Comp 1835 Coursework

Khassan Nazhem

COMP 1835
Dr Tatiana Simmonds
University of Greenwich
Greenwich, London
01.04.2021

Table of Contents

List of Figures

Part 1:

A Peer Review on "Comparison of NoSQL Databases"

The manuscript aims to inform the audience about several popular NoSQL databases and how they diverse. The writer successfully managed to choose six of the most famous NoSQL databases used today and focused on their differences. The writer states how each database excels at solving scenario-based problems, the drawback caused to solve the obstacle, and finally shaping the writer's evaluation after comparing how effective each database can be. In addition, the conclusion is merely informative and self-derived as it doesn't present any findings. According to the writer, each database has its use, where it depends on the requirements of the project, as some would demand using document-based databases whilst others would need graph-based databases. Besides, the manuscript presents easy-to-understand relevant information where the reader can benefit from discovering some but not all NoSQL databases. However, the writer didn't present a well-structured article since there are several points that the writer failed to touch on. Also, the minimal use of examples and experiments provides a low chance for the reader to engage with content. The audience wouldn't know how the mentioned databases run under certain scenarios and circumstances, how they facilitate writing and reading data and perform on their respective platforms. Moreover, the sources used for information are unreliable and the lack of concrete experiments make the manuscript lack credibility. Briefly, this peer review will recollect the major issues that the writer tackled, in addition to the minor issues. The major issues emphasize on the lack of presentation on NoSQL database types, lack of examples and experiments, and finally issues on presenting the information. Whereas the minor flaws shed light on the writer's structure like the abstract and introduction, references and presentation skills. Finally, the following review suggests edits as the manuscript holds critical issues that make it uncredible and less valuable to be published. The writer is encouraged to revise the presented research and adjust it according to this review.

To begin with, there is a major flaw when the writer starts presenting the three types of NoSQL databases: document-based, column family, and graph NoSQL databases. The writer has missed two other crucial types. NoSQL databases also offer the key-value model and the polyglot persistence of databases. A key-value model is a dictionary-like database, for example, Redis database. Redis is short for Remote Dictionary Server: it is a non-proprietary software database that is fast because it relies on in-memory key-value data store through caches. This model is flexible and widely used for schema-less prototypes since both the key and the value accept a wide range of formats that can include logical paths, BLOB data and media. The polyglot NoSQL database is a data storage technology that allows the usage of multiple types of databases simultaneously, creating a super data warehouse ecosystem. Having different databases helps with coordinating complicated operations through selecting the best component according to the scenario. Further, program leveraging improves response time and efficiency though assigning relevant processes. Polyglot persistence is most suitable for online retail websites since such websites require data storing for various objects like product catalogs, user sessions and financial data.

Moreover, the technical specifications mentioned in the manuscript lack concrete examples that would reform the conclusion. Various online resources focus on ranking the databases according to how they perform. The performance ranking would include how each database varies at having consistency, durability, speed of query execution, size complexity for data, friendliness of the provided GUI, and price variations in some cases. It would have been helpful to show different NoSQL languages, stating the complexity of the query and what it can offer. Databases like MongoDB offer queries that include filters, projections, aggregations and map reduced queries. Cassandra comes handy in retrieving averages, sums, maxima and minima specifically on large datasets with great performance whereas HBase queries are great for analytics integrated within Hadoop's MapReduce. Finally, Neo4j provides graph technology to store and navigate relationships in a high computational speed.

Likewise, the writer briefly specified how Cassandra, MongoDB and HBase are used through stating "The differences among these databases make them more practical for different cases. For example, Cassandra is used in many fraud detection applications as well as large consumer websites such as Twitter and Netflix. MongoDB is better suited for real-time analytics. HBase also excels for analytics but is also an excellent choice for web applications for better customer targeting", lacking the reason behind using it. The writer should state clear examples about which type of NoSQL databases are being used by the top companies and why. Some examples would show how Netflix integrates two NoSQL databases which are Simple DB and HBase so that they can offer service that is highly available. The mentioned databases can scale large volumes of data and in real-time through batch jobs, creating replicas around various data centers to avoid any point of failure. In addition, UBER uses MongoDB because it provides them a highly scalable caching system. Pinterest relies on HBase because HBase can provide large sizes of data that can be read and written in real-time. Finally, eBay uses Neo4j as it offers a faster calculation to keep same-day delivery service at its best. These examples would give the readers a chance to explore the approach of a database. Importantly, it enhances the ability to understand the reasoning behind using the database. Thus, the writer's comparison does not satisfy all elements of NoSQL databases. The mentioned aspects would help the writer compare similar databases to gain a clear vision. The writer should compare similar types of NoSQL databases and segregate the others, leading to shifting the comparison to a different result.

When comparing the databases, the writer started the section by recollecting the types of databases mentioned previously, classifying whether the database belongs to the column family databases or document-based databases. Initially, the writer classified Cassandra and HBase to belong to the column family databases. However, during the comparison, Cassandra and HBase are set to belong to the document-based databases. The writer mentions that "Each of these databases uses variations of the document-oriented model. Cassandra uses key spaces, MongoDB uses a flexible schema, HBase uses a column-oriented model, and CouchDB uses the standard document model." To avoid confusion, the writer should clarify more on the base of the presented comparison. For the audience, it seems to be vague to why the writer classified Cassandra and HBase correctly as column family databases when initially presented, yet then included them with the document-based databases when comparing. With that being said, the conclusion is just a presentation of collective ideas, it doesn't carry any significant findings to

present. The conclusion should be much more expressed as it should motivate readers to ask important questions and allow experts in different fields to determine whether the information presented is relevant.

As for the minor flaws, according to the writer, the introduction should briefly describe RDBMS and why most technologies are transitioning to NoSQL. However, the writer highly focuses on RDBMS and its history. Ideally, the writer should shed light on why NoSQL databases are more reliable than RDBMS to adapt and manage Big Data. NoSQL databases can handle huge loads of data because of their scalability as they use sharding for horizontal scaling. Sharding is the partitioning of data chunks and separating them on different machines for parallel processing: as data increases, the number of machines can be increased to handle it (horizontal scaling). This increase in data amounts requires a database that is highly available in case of system failure. NoSQL provides an auto-replication feature that provides data replicas for previous states.

The abstract includes what the writer will focus on in the introduction, the objective behind conducting this research and that the conclusion will be stated by the end of this research. So, the writer only managed to report one point in the abstract and failed to mention three others. Normally, a well-formed abstract does not only report the objective behind conducting the research, but also describes what methods were used for the analysis, what were the bases of the results for the arguments and also presents an insight on the main conclusion. Also, the writer presents keywords that seem unused in the manuscript like “Redis” and “Couchbase”; even though the keywords are related to NoSQL since both Redis and Couchbase are NoSQL databases, the chosen terms should be relevant to the main topic. Such words should describe the general topic like “NoSQL”, "Big Data" and "Database”, keeping in mind that the article is not only targeted for expertise on the topic, but also the public who use general keywords to search for related articles and topics.

The information presented by the writer in the manuscript is inaccurate due to using sources that are limited and not based on scholarly research. Websites like W3Schools and Tutorial Point mainly serve as tutorial and guide websites. In addition, this referencing does not follow Harvard style of referencing. The writer avoided writing the authors names and mentioned the blogs names instead. Even though the mentioned references are well-presented and do not defy the rules of the information presented, it is still preferred not to use them because they don’t carry scientific significance. Besides, good references present to the readers the quality of the work since it removes any possibilities for plagiarism. The writer should rely more on peer-reviewed sources. Peer-reviewed sources are credible since they have been approved and acknowledged by expertise in the specified field. Hence, references and key terms are very important to their source because they adhere by intellectual property laws and copyrights.

To sum up, this review proposes some changes to the original copy since it holds some issues that can make it uncredible. The author is urged to amend the presented article and change it as per this audit. The author didn't present an all-around organized article since there are a few

main points that were not addressed such as the other types of databases. Moreover, the technical qualifications lack examples on sustainability, durability and availability. Additionally, the sources used are questionable and need some solid expertise opinions to give the reader a better view to grasp the content. Regardless, the writer means to educate the crowd on few famous NoSQL databases through effectively informing how they utilize hardware components for tackling situation-based issues and the downside caused to address the problem.

Part 2:

JSON

The JSON Schema demonstrates a student information data set using various functionalities provided by JSON. These structures include definitions and objects of multiple data types like arrays, string patterns, ranges and conditions.

The structure of the schema demonstrates a university database built as a complex assembly starting with a definition for the address. The JSON definition is an auxiliary schema that can be reused instead of having excess lines of code. The address will hold 3 attributes: Street, City & State where only the City attribute is mandatory, whereas the others are optional. The address definition is used twice when defining the object by calling "$ref": "#/definitions/address" since the object consists of two addresses, permanent address and term address. The term address is required whereas the permanent address is not since the university cares about the current address of the student.

The point of such schema is that all students by default have an ID, name, age, classes registered, and degree being obtained, in addition to the term and permanent address definition. All of the mentioned attributes are mandatory except for classes registered and permanent address.

Moving on to how the student attributes where built, the ID is a string formed in a regex pattern containing “id#[0-9]{4}” this regex forces the ID to be arranged as id#2132, thus having more integrity and diversity between student IDs. The name is a string and doesn’t follow any condition. As for the age section, it is of type integer and the only condition existing is that the student age should range between 18-60 using “exclusiveMinimum”: 17 and “maximum”: 60, since it doesn’t make sense to have students who are less than 18 or older than 60 as university students. The classes data entry is an array that should include at least one up to four unique items/classes chosen by the student. Such conditions are applied by using “minItems”: 1, “maxItems”: 4, “uniqueItems”: true. The level of degree being obtained is a Enum type, where the input can only be specified as “Undergrad”,” Postgrad” or “Unspecified”, any other input would be considered as invalid.

The appendix includes two JSON files: StudentSchema.json and students.json. The first file presents the explained schema and the second includes 20 data entry of students which are fully validated by the schema. The student’s schema has a wide variety of application where all conditions set are demonstrated through different objects.

The Schema was successfully validated using: https://www.liquid-technologies.com/online-json-schema-validator. A screenshot is available in the appendix.

Redis Database

The following Redis Database demonstrates several key-value types to portrait a messaging application database. To preview the diversity of data types in Redis, this database includes a normal key-value pair “project”, 3 hashes “user:1 – 2”, 1 sorted set “users”, 1 unsorted set “accounts”, a list “messages”, a publish subscribe “broadcast” and finally an expiring key. The data was imported using the cat Data command in the shell. The command lines are used in Redis to demonstrate the complexity and how data can be retrieved then manipulated.

```
[nosql@nosql Desktop]$ cat Data | redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 14
```

Figure 1. Batch insert

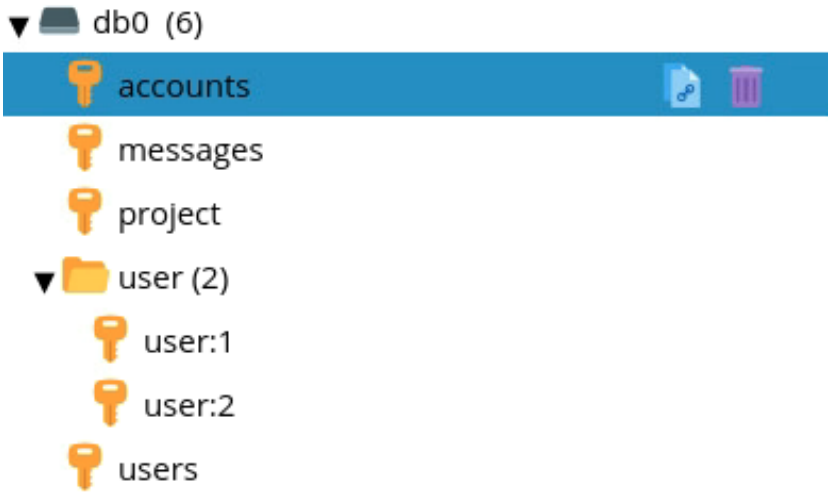

Figure 2: Redis data elements

Open ▾ | Data ~/Desktop | Save

```
SET project "Messaging Redis"
HMSET user:1 id 1 username "John" password "12534"
HMSET user:2 id 2 username "Smith" password "12312"
HMSET user:2 id 3 username "Sam" password "56456"
zadd users 1 John
zadd users 2 Smith
zadd users 3 Sam
sadd accounts personal
sadd accounts business
sadd accounts premium
lpush messages Hello
rpush messages "Hello there!"
```

Figure 3: Redis data creation

```
Local:0>KEYS *
1)" accounts"
2)" project"
3)" user:1"
4)" user:2"
5)" users"
6)" messages"

Local:0>GET project
"Messaging Redis"

Local:0>HEXISTS user:1 id
"1"

Local:0>HGET user:2 username
"Smith"

Local:0>HGET user:1 id
"3"

Local:0>HINCERBY user:1 id 4
"7"

Local:0>HGET user:1 id
"7"
```

Figure 4: Redis queries part 1

To begin with the database, KEYS * was used to show all available keys in our database to make sure that all the keys are imported. This is then proceeded with applying a basic GET for the project to retrieve the value "Messaging Redis". Furthermore, the HashMap serves as a data storage for users, where it saves information like ID, username and password. The HashMap was manipulated through using functions like HEXISTS to check whether the user has a registered ID. In this case, it was checked whether ID exists in user:1, outputting "1" which means TRUE. HGET, a command that permits access to the data values in a HashMap, was implemented to retrieve the username for the second user with value "Smith" and then the ID value for user:1 which gave 3. Finally, HINCERBY is used to increment the ID value of a user, where user:1 ID was increased by 4 so that it becomes 7.

```
Local:0>HGET user:1 id
“7”

Local:0>SISMEMBER accounts personal
“1”

Local:0>SISMEMBER accounts free
“0”

Local:0>SADD accounts_type free
“1”

Local:0>SUNION accounts accounts_type
1)” personal”
2)” free”
3)” premium”
4)” business”
```

Figure 5:Redis queries part 2

Furthermore, the unsorted set purpose is to store information of the type of the accounts/subscriptions existing. Several commands were applied on the unsorted sets. Such commands include SISMEMBER to verify whether the value “personal” exists in accounts which resulted in TRUE. However, when applying SISMEMBER on accounts free it gave as FALSE because it doesn’t exist in the list. Thus, SADD was used to create a new set named accounts type with value “free”. Followed by using SUNION to merge accounts and accounts type values in one set as it shows above.

```
Local:0>ZRANGE users 0 5
1)" John"
2)" Smith"
3)" Sam"

Local:0>ZRANK users John
"0"

Local:0>ZRANK users Smith
"1"

Local:0> LINDEX messages 0
"Hello"

Local:0> LINDEX messages 1
"Hello there!"
```

Figure 6: Redis queries part 3

The sorted list manages to save the user's name; it is a ranking system based on the user's interaction. The first command applied is ZRANGE, it retrieves all values between 0 and 5 in users set and prints them. Afterwards, ZRANK was used to give the rank of "John" in users. In addition, the messages list serves as a message's directory, where two users can be chatting. Messages are saved using LPUSH and RPUSH depending on the sender and receiver. To retrieve the values inside the messages list, LINDEX was applied on the list and specify the index of the message.

```
Local:0>PUBLISH broadcast "Hello users ☺"
(integer) 1

Local:0>SUBSCRIBE broadcast
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "broadcast"
3) (integer) 1
1) "message"
2) "broadcast"
3) "Hello users ☺"
```

Figure 7: Redis queries part 4

Redis also provides a publish and subscribe feature, it allows the senders/publishers to post a message into the channel whereas the receivers/subscribers can read and retrieve the message through subscribing to the same channel. To send a message, the sender publishes a message using PUBLISH broadcast "Hello users ☺". On the other hand, the receiver opts-in using SUBSCRIBE broadcast to subscribe to a channel and read the incoming messages.

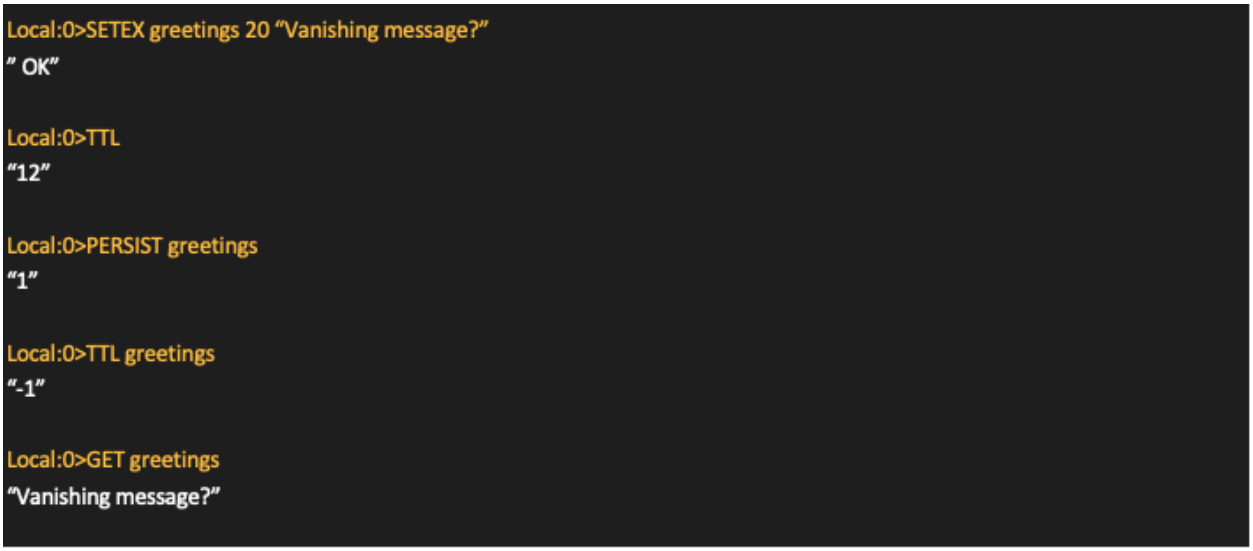

Figure 8: Redis queries part 5

For the final part, SETEX featured can be used as a text message which can be sent and deleted after a specified time. To preview this feature, SETEX creates new message called greetings with value “Vanishing message?” that can vanish in 20 seconds. After that, TLL is used to view how long is left for the message to vanish and finally PERSIST to prevent the greetings message from vanishing. When applying GET greetings, the query retrieved “Vanishing message?” normally.

Cassandra Database

The following Cassandra database is used to provide the user with data about some brands and their sales. The following column-family includes 6 columns initially (altered to 7 later) and 30 rows of data. The columns are separated as follows: ID int, monthly sales int, price double, product name varchar, supplier(brand) varchar and weekly sales int. This database comes handy in retrieving averages, sums, maxima and minima specifically on large datasets with great performance. Such queries will be demonstrated below on this database using terminal since Cassandra GUI tends to be very buggy and still in development mode.

First of all, to avoid time consumption, the data was inserted in bulks into the database through using batch queries. Because the database queries will rely heavily on condition that refer to the supplier, the index of the database was set to be supplier through using “CREATE INDEX supplier ON shop. sales(supplier); “.

id	monthlysales	price	productname	supplier	weeklysales
23	25	45	Force	Adidas	10
33	25	45	Force	Martens	14
5	5	5.99	Flip Flops	Nike	1
28	25	45	Rocks	Martens	19
10	0	120	Waswojk	Adidas	0
16	30	60	Jordans	Nike	6
13	25	45	Force	Adidas	10
30	5	5.99	Flip Flops	Levi	1
11	30	60	Jordans	Nike	6
1	30	60	Jordans	Nike	6
19	16	14	X	Converse	4
8	4	90	Ozweggo	Levi	1
2	12	30	JD	Adidas	3
4	16	67.99	X Ray	Puma	4
18	25	45	Force	Adidas	10
15	5	5.99	Flip Flops	Levi	1
22	12	30	JD	Nike	3
27	12	30	JD	Nike	3
20	5	5.99	Flip Flops	Levi	1
7	11	50	Ozweggo	Adidas	5
6	14	59.99	Provoke	Puma	3
29	16	14	X	Martens	4
9	10	34	Supii	Levi	4
14	16	15	X Ray	Puma	4
26	30	60	Jordans	Martens	6
21	30	60	Shoes	Converse	6
17	12	30	JD	Nike	3
35	5	5.99	Flips	Martens	1
31	30	60	Shoes	Martens	6
24	16	14	X Ray	Puma	4
32	12	30	JD	\nMartens	3
25	5	5.99	Flips	Converse	1
34	16	14	X Ray	Puma	4
12	12	30	JD	Nike	3
3	25	45	Air Force	Nike	10

Figure 9: Cassandra data fields

```
cqlsh:shop> SELECT count(*) FROM sales where supplier = 'Nike';

 count
-------
     9

(1 rows)
```

Figure 10: Cassandra select Nike

In order to get the number of products of a specific brand, select count was applied for all the products where the supplier is equal to that brand. An example is “SELECT COUNT (*) FROM sales WHERE supplier=’Nike’;”, giving us a result of count equal to 9, which is the number of products by Nike.

```
cqlsh:shop> SELECT monthlysales, productName, supplier FROM sales where monthlysales > 20

 monthlysales | productname | supplier
--------------+-------------+----------
           25 |       Force |   Adidas
           25 |       Force |  Martens
           25 |       Rocks |  Martens
           30 |     Jordans |     Nike
           25 |       Force |   Adidas
           30 |     Jordans |     Nike
           30 |     Jordans |     Nike
           25 |       Force |   Adidas
           30 |     Jordans |  Martens
           30 |       Shoes | Converse
           30 |       Shoes |  Martens
           25 |   Air Force |     Nike

(12 rows)
```

Figure 11: Cassandra select sales

Through querying “SELECT monthly sales, product name, supplier FROM sales where monthly sales > 20;” it was possible to retrieve the number of sales, product name and respective brand that has a very high monthly sale.

All of the below queries are very helpful for the sales department since they can provide a lot of information on sales.

```
cqlsh:shop> SELECT productName, price FROM sales WHERE supplier='Martens';

 productname | price
-------------+-------
       Force |    45
       Rocks |    45
           X |    14
     Jordans |    60
       Flips |  5.99
       Shoes |    60

(6 rows)
```

Figure 12: Cassandra select Martens

When using “SELECT product name, price FROM sales WHERE supplier = ‘Martens’;” a user can get all the products listed with their prices filtered by this brand.

```
cqlsh:shop> SELECT AVG(monthlysales) from sales where supplier='Adidas';

 system.avg(monthlysales)
--------------------------
                       16

(1 rows)
```

Figure 13: Cassandra select Adidas

Moreover, to retrieve values like averages, maximum and minimums, the user can apply queries like “SELECT AVG (monthly sales) FROM sales WHERE supplier = ‘Adidas’;”

```
cqlsh:shop> SELECT MIN(monthlysales), MAX(monthlysales) FROM sales where supplier = 'Levi';

 system.min(monthlysales) | system.max(monthlysales)
--------------------------+--------------------------
                        4 |                       10

(1 rows)
```

Figure 14: Cassandra select Levi

Finally, using “SELECT MIN (monthly sales), MIN (monthly sales) FROM sales WHERE supplier = ‘Levi’;” will result in the maximum and the minimum sales hit in the data for Levi brand.

```
cqlsh:shop> ALTER TABLE shop.sales ADD onsale varchar;
cqlsh:shop> select * from sales;

 id | monthlysales | onsale | price | productname | supplier  | weeklysales
----+--------------+--------+-------+-------------+-----------+-------------
 23 |           25 |   null |    45 |       Force |    Adidas |          10
 33 |           25 |   null |    45 |       Force |   Martens |          14
  5 |            5 |   null |  5.99 |  Flip Flops |      Nike |           1
 28 |           25 |   null |    45 |       Rocks |   Martens |          19
 10 |            0 |   null |   120 |     Waswojk |    Adidas |           0
 16 |           30 |   null |    60 |     Jordans |      Nike |           6
 13 |           25 |   null |    45 |       Force |    Adidas |          10
 30 |            5 |   null |  5.99 |  Flip Flops |      Levi |           1
 11 |           30 |   null |    60 |     Jordans |      Nike |           6
  1 |           30 |   null |    60 |     Jordans |      Nike |           6
```

Figure 15: Cassandra alter table

```
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=5;
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=10;
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=15;
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=20;
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=25;
cqlsh:shop> UPDATE sales SET onsale='yes' WHERE id=30;
cqlsh:shop> select * from sales;

 id | monthlysales | onsale | price | productname | supplier  | weeklysales
----+--------------+--------+-------+-------------+-----------+-------------
 23 |           25 |   null |    45 |       Force |    Adidas |          10
 33 |           25 |   null |    45 |       Force |   Martens |          14
  5 |            5 |    yes |  5.99 |  Flip Flops |      Nike |           1
 28 |           25 |   null |    45 |       Rocks |   Martens |          19
 10 |            0 |    yes |   120 |     Waswojk |    Adidas |           0
 16 |           30 |   null |    60 |     Jordans |      Nike |           6
 13 |           25 |   null |    45 |       Force |    Adidas |          10
 30 |            5 |    yes |  5.99 |  Flip Flops |      Levi |           1
 11 |           30 |   null |    60 |     Jordans |      Nike |           6
  1 |           30 |   null |    60 |     Jordans |      Nike |           6
```

Figure 16: Cassandra add elements

The last section focuses on how Cassandra permits columns to have unequal amount of data in the rows. To demonstrate this flexibility, a new column is created called on sale by using “ALTER TABLE shop. sales ADD on sale varchar”, this column will include if the listed product is on sale or not. Initially, all the values in the column are set to be null. Afterwards, it is possible to manually start inserting values inside the rows. Through choosing IDs 5 10 15 20 25 30, the value is now updated to “yes” on the products with the respective ID, however on other rows it is still null and it’s not compulsory to have data on all rows. The query used is “UPDATE sales SET on sale = ‘yes’ WHERE id = 5;”

MongoDB

The following MongoDB database establishes a hospital data set with a collection for the patient's records. It includes twenty patients with a variety of details. Such details include:

- The built-in unique MongoDB "_id".
- A unique patient's ID "pat_id" as a 32Bit integer.
- The patient's full name "full_name" as a string.
- The patient's age "age" as an 32Bit integer.
- The patient's gender "gender" as a string.
- Date of admission "admission_date" as a Date type.
- Checking whether the patient left the hospital or still in-care "checked_out" as a boolean.
- Symptoms shown on the patient "symptoms" as an array of strings.
- Medications prescribed "medications" as an array of objects including name and price.

The queries demonstrate a variety of functionalities supported by MongoDB; these queries include filters, projections, aggregations and map reduced queries. Most of the queries are presented through Compass. However, under every figure the query's syntax for the terminal will also be given. In addition, to prevent MongoDB from scanning every document, indexes are used to offer efficient execution of queries. Using db.patients_records.ensureIndex({ "full_name": 1 }, { "symptoms": 1 }, { "medications" :1 }).

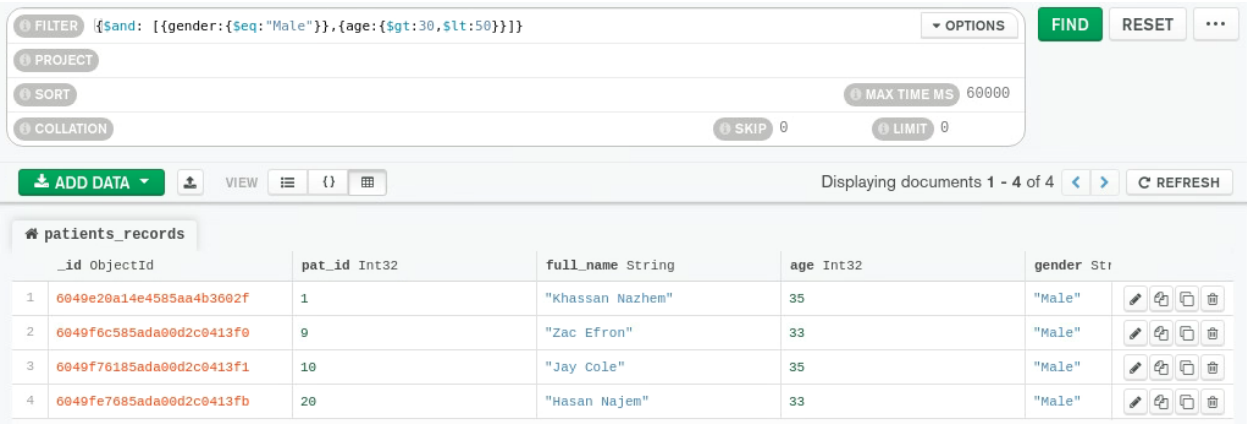

Figure 17: MongoDB select filter

Starting with filtering, this query functions as a filter query that retrieves all the patients that are males aged between 30 to 50 with all their information. Such filtering can be applied by using $and in order to make sure it satisfies both of the conditions. The $eq is used to make sure that the gender is equal to male. In addition, the $gt and $lt filters out the ages that are greater than 30 and less than 50.

```
Query: db.patients_records.find( { $and: [{ gender: { $eq: 'Male' }}, { age: { $gt:30, $lt:50 }}]} ).pretty()
```

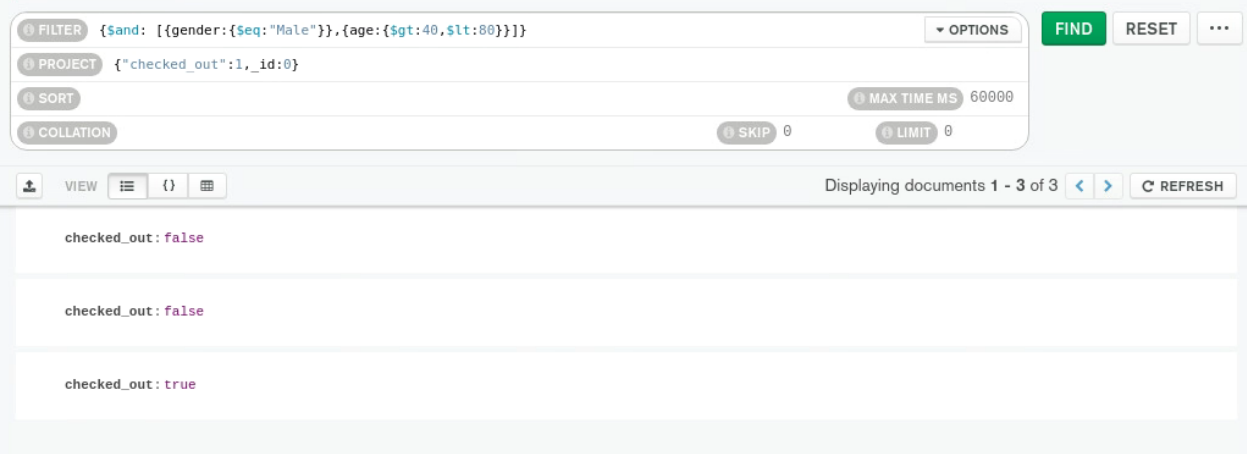

Figure 18: MongoDB select filter & projection

The first part of the query, the filtering, is still the same in this case. However, the difference is that the projection is applied. It manages to output specific information related to the data filtered. Hence, after retrieving all males whose age range between 30 and 50 it only shows the field which is checked out by checking it 1 and put _id 0 to avoid retrieving it.

```
Query: db.patients_records.find( { $and: [{ gender: { $eq: 'Male' }}, { age: { $gt:30, $lt:50 }}]} , { checked_out: 1, _id: 0} ).pretty()
```

Figure 19: MongoDB select filter, projection & sort

This query starts with a filter query that retrieves all the patients that are females and their age range between 20 to 40 with all their information. Next, a projection was added to the query which aims to retrieve only the name and the age. Finally, this query was sorted according to the age in an ascending manner.

```
Query: db.patients_records.find( { $and: [{ gender: { $eq: Female }}, { age: { $gt:20, $lt:40 }}]} ).sort(age:1).pretty()
```

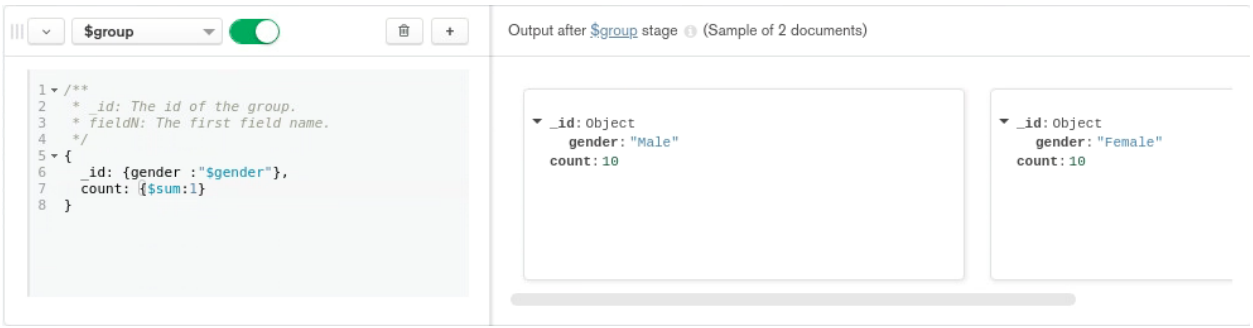

Figure 20: MongoDB aggregate $group

Moving to applying aggregation in MongoDB, this query functions as an aggregation query that retrieves the count of patients according to their gender. To achieve such aggregation, $group is applied followed by creating a field called count which sums the records, $sum:1 for each record.

```
Query: db.patients_records.aggregate( { $group: { gender: ”$gender”, count: { $sum:1 }}} )
```

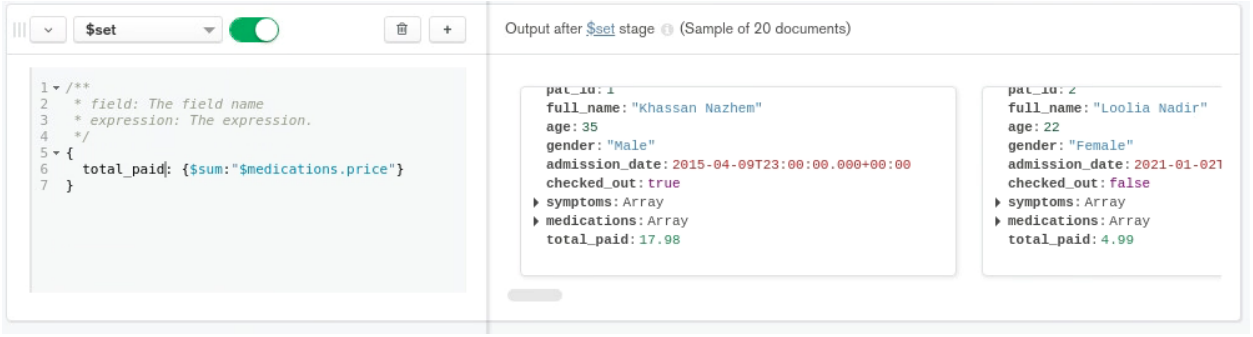

Figure 21: MongoDB aggregate $set

The $set aggregate is used to create a new field called total paid for each patient. This field includes the summation of medicine prices prescribed for each patient.

```
Query: Db.patients_records.aggregate( { $set: total_paid: { $sum: “$medications.price” }} )
```

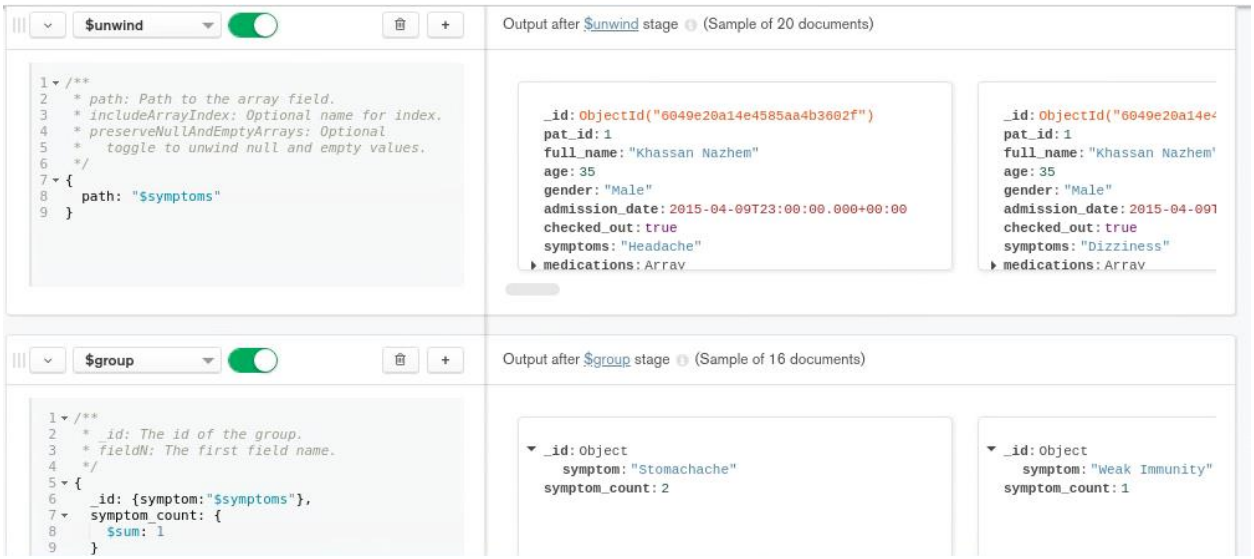

Figure 22: MongoDB aggregate $unwind & $group

The aim of this query is to count the number of symptoms for each patient. Using aggregation functions, $unwind is used in order to separate the elements of the symptoms array. Afterwards, the symptoms are grouped into a symptom counter by using the $group and $sum operations.

```
Query: db.patients_records.aggregate( [ { $unwind: “$symptoms” }, { $group: { symptoms: “$symptoms”, symptom_count: { $sum :1} } ])
```

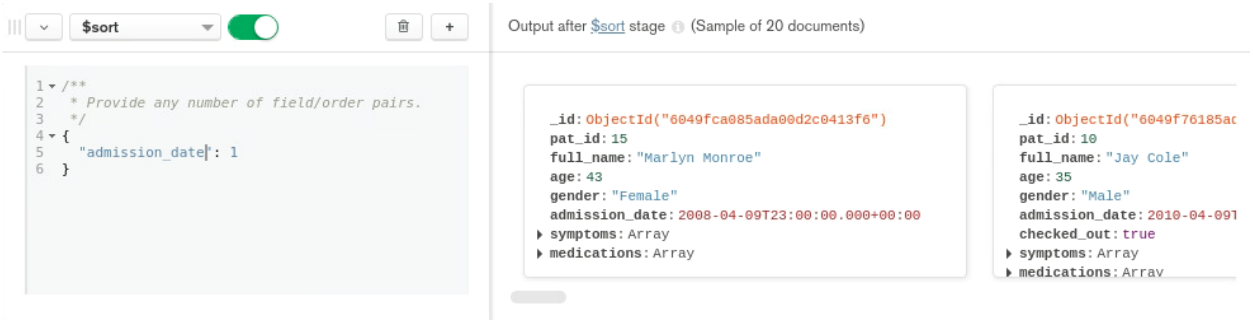

Figure 23: MongoDB aggregate $set

Finally, the aim of this query is to sort the records based on their admission dates using the $sort aggregation function.

```
Query: db.patients_records.aggregate(  $sort: { admission_date: 1 })
```

```
> db.patients_records.mapReduce( function(){if(this.hasOwnProperty("medications"
)){ this.medications.forEach(function(medication){emit(medication.name,medicatio
n.price);});}}, function(key,price){ return Array.sum(price);}, {out:"map_exampl
e"})
> db.map_example.find().pretty()
{ "_id" : "Almozhel", "value" : 55.99 }
{ "_id" : "Chemo2", "value" : 121.99 }
{ "_id" : "Advil", "value" : 11.98 }
{ "_id" : "Antibiotics", "value" : 91.97999999999999 }
{ "_id" : "Chemo1", "value" : 149.99 }
{ "_id" : "Pencilin", "value" : 59.98 }
{ "_id" : "Chemo3", "value" : 111.99 }
{ "_id" : "Surgery", "value" : 100.99 }
{ "_id" : "Syrenges", "value" : 45.99 }
{ "_id" : "Morphine", "value" : 99.99 }
{ "_id" : "Pills", "value" : 31.98 }
{ "_id" : "Teracylines", "value" : 53.98 }
{ "_id" : "Gum teeth", "value" : 49.99 }
{ "_id" : "Colgate", "value" : 1.99 }
{ "_id" : "Bandaid", "value" : 10.99 }
{ "_id" : "Panadol", "value" : 23.98 }
{ "_id" : "Herbs", "value" : 11.98 }
{ "_id" : "Ficodin", "value" : 21.99 }
{ "_id" : "Prophinal", "value" : 4.99 }
```

Figure 24: MongoDB MapReduce part 1

The goal of this map reduced function is to get all the medications that were prescribed for patients and calculate the profit made by each medication. In order to do so, the map function serves as a collector by looping over the medication array and collect the medicine's name and price, then emit the key-value. However, it is noteworthy to mention that because not all patients have medications prescribed, then it is important to use hasOwnProperty function to make sure that the document in hand satisfies the condition. In addition, the reduce function serves for summing the prices for each key (medication given). The output of this function is saved in map example, which can be retrieved to print the data.

```
> db.patients_records.mapReduce( function(){ this.symptoms.forEach(function(symp
tom){emit(symptom,1);});}, function(key,price){ return Array.sum(price);}, {out:
"map_example"})
{ "result" : "map_example", "ok" : 1 }
> db.map_example.find().pretty()
{ "_id" : "Hair loss", "value" : 1 }
{ "_id" : "Swollen neck", "value" : 2 }
{ "_id" : "Ingrown Nail", "value" : 1 }
{ "_id" : "Dizziness", "value" : 2 }
{ "_id" : "Titonnis", "value" : 2 }
{ "_id" : "Waist pain", "value" : 2 }
{ "_id" : "Blurred Vision", "value" : 1 }
{ "_id" : "Stomachache", "value" : 2 }
{ "_id" : "Weak Immunity", "value" : 1 }
{ "_id" : "Broken arm", "value" : 1 }
{ "_id" : "Wisdom teeth", "value" : 1 }
{ "_id" : "Intestinal pain", "value" : 1 }
{ "_id" : "Nausea", "value" : 2 }
{ "_id" : "Headache", "value" : 2 }
{ "_id" : "Heartache", "value" : 1 }
{ "_id" : "Cancer", "value" : 1 }
```

Figure 25: MongoDB MapReduce part 2

The following MapReduce functions as the aggregation count by sum. The goal is to retrieve all the symptoms available in the database and count the repetition of each one. So, for each symptom in the symptoms array we emit a key value pair made of symptom name and count of 1. The Reduce function then sums up all the counts and stores the output in map example.

Neo4J

The constructed Neo4J graph databases serve as a storing asset for a farming company. Through this database, the users can retrieve information about the nodes and their relationships. The nodes include farmers, plants, animals, pests, chemicals and diseases. So, through such a graph, the user will be able to retrieve information about the role of farmers in growing plants, treating them, and raising animals. In addition, it is also possible to retrieve information on which plants are affected by which diseases or pests and what chemical can be used for treatment. Also, information can be recalled regarding the origin of some pests and some chemicals in use.

Before going through the queries section and retrieving information, it is important to talk about the creation of the nodes stage in which it focuses on data uniqueness and the pre indexing. To reassure uniqueness, MERGE command was used to create nodes based on label and key information for the properties, it ensures that for every plant, animal node has a unique name, thus avoiding duplicates. Finally, to make the search queries more efficient, it is recommended to apply indexing on some nodes that have an existing label. Since the query may rely heavily on plants, the indexing is applied on plant's "Fresh" and "Leftovers" and on the farmer's "Livestock", "Gardener", "Exterminator", and" Medicator".

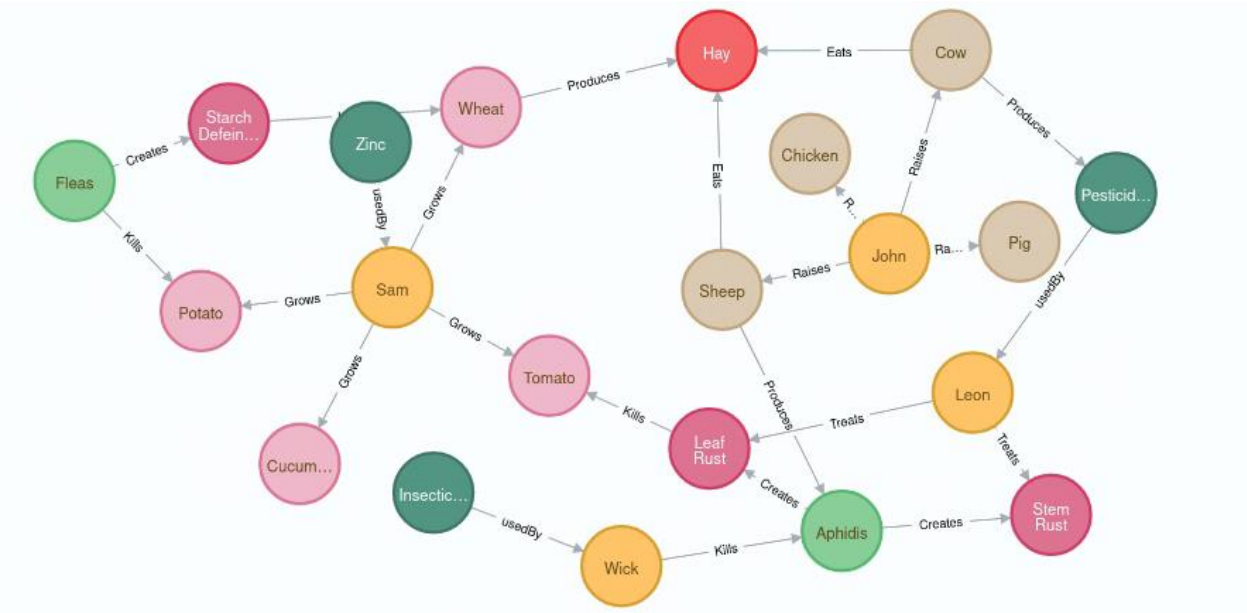

Figure 26: Neo4J map

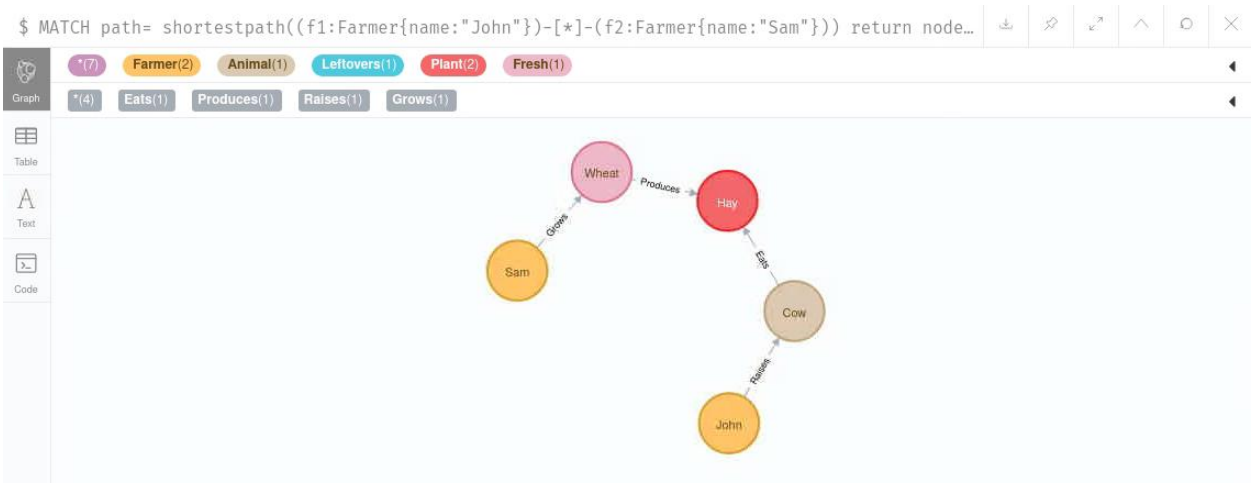

Figure 27: Neo4J shortest path

The following query finds the shortest path between the farmers John and Sam. To receive such information, the user should apply the shortest path function and specify between which two nodes (Farmers) the connection is wanted. The connection of relationship is set as * since the length specific relationships existing is unknown. As seen from the graph, 3 nodes and 4 relationships separate the two farmers. Sam grows wheat and wheat produces hay, cows are raised by John and cows eat hay.

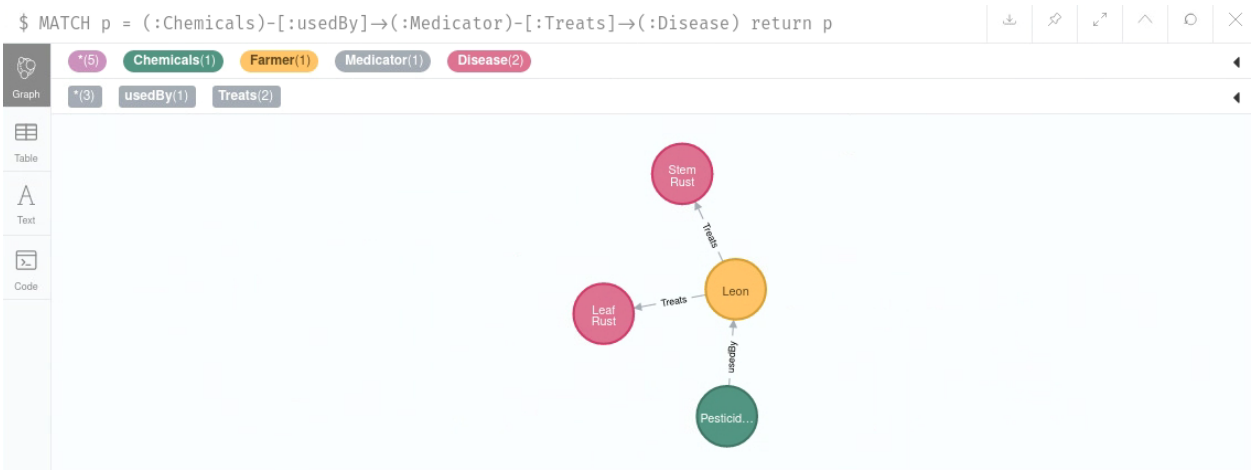

Figure 28: Neo4J match path part 1

In this query, the goal is to find the path for the chemicals that are used by Leon and what chemicals treat which diseases. In the query the only specified parameter is the farmer, it is set to be a medicator using the medicator label since the query is finding all chemicals and all respective diseases. Output: Leon uses pesticides to treat both leaf rust and stem rust.

Figure 29: Neo4J match disease

Through such query, the user can gain information about the diseases that target fresh plants, tomato specifically. Hence, in a real-life scenario, the user will know which treatment to use for this disease. Through specifying the plants name as tomato since it's our target, plus using the AS annotation to transform d.name to Disease for a better presentation, the output is observed to be Leaf Rust.

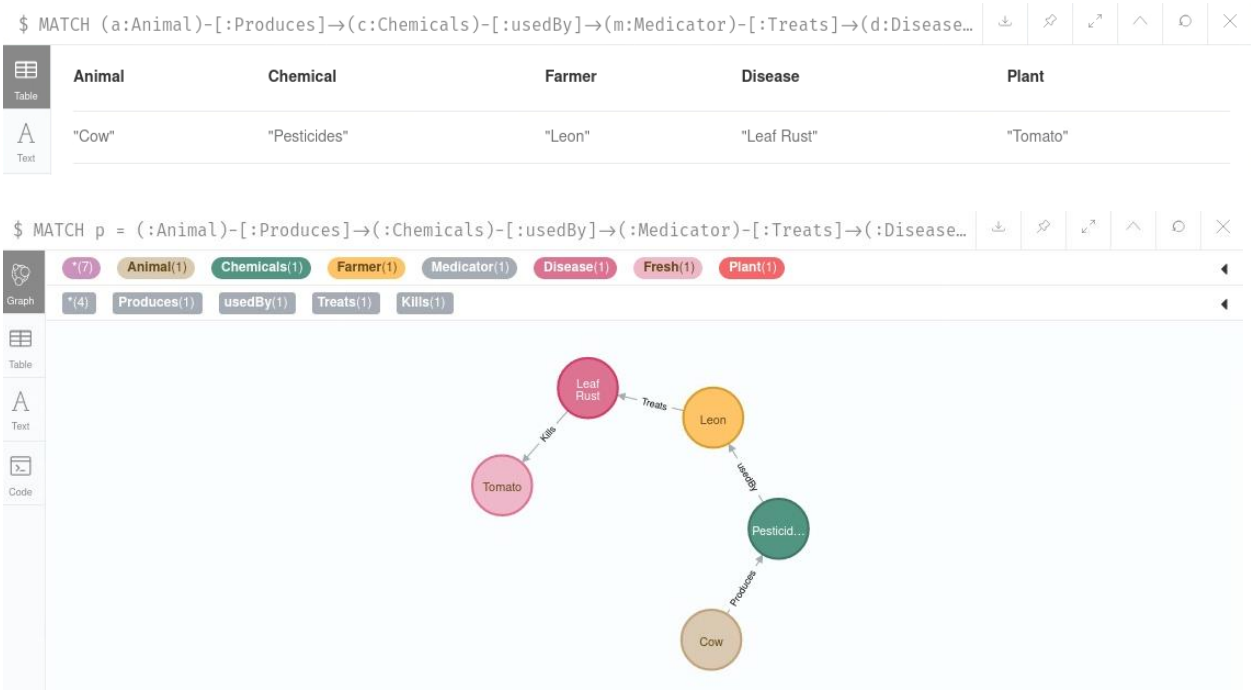

Figure 30: Neo4J match path part 2

The following query is a set of relationships. These relationships occur between the animals that can produce chemicals which are used by the farmer/medicator to treat the plant from a certain disease. Hence, through matching all the mentioned nodes and their relationships, and output each node's information accompanied with the AS annotation. To elaborate more on the output, it means that's cows produce pesticides, and these pesticides are used by Leon to treat tomatoes from leaf rust.

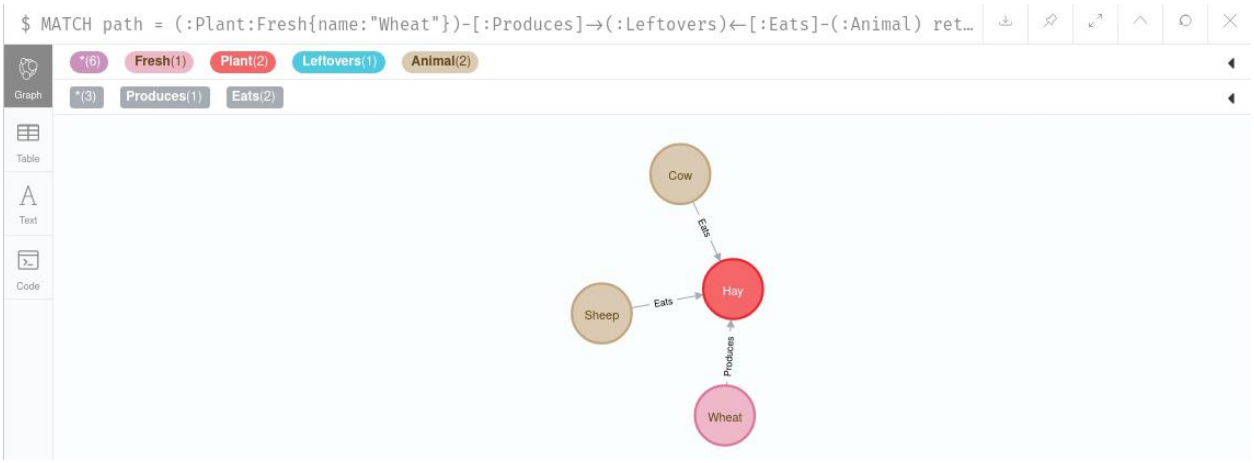

Figure 31: Neo4J match part 3

The following query is good to understand what wheat produces as leftovers and retrieve which animals will feed on these leftovers. In this case, wheat produces leftovers of hay, and hay can be used as food for sheep and cows.

Figure 32: Neo4J match pattern

The gardener is this farm is responsible to grow plants and support with them with chemicals for better products. In this query, the aim is to find the chemicals that are used by the gardener on fresh plants However, the data collected only focuses on the chemicals that are used in amounts higher than 160 ml. The output shows the plants and the respective chemical used to increase the plant's growth.

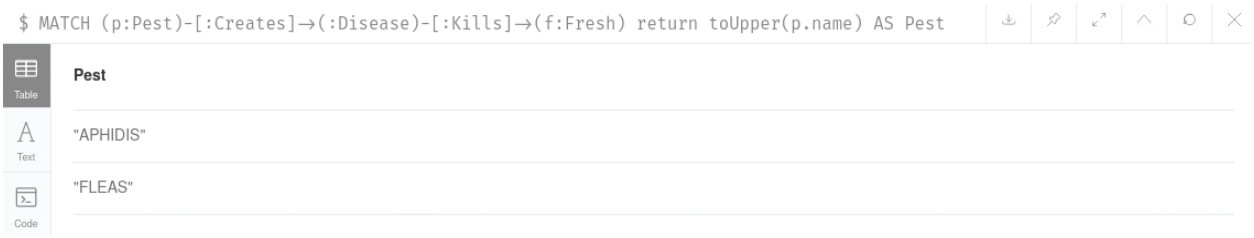

Figure 33: Neo4J match pests

Neo4J also supports string functions, so demonstrate it, to upper function is applied on pest's name that are creating diseases which directly kill the fresh plants.

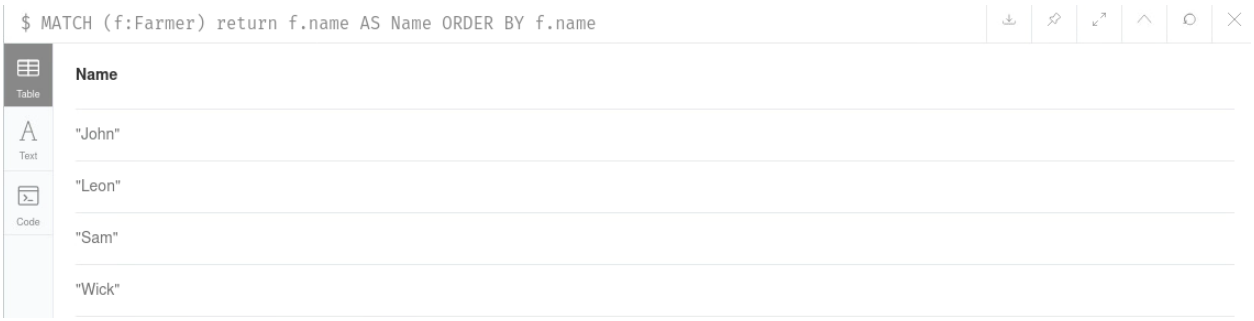

Figure 34: Neo4J match names sorted

In addition, Neo4J also supports aggregation function to sort out the output based on a chosen parameter. This query retrieves all the farmers name that exist in this database and sort the names in alphabetical order using ORDER BY aggregation.

Figure 35: Neo4J match pests count

Besides to having the ability to sort the output, Neo4J also provides functions like sum, maximum, minimum and count. This query is used to get the count/number of diseases created by each pest.

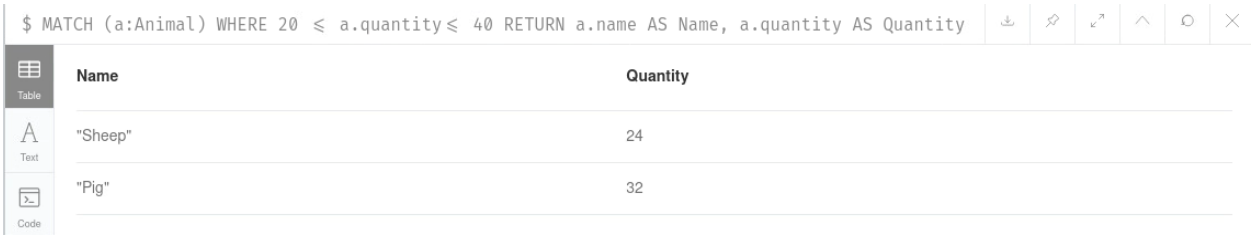

Figure 36: Neo4J match animals count

In this final query, the query retrieves all the animals that a property quantity between 20 and 40. Then proceed by outputting the name and the quantity of each animals that fall under those conditions.

Appendix:

Successful validation:

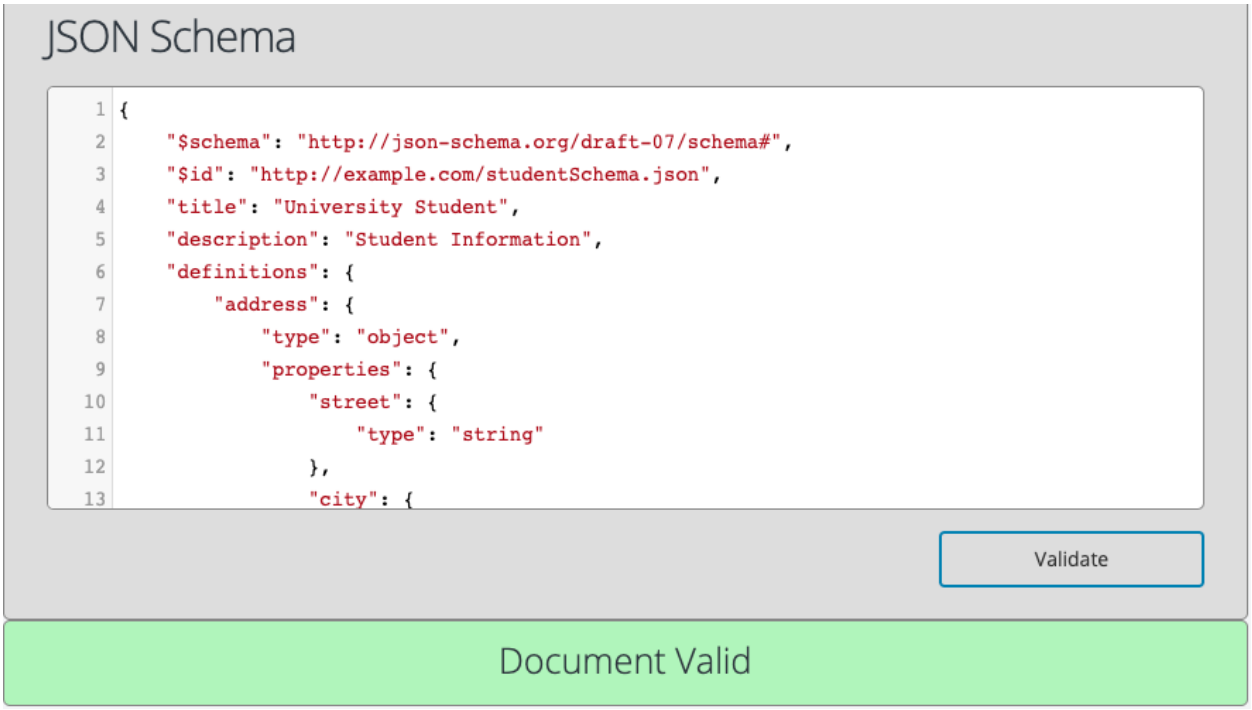

Figure 37: JSON validation

Schema Document: (StudentSchema.json)

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://example.com/studentSchema.json",
  "title": "University Student",
  "description": "Student Information",
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street": {
          "type": "string"
        },
        "city": {
          "type": "string"
        },
        "state": {
          "type": "string"
        }
      },
```

```
      "required": [
        "city"
      ]
    }
  },
  "type": "object",
  "properties": {
    "studentID": {
      "description": "Unique identifier",
      "type": "string",
      "pattern": "id#[0-9]{4}"
    },
    "studentName": {
      "description": "Name of student",
      "type": "string"
    },
    "studentAge": {
      "description": "Age of student",
      "type": "integer",
      "exclusiveMinimum": 17,
      "maximum": 60
    },
    "classesTaken": {
      "description": "Which classes the student is taking",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "maxItems": 4,
      "uniqueItems": true
    },
    "level": {
    "description": "Level of education",
      "enum": [
        "Undergrad",
        "Postgrad",
        "Unspecified"
      ]
    },
    "permanent_address": {
      "$ref": "#/definitions/address"
    },
    "term_address": {
      "$ref": "#/definitions/address"
    }
  },
  "required": [
```

```
    "studentID",
    "studentName",
    "studentAge",
    "level",
    "term_address"
  ]
}
```

Data Document: (students.json)

```
{
 "student1": {
  "studentID": "id#4534",
  "studentName": "John",
  "studentAge": 23,
  "classesTaken":["Astronomy","Maths"],
  "level": "Undergrad",
  "permanent_address":{
   "street": "Paris",
   "city": "France",
   "state": "Central"
  },
  "term_address": {
   "street": "Greenford",
   "city": "London",
   "state": "East",
   "post": "TW5"
  }
 },
 "student2": {
  "studentID": "id#5345",
  "studentName": "Karen",
  "studentAge": 19,
  "classesTaken":["Physics","Biology"],
  "level": "Postgrad",
  "term_address": {
   "street": "Oxford",
   "city": "London",
   "state": "West"
  }
 },
 "student3": {
  "studentID": "id#3123",
  "studentName": "Sam",
  "studentAge": 38,
  "level": "Unspecified",
  "term_address": {
   "city": "London",
```

```
    "state": "Central",
    "post": "UB9"
   }
  },
  "student4": {
   "studentID": "id#7657",
   "studentName": "Khassan",
   "studentAge": 45,
   "level": "Undergrad",
   "permanent_address":{
    "street": "Loral",
    "city": "Beirut"
   },
   "term_address": {
    "street": "Oxford",
    "city": "London"
   }
  },
  "student5": {
   "studentID": "id#4535",
   "studentName": "Fred",
   "studentAge": 21,
   "classesTaken":["Chemistry","Art"],
   "level": "Postgrad",
   "permanent_address":{
    "street": "Karlsruhe",
    "city": "Frankfurt",
    "post":"HG9"
   },
   "term_address": {
    "city": "London",
    "post":"UF2"
   }
  },
  "student6": {
   "studentID": "id#1231",
   "studentName": "Jeff",
   "studentAge": 59,
   "level": "Unspecified",
   "term_address": {
    "city": "London"
   }
  },
  "student7": {
   "studentID": "id#9797",
   "studentName": "Lily",
   "studentAge": 19,
   "level": "Postgrad",
```

```
  "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "South"
  }
 },
 "student8": {
  "studentID": "id#7657",
  "studentName": "Samantha",
  "studentAge": 33,
  "classesTaken":["Chemistry","Art"],
  "level": "Undergrad",
  "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "Central"
  }
 },
 "student9": {
  "studentID": "id#4324",
  "studentName": "Silvia",
  "studentAge": 37,
  "level": "Unspecified",
  "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "Central"
  }
 },
 "student10": {
  "studentID": "id#7456",
  "studentName": "Eren",
  "studentAge": 46,
  "level": "Postgrad",
  "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "Central"
  }
 },
 "student11": {
  "studentID": "id#9872",
  "studentName": "Mikasa",
  "studentAge": 23,
  "classesTaken":["Astronomy","Maths"],
  "level": "Undergrad",
  "term_address": {
    "street": "Greenford",
```

```
    "city": "London",
    "state": "East",
    "post": "TW5"
   }
  },
  "student12": {
   "studentID": "id#7657",
   "studentName": "Armin",
   "studentAge": 19,
   "classesTaken":["Physics","Biology"],
   "level": "Postgrad",
   "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "West"
   }
  },
  "student13": {
   "studentID": "id#1232",
   "studentName": "Levi",
   "studentAge": 38,
   "level": "Unspecified",
   "term_address": {
    "city": "London",
    "state": "Central",
    "post": "UB9"
   }
  },
  "student14": {
   "studentID": "id#9879",
   "studentName": "Hange",
   "studentAge": 45,
   "level": "Undergrad",
   "term_address": {
    "street": "Oxford",
    "city": "London"
   }
  },
  "student15": {
   "studentID": "id#6566",
   "studentName": "Fred",
   "studentAge": 21,
   "classesTaken":["Football","Swimming","Tenis"],
   "level": "Postgrad",
   "term_address": {
    "city": "Barcelona",
    "post":"JF2"
   }
```

```
},
"student16": {
 "studentID": "id#3444",
 "studentName": "Jeff",
 "studentAge": 59,
 "level": "Unspecified",
 "term_address": {
  "city": "Manchester"
 }
},
"student17": {
 "studentID": "id#7777",
 "studentName": "Lily",
 "studentAge": 19,
 "level": "Postgrad",
 "permanent_address":{
  "city": "Brussels"
 },
 "term_address": {
  "street": "Oxford",
  "city": "London",
  "state": "South"
 }
},
"student18": {
 "studentID": "id#4234",
 "studentName": "Samantha",
 "studentAge": 33,
 "classesTaken":["Chemistry","Art"],
 "level": "Undergrad",
 "term_address": {
  "street": "Oxford",
  "city": "London",
  "state": "Central"
 }
},
"student19": {
 "studentID": "id#7675",
 "studentName": "Silvia",
 "studentAge": 37,
 "level": "Unspecified",
 "term_address": {
  "street": "Oxford",
  "city": "London",
  "state": "Central"
 }
},
"student20": {
```

```
  "studentID": "id#3434",
  "studentName": "Eren",
  "studentAge": 46,
  "level": "Postgrad",
  "permanent_address":{
    "street": "Sepal",
    "city": "Berlin",
    "post":"HG9"
  },
  "term_address": {
    "street": "Oxford",
    "city": "London",
    "state": "Central"
  }
}
```

Evaluation:

Through the COMP1835 Graph and Modern Databases, I was able to gain a vast amount of information about NoSQL and Graph databases. This module successfully introduced all modern NoSQL databases and their respective usage. These databases include: Redis, Cassandra, JSON, MongoDB, Neo4J and Allegro Graph. Additionally, I was able to understand NoSQL which database fits to which scenario as well as the strengths and weaknesses of each of the mentioned database.

The coursework effectively presented a challenging environment to make use of the tools and practices taught from module in order to implement them in managing the required data system. In addition, I designed and built data systems belonging to the NoSQL databases and Graph databases. Finally, the peer review was an informative task to understand the requirements of academic articles and how to serve them.