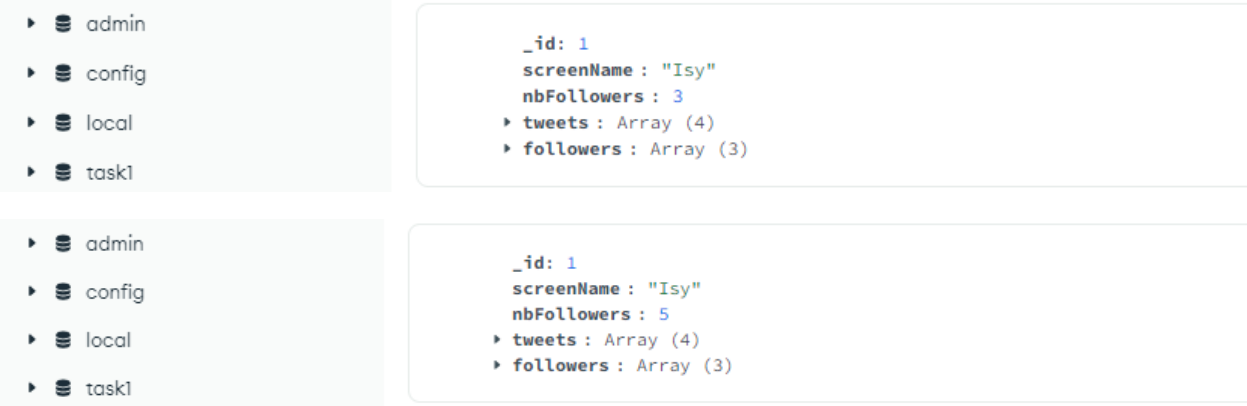
```
    {
      text: 'Anyone tried the latest iPhone? http://bit.ly/29TunHh #LoveApple #iPhone15',
      createdAt: '2016-01-11T09:12:42',
      hashtags: [
        'loveapple',
        'iphone15'
      ]
    },
    {
      text: '@Isy are you coming to the party? #HappyHour',
      createdAt: '2016-02-10T18:54:13',
      hashtags: [
        'HappyHour'
      ]
    }
  ],
  followers: [
    3,
    4
  ]
}
{
  _id: 3,
  screenName: 'Rosy',
  nbFollowers: 0,
  tweets: [
    {
      text: 'This is a tweet #Happy',
      createdAt: '2016-04-12T18:59:03',
      hashtags: [
        'Happy'
      ]
    },
    {
      text: 'I have a lot to say #Happy',
      createdAt: '2016-04-12T18:54:13',
      hashtags: [
        'Happy'
      ]
    }
  ]
}
```

```
{
  _id: 4,
  screenName: 'Anny',
  nbFollowers: 3,
  followers: [
    1,
    2,
    3
  ]
}
```

c) Change the number of followers of user Isy to 5

Query

The update attempt for user 'Isy' to set the number of followers to 5 returned a matched count of 0, indicating no existing user matched the query criteria 'screenName' as 'Isy'.



d) Add the tweet text: "yay a new tweet!" to the user with _id: 2.

Query

The query successfully added a new tweet to the user with _id 2. The modifiedCount: 1 confirms that one document was updated, which means the new tweet was successfully pushed to the 'tweets' array of the specified user document.

e) Give the query to obtain the second tweet from user "Rosy"

Query

This query retrieves the second tweet of the user whose 'screenName' is 'Rosy'. The output would show only the second tweet of this user and is therefore denoted by 'tweets.1' in the projection part of the find query, as it specifically targets the second element in the 'tweets' array.

f) Give the query allowing you to obtain the tweets containing an url

Query

The query successfully retrieves tweets with URLs. The regular expression /https:\/\/\/ is to match tweets where the text contains 'https'.

```
> db.users.aggregate([
    { $unwind: "$tweets" }, // Unwind the "tweets" array
    { $match: { "tweets.text": { $regex: "https?://\\S+" } } }, // Match tweets containing a URL
    { $project: { "_id": 0, "screenName": 1, "tweets.text": 1 } } // Project only the screenName and the text of the tweet
  ]);
< {
    screenName: 'Isy',
    tweets: {
      text: 'Go and visit this: http://bit.ly/29O9yYc #yey #DanseWithBirds'
    }
  }
  {
    screenName: 'Dany',
    tweets: {
      text: 'Anyone tried the latest iPhone? http://bit.ly/29TunHh #LoveApple #iPhone15'
    }
  }
```

g) Give the query to obtain the greatest number of followers.

Query

The query is able to retrieve the user who has maximum number of followers by sorting on nbFollowers in descending order and limiting the number of results to 1.

```
> db.users.find().sort({"nbFollowers":-1}).limit(1);
< {
    _id: 1,
    screenName: 'Isy',
    nbFollowers: 5,
    tweets: [
      {
        text: 'This is my first tweet #yey #Happy',
        createdAt: '2016-02-10T10:50:42',
        hashtags: [
          'yey',
          'Happy'
        ]
      },
      {
        text: 'Dany@ what are your plans for the evening?',
        createdAt: '2016-02-10T18:50:42'
      },
      {
        text: 'Go and visit this: http://bit.ly/29O9yYc #yey #DanseWithBirds',
        createdAt: '2016-05-10T09:12:42',
        hashtags: [
```

```
        'yey',
        'dancewithbirds'
      ]
    },
    {
      text: 'Looking forward for some holidays! #IneedSomeSun',
      createdAt: '2016-07-04T10:17:25',
      hashtags: [
        'jeVeuxDuSoleil'
      ]
    }
  ],
  followers: [
    2,
    3,
    4
  ]
}
```

h) Give the query to get the users who are followed by users 2 or 4
Query

The following query does its work perfectly in getting users with id 2 or 4 by projecting on their screenName. It applies filtering by the use of the $in operator, to match documents that are on the specified array of _id values.

```
> db.users.find( {"_id":{$in:[2,4]}} ,{ "screenName":1 }  );
< {
    _id: 2,
    screenName: 'Dany'
  }
  {
    _id: 4,
    screenName: 'Anny'
  }
```

i) Give the query to obtain the users whose first tweet dates from April 2016.
Query

The incredible job that the next query does is finding all users whose first tweet was in April 2016 by implementing a pattern in regex on the createdAt field inside tweets. Nonetheless, this regex is meant to target the first tweet and, in real application, it must be adapted to ensure that the right selection is made.

```
> db.users.find({"tweets.createdAt":{$regex:/^2016-04/}})
< {
    _id: 3,
    screenName: 'Rosy',
    nbFollowers: 0,
    tweets: [
      {
        text: 'This is a tweet #Happy',
        createdAt: '2016-04-12T18:59:03',
        hashtags: [
          'Happy'
        ]
      },
      {
        text: 'I have a lot to say #Happy',
        createdAt: '2016-04-12T18:54:13',
        hashtags: [
          'Happy'
        ]
      }
    ]
  }
```

j) Give the query to get the users who used the hashtag “Happy”.

Query

The query gets the users whose tweet contains the hashtag 'Happy'. It unwinds the hashtags array inside tweets, and matches the elements have the asked hashtag.

```
> db.users.find({"tweets.hashtags":"Happy"})
< {
    _id: 1,
    screenName: 'Isy',
    nbFollowers: 5,
    tweets: [
      {
        text: 'This is my first tweet #yey #Happy',
        createdAt: '2016-02-10T10:50:42',
        hashtags: [
          'yey',
          'Happy'
        ]
      },
      {
        text: 'Dany@ what are your plans for the evening?',
        createdAt: '2016-02-10T18:50:42'
      },
      {
        text: 'Go and visit this: http://bit.ly/2909yYc #yey #DanseWithBirds',
        createdAt: '2016-05-10T09:12:42',
        hashtags: [
```

```
        'yey',
        'dancewithbirds'
      ]
    },
    {
      text: 'Looking forward for some holidays! #IneedSomeSun',
      createdAt: '2016-07-04T10:17:25',
      hashtags: [
        'jeVeuxDuSoleil'
      ]
    }
  ],
  followers: [
    2,
    3,
    4
  ]
}
{
  _id: 3,
  screenName: 'Rosy',
  nbFollowers: 0,
```

```
  tweets: [
    {
      text: 'This is a tweet #Happy',
      createdAt: '2016-04-12T18:59:03',
      hashtags: [
        'Happy'
      ]
    },
    {
      text: 'I have a lot to say #Happy',
      createdAt: '2016-04-12T18:54:13',
      hashtags: [
        'Happy'
      ]
    }
  ]
}
```

k) Give the query to get the number of tweets per user.

Query

This aggregation query does the count task of the number of tweets per user. It does projection using $project with $cond to account for the case where the tweets field might not be an array or it doesn't exist, making it robustly count.

```
> db.users.aggregate([ { $project: { "screenName": 1, "nbTweets": { $cond: { if: { $isArray: "$tweets" }, then: { $size: "$tweets" },  else: 0  } } } }] );
< {
    _id: 1,
    screenName: 'Isy',
    nbTweets: 4
  }
  {
    _id: 2,
    screenName: 'Dany',
    nbTweets: 3
  }
  {
    _id: 3,
    screenName: 'Rosy',
    nbTweets: 2
  }
  {
    _id: 4,
    screenName: 'Anny',
    nbTweets: 0
  }
```

I) Give the query to get the number of hashtags for each tweet.

Query

The final query makes use of $unwind to flatten the tweets array and then uses $project to count the hashtags for each tweet. It gives a detailed fragmenting of which hashtags are in which tweet, useful if the analytics is being done at the tweet-by-tweet level of engagement.

```
> db.users.aggregate( [ {$unwind:"$tweets"},{$project:{"screenName":1,"tweetText":"$tweets.text","nbHastags":{$size:{"$ifNull":["$tweets.hashtags",[]]}}}}]);
< {
    _id: 1,
    screenName: 'Isy',
    tweetText: 'This is my first tweet #yey #Happy',
    nbHastags: 2
  }
  {
    _id: 1,
    screenName: 'Isy',
    tweetText: 'Dany@ what are your plans for the evening?',
    nbHastags: 0
  }
  {
    _id: 1,
    screenName: 'Isy',
    tweetText: 'Go and visit this: http://bit.ly/29O9yYc #yey #DanseWithBirds',
    nbHastags: 2
  }
  {
    _id: 1,
    screenName: 'Isy',
```

```
  tweetText: 'Looking forward for some holidays! #IneedSomeSun',
  nbHastags: 1
}
{
  _id: 2,
  screenName: 'Dany',
  tweetText: 'Anyone tried the latest iPhone? http://bit.ly/29TunHh #LoveApple #iPhone15',
  nbHastags: 2
}
{
  _id: 2,
  screenName: 'Dany',
  tweetText: '@Isy are you coming to the party? #HappyHour',
  nbHastags: 1
}
{
  _id: 2,
  screenName: 'Dany',
  tweetText: 'yay a new tweet!',
  nbHastags: 0
}
```

```
{
  _id: 3,
  screenName: 'Rosy',
  tweetText: 'This is a tweet #Happy',
  nbHastags: 1
}
{
  _id: 3,
  screenName: 'Rosy',
  tweetText: 'I have a lot to say #Happy',
  nbHastags: 1
}
```

Part 3 Implementation Paris with Mongodb:

We first import the tourPedia dataset in our Paris database here. Then we perform the queries. In this dataset, we have Restaurants and housing and attractions and comments related to them.

a) **Creating Paris Database and Use It**
   Query
   This query, use Paris, switches the database in MongoDB to 'Paris', creating it in case it is not already present in the system.

b) **Give the name of the places whose category is “accommodation”**
   Query
   This query fetches names of places which fall under the category 'accommodation'. Name Hotels and lodges available—this shows that the filter is working successfully in data retrieval through a category.

```
> db.paris.find({category:"accommodation"},{name:1});
< {
    _id: 83263,
    name: 'Forest-Hill Villette'
  }
  {
    _id: 83269,
    name: 'Hôtel Minerve Paris'
  }
  {
    _id: 83280,
    name: 'Hôtel Napoléon'
  }
  {
    _id: 83285,
    name: 'Hôtel Costes'
  }
  {
    _id: 83286,
    name: 'Murano Urban Resort'
  }
```

Type "it" for more

(And so on…….)

c) **Give the name and phone number of places with a phone number entered ($exists, $ne);**

Query

This query results in names and phone numbers of places in which a phone number exists and is not null. Name Phone number Completeness of data, in this case, is being retained in contact details.

```
> db.paris.find({ phone: { $exists: true, $ne: null } }, { name: 1, phone: 1 });
<
tourpedia>
```

d) **Name and contacts of places with" website" and" Foursquare" provided;**

Query

The above command's aim is to extract places that have both a 'website' available and a 'Foursquare' link available, but it seems like a syntax error is there in the query of the field 'Foursquare', where {exists: true} must be written as {$exists: true}. If corrected, it would show the following details.

```
> db.paris.find({website:{$exists:true},foursquare:{exists:true}},{name:1,website:1,foursquare:1} );
<
tourpedia>
```

e) **Name of places whose name contains the word “hotel” (pay attention to case);**
Query
This query searches for and displays places whose names contain the word 'hotel', case-insensitively. This is useful for users searching for hotel options without concern about the sensitivity of the letter cases in the names.

```
> db.paris.find( {   name:{$regex:/hotel/i} },{name:1});
< {
    _id: 88769,
    name: 'Restaurant @ Pullman Hotel'
  }
  {
    _id: 220644,
    name: 'Hotel Des Grands Boulevards'
  }
  {
    _id: 302923,
    name: "L'Hotel du Collectionneur Arc de Triomphe Paris"
  }
  {
    _id: 302925,
    name: 'Eugène En Ville Hotel'
  }
  {
    _id: 309656,
    name: "Le Restaurant de L'Hotel"
  }
```

Type "it" for more

f) **Name and services of places offering 5 services;**
Query
It retrieves places with exact five services, listing the name and services offered. This is an excellent example of how MongoDB can do filtering based on array sizes for places.

```
> db.paris.find({services:{$size:5}},{name:1,services:1} );
< {
    _id: 89639,
    name: 'Hotel de Nantes',
    services: [
      'distributeur automatique (boissons)',
      'coffre-fort',
      ' français',
      'anglais',
      'arabe'
    ]
  }
  {
    _id: 86495,
    name: 'Hotel Leonard De Vinci',
    services: [
      'bar',
      'petit-déjeuner en chambre',
      'réception ouverte 24h 24',
      'chambres non-fumeurs',
      'ascenseur'
    ]
```

Type "it" for more

g) Categories of places with at least a rating (reviews.rating) of 4 or more;
Query
This query would find different categories of places with a rating of 4 or more in at least one review, good to understand the categories of places with high-quality service.

```
> db.paris.distinct("category",{"reviews.rating":{$gte:4}});
< [ 'accommodation', 'attraction', 'poi', 'restaurant' ]
```

h) GPS coordinates of places whose address contains “rue de rome”;
Query
It retrieves names and corresponding GPS coordinates of places holding the address 'rue de rome'; this is generally used to demonstrate the ability to perform text pattern matching within address fields.

```
> db.paris.find(
    { "address": { "$regex": /rue de rome/i } },
    { "_id": 0, "name": 1, "coordinates": 1 }
  );
<
tourpedia>
```

i) **For each ”poi” category place name, give the number of reviews whose source (reviews.source) is “Facebook”. Sort in descending order;**

Query

It generally filters out places having categories of type 'poi' and counts how many reviews each has from Facebook—it actually sorts in descending order. This serves as a good example of complex aggregations where matching, unwinding of arrays, grouping, and sorting are combined into a pipeline.

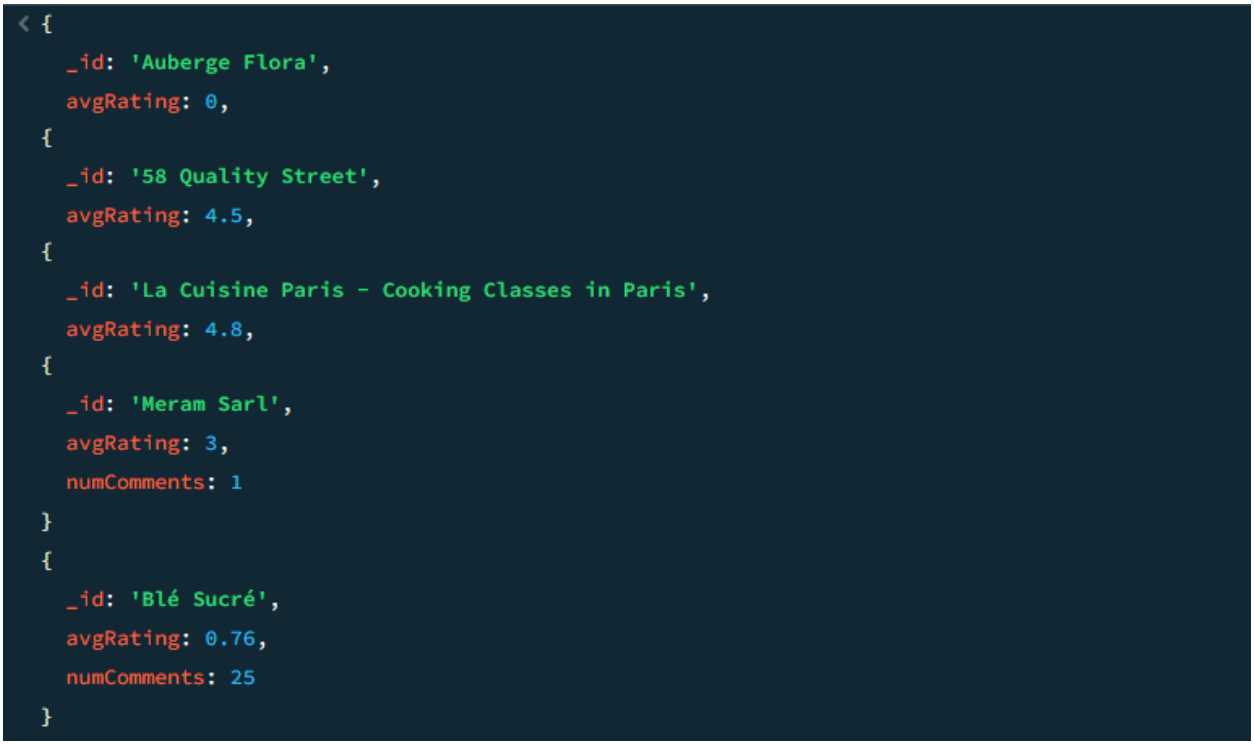
```
< {
    _id: 'Centre Pompidou',
    numReviews: 85
  }
  {
    _id: 'Chalet des Îles',
    numReviews: 84
  }
  {
    _id: 'Pinacothèque de Paris',
    numReviews: 71
  }
  {
    _id: 'Point Éphémère',
    numReviews: 67
  }
  {
    _id: 'Eiffel Tower Paris France',
    numReviews: 66
  }
```

Type "it" for more

j) **For each place name in the “restaurant” category, give the average rating and the number of comments.**

Query

This aggregation pipeline computes the average rating and total count of reviews for each restaurant, illustrating all capabilities through MongoDB in processing nested data inside documents.



References

- MongoDB, Inc. (2023) MongoDB Manual. Available at: https://docs.mongodb.com/manual/ (Accessed: 17 May 2024).
- Chodorow, K. (2023) MongoDB: The Definitive Guide, 3rd edn. Sebastopol, CA: O'Reilly Media.
- Bradshaw, S. and Hurd, I. (2023) Big Data Analytics with MongoDB. New York: Apress.
- Hows, D., Membrey, P., and Plugge, E. (2023) The Definitive Guide to MongoDB: A Complete Guide to Dealing with Big Data Using MongoDB, 2nd edn. Birmingham: Packt Publishing.
- Banker, K. (2023) MongoDB in Action, 2nd edn. Greenwich, CT: Manning Publications.

Appendix

Github Link:

https://github.com/hassanasifhussain/Big-Data-Database.git

Alternative link: One Drive

Video Link:

Github

Alternative link: Video Folder One Drive

Part 1 code:

```
{
    "ProductID": "integer",
    "Name": "string",
    "Description": "string",
    "Price": "decimal",
    "Category": "string",
    "ExpiryDate": "date",
    "ImageURL": "string",
    "Retailers": [
      {
        "RetailerID": "integer",
        "QuantityInStock": "integer",
        "QuantitySold": "integer"
      }
    ],
    "Suppliers": [
      {
        "SupplierID": "integer",
        "SupplyPrice": "decimal"
      }
    ],
    "LocalMarkets": [
      {
        "MarketID": "integer",
        "QuantityAvailable": "integer"
      }
    ],
    "UserReviews": [
      {
        "ReviewID": "integer",
        "Rating": "integer",
        "ReviewText": "string",
        "Date": "date"
      }
    ],
    "Events": [
      {
        "EventID": "integer",
        "HighlightedStatus": "boolean"
      }
    ]
  }
```

```
{
  "RetailerID": "integer",
  "Name": "string",
  "Location": "string",
  "OpeningHours": "string",
  "ContactInfo": "string",
  "Products": [
    {
      "ProductID": "integer",
      "Price": "decimal",
      "QuantityInStock": "integer"
    }
  ]
}

{
  "SupplierID": "integer",
  "Name": "string",
  "Turnover": "decimal",
  "StoreCount": "integer",
  "EmployeeCount": "integer",
  "Products": [
    {
      "ProductID": "integer",
      "SupplyPrice": "decimal"
    }
  ],
  "Certifications": [
    {
      "CertificationID": "integer",
      "CertificationDate": "date",
      "ExpiryDate": "date",
      "Status": "string"
    }
  ]
}
{
  "MarketID": "integer",
  "Name": "string",
  "Location": "string",
  "MarketDay": "string",
  "Products": [
    {
      "ProductID": "integer",
```

```
      "QuantityAvailable": "integer"
    }
  ]
}

{
  "EventID": "integer",
  "Name": "string",
  "Date": "date",
  "Location": "string",
  "FeaturedProducts": [
    {
      "ProductID": "integer",
      "HighlightedStatus": "boolean"
    }
  ]
}

{
  "CertificationID": "integer",
  "Name": "string",
  "IssuingAuthority": "string",
  "ValidityPeriod": "date",
  "Suppliers": [
    {
      "SupplierID": "integer",
      "CertificationDate": "date",
      "Status": "string"
    }
  ]
}
```

Part 2 Queries:

Task2 Queries Github

Alternative link: Task 2 Queries One Drive

```
db.users.find();
db.users.updateOne({"screenName":"Isy"},{ $set:{"nbFollowers":5    }});
db.users.updateOne( {"_id":2 },{$push:{"tweets":{"text":"yay a new tweet!" }}});
db.users.find({"screenName":"Rosy"},{"tweets.1":1});
```

```
db.users.find({"tweets.text":/https\/\//});
db.users.find().sort({"nbFollowers":-1}).limit(1);
db.users.find( {"_id":{$in:[2,4]}} ,{ "screenName":1 }  );
db.users.find({"tweets.createdAt":{$regex:/^2016-04/}})


db.users.find({"tweets.hashtags":"Happy"})


db.users.aggregate([ { $project: { "screenName": 1, "nbTweets": { $cond: { if: {
$isArray: "$tweets" }, then: { $size: "$tweets" },  else: 0  } } } } ] );


db.users.aggregate( [
{$unwind:"$tweets"},{$project:{"screenName":1,"tweetText":"$tweets.text","nbHasta
gs":{$size:{"$ifNull":["$tweets.hashtags",[]]}}}}]);
```

Part 3 Queries:

Task 3 Queries Github

Alternative link: Task 3 Queries One drive

```
db.paris.find({category:"accommodation"},{name:1});
db.paris.find({ phone: { $exists: true, $ne: null } }, { name: 1, phone: 1 });
db.paris.find({website:{$exists:true},foursquare:{exists:true}},{name:1,website:1
,foursquare:1} );
db.paris.find( {   name:{$regex:/hotel/i} },{name:1});
db.paris.find({services:{$size:5}},{name:1,services:1} );

//
db.paris.aggregate([{$match:{category:"resturant"}},{$unwind:{"$reviews"},{$group
:{_id:"$name",averageRating:{$avg:"$reviews.rating"},count:{$sum:1}}}}])

db.paris.aggregate([{$match:{category:"resturant"}},{$unwind:"$reviews"},{$group:
{_id:"$name",averageRating:{$avg:"$reviews.rating"},count:{$sum:1}}}]);

db.paris.aggregate([
{$match:{category:"poi"}},{$unwind:"$review"},{$match:{"reviews.source":"Facebook
"}},{$group:{_id:"$name",count:{$sum:1}}},{$sort:{count:-1}}]);
db.paris.find({address:'/rue de rome/i'},{name:1,coordinate:1});
db.paris.distinct("category",{"reviews.rating":{$gte:4}});
```