CSY3024 – DATABASES 3

LEVEL 6

Assignment 2 (2019/20)

Diwas Lamsal

18406547

University of Northampton,

NAMI College

2020

Table of Contents

PART II – MongoDB task

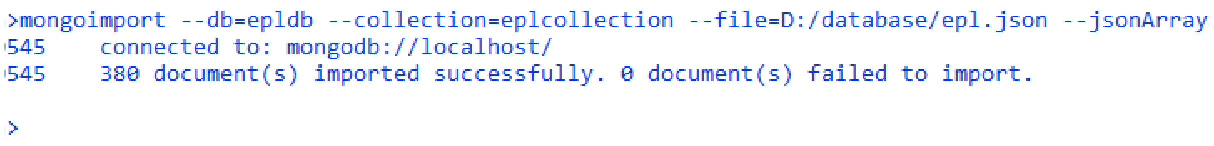
The contents for part one which includes the lab exercises and diary contents are included in the Appendix.

Loading the Dataset JSON File

The assignment package included a JSON file which was mentioned as being the EPL dataset used for Cypher previously but converted to JSON for MongoDB usage. The dataset is said to contain the EPL league match information. The first step was to load the JSON file to MongoDB database. Initial attempts introduced an error that said, "cannot decode array into a D". It was solved with the help of an answer found at Stack Overflow by daydreamer by simply adding "--jsonArray" at the end of the import command (daydreamer, 2019).

Final Import Command:

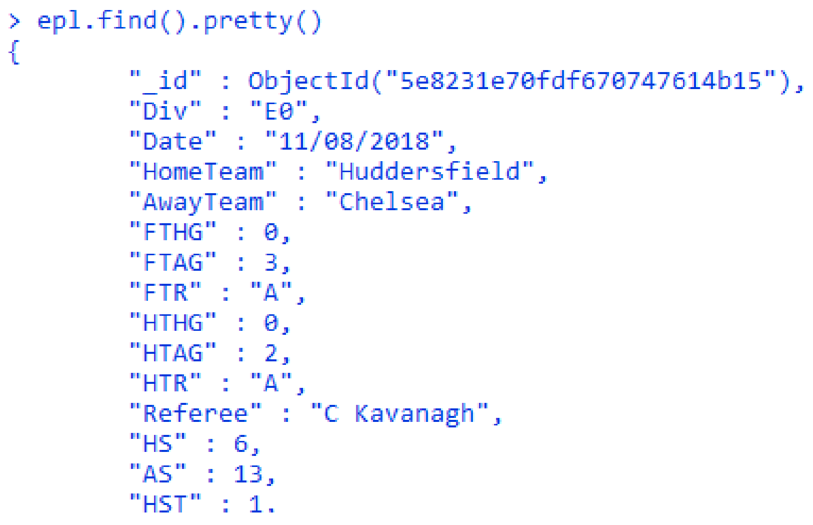
mongoimport --db=epldb --collection=eplcollection --file=D:/database/epl.json --jsonArray



```
>mongoimport --db=epldb --collection=eplcollection --file=D:/database/epl.json --jsonArray
545    connected to: mongodb://localhost/
545    380 document(s) imported successfully. 0 document(s) failed to import.

>
```

Figure 1 – Importing JSON to MongoDB Database

This created a new database epldb and a collection eplcollection where all the JSON data was imported as BSON documents ready to be queried with MongoDB commands. This was tested by first navigating to the database and assigning a variable epl to the collection to be used later. The initial epl.find().pretty() command resulted in the following output:



```
> epl.find().pretty()
{
        "_id" : ObjectId("5e8231e70fdf670747614b15"),
        "Div" : "E0",
        "Date" : "11/08/2018",
        "HomeTeam" : "Huddersfield",
        "AwayTeam" : "Chelsea",
        "FTHG" : 0,
        "FTAG" : 3,
        "FTR" : "A",
        "HTHG" : 0,
        "HTAG" : 2,
        "HTR" : "A",
        "Referee" : "C Kavanagh",
        "HS" : 6,
        "AS" : 13,
        "HST" : 1.
```

Figure 2 – Find().Pretty() screenshot

Querying the Database to Answer the Questions

Before answering any questions, it was necessary to take note of the fields and values and what they represent. The assumption was that these are equivalent to what was provided in the previous dataset and were used accordingly.

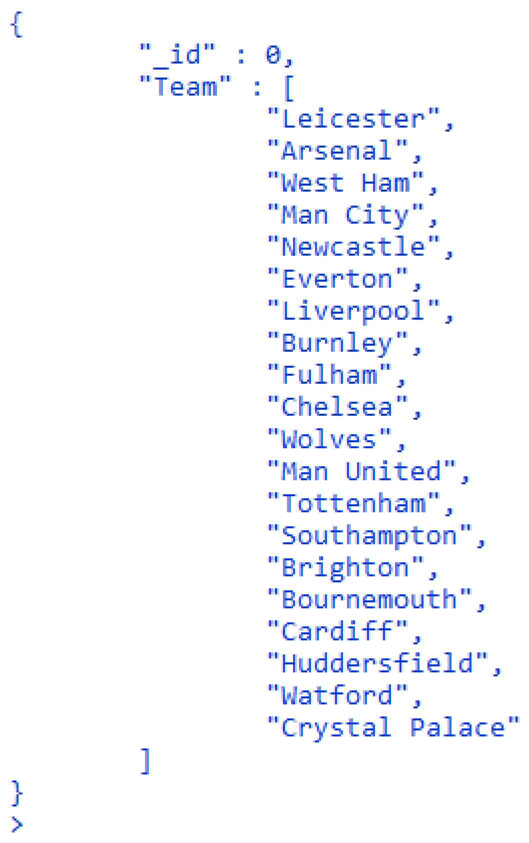
1) Show all the EPL teams involved in the season.

- In order to display unique records, the availability of "$addToSet" array operator was found in the MongoDB documentation ($addToSet — MongoDB Manual, 2020). The documentation mentions that this operator adds unique values to an array. This is similar to implementation of DISTINCT LISTs in the chapter Cypher 7. This could be used to first add all the team names to an array and display the array. The problem faced initially without using $addToSet was duplication of teams in the output as both Home and Away teams had to be displayed. Even though the same twenty teams are available for both Home and Away matches, using only one of these fields could have a different outcome when some teams have not played one of these matches. Due to this, both the home and away teams were added to a single array which only holds unique elements. This would prevent duplication as well as omission.

Answer: Twenty teams as shown in the screenshot.

Code:

```
epl.aggregate([
   {
     $group:
     {
        _id:0,
        Team: {$addToSet: '$HomeTeam'},
        Team: {$addToSet: '$AwayTeam'}
   }}
 ]).pretty();
```



```
{
        "_id" : 0,
        "Team" : [
                "Leicester",
                "Arsenal",
                "West Ham",
                "Man City",
                "Newcastle",
                "Everton",
                "Liverpool",
                "Burnley",
                "Fulham",
                "Chelsea",
                "Wolves",
                "Man United",
                "Tottenham",
                "Southampton",
                "Brighton",
                "Bournemouth",
                "Cardiff",
                "Huddersfield",
                "Watford",
                "Crystal Palace"
        ]
}
>
```

Figure 3 – All the Teams involved

The screenshot shows the list of teams involved in the EPL by assuming that all the teams have played at least a Home or an Away match.

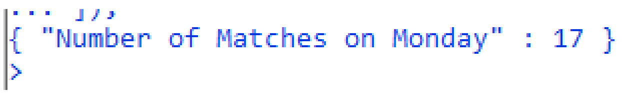
2) How many matches were played on Mondays?

- This question required further research into MongoDB Date object. The first step was to convert the provided string in dd/mm/yyyy format to a MongoDB date object after failed attempts to use the string directly. While searching for a way to do this, the $dateFromString operator was found among the Aggregation Pipeline Operators ($dateFromString — MongoDB Manual, 2020). The documentation provides the required fields and values and examples of usage. It was also found that another operator $dayOfWeek could be used to retrieve day of a week as a numeric value (2 for Monday) ($dayOfWeek — MongoDB Manual, 2020). An extensive study of the MongoDB Aggregation Pipeline Stages allowed implementation of the stages such as $match, $project, $group, $count, etc (Aggregation Pipeline Stages — MongoDB Manual, 2020). $match was used here to restrict the values to 2 (Monday) and $count was used to count the number of results.

Answer: 17 matches.

Code:

```
epl.aggregate( [
{
        "$project" : {
                "dofweek" : {
                        "$dayOfWeek" : {
                                "$dateFromString" : {
                                        "dateString": "$Date",
                                        "format": "%d/%m/%Y"
                                }
                        }
                }
        }
},
{"$match": {"dofweek": 2}},
{"$count":"Number of Matches on Monday"}
]);
```



```
... ]);
{ "Number of Matches on Monday" : 17 }
>
```

Figure 4 -Total Matches on Monday

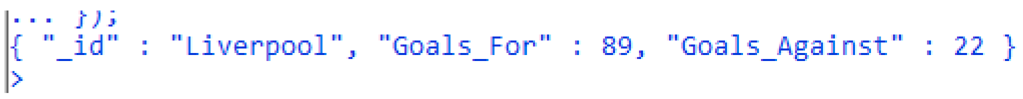
3) Display the total number of goals "Liverpool" had scored and conceded in the season.

- This question required knowing whether Liverpool was the Home team or the Away team in any match between Liverpool and another opponent. The initial $match is used to select only the games where Liverpool is playing. The $cond operator is equivalent to using CASE statement or IF-ELSE statement. In the $project pipeline stage, the goals for or against Liverpool were filtered using this operator. If Liverpool is the home team, the goals scored by Liverpool should be incremented by the FTHG number (Full Time Home Team Goals) and the same goes as FTAG while being an away team. The conceded goals are also filtered in the same manner just reversing the condition. By aggregating the results using $sum operator, the total number of goals scored and conceded can be retrieved.

Answer: Scored: 89, Conceded: 22

Code:

```
epl.aggregate({
$match: {$or : [
                        {"HomeTeam" : "Liverpool"},
                        {"AwayTeam" : "Liverpool"}
                ]
        }
},
{"$project" : {
                _id: 0,
                Team: "Liverpool",
                "HomeTeam": 1,
                "AwayTeam": 1,
                Goals_For: {$cond: [{
                        $eq: ['$HomeTeam', 'Liverpool']}, '$FTHG', '$FTAG']
                },
                Goals_Against: {$cond: [{
                        $eq: ['$AwayTeam', 'Liverpool']}, '$FTHG', '$FTAG']
                }
        }
},
{$group: {
                _id: "$Team",
                Goals_For: {$sum: '$Goals_For'},
                Goals_Against: {$sum: '$Goals_Against'}
        }
});
```



```
... });
{ "_id" : "Liverpool", "Goals_For" : 89, "Goals_Against" : 22 }
>
```

Figure 5 – Goals for and against by Liverpool

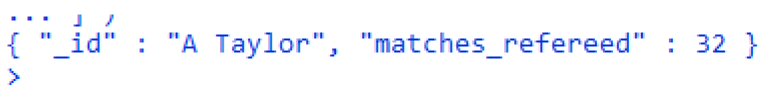
4) Who refereed the most matches?

- This answer made simple use of group, sort and limit pipelines. The $group stage finds the number of occurrences for each referee which would refer to the number of matches refereed by any particular referee. The $sort in descending order and $limit of 1 are used to display only the referee with most matches.

Answer: A Taylor refereed the most matches with 32 matches.

Code:

```
epl.aggregate( [
  {
    $group: {
        _id : "$Referee",
       matches_refereed: { $sum: 1 }
    }
  },
  { $sort: { matches_refereed: -1 } },
  { $limit : 1 }
] )
```



```
... ] )
{ "_id" : "A Taylor", "matches_refereed" : 32 }
>
```

Figure 6 – Refereed most matches

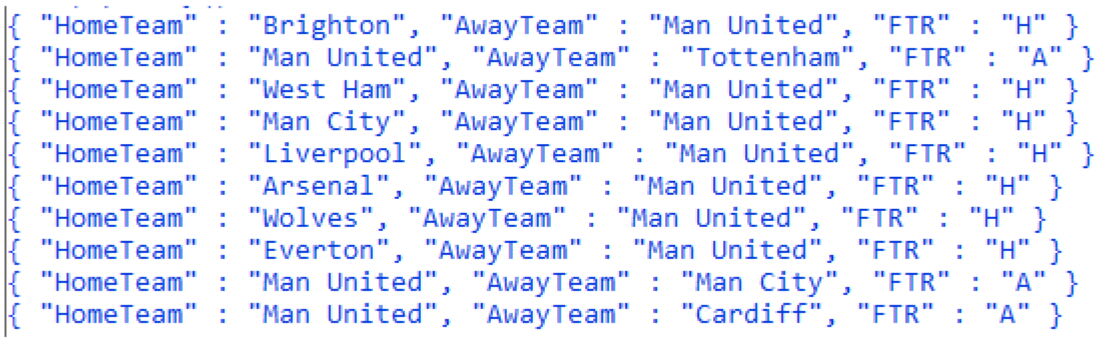
5) Display all the matches that “Man United” lost.

- This answer is different from the rest because find() is used instead of aggregate() and simple and-or operators are used to filter the records where Man United lost. The condition where Man United loses is if it is the home team and the full-time result is A or if it is the away team and the full-time result is H. These records are filtered and projected by displaying only relevant information.

Answer: The 10 matches as shown in the screenshot.

Code:

```
epl.find(
{
$or : [
   {
          $and : [
                          {"HomeTeam" : "Man United"},
                          {"FTR" : "A"}
                  ]
   },
   {
          $and : [
                          {"AwayTeam" : "Man United"},
                          {"FTR" : "H"}
                  ]
   }
          ]
},
{
         HomeTeam: 1, AwayTeam: 1, FTR: 1, _id: 0
}).pretty()
```



```
{ "HomeTeam" : "Brighton", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Man United", "AwayTeam" : "Tottenham", "FTR" : "A" }
{ "HomeTeam" : "West Ham", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Man City", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Liverpool", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Arsenal", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Wolves", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Everton", "AwayTeam" : "Man United", "FTR" : "H" }
{ "HomeTeam" : "Man United", "AwayTeam" : "Man City", "FTR" : "A" }
{ "HomeTeam" : "Man United", "AwayTeam" : "Cardiff", "FTR" : "A" }
```

Figure 7 - Man United Lost Games

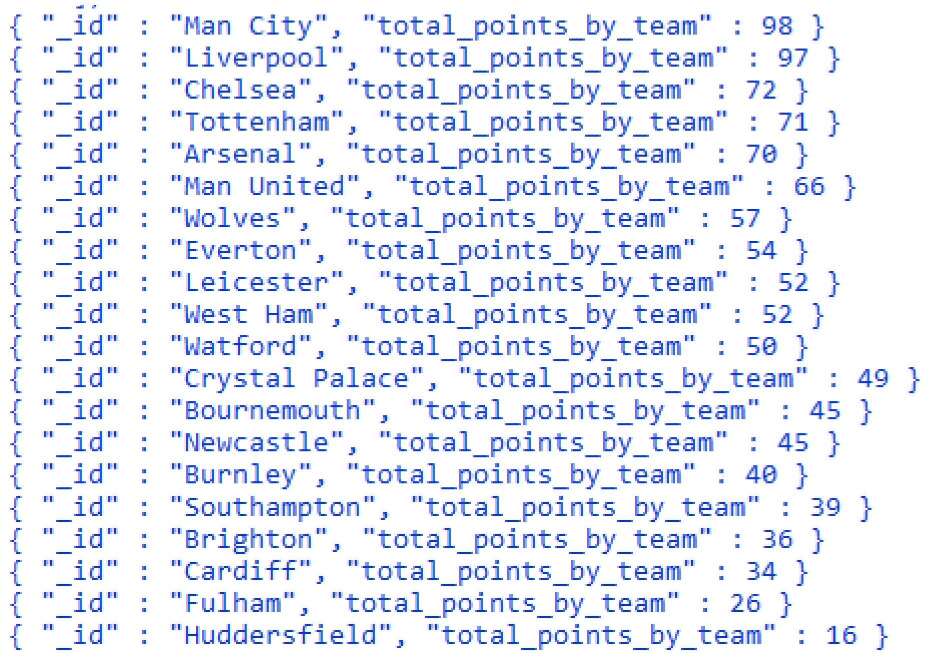
6) Write a query to display the final ranking of all the teams based on their total points.

This was arguably the toughest and most time-consuming question in terms of applying logic and knowledge from previous queries. The challenge faced at first was to group the teams according to them being home and away in a single attempt. The early attempt only succeeded in finding points gained from home games and away games separately. A different attempt was made to include the home or away teams in an array according to who won the game and both if the game was a draw using $switch statement referring to the MongoDB documentation ($switch — MongoDB Manual, 2020). The point variable stores 3 if the game is a win and 1 if the game is a draw. It was assumed that winning gives a team 3 points, 1 point for draw and no points for a loss. The $unwind aggregation operator is used to deconstruct the array ($unwind — MongoDB Manual, 2020). In the $group pipeline, the points are summed and grouped according to "Team" where the array held winning, losing or drawing teams. This results in allocating and adding the points for all the teams involved in the EPL. The results are sorted in descending order to display the highest ranked team first.

Answer: The results in the screenshot for all the 20 teams in descending order.

Code:

```
epl.aggregate([{
$project:
  {
        _id: 0,
        Team: {
                $switch:
                {
                  branches: [
                        {
                          case: { $eq : [ '$FTR', 'H' ] },
                          then: ['$HomeTeam']
                        },
                        {
                          case: { $eq : [ '$FTR', 'A' ] },
                          then: ['$AwayTeam']
                        },
                        {
                          case: { $eq : [ '$FTR', 'D' ] },
                          then: ['$HomeTeam', '$AwayTeam']
                        }
                  ],
                  default: 0
                }
        },
          points: {$cond: [{$eq: ['$FTR', 'D']}, 1, 3]}
  }
},{"$unwind": "$Team"},
{$group: {
                    _id: "$Team",
                    total_points_by_team: {$sum: '$points'},
          }
},
{ $sort: { total_points_by_team: -1 } }
])
```



```
{ "_id" : "Man City", "total_points_by_team" : 98 }
{ "_id" : "Liverpool", "total_points_by_team" : 97 }
{ "_id" : "Chelsea", "total_points_by_team" : 72 }
{ "_id" : "Tottenham", "total_points_by_team" : 71 }
{ "_id" : "Arsenal", "total_points_by_team" : 70 }
{ "_id" : "Man United", "total_points_by_team" : 66 }
{ "_id" : "Wolves", "total_points_by_team" : 57 }
{ "_id" : "Everton", "total_points_by_team" : 54 }
{ "_id" : "Leicester", "total_points_by_team" : 52 }
{ "_id" : "West Ham", "total_points_by_team" : 52 }
{ "_id" : "Watford", "total_points_by_team" : 50 }
{ "_id" : "Crystal Palace", "total_points_by_team" : 49 }
{ "_id" : "Bournemouth", "total_points_by_team" : 45 }
{ "_id" : "Newcastle", "total_points_by_team" : 45 }
{ "_id" : "Burnley", "total_points_by_team" : 40 }
{ "_id" : "Southampton", "total_points_by_team" : 39 }
{ "_id" : "Brighton", "total_points_by_team" : 36 }
{ "_id" : "Cardiff", "total_points_by_team" : 34 }
{ "_id" : "Fulham", "total_points_by_team" : 26 }
{ "_id" : "Huddersfield", "total_points_by_team" : 16 }
```

Figure 8 - All teams and their points in descending order

Conclusions:

Querying the EPL dataset was quite challenging as the answers were somewhat different to what was practiced in the lectures. The aggregation pipeline and operators, which were new at the start, were later used effectively, answering all the questions. These were gradually easy to use. It was discovered that a lot of other MongoDB features are yet to be explored and that MongoDB is very powerful yet fun to use.

The answers were also cross-checked and verified by loading the dataset to Neo4j after converting the JSON file to CSV and running cypher commands.

APPENDIX – REFERENCES

JSON Array – Stack Overflow

daydreamer (2019). [Stack Overflow] *Mongodb Atlas Mongoimport Issues Cannot Decode Array Into A D*. [online]. Available at: https://stackoverflow.com/questions/58150528/mongodb-atlas-mongoimport-issues-cannot-decode-array-into-a-d [Accessed 31 March 2020].

$addToSet – MongoDB Documentation

Docs.mongodb.com (2020). *$addToSet — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/update/addToSet/ [Accessed 31 March 2020].

$dateFromString – MongoDB Documentation

Docs.mongodb.com (2020). *$dateFromString — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/aggregation/dateFromString/ [Accessed 31 March 2020].

$dayOfWeek – MongoDB Documentation

Docs.mongodb.com. (2020). *$dayOfWeek — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/aggregation/dayOfWeek/ [Accessed 31 March 2020].

Aggregation Pipeline – MongoDB Documentation

Docs.mongodb.com. (2020). *Aggregation Pipeline Stages — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/ [Accessed 31 March 2020].

$switch – MongoDB Documentation

Docs.mongodb.com. (2020). *$switch — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/aggregation/switch/ [Accessed 31 March 2020].

$unwind – MongoDB Documentation

Docs.mongodb.com. (2020). *$unwind — Mongodb Manual*. [online] Available at: https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/ [Accessed 31 March 2020].

PART III - ESSAY

Bigdata Analytics & the Role of NoSQL Databases

Diwas Lamsal

***Abstract* – Data has evolved a lot in the current world. In this paper, a trending topic "Big Data" is discussed followed by discussions of NoSQL databases and their role in Big Data Analytics. Detailed study of their history, characteristics, types and differences from Relational Databases is presented. The paper concludes that NoSQL databases play a vital role in Big Data Analytics.**

Background

The world is full of data. Every minute, hundreds of thousands of photos are shared among Snapchat users, similar number of tweets are tweeted on Twitter, tens of thousands of pictures are posted on Instagram and a lot more of data is generated from other sources (Lackey, 2019). The pace at which we are generating data is only increasing and we should not expect it to decelerate any time soon. As of now, it is not just us who generate this enormous amount of data. With the rise of technology, IoT devices and Artificial Intelligence, the machines themselves are producing so much data. We have come to a point where data has come close to being one of our basic needs. The computerized world cannot exist without data. With so much data around, a new topic is trending amongst people in the tech industry.

Big Data

Fig 1 – Big Data (Mihajlovic, 2019)

"Big Data" is, as one would guess, huge data. Guru99 defines Big Data as "a term used to describe a collection of data that is huge in volume and yet growing exponentially with time" (Guru99, 2020). Like we talked about earlier, this is the type of data the world is dealing with right now. Big Data has a lot of history in itself. Oracle talks about how after 2005 people started to realize about generation of large amount of data and Hadoop and NoSQL databases started to gain popularity at that time (Oracle, 2020). It is very likely that with so much data, there is something meaningful within it. This thought has motivated people into "Big Data Analytics".

Big Data Analytics

Analyzing Big Data involves usage of advanced analytical techniques to gain insights from large datasets. The data could be structured, semi-structured, or unstructured (IBM, 2020). Big Data Analytics has proved very promising as it has set great drifts in the field of Business Intelligence. Benefits of Big Data Analytics lie in a lot of areas including marketing, learning sales opportunities, business planning and forecasting, risk detection, etc. (Russom, 2011). Some papers also discuss the potential of Big Data Analytics as applied to healthcare industries (Wang, et. al., 2018; Raghupati, et. al, 2014). There are several other uses and potentials of Big Data Analytics.

The take here is that Big Data Analytics has evolved very much since its rise and is currently a big topic in the technical industry. It is quite evident that traditional Relational Databases cannot be used to effectively handle Big Data as relational methods are ineffective on handling large scale datasets which could comprise of diverse configurations. This has given rise to another database term for Big Data Analytics called NoSQL databases (Zafar et. al., 2016). In this essay, we will talk about how these NoSQL databases play a big role in Big Data Analytics and also talk about their characteristics and types and benefits over relational databases while doing so.

NoSQL Databases

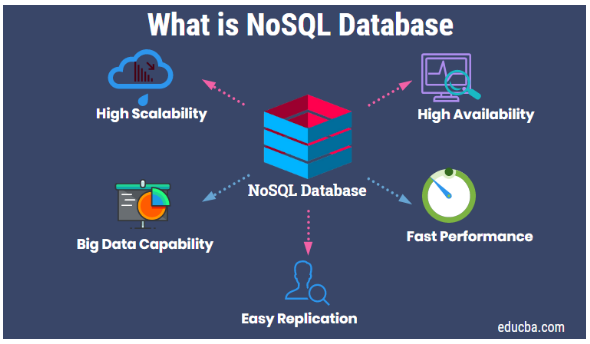


Fig 2 – What is NoSQL Database (EDUCBA, 2020)

NoSQL databases, or Not only SQL databases are called so because they do not inherit properties of traditional Relational systems and the SQL. These are generally used while working with larger datasets and where relational properties are not necessary. These databases are not primarily built on tables (Moniruzzaman et al., 2013) and focus on horizontal scaling. There could be beliefs that NoSQL is aimed to replace relational databases, however, that is not the case as Not only SQL aims to coexist with SQL and the use of these databases depends on context of their application (Bokhari & Khan, 2016).

The term NoSQL was first introduced by Carlo Strozzi when he created his database - “Strozzi NoSQL Open Source Relational Database” which was a relational database without an SQL interface (Lacefield, 2018). It started to get popular later in 2009. Eric Evans, who was a blogger and a Rackspace employee, described NoSQL movement’s goal as "the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for" (Bokhari & Khan, 2016). It was reported that NoSQL had the ability to write 2500 times faster than MySQL in a 50 GB database at that time (Lakhsman & Malik, 2009).

The increasing amount of data made relational databases less reliable for Big Data as they have poor horizontal scalability and were mainly targeted for robust data. Despite being somewhat possible for using these databases for Big Data, it could prove costly and require newer methods to tackle the unstructured datasets (Bokhari & Khan, 2016). It is seen that many organizations that deal with larger datasets and unstructured data are turning towards NoSQL databases (Leavitt, 2010). The NoSQL databases, with their decentralized environment, and parallel processing, provide many benefits over the relational databases and are the go-to for Big Data. Some popular NoSQL databases include MongoDB, Cassandra, HBase, Neo4j, Redis, etc (Mayo, 2016). While NoSQL databases are better and reliable, we should still consider the popularity of SQL databases for handling structured datasets. According to ScaleGrid, as of March 2019, 60.5% of the world's databases comprised of SQL databases while SQL still held rising demands. Many organizations were not migrating to NoSQL (ScaleGrid, 2019).

NoSQL Databases – What are their characteristics?

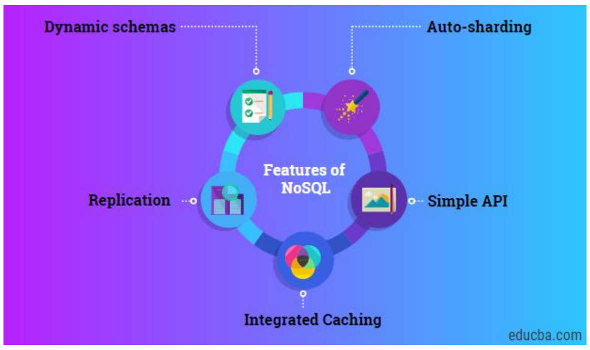


Fig 3 – Features of NoSQL Database (EDUCBA, 2020)

The NoSQL Databases have many characteristics and features that separate them from traditional RDBs. Some of the most prominent ones will be discussed here. The first property that comes to mind while talking about NoSQL databases is perhaps them being non-relational and schema-less. As it is the first thing that separates them from relational databases, the NoSQL databases do not primarily deal with tables.

It is known that relational databases are designed to follow ACID (Atomicity, Consistency, Isolation and Durability) properties. The purpose of this was to ensure Consistency, Availability and Partition Tolerance. However, according to the CAP theorem, only two of these three aspects can be achieved at once in a distributed system (Narzul, 2018). Most of the NoSQL databases have considered less priority for consistency to provide good Availability and Partition Tolerance, which has led to systems being BASE (Basically Available, Soft-state, Eventually consistent) (Moniruzzaman et al., 2013).

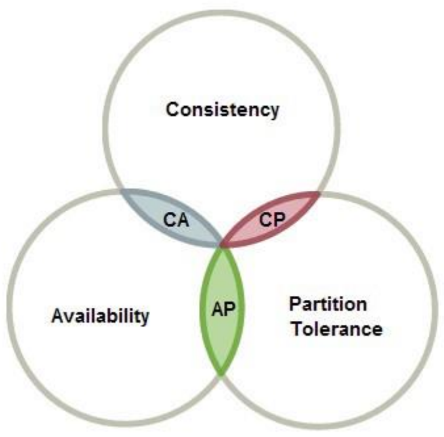
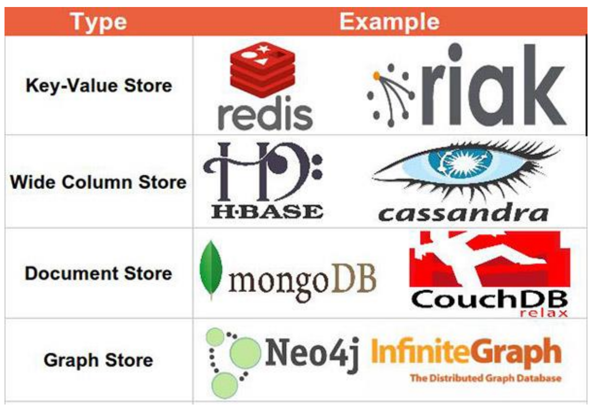


Fig 4 – CAP Theorem (Narzul, 2018).

As mentioned before, horizontal scalability is another important factor about NoSQL databases which relational databases fail to provide. As data can be stored across multiple devices, failure of one device would not affect data availability. Similarly, it would also allow partitioning and parallel processing, which is beneficial while dealing with larger datasets (Big Data). Moreover, the NoSQL databases are cheaper and easier to implement as most of these are open source. With the lack of SQL, these languages have more programming and are easier to get into with past programming experience. Some NoSQL databases such as MongoDB allow direct use of programming languages such as JavaScript (scripts for mongo shell - MongoDB Manual, 2020). Some of these have support for external libraries and can be used very effectively. The lack of SQL can also be made up for by using external tools such as XQuery (Zafar et al., 2016). On the other hand, relational features such as JOIN commands cannot be used in NoSQL databases as they would require a lot of computational power for large datasets. Instead of tables, NoSQL databases deal with different types of applications for primary data exchange.

NoSQL Databases – Types



Type	Example	
Key-Value Store	redis	riak
Wide Column Store	HBASE	cassandra
Document Store	mongoDB	CouchDB relax
Graph Store	Neo4j	InfiniteGraph The Distributed Graph Database

Fig 5 – NoSQL Database Types (TipsMake, 2019).

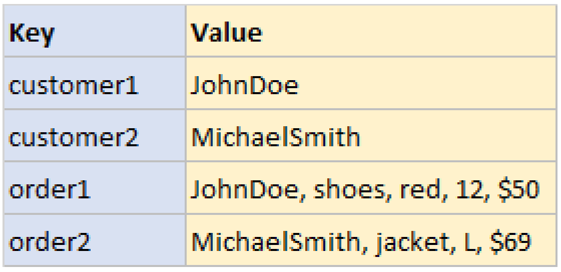
The NoSQL Databases have generally been classified into four types:

Key-value:

The main focus of key-value databases to scale horizontally for large amount of data. These databases tend to provide high reliability and availability over consistency.

These databases have been commonly found getting used for online games and web applications (Zafar et al., 2016). The data model consists of key and value pairs. The keys are usually alphanumeric IDs whereas the values could range from integers, strings to complex sets (Moniruzzaman et al., 2013). While these databases are fast, scalable and provide good fault-tolerance, they cannot model objects.

Examples: Redis, Dynamo, etc.



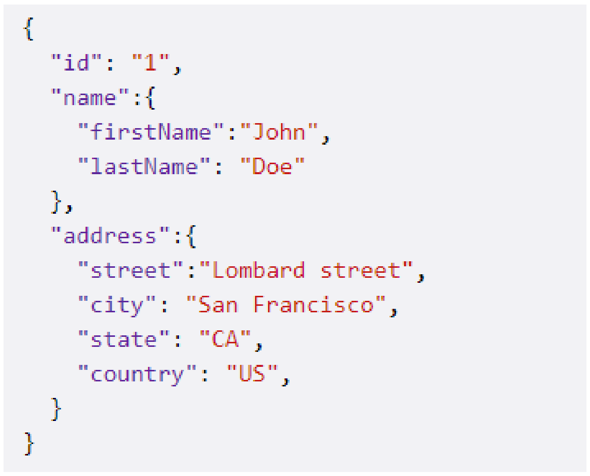
Key	Value
customer1	JohnDoe
customer2	MichaelSmith
order1	JohnDoe, shoes, red, 12, $50
order2	MichaelSmith, jacket, L, $69

Fig 6 – Key-Value example (Kononow, 2018)

Document-based:

This kind of database stores data in the form of documents, which typically contain key-value pairs while also supporting objects. The basic key-value concepts are extended to store complex type of data. These document stores usually do not comprise of schema and could be accessed through unique keys. These databases provide higher scalability, performance and flexibility (Venkatraman et al., 2016). One major weakness of key-value pairs of usually only being able to search against keys is made up in document-based databases by allowing both keys as well as values to be fully searched (Moniruzzaman et al., 2013).

Examples: MongoDB, CouchDB, etc.



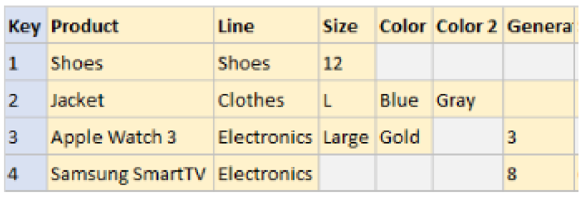
```
{
  "id": "1",
  "name":{
    "firstName":"John",
    "lastName": "Doe"
  },
  "address":{
    "street":"Lombard street",
    "city": "San Francisco",
    "state": "CA",
    "country": "US",
  }
}
```

Fig 7 – Document-Based example (Kononow, 2018)

Column-based:

These databases are also called Wide-column databases and are ideally used for data mining and Big Data Analytics (Venkatraman et al., 2016). These databases store the data in tables but are column-based instead of row-based as in relational databases. The data model comprises of column families and row keys (additionally timestamps). They are great for distributed data and versioning as they support timestamping features (Zafar et al., 2016). Different versions of values within a cell are separated by timestamps. Other features of column-based databases include large-scale operations such as sorting, converting, parsing and algorithmic crunching (Moniruzzaman et al., 2013).

Examples: BigTable, Cassandra, etc.



Key	Product	Line	Size	Color	Color 2	Genera
1	Shoes	Shoes	12			
2	Jacket	Clothes	L	Blue	Gray	
3	Apple Watch 3	Electronics	Large	Gold		3
4	Samsung SmartTV	Electronics				8

Fig 8 – Column-Based example (Kononow, 2018)

Graph-based:

Graph databases focus more on connections between data. These are inspired by Graph Theory and are appropriate for data which can be labelled suitably using nodes, relationships and properties (where relationships are important than data itself). The data model comprises of nodes with labels and properties, edges and key-value pairs for both of these. These databases are very suitable for social-networking systems such as Twitter. These are arguably the most user-friendly due to visual representation of the data model (Moniruzzaman et al., 2013).

Examples: Neo4j, GraphDB, etc.

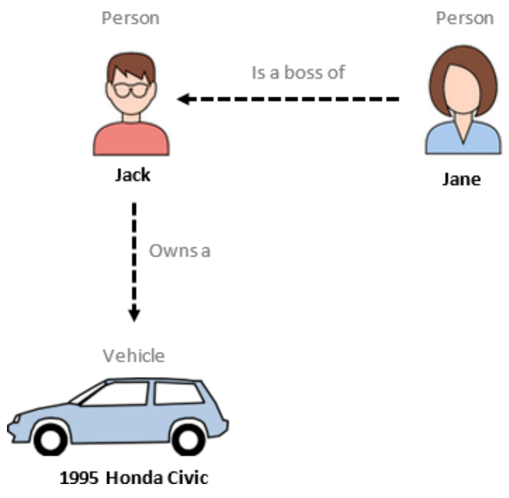


Fig 9 – Graph-Based example (Kononow, 2018)

NoSQL Databases & Big Data

As Big Data has grown so much, there is now the need for horizontally scalable databases. Due to the same reason, NoSQL databases in general have gained a lot of popularity (Gudivada, 2014). Relational databases being inadequate in handling huge amount of data, these NoSQL databases have evidentially shown that they can be very effective for Big Data Analytics. Many companies have converted to NoSQL databases and gained huge boost in performance (Zafar et. al., 2016; Gudivada, 2014). Where the main focus is about reliability, fault tolerance and readiness, the NoSQL databases clearly win out.

In a nutshell, we talked about the three key topics in Big Data, Analytics and NoSQL databases. We talked about how the world is diverting towards Big Data Analytics and how NoSQL databases play a major role in managing very large datasets such as the Big Data itself. We looked at the key characteristics and four types of NoSQL databases available while doing so. Overall, from this research and literature, it can be concluded that the previous generation of relational databases as well as hardware cannot handle the rapid growth of complex data. The new and improvised, horizontally scalable and decentralized NoSQL databases, that have many other benefits play a major role in the field of Big Data Analytics.

References

1. Lackey, D. (2019) *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read.* [online]. blazon.online. Available at: https://blazon.online/data-marketing/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/ [Accessed 16 April 2020]

2. Guru99. (2020) *What is BIG DATA? Introduction, Types, Characteristics & Example* [online]. Guru99. Available at: https://www.guru99.com/what-is-big-data.html [Accessed 16 April 2020]

3. Mihajlovic, I. (2019) *What is Big Data? Let's answer this question!* [online] Towards Data Science. Available at: https://towardsdatascience.com/what-is-big-data-lets-answer-this-question-933b94709caf [Accessed 16 April 2020]

4. Oracle. (2020) *What Is Big Data?* [online]. Oracle. Available at: https://www.oracle.com/big-data/guide/what-is-big-data.html [Accessed 16 April 2020]

5. IBM. (2020) *Big Data Analyics | IBM* [online]. IBM. Available at: https://www.ibm.com/analytics/hadoop/big-data-analytics [Accessed 16 April 2020]

6. Russom, P. (2011) *BIG DATA ANALYTICS*. 1201 Monster Road SW Suite 250, Renton, WA: TDWI Research

7. Wang, Y., Kung, L., & Byrd, T. A. (2018). *Big data analytics: Understanding its capabilities and potential benefits for healthcare organizations*. Technological Forecasting and Social Change, vol. **126**, pp.3–13. Available from DOI: 10.1016/j.techfore.2015.12.019 [Accessed 16 April 2020]

8. Raghupathi, W., & Raghupathi, V. (2014). *Big data analytics in healthcare: promise and potential*. Health Information Science and Systems, vol. **2**(1). Available from DOI: 10.1186/2047-2501-2-3 [Accessed 16 April 2020]

9. Zafar, R., Yafi, E., Zuhairi, M. F., & Dao, H. (2016). *Big Data: The NoSQL and RDBMS review*. 2016 International Conference on Information and Communication Technology (ICICTM). pp. 120-126. Available from DOI :10.1109/icictm.2016.7890788 [Accessed 16 April 2020]

10. EDUCBA. (2020) *What is NoSQL Database* [online]. EDUCBA. Available from: https://www.educba.com/what-is-nosql-database/ [Accessed 16 April 2020]

11. Lacefield, J. (2018) *The Evolution of NoSQL* [online]. DATASTAX. Available from: https://www.datastax.com/blog/2018/08/evolution-nosql [Accessed 17 April 2020]

12. Moniruzzaman, A.B.M, Hossain, S.A. (2013) *NoSQL Database: New Era of Databases for Big data Analytics – Classification, Characteristics and Comparison*. International Journal of Database Theory and Application, vol. **6**(4).

13. Bokhari, M.U., Khan, A. (2016) *The NoSQL Movement.* IOSR Journal of Computer Engineering (IOSR-JCE). vol. **18**(6), pp. 6-12. Available from DOI: 10.9790/0661-1806040612 [Accessed 17 April 2020]

14. Lakshman, A., Malik, P. (2009) Cassandra-Structured Storage System over a P2P Network.

Presentation at NoSQL meet-up in San Francisco on 2009-06-11. Available from: http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf [Accessed: 17 April 2020]

15. Leavitt, N. (2010). *Will NoSQL databases live up to their promise?* Computer. vol. **43**(2), pp. 12-14. Available from DOI: 10.1109/MC.2010.58 [Accessed 17 April 2020]

16. Mayo, M. (2016) op NoSQL Database Engines. [online] KDnuggets. Available from: https://www.kdnuggets.com/2016/06/top-nosql-database-engines.html [Accessed 17 April 2020]

17. ScaleGrid. (2019) 2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use [online] ScaleGrid. Available from: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/ [Accessed 17 April 2020]

18. Narzul, S.S. (2018) CAP Theorem and Distributed Database Management Systems [online] Towards Data Science. Available at: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e [Accessed 18 April 2020]

19. MongoDB Manual (2020) *Write Scripts for the mongo Shell* [online] MongoDB Manual – docs.mongodb.com Available at: https://docs.mongodb.com/manual/tutorial/write-scripts-for-the-mongo-shell/ [Accessed 18 April 2020]

20. TipsMake, (2019) Learn about Non-relational Database – NoSQL [image] TipsMake. Available from: https://tipsmake.com/learn-about-nonrelational-database-nosql [Accessed 18 April 2020]

21. Kononow, P. (2018) *What is NoSQL (different types)* [image] Dataedo. Available from: https://dataedo.com/kb/data-glossary/what-is-nosql [Accessed 18 April 2020]

22. Venkatraman, S., Fahd, K., Kaspi, S., Venkatraman, P. (2016) *SQL Versus NoSQL Movement with Big Data Analytics*. I.J. Information Technology and Computer Science, 2016, vol. **12**, pp. 59-66, Available from DOI: 10.5815/ijitcs.2016.12.07 [Accessed 18 April 2020]

23. Gudivada, V. N., Rao, D., & Raghavan, V. V. (2014). *NoSQL Systems for Big Data Management*. 2014 IEEE World Congress on Services. Available from DOI:10.1109/services.2014.42 [Accessed 18 April 2020]

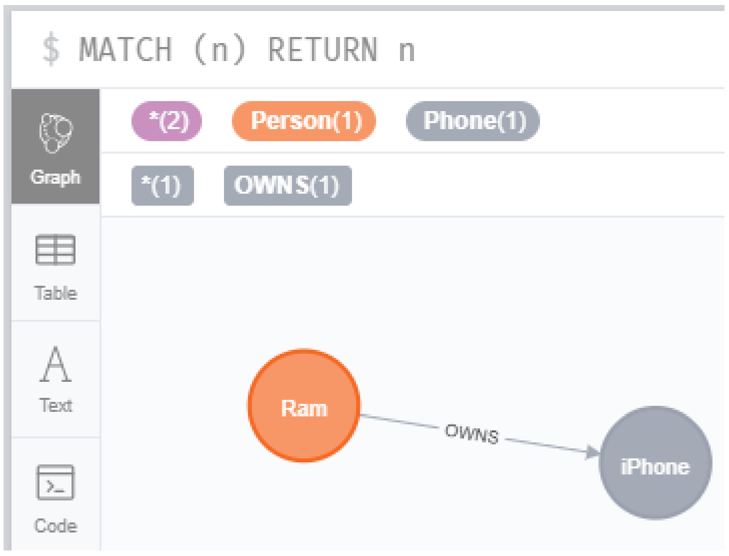
APPENDIX – PART I - LAB EXERCISES & LEARNING DIARY

CYPHER 7

This chapter talked about some new as well as previously used cypher commands. Most of the commands such as RETURN, ORDER BY, LIMIT, etc. were used frequently previously and were familiar.

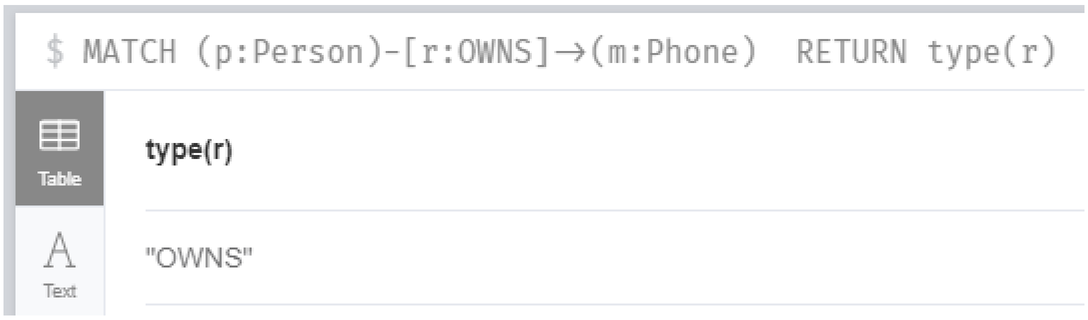
RETURN clause is used to display the output from a query. It can be used to return full nodes, relationships, properties or every element.

MATCH (nodes) RETURN nodes
Will return all the nodes in the database.


```
$ MATCH (n) RETURN n
```

MATCH (p:Person)-[r:OWNS]->(m:Phone)
RETURN type(r)

This should return the type of relationship “r” is.

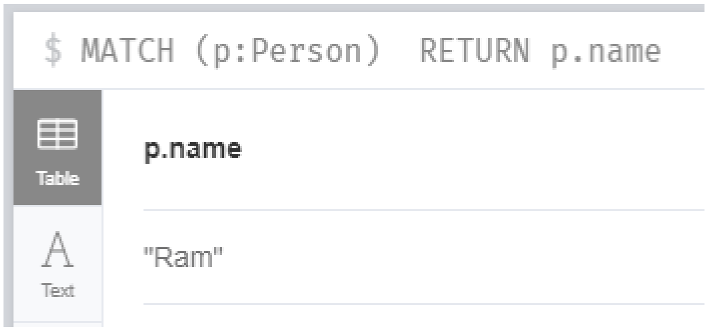

```
$ MATCH (p:Person)-[r:OWNS]→(m:Phone)  RETURN type(r)
```

type(r)
"OWNS"

MATCH (p:Person)

RETURN p.name

This should display the value in “name” property of all the Person nodes in the database.

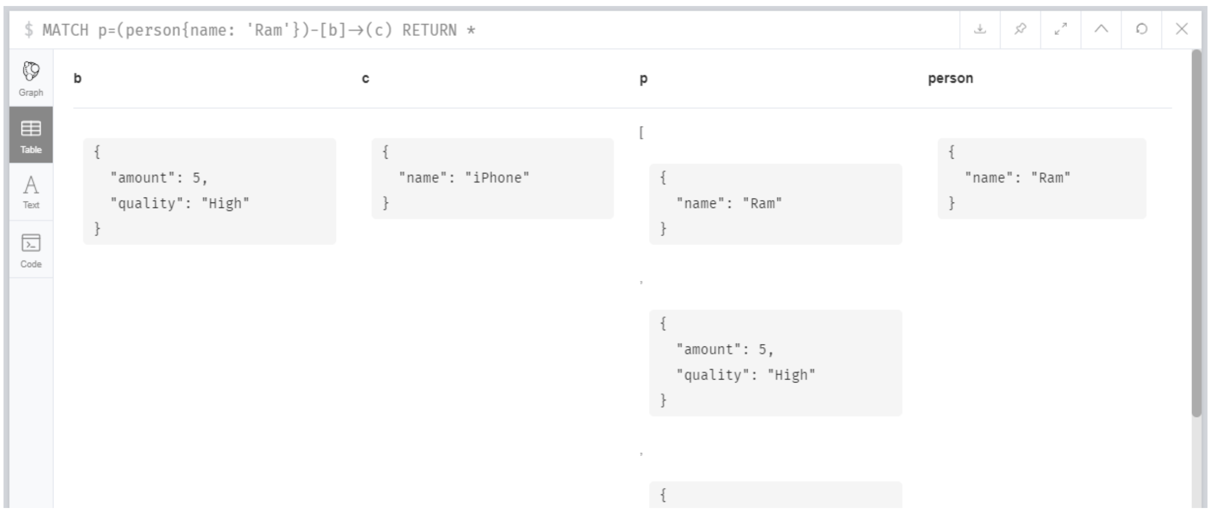

```
$ MATCH (p:Person)  RETURN p.name
```

p.name
"Ram"

MATCH p=(person{name: 'Ram'})-[b]->(c)

RETURN *

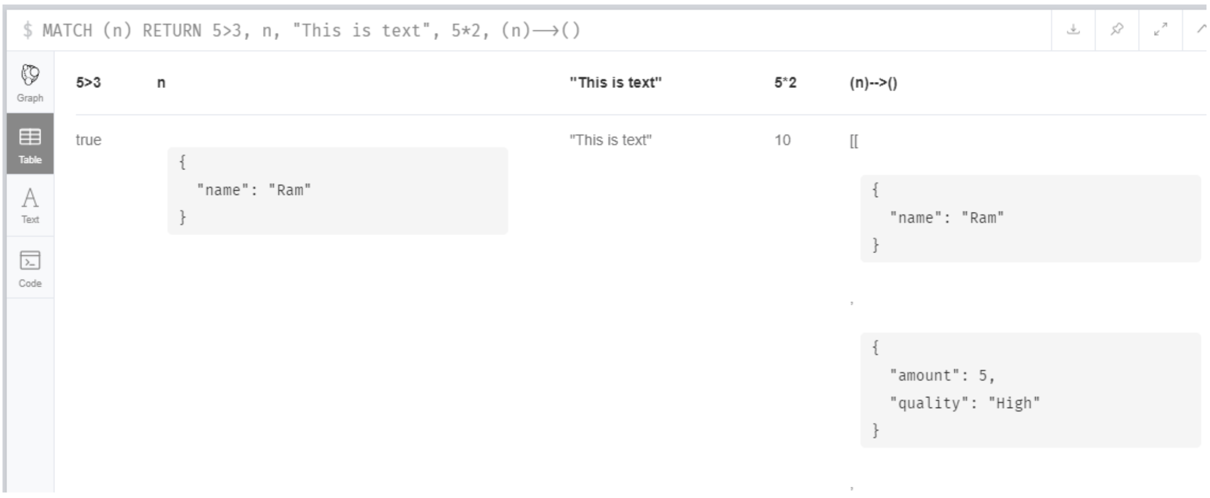
All the properties are returned and listed in columns.


```
$ MATCH p=(person{name: 'Ram'})-[b]→(c) RETURN *
```

b	c	p	person
{ "amount": 5, "quality": "High" }	{ "name": "iPhone" }	[{ "name": "Ram" } , { "amount": 5, "quality": "High" } , {	{ "name": "Ram" }

```
MATCH (n)
RETURN 5>3, n, "This is text", 5*2, (n)-->()
```

RETURN can be used to display functions, calculations, texts (string literals), predicates and mostly anything.

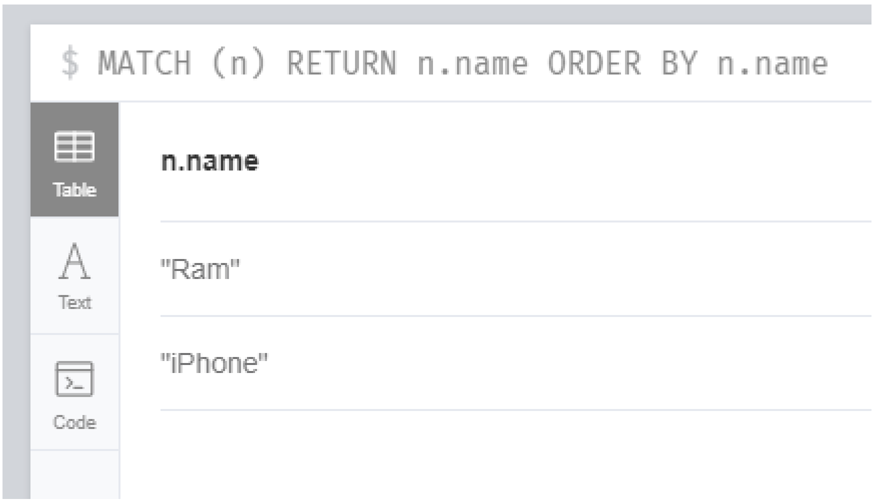


```
$ MATCH (n) RETURN 5>3, n, "This is text", 5*2, (n)⟶()
```

5>3	n	"This is text"	5*2	(n)-->()
true	{ "name": "Ram" }	"This is text"	10	[[{ "name": "Ram" } , { "amount": 5, "quality": "High" } ,

ORDER BY is another very commonly used clause which is used in conjunction with RETURN or WITH clause. This is basically used to sort the output. The sorting can only be done according to properties and not nodes or relationship. It was found that NULL results are displayed at last while sorting properties in ascending order.

```
MATCH (n)
RETURN n.name
ORDER BY n.name
```



```
$ MATCH (n) RETURN n.name ORDER BY n.name
```

n.name
"Ram"
"iPhone"

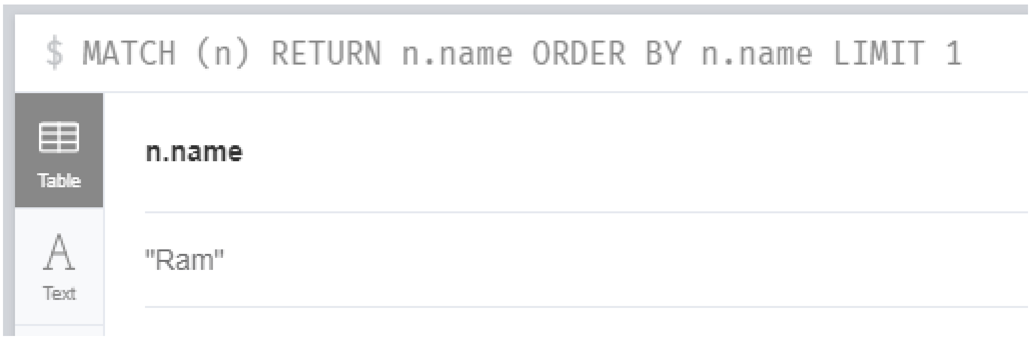
LIMIT is a constraint set on the number of outputs.

MATCH (n)

RETURN n.name

ORDER BY n.name

LIMIT 1



```
$ MATCH (n) RETURN n.name ORDER BY n.name LIMIT 1
```

n.name
"Ram"

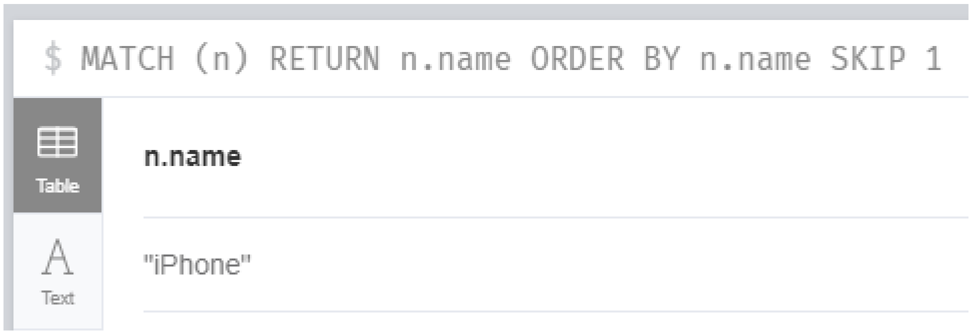
SKIP can be used to skip the required number of outputs.

MATCH (n)

RETURN n.name

ORDER BY n.name

SKIP 1

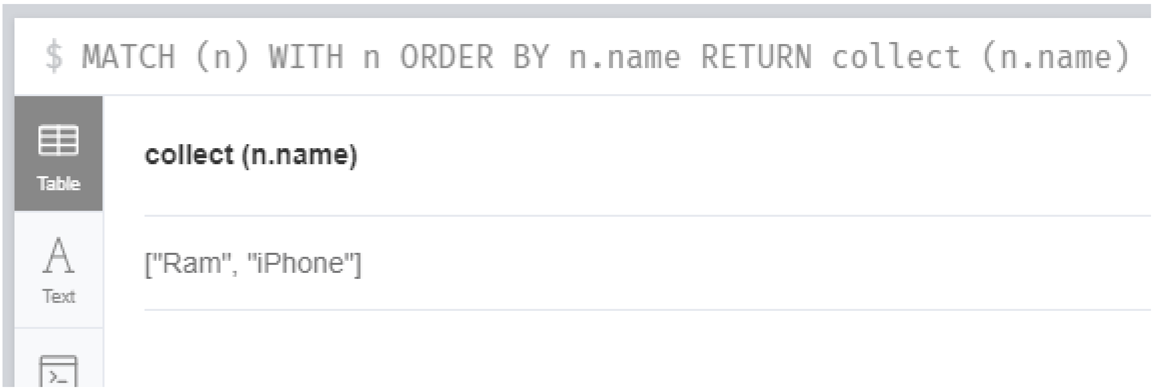


```
$ MATCH (n) RETURN n.name ORDER BY n.name SKIP 1
```

n.name
"iPhone"

WITH was a very useful clause for Assignment I and has been used extensively. It is used to send results from first operation to the second and so on (pipelining).

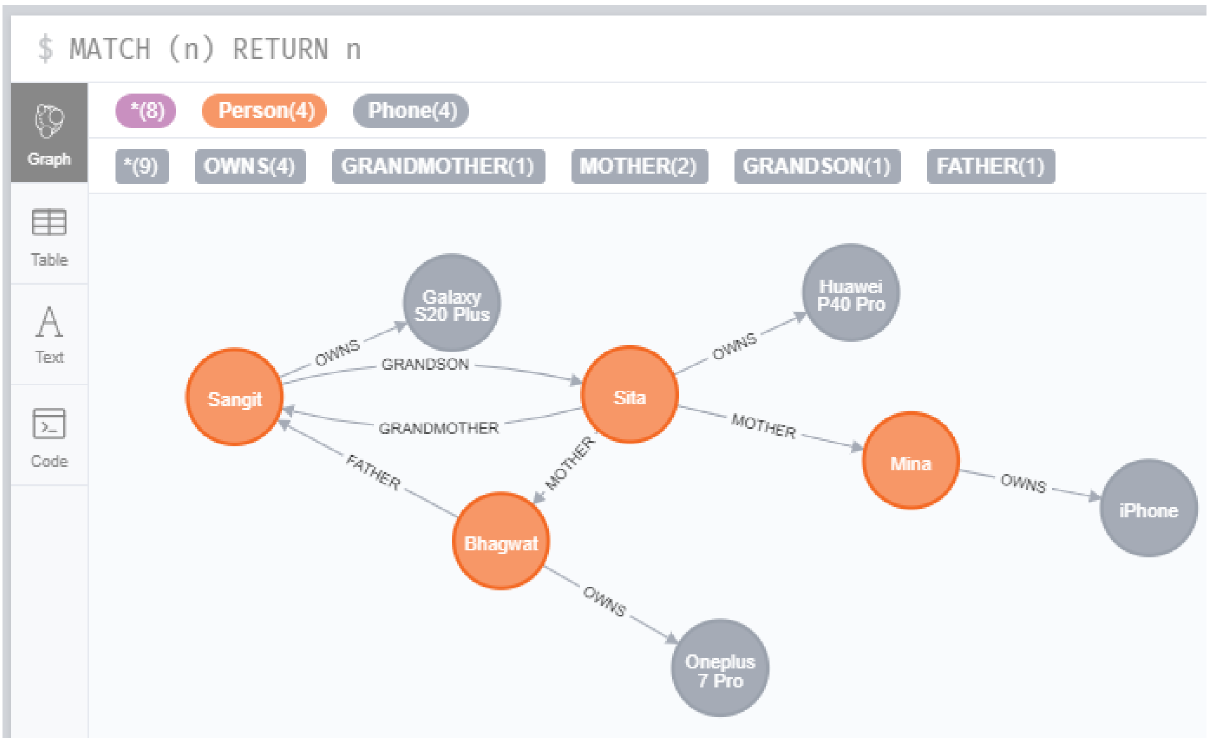
We can use WITH to perform sort in order to store them in a collection with the use of COLLECT.



```
$ MATCH (n) WITH n ORDER BY n.name RETURN collect (n.name)
```

collect (n.name)
["Ram", "iPhone"]

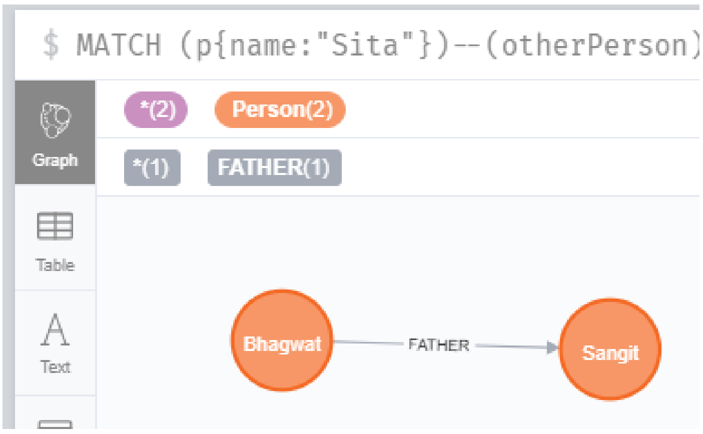
A new database is created to practice the WITH clause.



```
$ MATCH (n) RETURN n
```

The following query returns the relationship where “Sita” has a relationship and the person/phone she has a relationship with further has another relationship with something/someone else. The second relationship is the result of this query.

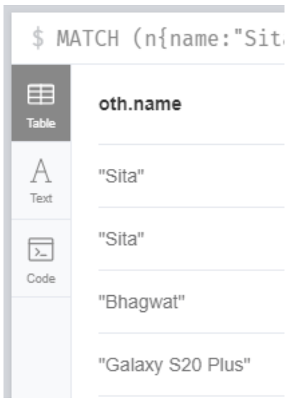
```
MATCH (p{name:"Sita"})--(otherPerson)-->()
WITH otherPerson, COUNT(*) AS roar
WHERE roar>1
RETURN otherPerson
```



```
$ MATCH (p{name:"Sita"})--(otherPerson)
```

Similarly, we can use WITH for limiting the branching of a query result.

```
MATCH (n{name:"Sita"})--(p)
WITH p
ORDER BY DESC LIMIT 1
MATCH (p)--(oth)
RETURN oth.name
```

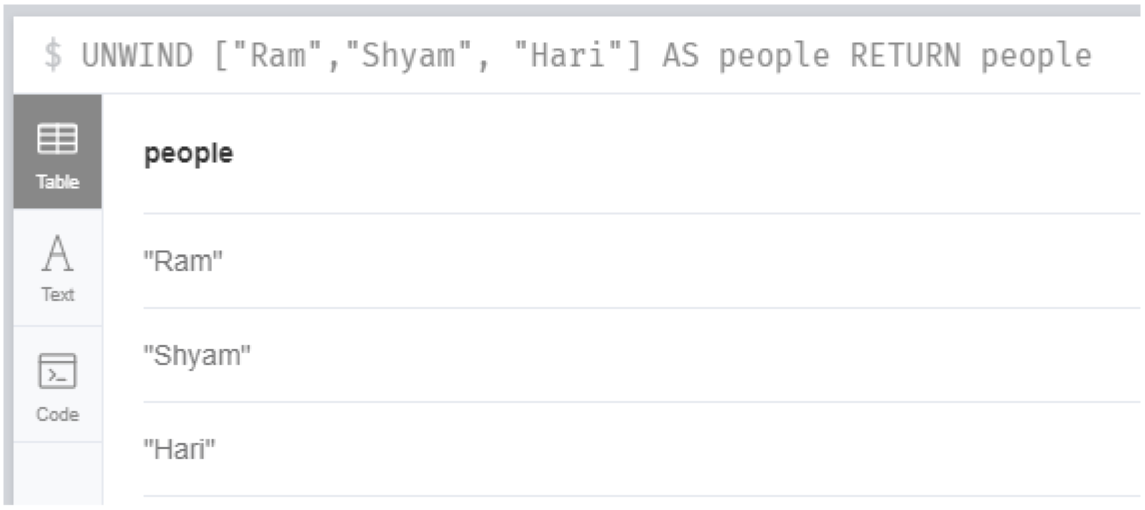


```
$ MATCH (n{name:"Sit
```

oth.name
"Sita"
"Sita"
"Bhagwat"
"Galaxy S20 Plus"

UNWIND can be used to open or deconstruct a list(array) into a row. It is mentioned that a new variable name is needed to use the elements in the list.

UNWIND ["Ram","Shyam", "Hari"] AS people
RETURN people

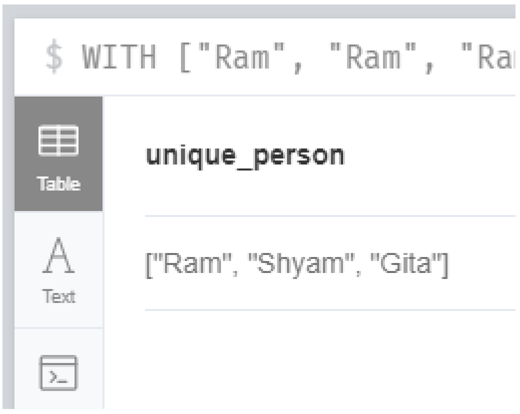


```
$ UNWIND ["Ram","Shyam", "Hari"] AS people RETURN people
```

people
"Ram"
"Shyam"
"Hari"

A LIST can be made DISTINCT by first unwinding it and then passing it through WITH DISTINCT clause.

WITH ["Ram", "Ram", "Ram", "Shyam", "Gita", "Gita"] AS people
UNWIND people AS person
WITH DISTINCT person
RETURN collect(person) AS unique_person



```
$ WITH ["Ram", "Ram", "Ra
```

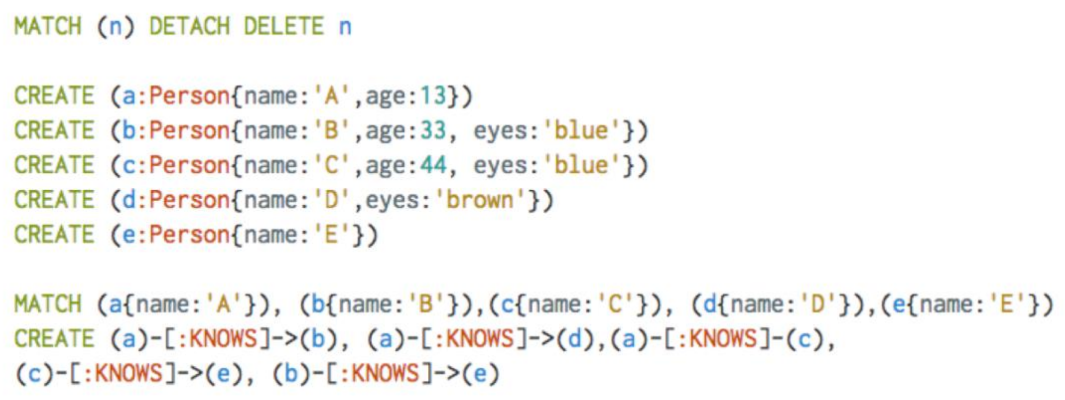
unique_person
["Ram", "Shyam", "Gita"]

Chapter Summary:

The chapter Cypher 7 basically taught about more cypher commands. All the commands provided in the lecture slides were replicated and practiced, making extra attempts on a different dataset than provided. Basically, from the chapter, all the commands such as RETURN, ORDER BY, LIMIT, SKIP, WITH and UNWIND were learned further as most of these commands were already used in the Term I assignment by referring to this chapter and online documentation.

CYPHER 8

The beginning of Cypher 8 talks about CASE expression which is equivalent to IF ELSE statements in PL-SQL or most programming languages. The slides provide code to prepare a database for querying using the CASE statement.



```
MATCH (n) DETACH DELETE n

CREATE (a:Person{name:'A',age:13})
CREATE (b:Person{name:'B',age:33, eyes:'blue'})
CREATE (c:Person{name:'C',age:44, eyes:'blue'})
CREATE (d:Person{name:'D',eyes:'brown'})
CREATE (e:Person{name:'E'})

MATCH (a{name:'A'}), (b{name:'B'}),(c{name:'C'}), (d{name:'D'}),(e{name:'E'})
CREATE (a)-[:KNOWS]->(b), (a)-[:KNOWS]->(d),(a)-[:KNOWS]-(c),
(c)-[:KNOWS]->(e), (b)-[:KNOWS]->(e)
```

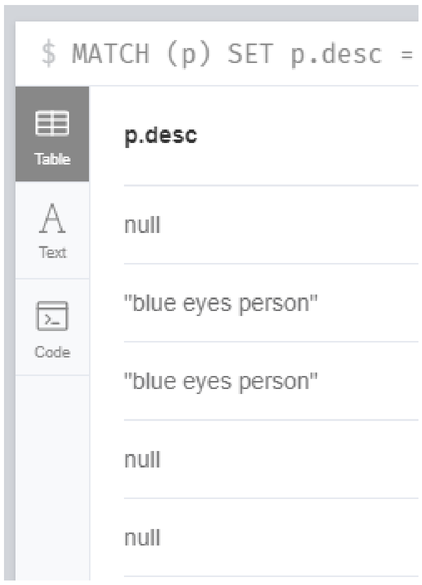
This was used to prepare a database for querying and answering the lab exercises.

Exercise

Write some Cypher commands with CASE expression to add a new attribute called ‘desc’

-- If the ’eyes’ is ‘blue’, set value of ‘desc’ to ‘blue eyes person’

```
MATCH (p)
SET p.desc = CASE
        WHEN p.eyes='blue' THEN "blue eyes person"
        END
RETURN p.desc
```

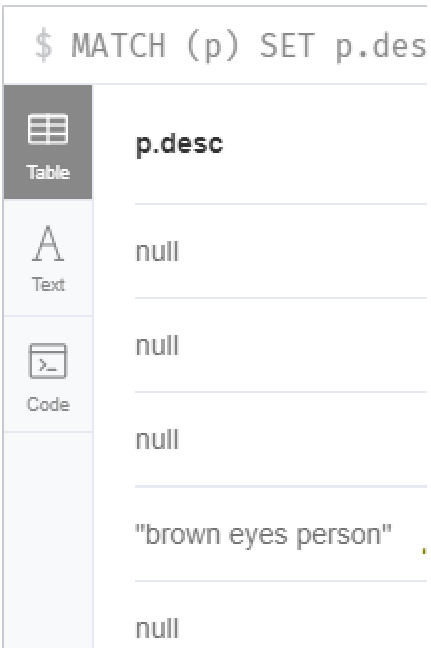


```
$ MATCH (p) SET p.desc =
```

```
p.desc
null
"blue eyes person"
"blue eyes person"
null
null
```

-- • If the 'eyes' is 'brown', set value of 'desc' to 'brown eyes person'

```
MATCH (p)
SET p.desc = CASE
        WHEN p.eyes='brown' THEN "brown eyes person"
        END
RETURN p.desc
```



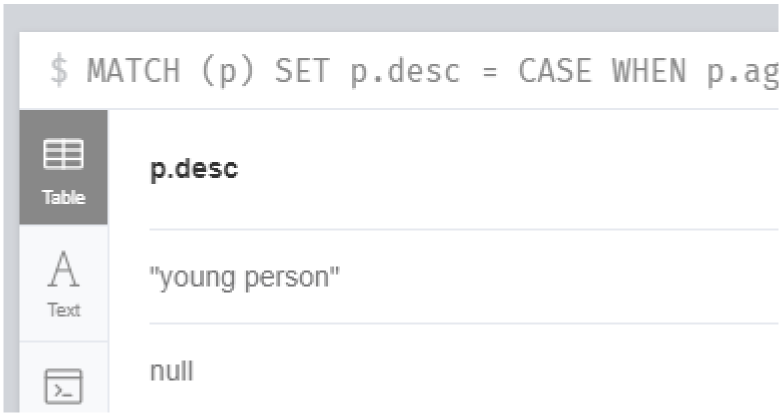
```
$ MATCH (p) SET p.des
```

```
p.desc
null
null
null
"brown eyes person"
null
```

-- If the 'age' is smaller than 30, set value of 'desc' to 'young person'

```
MATCH (p)
SET p.desc = CASE
```

```
          WHEN p.age<30 THEN "young person"
          END
RETURN p.desc
```



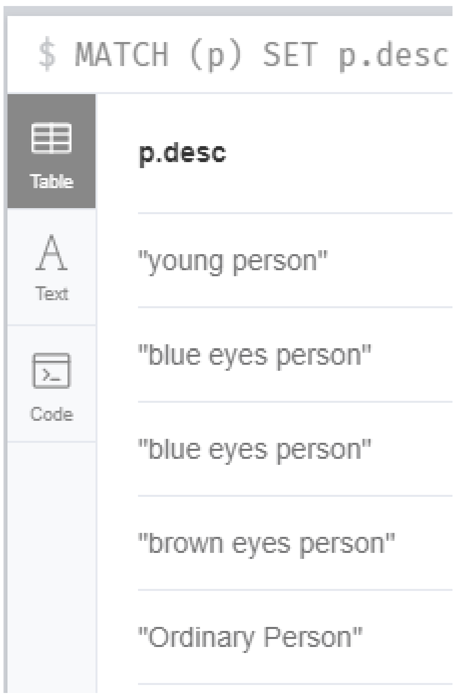
```
$ MATCH (p) SET p.desc = CASE WHEN p.ag
```

p.desc
"young person"
null

-- Otherwise set the value of ‘desc’ to ‘ordinary person’

Combining everything from above, the result will now be:

```
MATCH (p)
SET p.desc = CASE
          WHEN p.eyes='blue' THEN "blue eyes person"
   WHEN p.eyes='brown' THEN "brown eyes person"
   WHEN p.age<30 THEN "young person"
   ELSE "Ordinary Person"
          END
RETURN p.desc
```

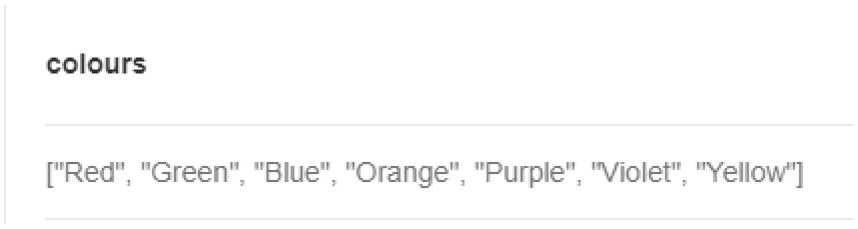


```
$ MATCH (p) SET p.desc
```

p.desc
"young person"
"blue eyes person"
"blue eyes person"
"brown eyes person"
"Ordinary Person"

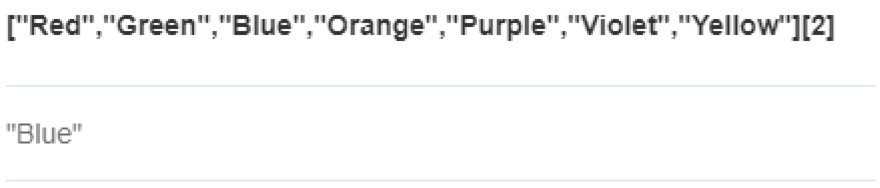
The lists were also talked about in the previous chapter while talking about UNWIND. List is quite similar to arrays in many programming languages.

RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"] AS colours



colours
["Red", "Green", "Blue", "Orange", "Purple", "Violet", "Yellow"]

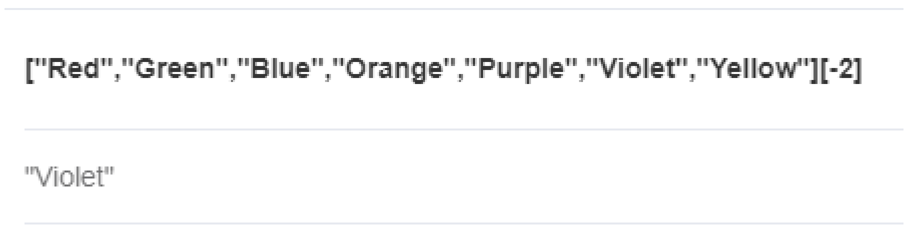
Indexes can be provided to retrieve a particular element from the list.
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][2]



["Red","Green","Blue","Orange","Purple","Violet","Yellow"][2]
"Blue"

These indexes can also have negative values to count the index from the last.

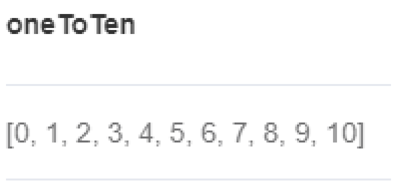
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][-2]



```
["Red","Green","Blue","Orange","Purple","Violet","Yellow"][-2]
"Violet"
```

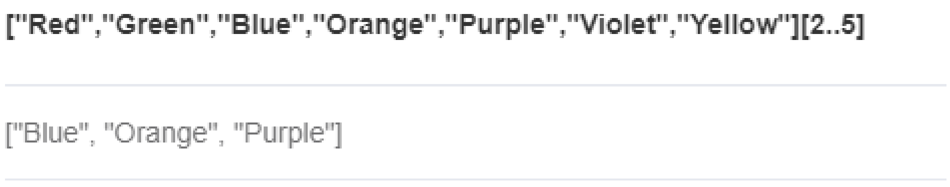
Range can be used to create a list that comprises of a range of numbers.

RETURN range(0,10) AS oneToTen



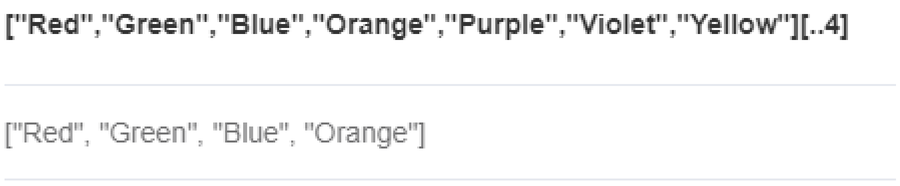
```
oneToTen
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Similarly, range can also be used as index for displaying the records from a list.
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][2..5]



```
["Red","Green","Blue","Orange","Purple","Violet","Yellow"][2..5]
["Blue", "Orange", "Purple"]
```

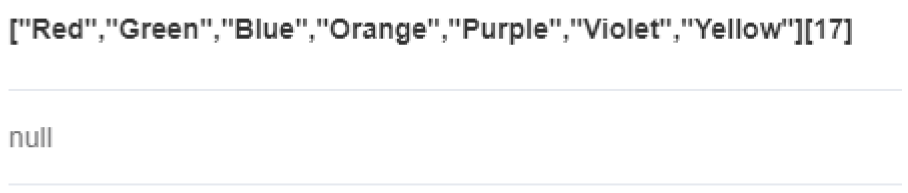
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][..4]



```
["Red","Green","Blue","Orange","Purple","Violet","Yellow"][..4]
["Red", "Green", "Blue", "Orange"]
```

If an attempt is made to retrieve an element from the list that is out of bounds (the size of the list), NULL is returned.

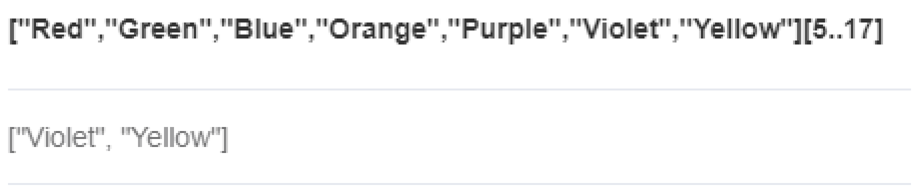
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][17]



```
["Red","Green","Blue","Orange","Purple","Violet","Yellow"][17]
null
```

If an attempt is made to retrieve a range of elements from the list which include the elements within the bounds of the list as well as outside the bounds, the null results are truncated.

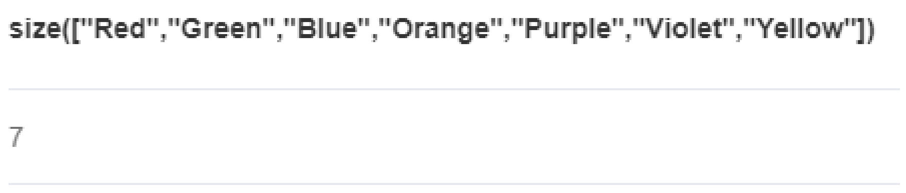
RETURN ["Red","Green","Blue","Orange","Purple","Violet","Yellow"][5..17]



```
["Red","Green","Blue","Orange","Purple","Violet","Yellow"][5..17]
["Violet", "Yellow"]
```

The function size returns the size of a list.

RETURN size(["Red","Green","Blue","Orange","Purple","Violet","Yellow"])

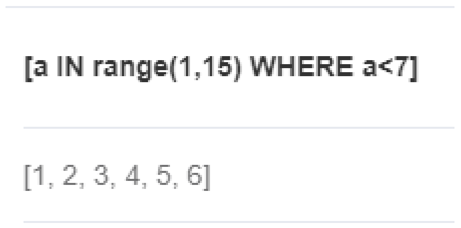


```
size(["Red","Green","Blue","Orange","Purple","Violet","Yellow"])
7
```

List comprehension is available in Cypher which is similar to set comprehension. A list can be produced as map or filtered result from an existing list.

Filtering:

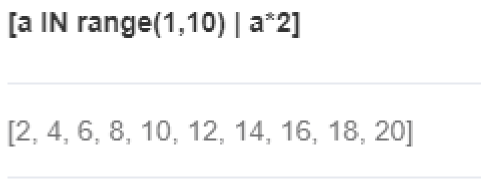
RETURN [a IN range(1,15) WHERE a<7]



```
[a IN range(1,15) WHERE a<7]

[1, 2, 3, 4, 5, 6]
```

Mapping:

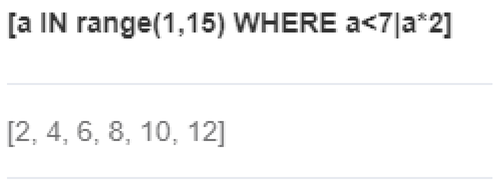
RETURN [a IN range(1,10) | a*2]



```
[a IN range(1,10) | a*2]

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Both:

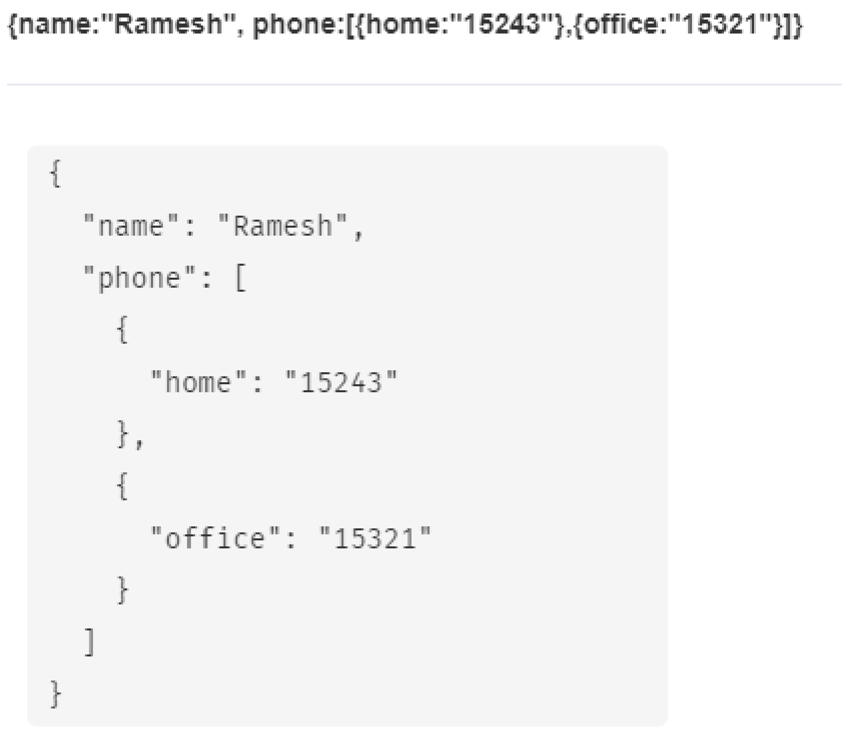
RETURN [a IN range(1,15) WHERE a<7|a*2]



```
[a IN range(1,15) WHERE a<7|a*2]

[2, 4, 6, 8, 10, 12]
```

It was also mentioned about how maps can also be created in cypher.

RETURN {name:"Ramesh", phone:[{home:"15243"},{office:"15321"}]}



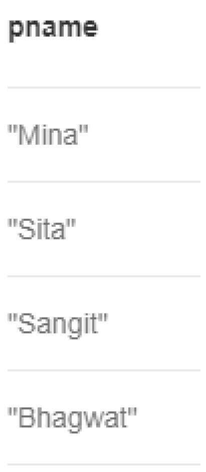
```
{name:"Ramesh", phone:[{home:"15243"},{office:"15321"}]}
```

```
{
  "name": "Ramesh",
  "phone": [
    {
      "home": "15243"
    },
    {
      "office": "15321"
    }
  ]
}
```

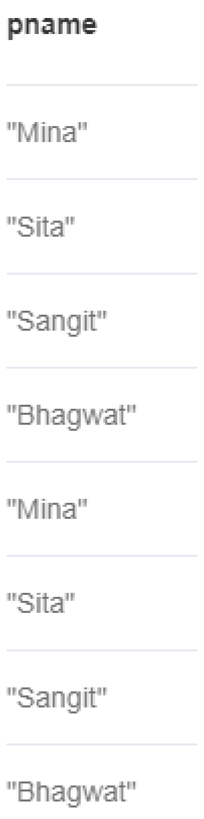
The relatively familiar UNION clause was also talked about which is used to combine results from multiple queries. The title of the column should however be common for all the queries involved. UNION ALL can be used to include duplicated rows as well whereas only UNION would display distinct records.

The duplicated records are not redisplayed in the UNION clause and the result contains 4 rows whereas the UNION ALL clause is returning 8 rows.

MATCH (n:Person) RETURN n.name AS pname UNION MATCH (p:Person)-[:OWNS]-(:Phone) RETURN p.name AS pname	MATCH (n:Person) RETURN n.name AS pname UNION ALL MATCH (p:Person)-[:OWNS]-(:Phone) RETURN p.name AS pname



```
pname
"Mina"
"Sita"
"Sangit"
"Bhagwat"
```



```
pname
"Mina"
"Sita"
"Sangit"
"Bhagwat"
"Mina"
"Sita"
"Sangit"
"Bhagwat"
```

Chapter Summary:

The chapter Cypher 8, like Cypher 7, taught about more cypher commands. All the commands provided in the lecture slides were replicated and practiced, making extra attempts on a different dataset than provided in the slides, similar to Cypher 7. CASE expressions were used to answer the lab exercises. Similarly, LISTs were created and manipulated as part of second exercise. Other clauses such as UNION and CALL were practiced at the end of the chapter. At this point, Cypher had become very familiar and could be used effectively for advanced queries as well as basic queries. Some improvements could still be made while using them for advanced purposes by going through the documentation and learning almost everything Cypher has to offer.

Migration from RDB to Neo4j

This chapter talks about why migration from a traditional relational database to Neo4j could be important and how to perform the migration. The weaknesses of a traditional RDB include not being able to handle complex relationships, degradation of performance with the increase in database size and relationship levels, complexity caused due to JOIN and sometimes having to change the entire schema to introduce something new. The Neo4j or a graph database does not require JOIN, will load only whatever is necessary and thus increase performance, allow to project or aggregate results (pipeline) at later stages of the queries and has many other advantages over RDB.

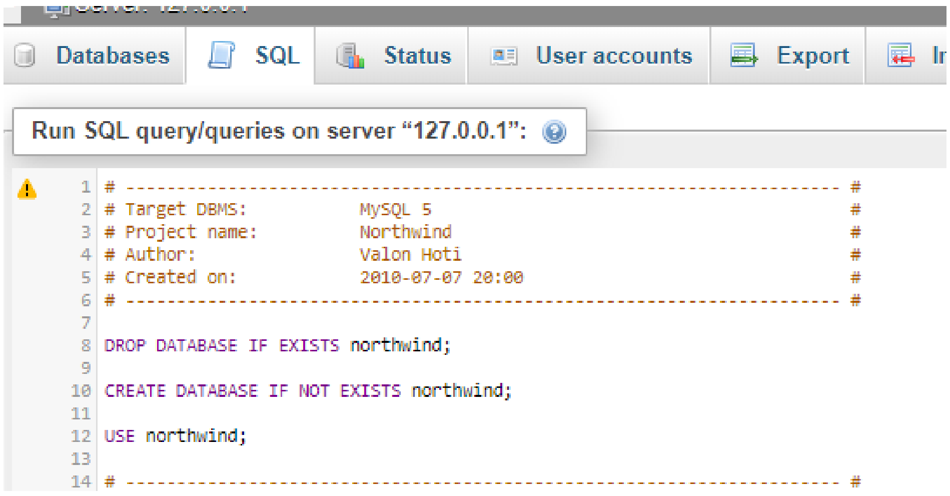
In order to migrate from RDB to Neo4j, there are some set rules that have to be followed. These rules include:

- Naming nodes as tables from the RDB
- The columns for tables in the RDB are used as properties for the nodes
- The values might need to be converted to support native datatypes for Neo4j
- The foreign keys are converted into relationships between nodes

Other set of rules for cleaning up, not including unwanted data in the new database and ensuring everything is correct are also defined. These rules mention about removing existing auto-increment primary keys as the nodes are unique in the graph database. However, other IDs such as ISBN number could be necessary and should not be removed. The names for Label, relationship type and property names which might not be included in the existing relational database should also be adjusted.

The two steps to migrate data from an RDB to Neo4j would be to export the data as CSV file and then import them using LOAD CSV command. An example Northwind database has been migrated as part of the tutorial in this chapter.

The provided SQL was copied and run in the MySQL server.



```
# ------------------------------------------------------------------------ #
# Target DBMS:            MySQL 5                                           #
# Project name:           Northwind                                         #
# Author:                 Valon Hoti                                        #
# Created on:             2010-07-07 20:00                                  #
# ------------------------------------------------------------------------ #

DROP DATABASE IF EXISTS northwind;

CREATE DATABASE IF NOT EXISTS northwind;

USE northwind;

# ------------------------------------------------------------------------ #
```

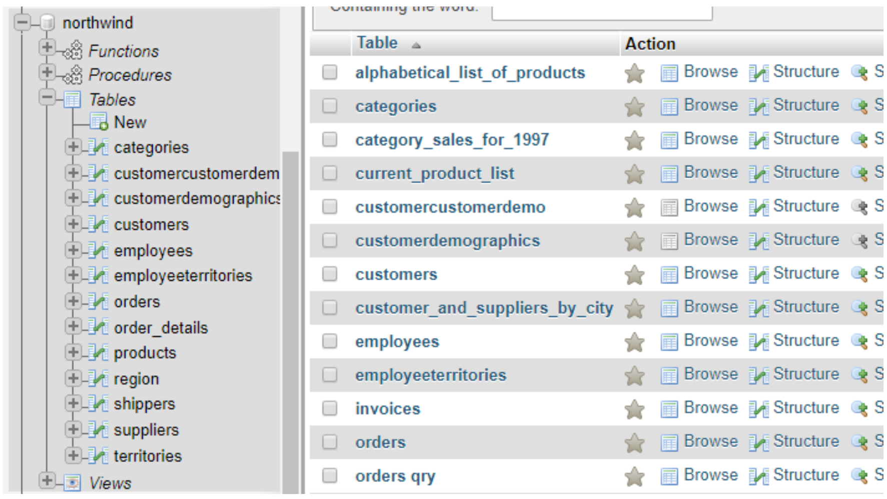
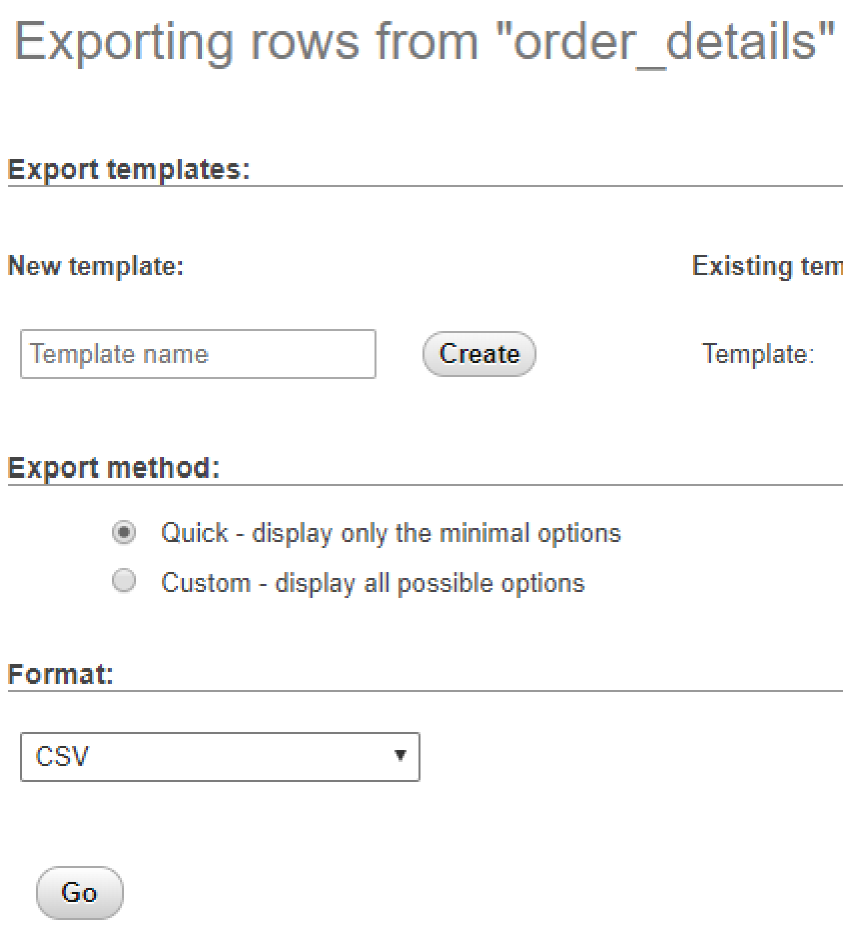


Table	Action
alphabetical_list_of_products	Browse Structure
categories	Browse Structure
category_sales_for_1997	Browse Structure
current_product_list	Browse Structure
customercustomerdemo	Browse Structure
customerdemographics	Browse Structure
customers	Browse Structure
customer_and_suppliers_by_city	Browse Structure
employees	Browse Structure
employeeterritories	Browse Structure
invoices	Browse Structure
orders	Browse Structure
orders qry	Browse Structure

This resulted in creation of the northwind database. The products and order_details table were exported as CSV files.


Exporting rows from "order_details"

Export templates:

New template:

Template name

Existing tem

Template:

Export method:

[✓ Quick - display only the minimal options]

Custom - display all possible options

Format:

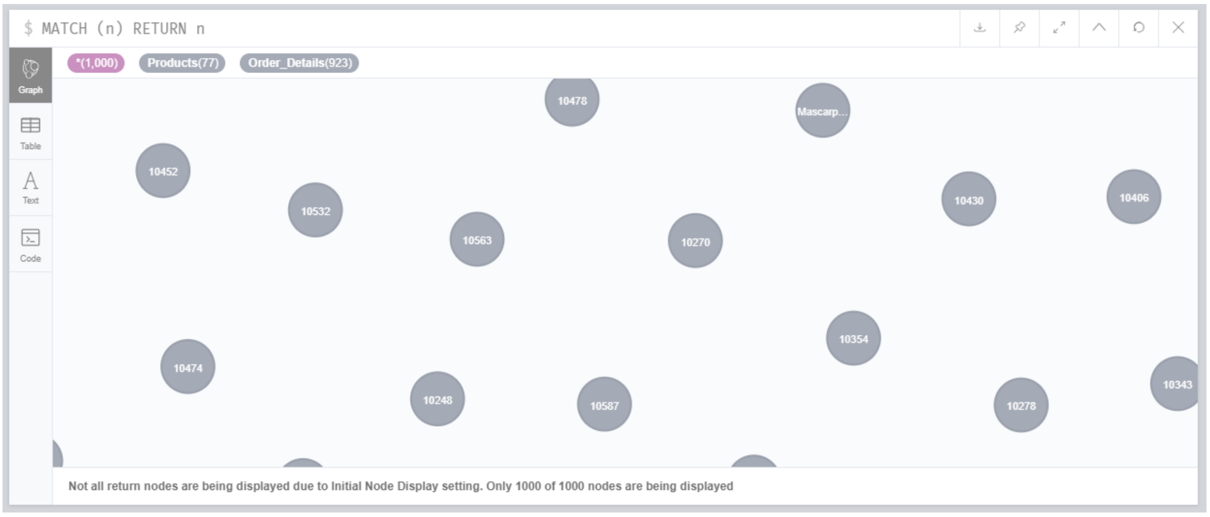
CSV

The files were exported to the Neo4j import directory. The provided LOAD CSV code was run to import the CSV files into the graph database.

```
LOAD CSV FROM "file:///Products.csv" AS row
CREATE (:Products{product_id:row[0],product_name:row[1]})
```

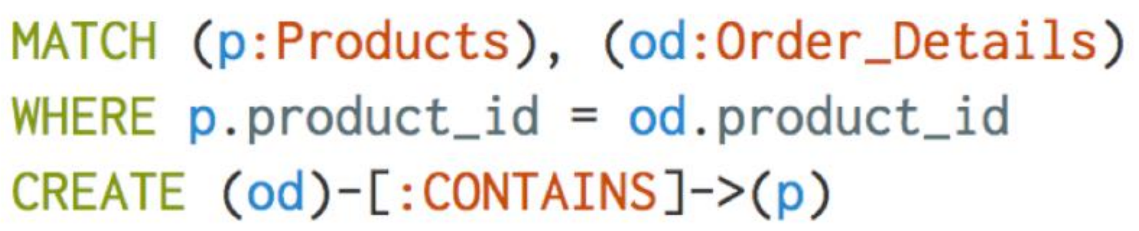
```
LOAD CSV FROM "file:///Order_Details.csv" AS row
CREATE (:Order_Details{order_id:row[0],product_id:row[1]})
```

The MATCH (n) RETURN n command displays the following result.

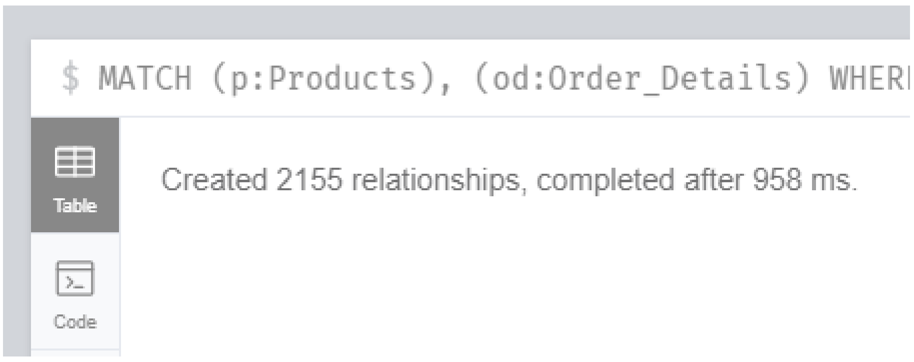

```
$ MATCH (n) RETURN n
```

Not all return nodes are being displayed due to Initial Node Display setting. Only 1000 of 1000 nodes are being displayed

The next step is to create the relationship between products and order_details.

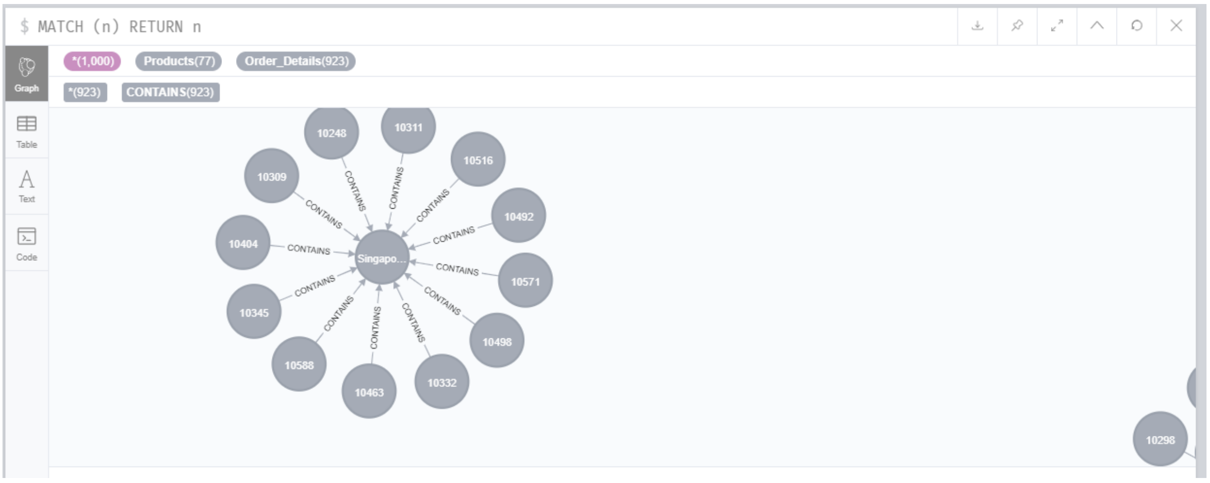


```
MATCH (p:Products), (od:Order_Details)
WHERE p.product_id = od.product_id
CREATE (od)-[:CONTAINS]->(p)
```



```
$ MATCH (p:Products), (od:Order_Details) WHERI
```

Created 2155 relationships, completed after 958 ms.



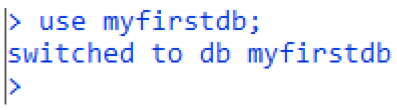
```
$ MATCH (n) RETURN n
```

MongoDB

This lesson introduced MongoDB. It was taught how MongoDB became the leading NoSQL database. MongoDB is a document-based database which uses BSON documents that are quite similar to JSON documents. The exclusion of JOINs removes complexity and increases performance. The MongoDB database has a lot of advantages over relational databases as well as some NoSQL databases. Some of these include increased availability due to distribution and replication of servers, being highly scalable, not requiring a predefined schema as the documents could have different data unlike in relational databases, being open source makes the community expand, being very easy to use, having support of APIs, and many other advantages.

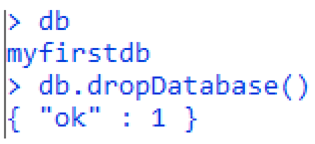
We can create a database if it does not exist by simply typing its name in the console after 'use' command.

use myfirstdb;



```
> use myfirstdb;
switched to db myfirstdb
>
```

Using the db command displays what database we are currently at and using the db.dropDatabase() command deletes the current database.



```
> db
myfirstdb
> db.dropDatabase()
{ "ok" : 1 }
```

The remaining slides talked about performing different operations (CRUD) in the database and how these relate to traditional SQL. The tutorial for this chapter included a restaurants.json file which was to be imported into a MongoDB database and queried.

- The JSON file was imported into a "restdb" database and inside "restaurants" collection.

mongoimport --db=restdb --collection=restaurants --file=D:/Database/restaurants.json

- Navigate to the restdb database

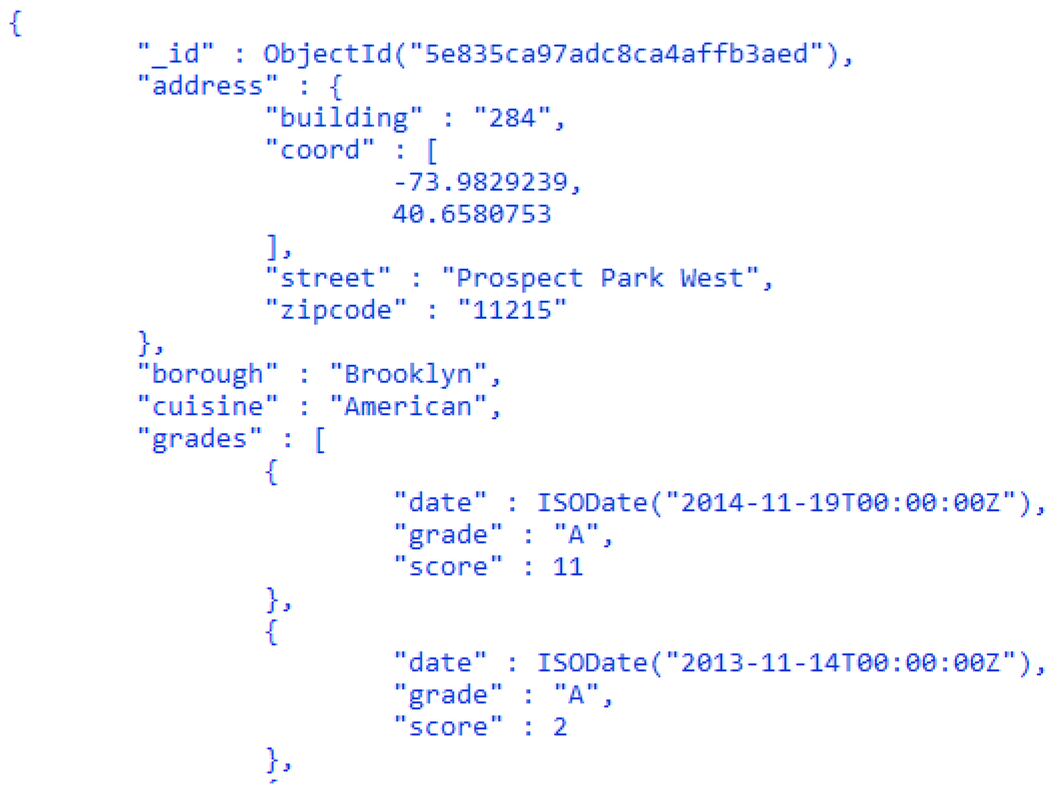
use restdb;

- Set up a variable for the collection

rest = db.restaurants;

- Test everything has run successfully till this point.

rest.find().pretty();



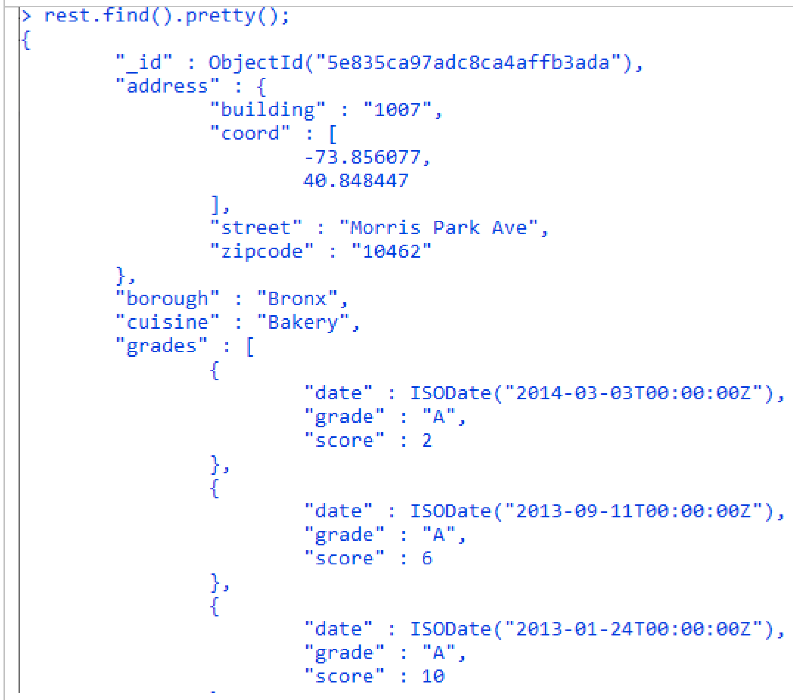
```
{
        "_id" : ObjectId("5e835ca97adc8ca4affb3aed"),
        "address" : {
                "building" : "284",
                "coord" : [
                        -73.9829239,
                        40.6580753
                ],
                "street" : "Prospect Park West",
                "zipcode" : "11215"
        },
        "borough" : "Brooklyn",
        "cuisine" : "American",
        "grades" : [
                {
                        "date" : ISODate("2014-11-19T00:00:00Z"),
                        "grade" : "A",
                        "score" : 11
                },
                {
                        "date" : ISODate("2013-11-14T00:00:00Z"),
                        "grade" : "A",
                        "score" : 2
                },
```

Now that the database is ready, we can start querying the database. The activities of MongoDB are included here.

MONGODB EXERCISES

Question: 1. Write a MongoDB query to display all the documents in the collection restaurants.

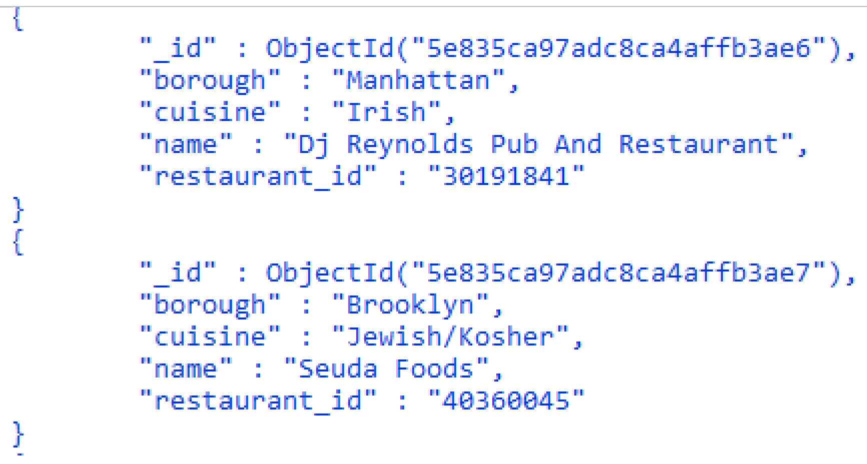
rest.find().pretty();



```
> rest.find().pretty();
{
        "_id" : ObjectId("5e835ca97adc8ca4affb3ada"),
        "address" : {
                "building" : "1007",
                "coord" : [
                        -73.856077,
                        40.848447
                ],
                "street" : "Morris Park Ave",
                "zipcode" : "10462"
        },
        "borough" : "Bronx",
        "cuisine" : "Bakery",
        "grades" : [
                {
                        "date" : ISODate("2014-03-03T00:00:00Z"),
                        "grade" : "A",
                        "score" : 2
                },
                {
                        "date" : ISODate("2013-09-11T00:00:00Z"),
                        "grade" : "A",
                        "score" : 6
                },
                {
                        "date" : ISODate("2013-01-24T00:00:00Z"),
                        "grade" : "A",
                        "score" : 10
```

Question: 2. Write a MongoDB query to display the fields restaurant_id, name, borough and cuisine for all the documents in the collection restaurant

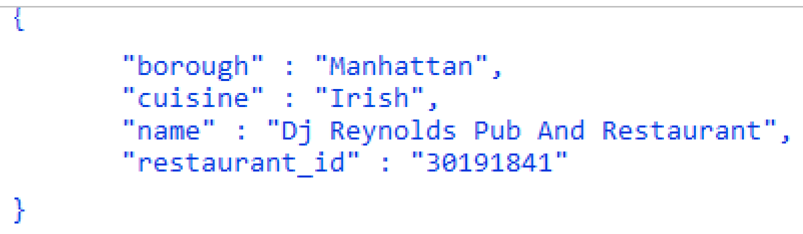
```
rest.find(
{},
{
        restaurant_id: 1, name: 1, borough: 1, cuisine: 1
}
).pretty()
```



```
{
        "_id" : ObjectId("5e835ca97adc8ca4affb3ae6"),
        "borough" : "Manhattan",
        "cuisine" : "Irish",
        "name" : "Dj Reynolds Pub And Restaurant",
        "restaurant_id" : "30191841"
}
{
        "_id" : ObjectId("5e835ca97adc8ca4affb3ae7"),
        "borough" : "Brooklyn",
        "cuisine" : "Jewish/Kosher",
        "name" : "Seuda Foods",
        "restaurant_id" : "40360045"
}
```

Question: 3. Write a MongoDB query to display the fields restaurant_id, name, borough and cuisine, but exclude the field _id for all the documents in the collection restaurant.

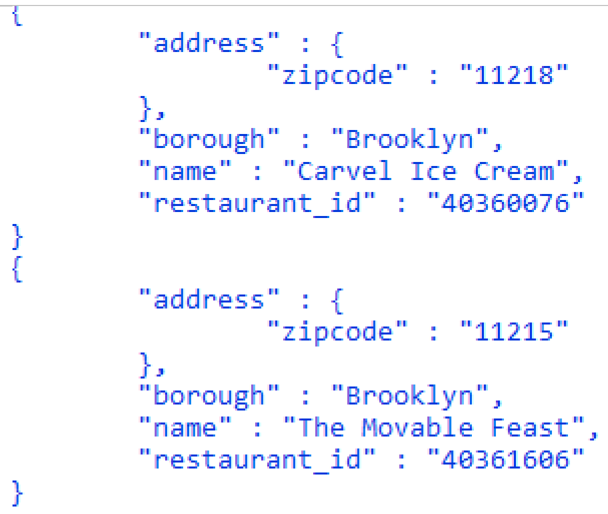
```
rest.find(
{},
{
        restaurant_id: 1, name: 1, borough: 1, _id: 0, cuisine: 1
}
).pretty()
```



```
{
        "borough" : "Manhattan",
        "cuisine" : "Irish",
        "name" : "Dj Reynolds Pub And Restaurant",
        "restaurant_id" : "30191841"

}
```

Question: 4. Write a MongoDB query to display the fields restaurant_id, name, borough and zip code, but exclude the field _id for all the documents in the collection restaurant.

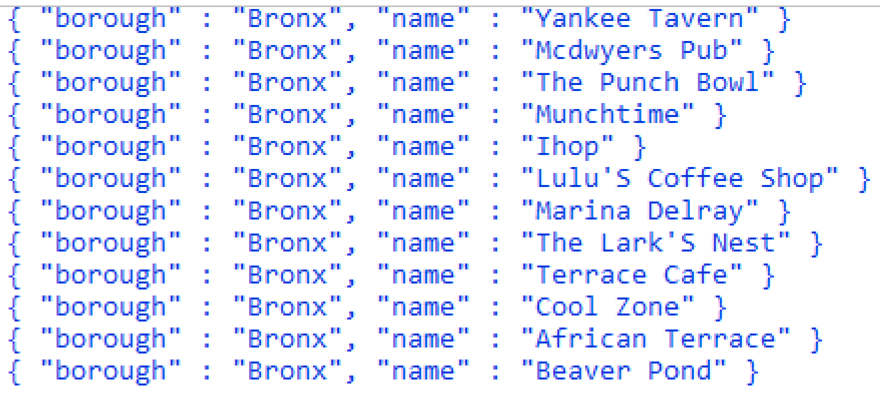
```
rest.find(
{},
{
        restaurant_id: 1, name: 1, borough: 1, _id: 0, "address.zipcode": 1
}
).pretty()
```



```
{
        "address" : {
                "zipcode" : "11218"
        },
        "borough" : "Brooklyn",
        "name" : "Carvel Ice Cream",
        "restaurant_id" : "40360076"
}
{
        "address" : {
                "zipcode" : "11215"
        },
        "borough" : "Brooklyn",
        "name" : "The Movable Feast",
        "restaurant_id" : "40361606"
}
```

Question: 5. Write a MongoDB query to display all the restaurant which is in the borough Bronx.

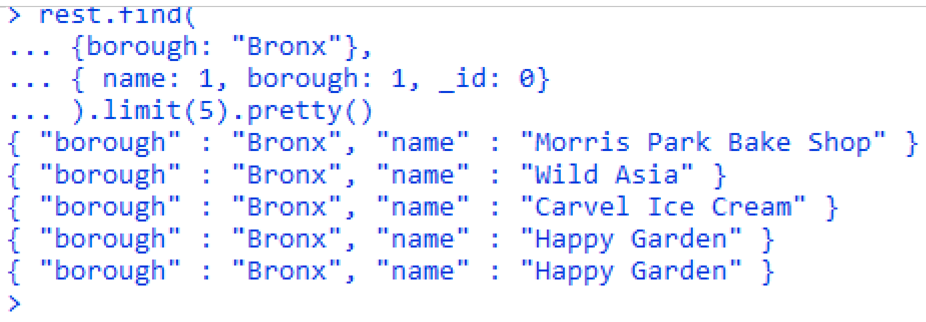
```
rest.find(
        {borough: "Bronx"},
        {name: 1, borough: 1, _id: 0}
).pretty()
```



```
{ "borough" : "Bronx", "name" : "Yankee Tavern" }
{ "borough" : "Bronx", "name" : "Mcdwyers Pub" }
{ "borough" : "Bronx", "name" : "The Punch Bowl" }
{ "borough" : "Bronx", "name" : "Munchtime" }
{ "borough" : "Bronx", "name" : "Ihop" }
{ "borough" : "Bronx", "name" : "Lulu'S Coffee Shop" }
{ "borough" : "Bronx", "name" : "Marina Delray" }
{ "borough" : "Bronx", "name" : "The Lark'S Nest" }
{ "borough" : "Bronx", "name" : "Terrace Cafe" }
{ "borough" : "Bronx", "name" : "Cool Zone" }
{ "borough" : "Bronx", "name" : "African Terrace" }
{ "borough" : "Bronx", "name" : "Beaver Pond" }
```

Question: 6. Write a MongoDB query to display the first 5 restaurant which is in the borough Bronx

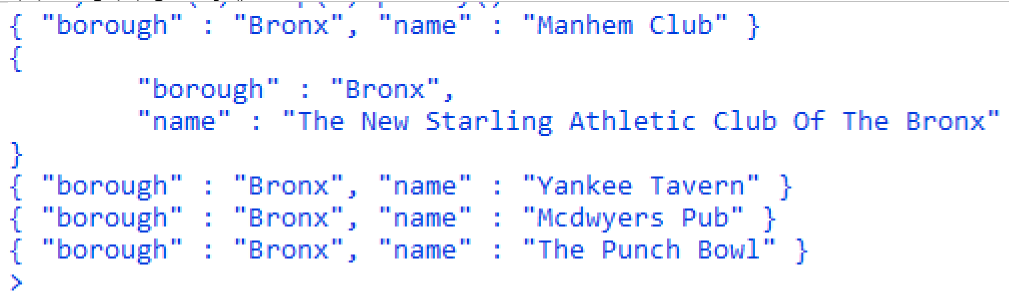
```
rest.find(
        {borough: "Bronx"},
        { name: 1, borough: 1, _id: 0}
).limit(5).pretty()
```



```
> rest.find(
... {borough: "Bronx"},
... { name: 1, borough: 1, _id: 0}
... ).limit(5).pretty()
{ "borough" : "Bronx", "name" : "Morris Park Bake Shop" }
{ "borough" : "Bronx", "name" : "Wild Asia" }
{ "borough" : "Bronx", "name" : "Carvel Ice Cream" }
{ "borough" : "Bronx", "name" : "Happy Garden" }
{ "borough" : "Bronx", "name" : "Happy Garden" }
>
```

Question: 7. Write a MongoDB query to display the next 5 restaurants after skipping first 5 which are in the borough Bronx.

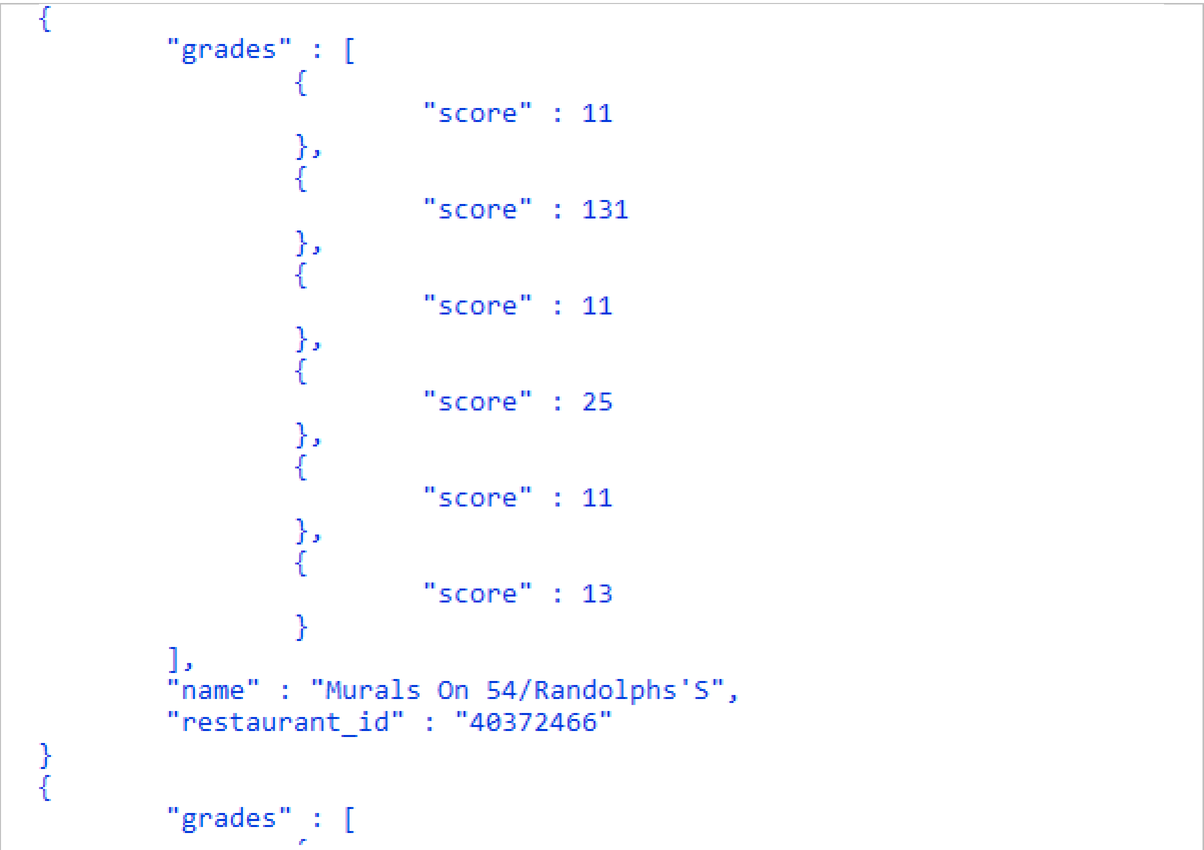
```
rest.find(
        {borough: "Bronx"},
        {name: 1, borough: 1, _id: 0}
).limit(5).skip(5).pretty()
```



```
{ "borough" : "Bronx", "name" : "Manhem Club" }
{
        "borough" : "Bronx",
        "name" : "The New Starling Athletic Club Of The Bronx"
}
{ "borough" : "Bronx", "name" : "Yankee Tavern" }
{ "borough" : "Bronx", "name" : "Mcdwyers Pub" }
{ "borough" : "Bronx", "name" : "The Punch Bowl" }
>
```

Question: 8. Write a MongoDB query to find the restaurants who achieved a score more than 90.

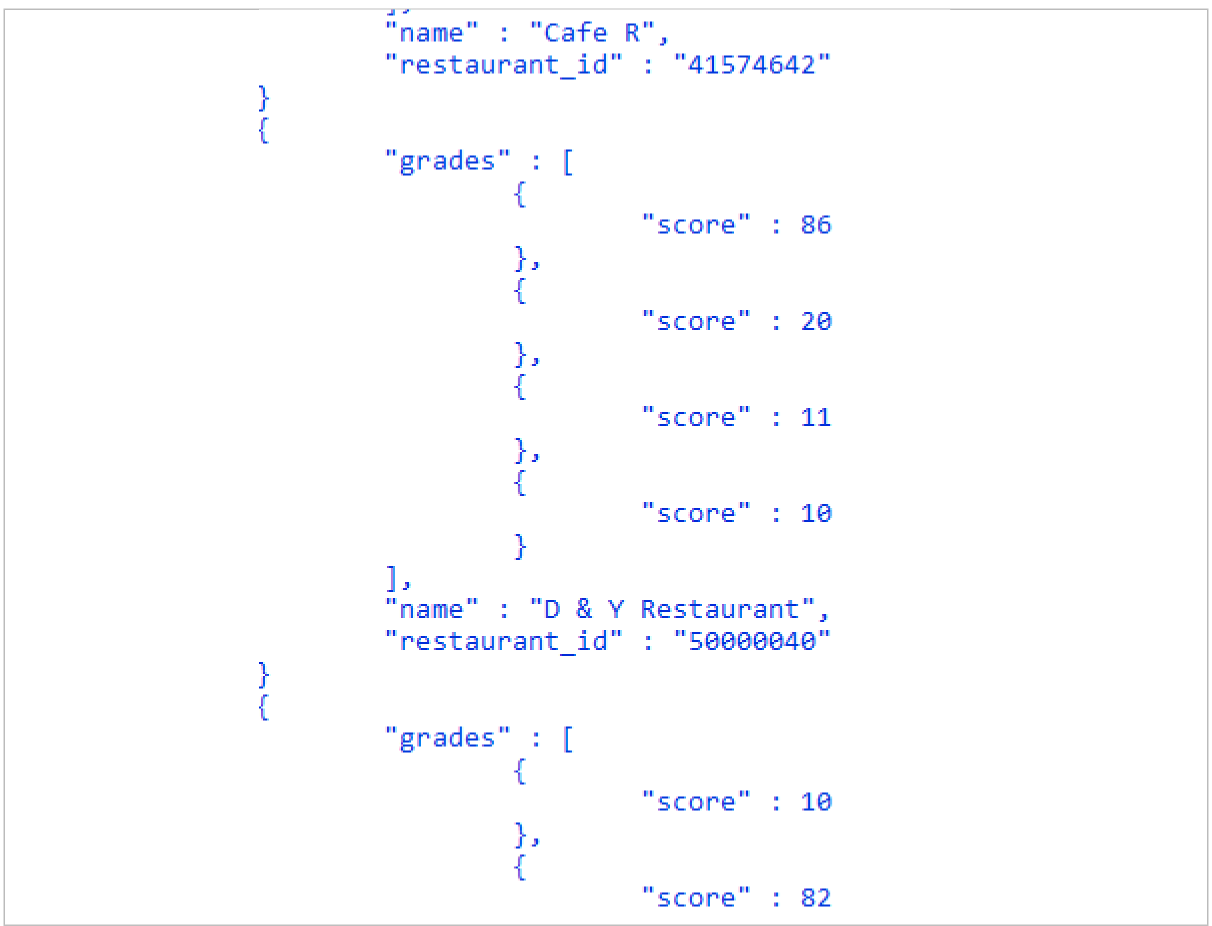
```
rest.find(
        {"grades.score": {$gt:90}},
        {restaurant_id: 1, name: 1, "grades.score": 1, _id: 0}
).pretty()
```



```
{
        "grades" : [
                {
                        "score" : 11
                },
                {
                        "score" : 131
                },
                {
                        "score" : 11
                },
                {
                        "score" : 25
                },
                {
                        "score" : 11
                },
                {
                        "score" : 13
                }
        ],
        "name" : "Murals On 54/Randolphs'S",
        "restaurant_id" : "40372466"
}
{
        "grades" : [
```

Question: 9. Write a MongoDB query to find the restaurants that achieved a score, more than 80 but less than 100.

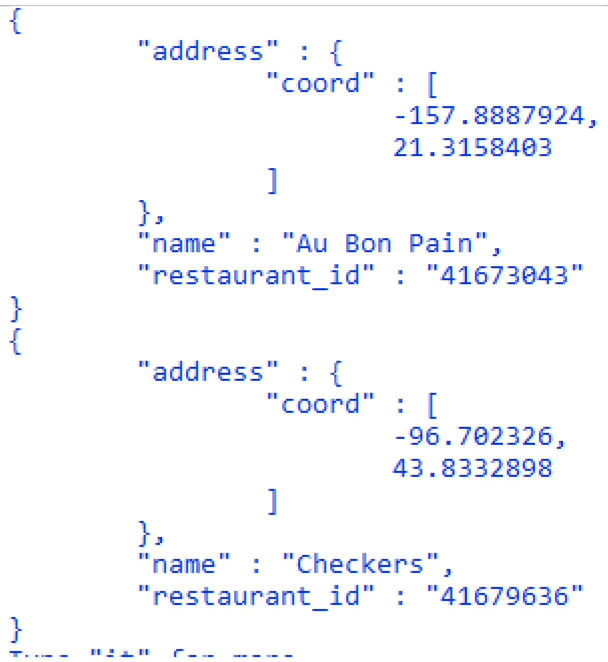
```
rest.find(
        {"grades.score": {$gt: 80, $lt: 100}},
        {restaurant_id: 1, name: 1, "grades.score": 1, _id: 0}
).pretty()
```



```
                        "name" : "Cafe R",
                        "restaurant_id" : "41574642"
                }
                {
                        "grades" : [
                                {
                                        "score" : 86
                                },
                                {
                                        "score" : 20
                                },
                                {
                                        "score" : 11
                                },
                                {
                                        "score" : 10
                                }
                        ],
                        "name" : "D & Y Restaurant",
                        "restaurant_id" : "50000040"
                }
                {
                        "grades" : [
                                {
                                        "score" : 10
                                },
                                {
                                        "score" : 82
```

Question: 10. Write a MongoDB query to find the restaurants which locate in latitude value less than -95.754168.

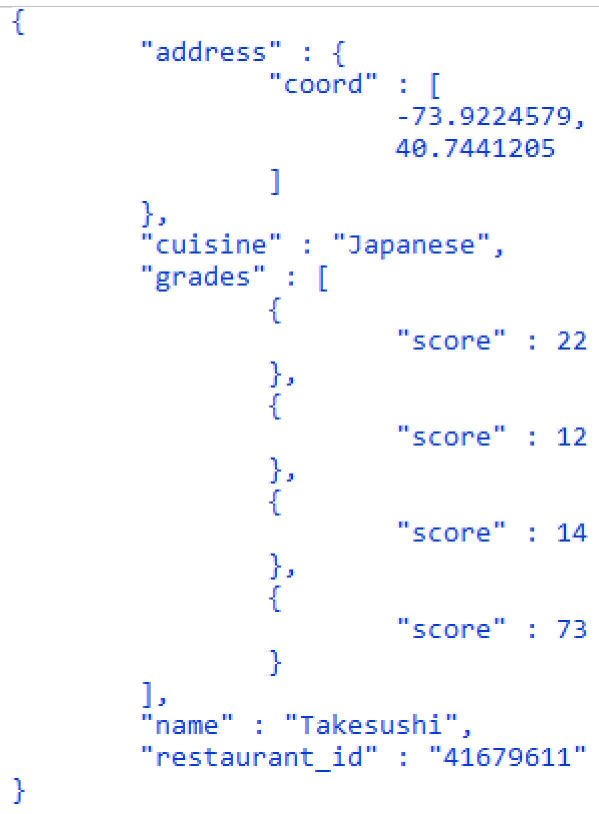
```
rest.find(
        {"address.coord.0": {$lt: -95.754168 }},
        {
                restaurant_id: 1, name: 1, _id: 0, "address.coord": 1
        }
).pretty()
```



```
{
        "address" : {
                "coord" : [
                        -157.8887924,
                        21.3158403
                ]
        },
        "name" : "Au Bon Pain",
        "restaurant_id" : "41673043"
}
{
        "address" : {
                "coord" : [
                        -96.702326,
                        43.8332898
                ]
        },
        "name" : "Checkers",
        "restaurant_id" : "41679636"
}
```

Question: 11. Write a MongoDB query to find the restaurants that do not prepare any cuisine of 'American' and their grade score more than 70 and latitude less than -65.754168.

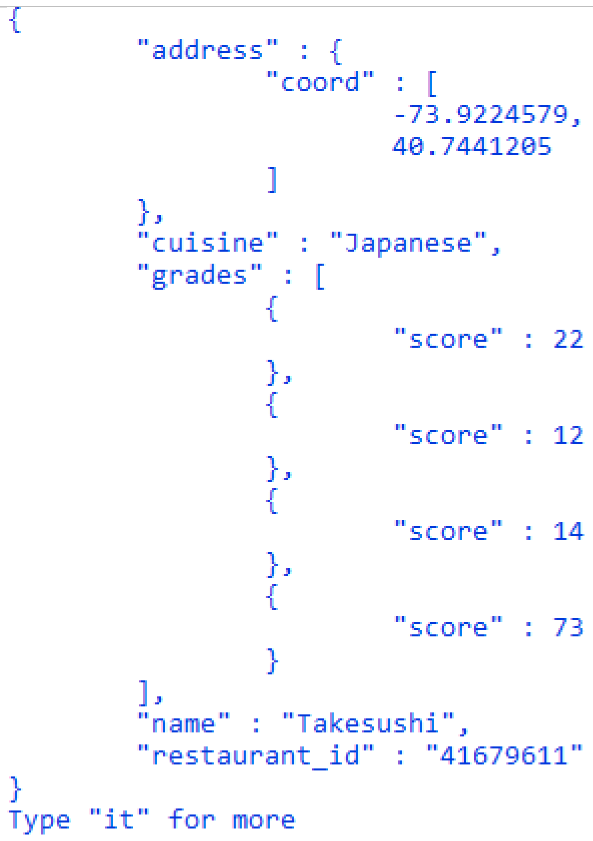
```
rest.find(
        {"address.coord.1": {$gte: -65.754168 }, "grades.score": {$gt:70}, cuisine: {$ne:"American"}},
        {
                restaurant_id: 1, name: 1, _id: 0, "address.coord": 1, "grades.score": 1, cuisine: 1
        }
).pretty()
```



```
{
        "address" : {
                "coord" : [
                        -73.9224579,
                        40.7441205
                ]
        },
        "cuisine" : "Japanese",
        "grades" : [
                {
                        "score" : 22
                },
                {
                        "score" : 12
                },
                {
                        "score" : 14
                },
                {
                        "score" : 73
                }
        ],
        "name" : "Takesushi",
        "restaurant_id" : "41679611"
}
```

Question: 12. Write a MongoDB query to find the restaurants which do not prepare any cuisine of 'American' and achieved a score more than 70 and not located in the longitude less than -65.754168. Note : Do this query without using $and operator.

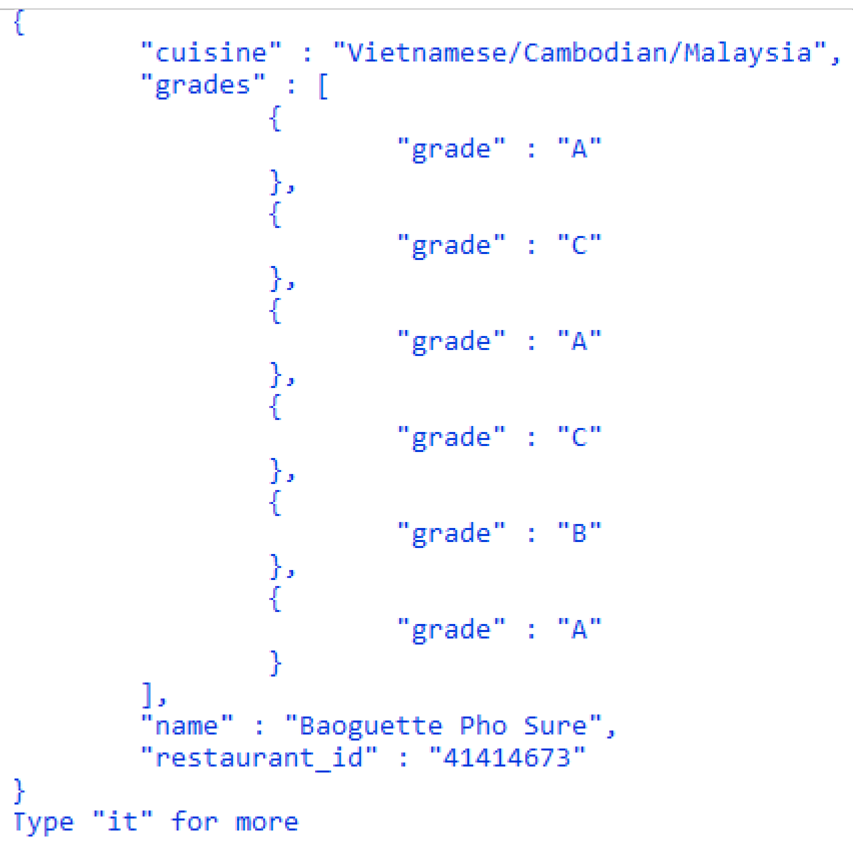
```
rest.find(
        {"address.coord.1": {$gte: -65.754168 }, "grades.score": {$gt:70}, cuisine: {$ne:"American"}},
        {
                restaurant_id: 1, name: 1, _id: 0, "address.coord": 1, "grades.score": 1, cuisine: 1
        }
).pretty()
```



```
{
        "address" : {
                "coord" : [
                        -73.9224579,
                        40.7441205
                ]
        },
        "cuisine" : "Japanese",
        "grades" : [
                {
                        "score" : 22
                },
                {
                        "score" : 12
                },
                {
                        "score" : 14
                },
                {
                        "score" : 73
                }
        ],
        "name" : "Takesushi",
        "restaurant_id" : "41679611"
}
Type "it" for more
```

Question: 13. Write a MongoDB query to find the restaurants which do not prepare any cuisine of 'American ' and achieved a grade point 'A' not belongs to the borough Brooklyn. The document must be displayed according to the cuisine in descending order.

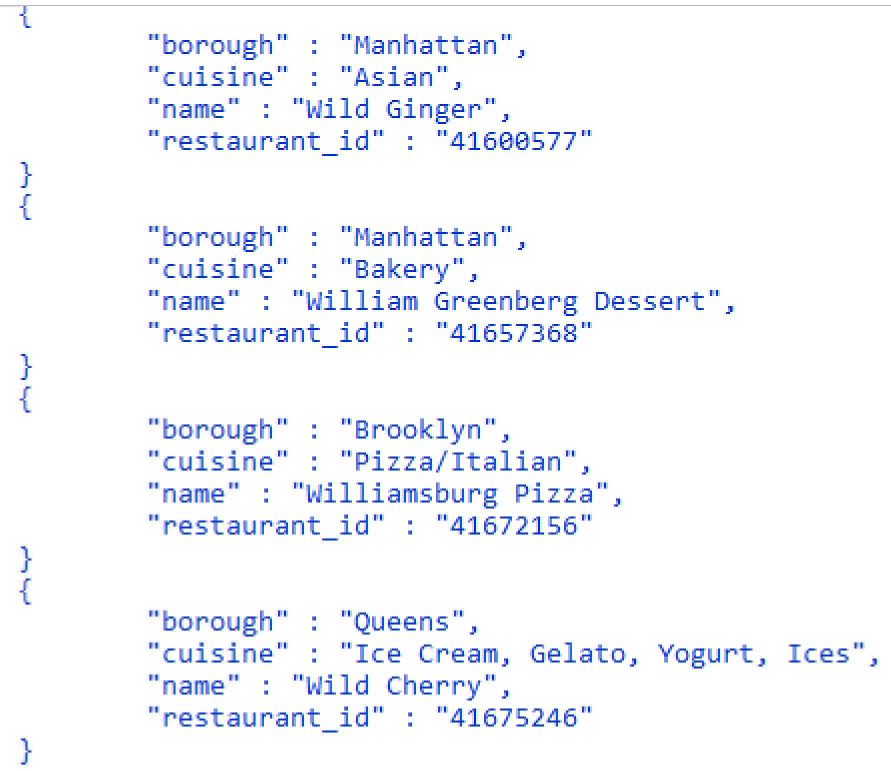
```
rest.find(
        {borough: {$ne:"Brooklyn"}, "grades.grade": "A", cuisine: {$ne:"American"}},
        {
                restaurant_id: 1, name: 1, _id: 0, "grades.grade": 1, cuisine: 1
        }
).sort({cuisine: -1}).pretty()
```



```
{
        "cuisine" : "Vietnamese/Cambodian/Malaysia",
        "grades" : [
                {
                        "grade" : "A"
                },
                {
                        "grade" : "C"
                },
                {
                        "grade" : "A"
                },
                {
                        "grade" : "C"
                },
                {
                        "grade" : "B"
                },
                {
                        "grade" : "A"
                }
        ],
        "name" : "Baoguette Pho Sure",
        "restaurant_id" : "41414673"
}
Type "it" for more
```

Question: 14. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'Wil' as first three letters for its name.

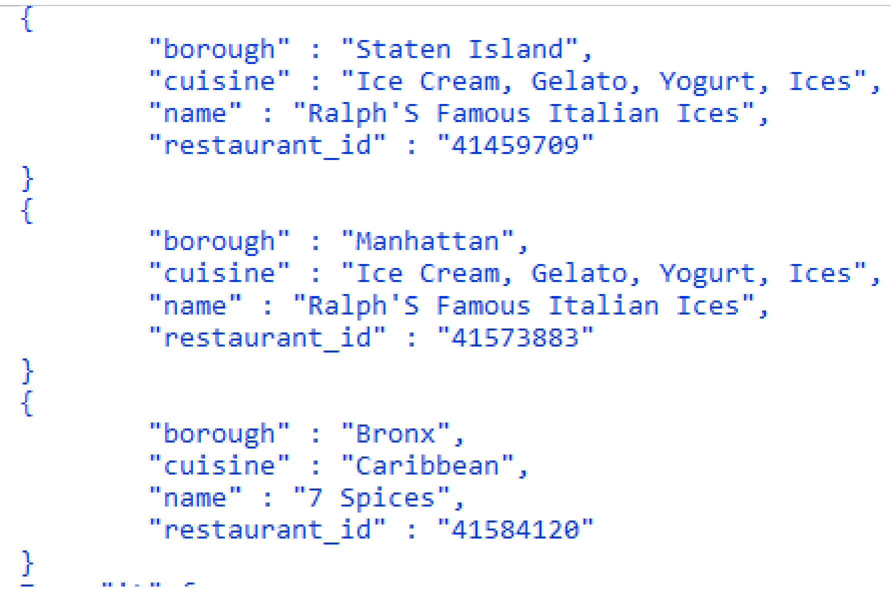
```
rest.find(
        {"name": /^Wil/},
        {restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
        {
                "borough" : "Manhattan",
                "cuisine" : "Asian",
                "name" : "Wild Ginger",
                "restaurant_id" : "41600577"
        }
        {
                "borough" : "Manhattan",
                "cuisine" : "Bakery",
                "name" : "William Greenberg Dessert",
                "restaurant_id" : "41657368"
        }
        {
                "borough" : "Brooklyn",
                "cuisine" : "Pizza/Italian",
                "name" : "Williamsburg Pizza",
                "restaurant_id" : "41672156"
        }
        {
                "borough" : "Queens",
                "cuisine" : "Ice Cream, Gelato, Yogurt, Ices",
                "name" : "Wild Cherry",
                "restaurant_id" : "41675246"
        }
```

Question: 15. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'ces' as last three letters for its name.

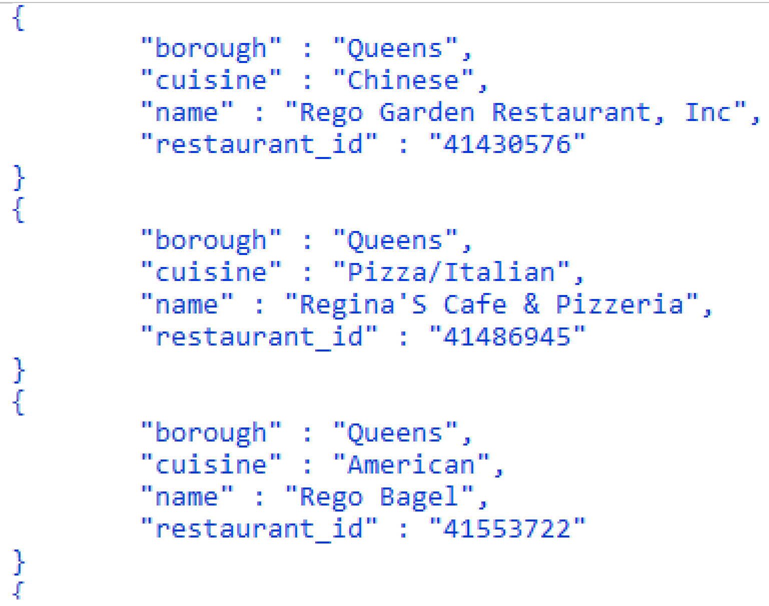
```
rest.find(
        {"name": /ces$/},
        {restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
        {
                "borough" : "Staten Island",
                "cuisine" : "Ice Cream, Gelato, Yogurt, Ices",
                "name" : "Ralph'S Famous Italian Ices",
                "restaurant_id" : "41459709"
        }
        {
                "borough" : "Manhattan",
                "cuisine" : "Ice Cream, Gelato, Yogurt, Ices",
                "name" : "Ralph'S Famous Italian Ices",
                "restaurant_id" : "41573883"
        }
        {
                "borough" : "Bronx",
                "cuisine" : "Caribbean",
                "name" : "7 Spices",
                "restaurant_id" : "41584120"
        }
```

Question: 16. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'Reg' as three letters somewhere in its name.

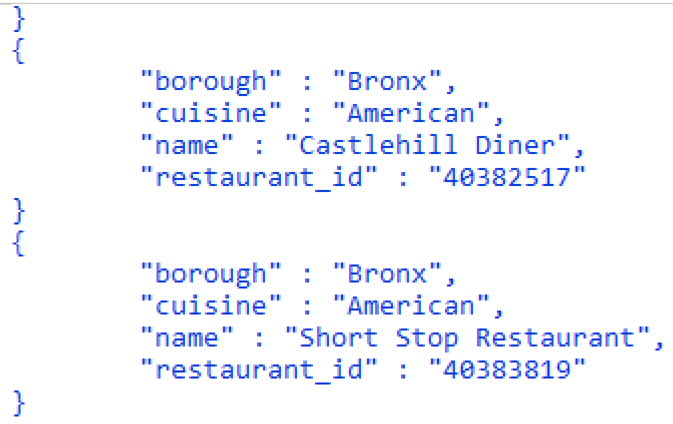
```
rest.find(
        {"name": /Reg/},
	{restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
{
        "borough" : "Queens",
        "cuisine" : "Chinese",
        "name" : "Rego Garden Restaurant, Inc",
        "restaurant_id" : "41430576"
}
{
        "borough" : "Queens",
        "cuisine" : "Pizza/Italian",
        "name" : "Regina'S Cafe & Pizzeria",
        "restaurant_id" : "41486945"
}
{
        "borough" : "Queens",
        "cuisine" : "American",
        "name" : "Rego Bagel",
        "restaurant_id" : "41553722"
}
{
```

Question: 17. Write a MongoDB query to find the restaurants which belong to the borough Bronx and prepared either American or Chinese dish.

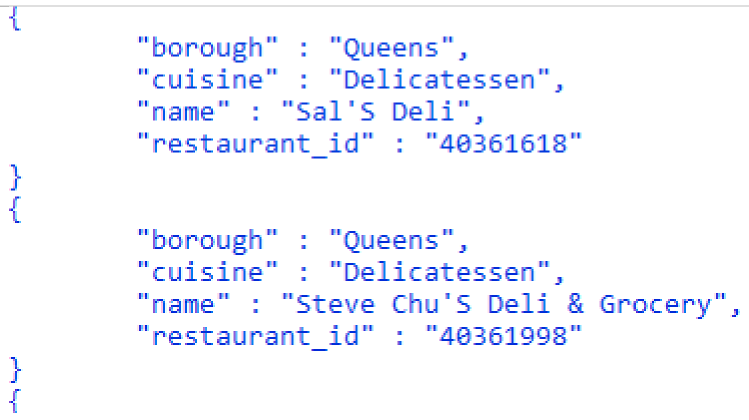
```
rest.find(
	{$or:[{cuisine: "American"}, {cuisine: "Chinese"}], borough:"Bronx"},
	{
		restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0
	}
).pretty()
```



```
}
{
        "borough" : "Bronx",
        "cuisine" : "American",
        "name" : "Castlehill Diner",
        "restaurant_id" : "40382517"
}
{
        "borough" : "Bronx",
        "cuisine" : "American",
        "name" : "Short Stop Restaurant",
        "restaurant_id" : "40383819"
}
```

Question: 18. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which belong to the borough Staten Island or Queens or Bronx or Brooklyn

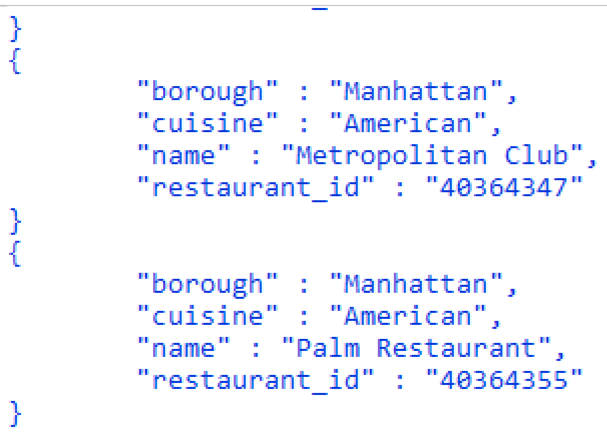
```
rest.find(
	{$or:[{borough: "Staten Island"}, {borough: "Queens"}, {borough: "Bronx"}, {borough: "Brooklyn"}]},
	{
		restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0
	}
).pretty()
```



```
{
        "borough" : "Queens",
        "cuisine" : "Delicatessen",
        "name" : "Sal'S Deli",
        "restaurant_id" : "40361618"
}
{
        "borough" : "Queens",
        "cuisine" : "Delicatessen",
        "name" : "Steve Chu'S Deli & Grocery",
        "restaurant_id" : "40361998"
}
{
```

Question: 19. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which are not belonging to the borough Staten Island or Queens or Bronx or Brooklyn.

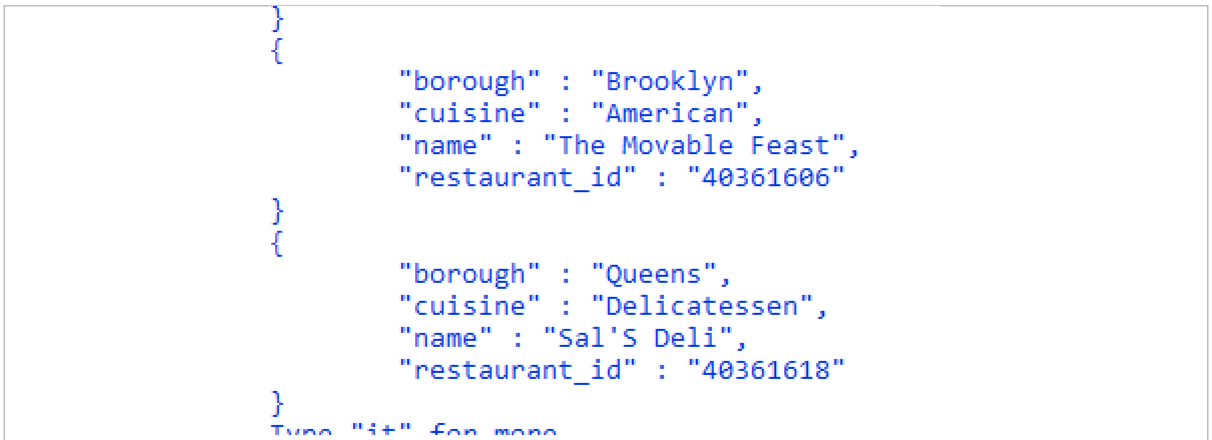
```
rest.find(
	{$nor:[{borough: "Staten Island"}, {borough: "Queens"}, {borough: "Bronx"}, {borough: "Brooklyn"}]},
	{
		restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0
	}
).pretty()
```



```
}
{
        "borough" : "Manhattan",
        "cuisine" : "American",
        "name" : "Metropolitan Club",
        "restaurant_id" : "40364347"
}
{
        "borough" : "Manhattan",
        "cuisine" : "American",
        "name" : "Palm Restaurant",
        "restaurant_id" : "40364355"
}
```

Question: 20. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which achieved a score which is not more than 10.

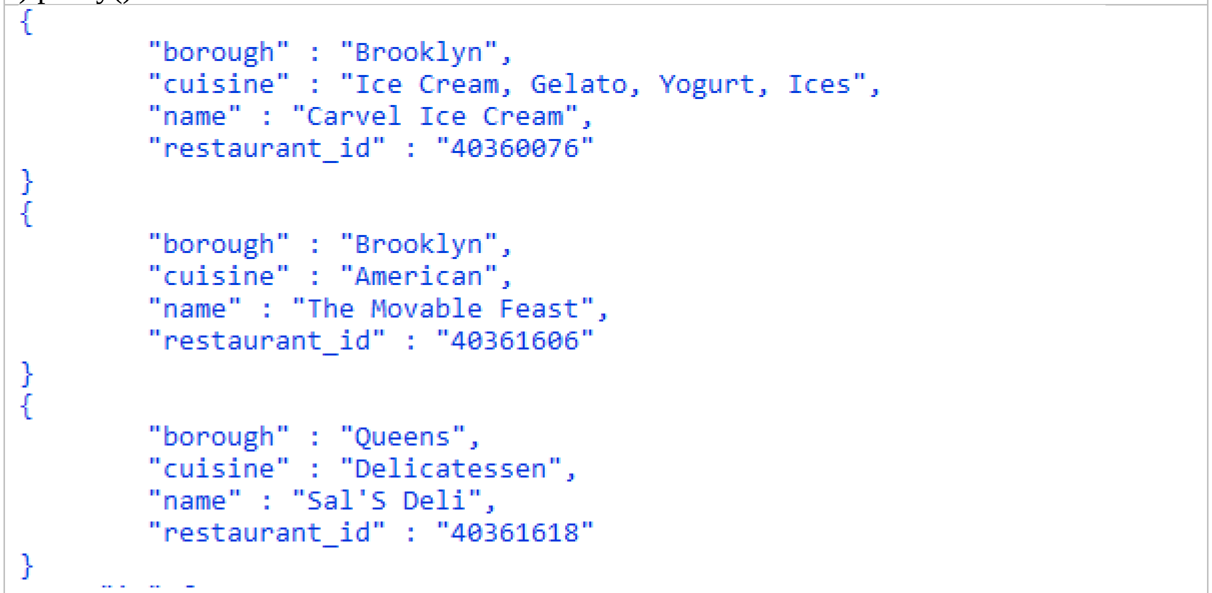
```
rest.find(
	{"grades.score": {$lte:10}},
	{restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
}
{
        "borough" : "Brooklyn",
        "cuisine" : "American",
        "name" : "The Movable Feast",
        "restaurant_id" : "40361606"
}
{
        "borough" : "Queens",
        "cuisine" : "Delicatessen",
        "name" : "Sal'S Deli",
        "restaurant_id" : "40361618"
}
Type "it" for more
```

Question: 21. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which prepared dish except 'American' and 'Chinese' or restaurant's name begins with letter 'Wil'.

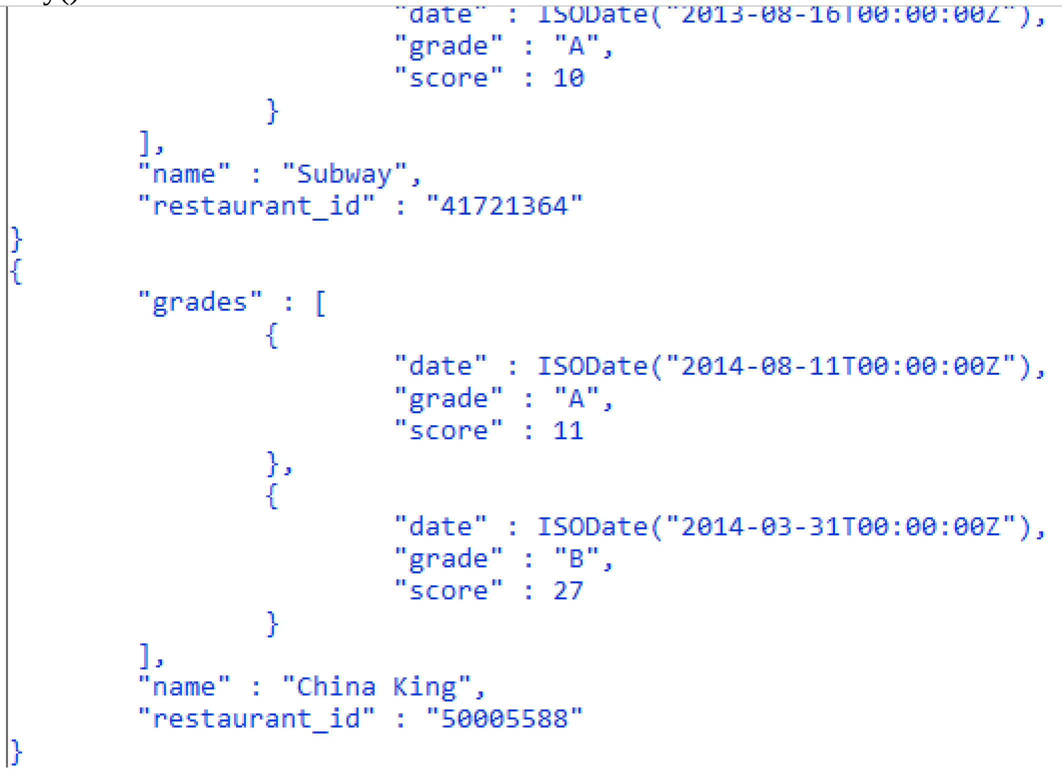
```
rest.find(
        {
                $or:[{cuisine: {$ne:"American", $ne:"Chinese"}}, {"name": /^Wil/}]
        },
        {
                restaurant_id: 1, name: 1, borough: 1, cuisine: 1, _id: 0
        }
).pretty()
```



```
{
        "borough" : "Brooklyn",
        "cuisine" : "Ice Cream, Gelato, Yogurt, Ices",
        "name" : "Carvel Ice Cream",
        "restaurant_id" : "40360076"
}
{
        "borough" : "Brooklyn",
        "cuisine" : "American",
        "name" : "The Movable Feast",
        "restaurant_id" : "40361606"
}
{
        "borough" : "Queens",
        "cuisine" : "Delicatessen",
        "name" : "Sal'S Deli",
        "restaurant_id" : "40361618"
}
```

Question: 22. Write a MongoDB query to find the restaurant Id, name, and grades for those restaurants which achieved a grade of "A" and scored 11 on an ISODate "2014-08-11T00:00:00Z" among many of survey dates.

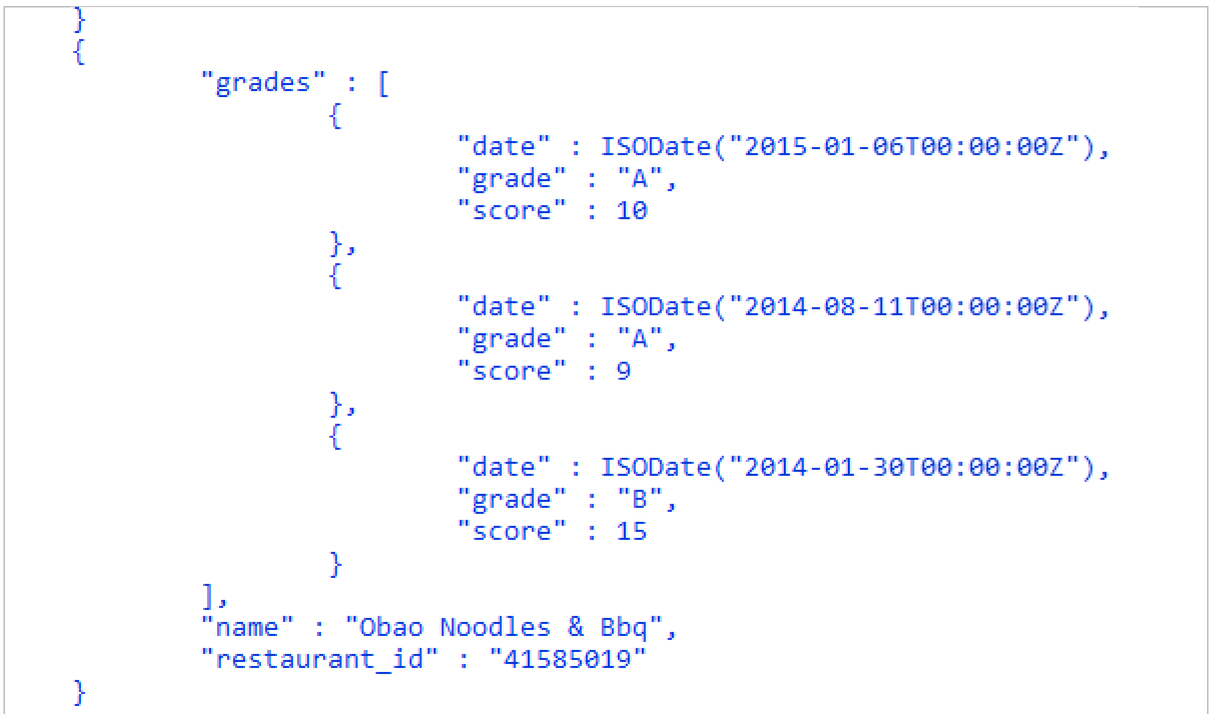
```
rest.find(
        {
                "grades.grade": "A",
                "grades": { $elemMatch: {score: 11,
                date: ISODate("2014-08-11T00:00:00Z")}}
        },
        {
                restaurant_id: 1, name: 1, _id: 0, "grades.grade": 1, "grades.score": 1,
"grades.date": 1
        }
).pretty()
```



```
                                "date" : ISODate("2013-08-16T00:00:00Z"),
                                "grade" : "A",
                                "score" : 10
                        }
                ],
                "name" : "Subway",
                "restaurant_id" : "41721364"
}
{
                "grades" : [
                        {
                                "date" : ISODate("2014-08-11T00:00:00Z"),
                                "grade" : "A",
                                "score" : 11
                        },
                        {
                                "date" : ISODate("2014-03-31T00:00:00Z"),
                                "grade" : "B",
                                "score" : 27
                        }
                ],
                "name" : "China King",
                "restaurant_id" : "50005588"
}
```

Question: 23. Write a MongoDB query to find the restaurant Id, name and grades for those restaurants where the 2nd element of grades array contains a grade of "A" and score 9 on an ISODate "2014-08-11T00:00:00Z".

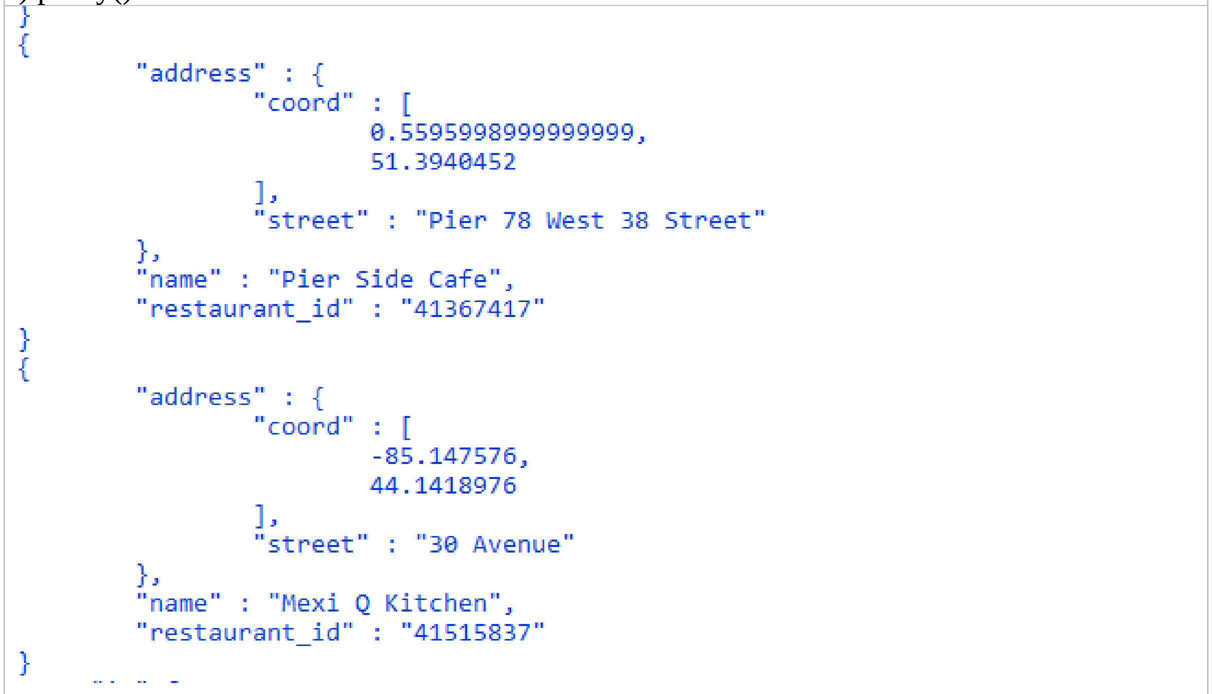
```
rest.find(
        {

                "grades.1.grade": "A",
                "grades.1.score": 9,
                "grades.1.date": ISODate("2014-08-11T00:00:00Z")
        },
        {
                restaurant_id: 1, name: 1, _id: 0, "grades.grade": 1, "grades.score": 1,
"grades.date": 1
        }
).pretty()
```



```
    }
    {
            "grades" : [
                    {
                            "date" : ISODate("2015-01-06T00:00:00Z"),
                            "grade" : "A",
                            "score" : 10
                    },
                    {
                            "date" : ISODate("2014-08-11T00:00:00Z"),
                            "grade" : "A",
                            "score" : 9
                    },
                    {
                            "date" : ISODate("2014-01-30T00:00:00Z"),
                            "grade" : "B",
                            "score" : 15
                    }
            ],
            "name" : "Obao Noodles & Bbq",
            "restaurant_id" : "41585019"
    }
```

Question: 24. Write a MongoDB query to find the restaurant Id, name, address and geographical location for those restaurants where 2nd element of coord array contains a value which is more than 42 and up to 52

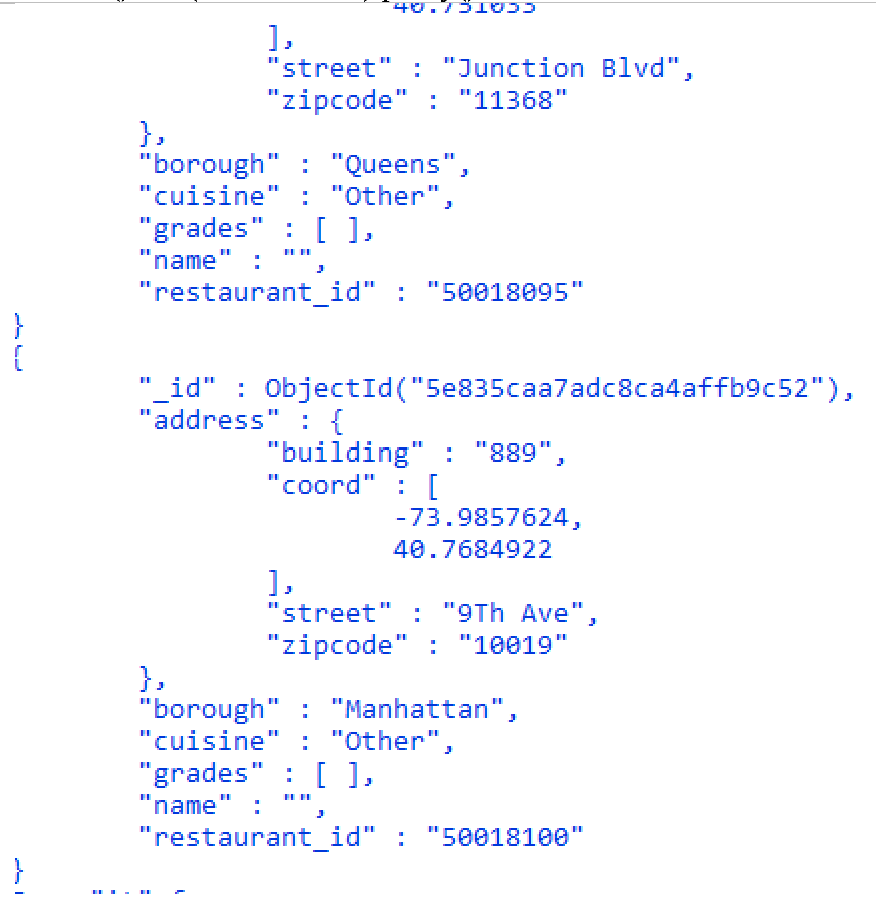
```
rest.find(
        {"address.coord.1": {$gt: 42, $lte: 52}},
        {
                restaurant_id: 1, name: 1, _id: 0, "address.coord": 1, "address.street": 1
        }
).pretty()
```



```
}
{
        "address" : {
                "coord" : [
                        0.5595998999999999,
                        51.3940452
                ],
                "street" : "Pier 78 West 38 Street"
        },
        "name" : "Pier Side Cafe",
        "restaurant_id" : "41367417"
}
{
        "address" : {
                "coord" : [
                        -85.147576,
                        44.1418976
                ],
                "street" : "30 Avenue"
        },
        "name" : "Mexi Q Kitchen",
        "restaurant_id" : "41515837"
}
```

Question: 25. Write a MongoDB query to arrange the name of the restaurants in ascending order along with all the columns.

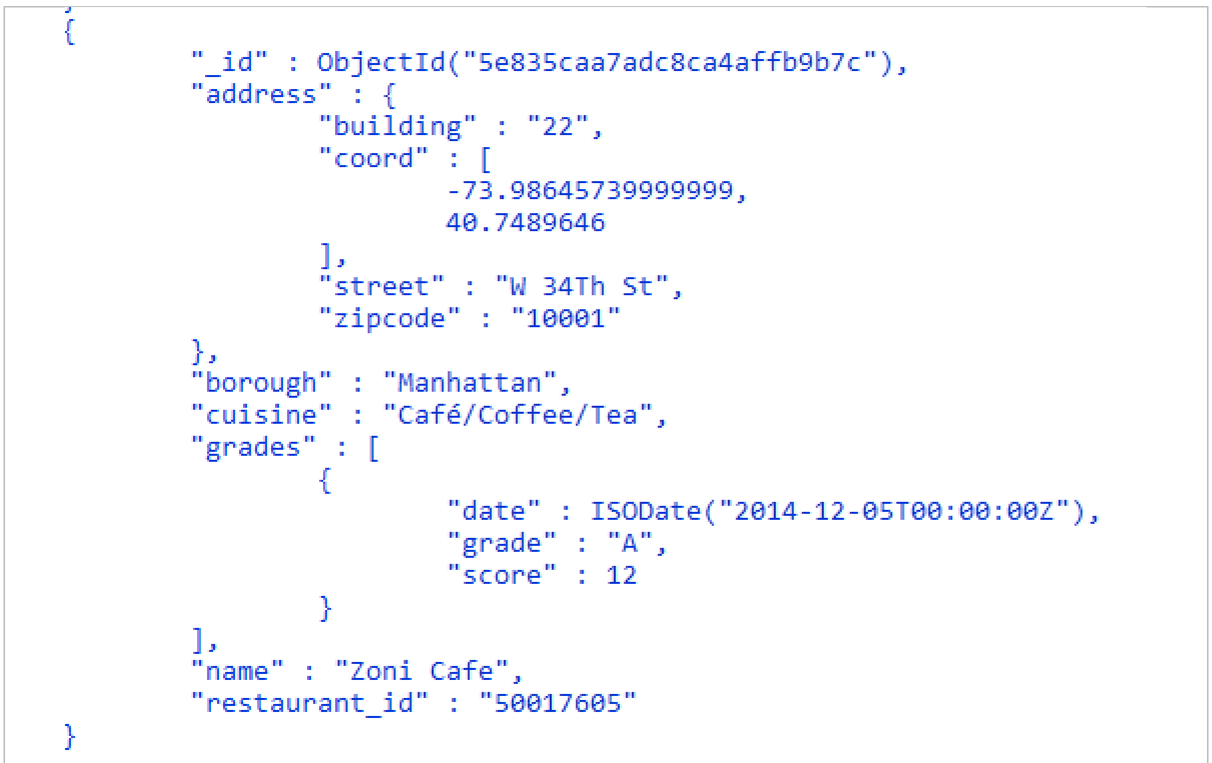
rest.find().sort({ name: 1 }).pretty();



```
                ],
                "street" : "Junction Blvd",
                "zipcode" : "11368"
        },
        "borough" : "Queens",
        "cuisine" : "Other",
        "grades" : [ ],
        "name" : "",
        "restaurant_id" : "50018095"
}
{
        "_id" : ObjectId("5e835caa7adc8ca4affb9c52"),
        "address" : {
                "building" : "889",
                "coord" : [
                        -73.9857624,
                        40.7684922
                ],
                "street" : "9Th Ave",
                "zipcode" : "10019"
        },
        "borough" : "Manhattan",
        "cuisine" : "Other",
        "grades" : [ ],
        "name" : "",
        "restaurant_id" : "50018100"
}
```

Question: 26. Write a MongoDB query to arrange the name of the restaurants in descending along with all the columns

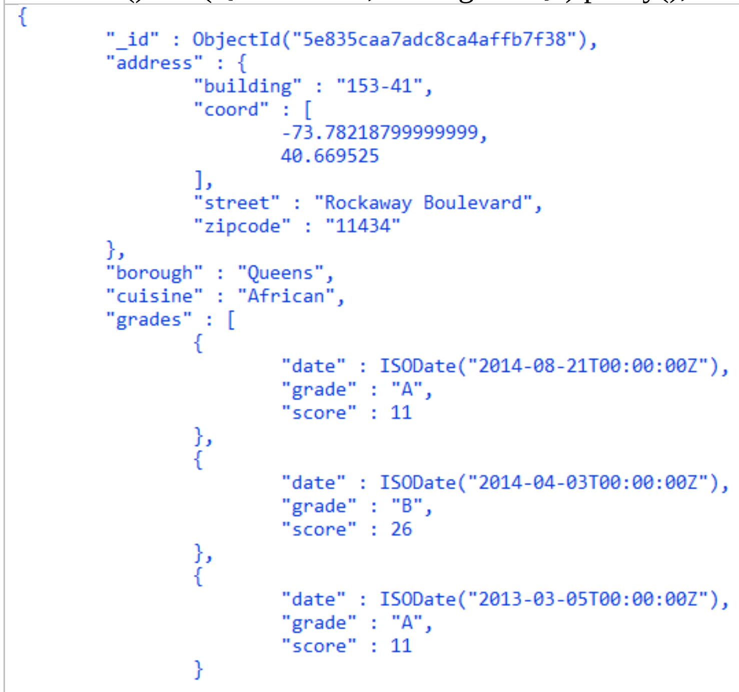
rest.find().sort({ name: -1 }).pretty();



```
{
        "_id" : ObjectId("5e835caa7adc8ca4affb9b7c"),
        "address" : {
                "building" : "22",
                "coord" : [
                        -73.98645739999999,
                        40.7489646
                ],
                "street" : "W 34Th St",
                "zipcode" : "10001"
        },
        "borough" : "Manhattan",
        "cuisine" : "Café/Coffee/Tea",
        "grades" : [
                {
                        "date" : ISODate("2014-12-05T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                }
        ],
        "name" : "Zoni Cafe",
        "restaurant_id" : "50017605"
}
```

Question: 27. Write a MongoDB query to arranged the name of the cuisine in ascending order and for that same cuisine borough should be in descending order.

```
rest.find().sort( { cuisine: 1, borough: -1 } ).pretty();
```



```
{
        "_id" : ObjectId("5e835caa7adc8ca4affb7f38"),
        "address" : {
                "building" : "153-41",
                "coord" : [
                        -73.78218799999999,
                        40.669525
                ],
                "street" : "Rockaway Boulevard",
                "zipcode" : "11434"
        },
        "borough" : "Queens",
        "cuisine" : "African",
        "grades" : [
                {
                        "date" : ISODate("2014-08-21T00:00:00Z"),
                        "grade" : "A",
                        "score" : 11
                },
                {
                        "date" : ISODate("2014-04-03T00:00:00Z"),
                        "grade" : "B",
                        "score" : 26
                },
                {
                        "date" : ISODate("2013-03-05T00:00:00Z"),
                        "grade" : "A",
                        "score" : 11
                }
```

Question: 28. Write a MongoDB query to know whether all the addresses contains the street or not.

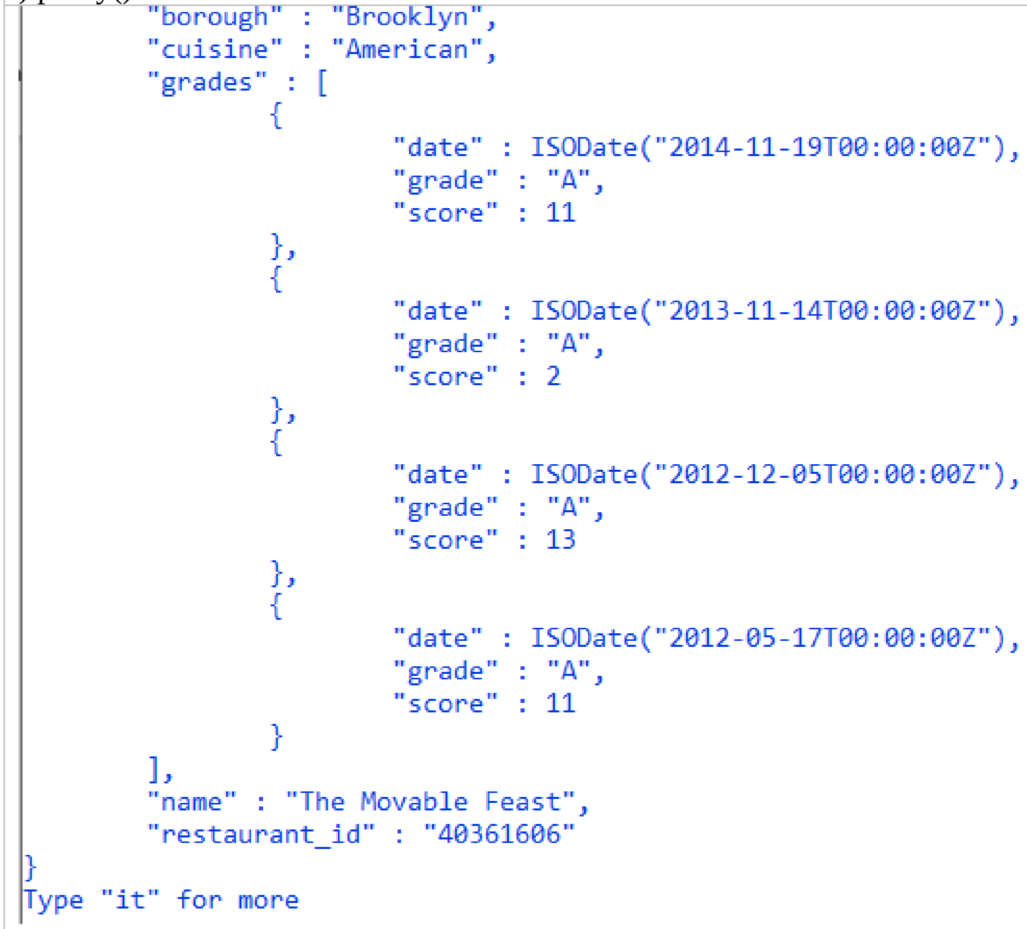
```
rest.find(
        {"address.street": { $exists: false }},
).pretty()
```

No records, meaning all the addresses contain street.

Question: 29. Write a MongoDB query which will select all documents in the restaurants collection where the coord field value is Double.

This question was skipped earlier because it was unknown what “Double” meant in the sense that whether it is the datatype, or it is 2 times the value or it is a duplicated value. Later it was assumed to be datatype.

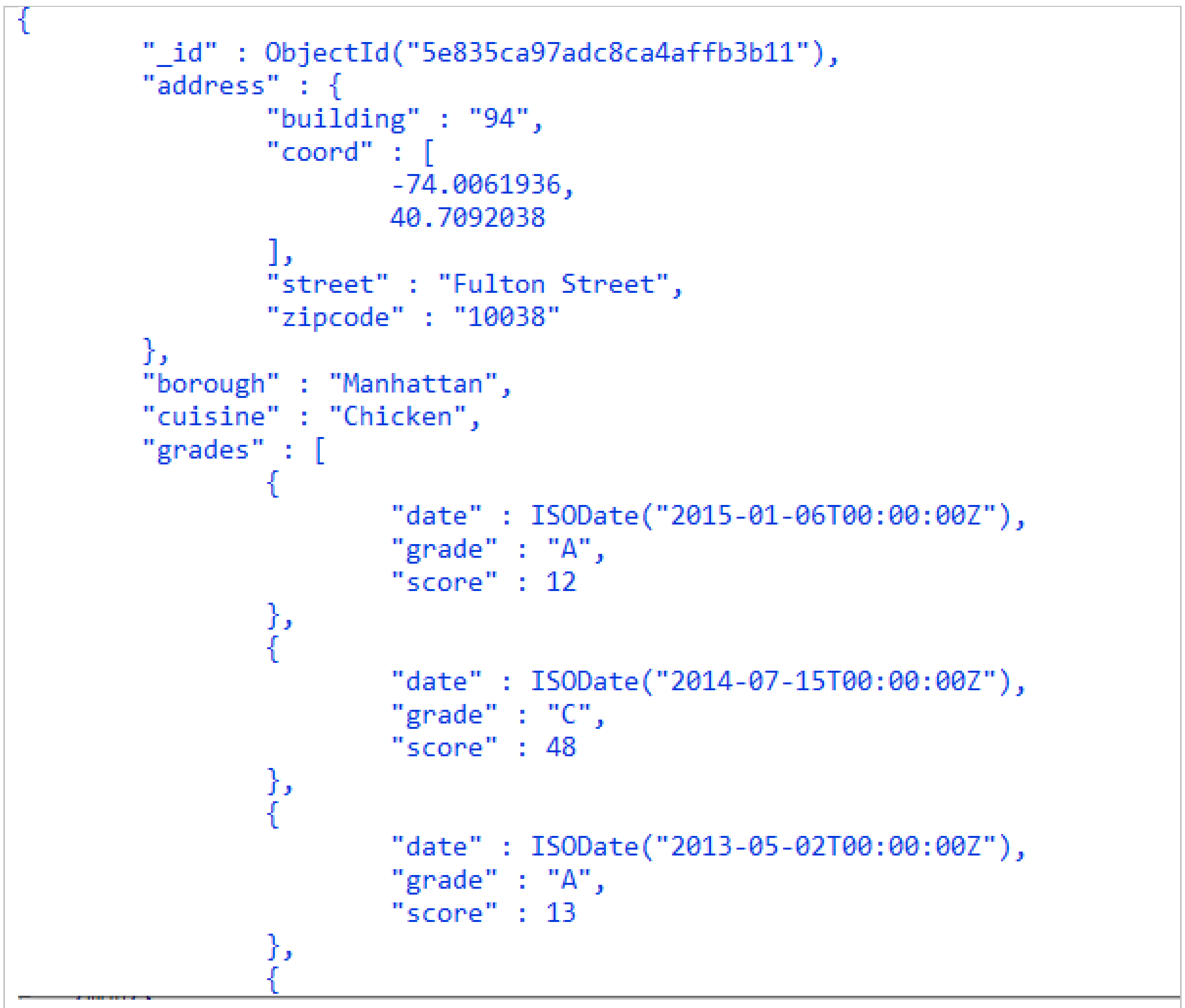
```
rest.find(
	{"address.coord": { $type: "double" }},
).pretty()
```



```
        "borough" : "Brooklyn",
        "cuisine" : "American",
        "grades" : [
                {
                        "date" : ISODate("2014-11-19T00:00:00Z"),
                        "grade" : "A",
                        "score" : 11
                },
                {
                        "date" : ISODate("2013-11-14T00:00:00Z"),
                        "grade" : "A",
                        "score" : 2
                },
                {
                        "date" : ISODate("2012-12-05T00:00:00Z"),
                        "grade" : "A",
                        "score" : 13
                },
                {
                        "date" : ISODate("2012-05-17T00:00:00Z"),
                        "grade" : "A",
                        "score" : 11
                }
        ],
        "name" : "The Movable Feast",
        "restaurant_id" : "40361606"
}
Type "it" for more
```

Question: 30. Write a MongoDB query which will select the restaurant Id, name and grades for those restaurants which returns 0 as a remainder after dividing the score by 7

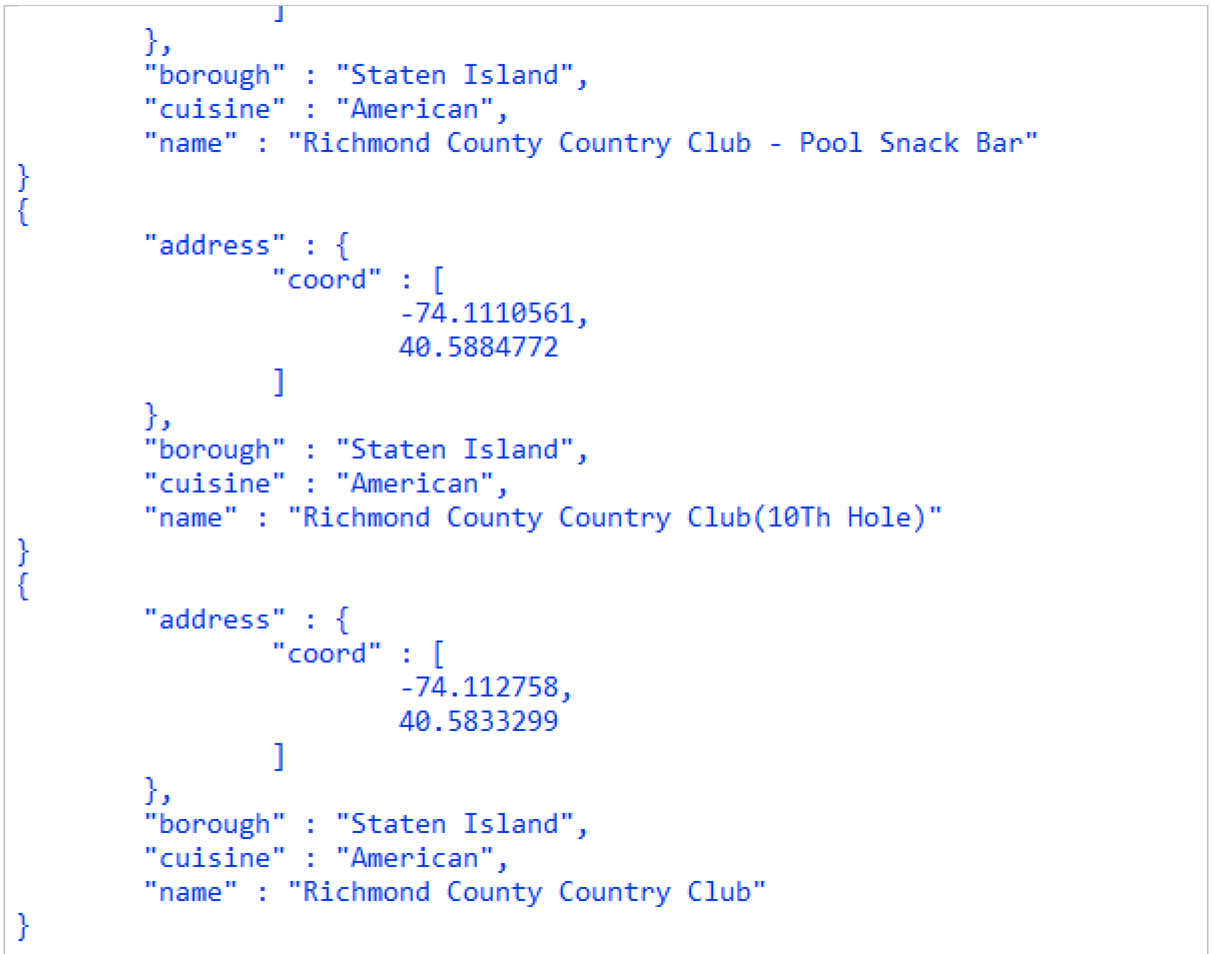
```
rest.find(
	{"grades.score": { $mod: [ 7, 0 ] }},
).pretty()
```



```
{
        "_id" : ObjectId("5e835ca97adc8ca4affb3b11"),
        "address" : {
                "building" : "94",
                "coord" : [
                        -74.0061936,
                        40.7092038
                ],
                "street" : "Fulton Street",
                "zipcode" : "10038"
        },
        "borough" : "Manhattan",
        "cuisine" : "Chicken",
        "grades" : [
                {
                        "date" : ISODate("2015-01-06T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                },
                {
                        "date" : ISODate("2014-07-15T00:00:00Z"),
                        "grade" : "C",
                        "score" : 48
                },
                {
                        "date" : ISODate("2013-05-02T00:00:00Z"),
                        "grade" : "A",
                        "score" : 13
                },
                {
```

Question: 31. Write a MongoDB query to find the restaurant name, borough, longitude and attitude and cuisine for those restaurants which contains 'mon' as three letters somewhere in its name.

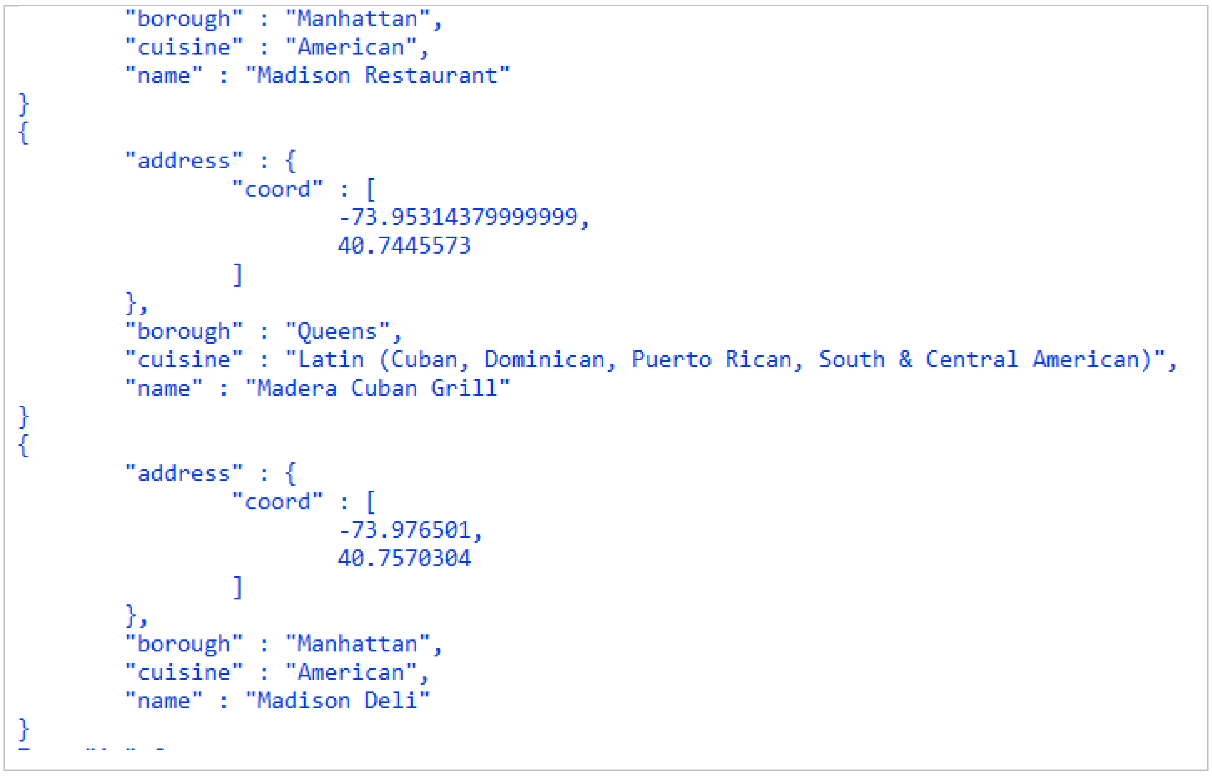
```
rest.find(
        {"name": /mon/},
        {"address.coord": 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
                ]
        },
        "borough" : "Staten Island",
        "cuisine" : "American",
        "name" : "Richmond County Country Club - Pool Snack Bar"
}
{
        "address" : {
                "coord" : [
                        -74.1110561,
                        40.5884772
                ]
        },
        "borough" : "Staten Island",
        "cuisine" : "American",
        "name" : "Richmond County Country Club(10Th Hole)"
}
{
        "address" : {
                "coord" : [
                        -74.112758,
                        40.5833299
                ]
        },
        "borough" : "Staten Island",
        "cuisine" : "American",
        "name" : "Richmond County Country Club"
}
```

Question: 32. Write a MongoDB query to find the restaurant name, borough, longitude and latitude and cuisine for those restaurants which contain 'Mad' as first three letters of its name.

```
rest.find(
        {"name": /^Mad/},
        {"address.coord": 1, name: 1, borough: 1, cuisine: 1, _id: 0}
).pretty()
```



```
        "borough" : "Manhattan",
        "cuisine" : "American",
        "name" : "Madison Restaurant"
}
{
        "address" : {
                "coord" : [
                        -73.95314379999999,
                        40.7445573
                ]
        },
        "borough" : "Queens",
        "cuisine" : "Latin (Cuban, Dominican, Puerto Rican, South & Central American)",
        "name" : "Madera Cuban Grill"
}
{
        "address" : {
                "coord" : [
                        -73.976501,
                        40.7570304
                ]
        },
        "borough" : "Manhattan",
        "cuisine" : "American",
        "name" : "Madison Deli"
}
```

Another tutorial was provided for practicing MongoDB. Some details were provided about a student database. The student collection/database consists of StdName, StdAddress, StdContactNo, Dateofjoin as keys, CourseEnrolled as array and CGPA, Grade and Status as objects.

student = db.students;

MONGODB EXERCISES II

Question: 1. Find the details of the student whose name is ‘Devoulder’.

```
student.find(
        {StdName: "Devoulder"},
        {}
).pretty()
```

Question: 2. When did Devoulder join as student?

```
student.find(
        {StdName: "Devoulder"},
        {Dateofjoin: 1, _id: 0}
).pretty()
```

Question: 3. Show the address and contact number only of Devoulder

```
student.find(
        {StdName: "Devoulder"},
        {StdAddress: 1, StdContactNo: 1, _id: 0}
).pretty()
```

Question: 4. List the course enrolled by Neuman

```
student.find(
        {StdName: "Neuman"},
        {StdName: 1, CourseEnrolled: 1, _id: 0}
).pretty()
```

Question: 5. What is the CGPA scored by Lee?

```
student.find(
        {StdName: "Lee"},
        {StdName: 1, CGPA: 1, _id: 0}
).pretty()
```

Question: 6. Find the average/max/min of the CGPA. Aggregation pipeline was referred to for answering this question.

```
student.aggregate([
  { "$group": {
    "_id": null,
    "max_cgpa": { "$max": "$CGPA" },
    "min_cgpa": { "$min": "$CGPA" },
    "avg_cgpa": { "$avg": "$CGPA" }
  }}
])
```

Question: 7. Find only the status and the name of the student who has scored more than 3.5 CGPA.

```
student.find(
        {CGPA: {$gt:90}},
        {StdName: 1, Status: 1, _id: 0}
).pretty()
```

Question: 8. List all the student whose has attain Grade “A”.

```
student.find(
        {Grade: 'A'},
        {StdName: 1, _id: 0}
).pretty()
```

Question: 9. Sort all the student according to the date of join first and then by CGPA.

```
student.find().sort( { Dateofjoin: 1, CGPA: 1 } ).pretty();
```

Question: 10. Find the number of student who has not enrolled COMP232 course.

```
student.find( {
                CourseEnrolled: { $not: { $elemMatch: {$eq: 'COMP32'}}}
```

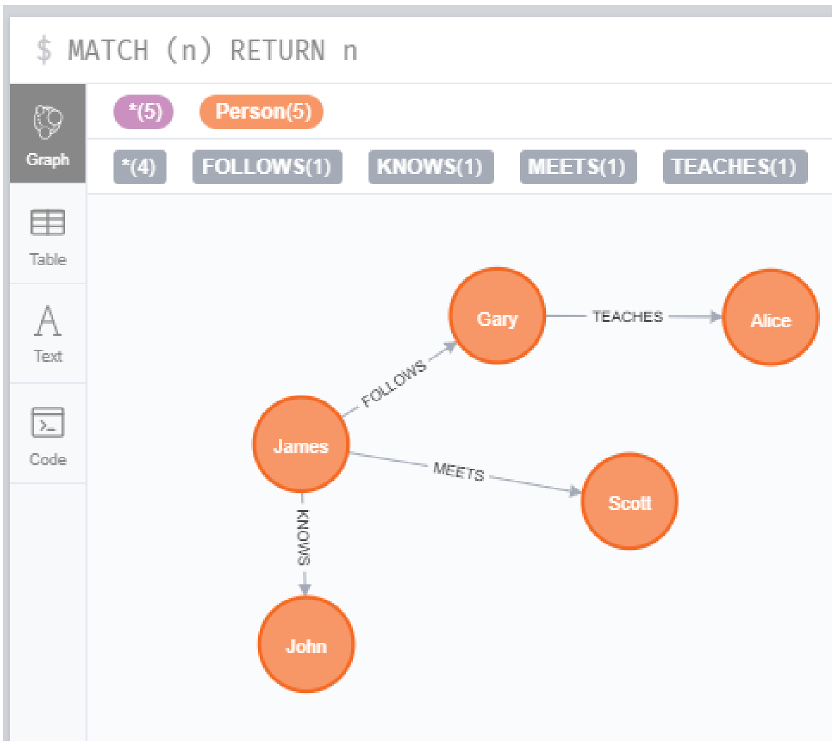
```
            },
            {}
).count()
```

Graph Modelling

This chapter elaborated more about graph modelling and graph databases in general. The nodes, properties, labels and relationships were discussed as to what are these used to represent in a database. It was discussed about deriving questions, identifying entities and relationships, producing a cypher path and then producing query commands from user stories. The conversion from a simple user story to a graph model was very interesting. Other discussions include when to and not to use relationships, different types of relationships and when to and not to use properties for a graph model.

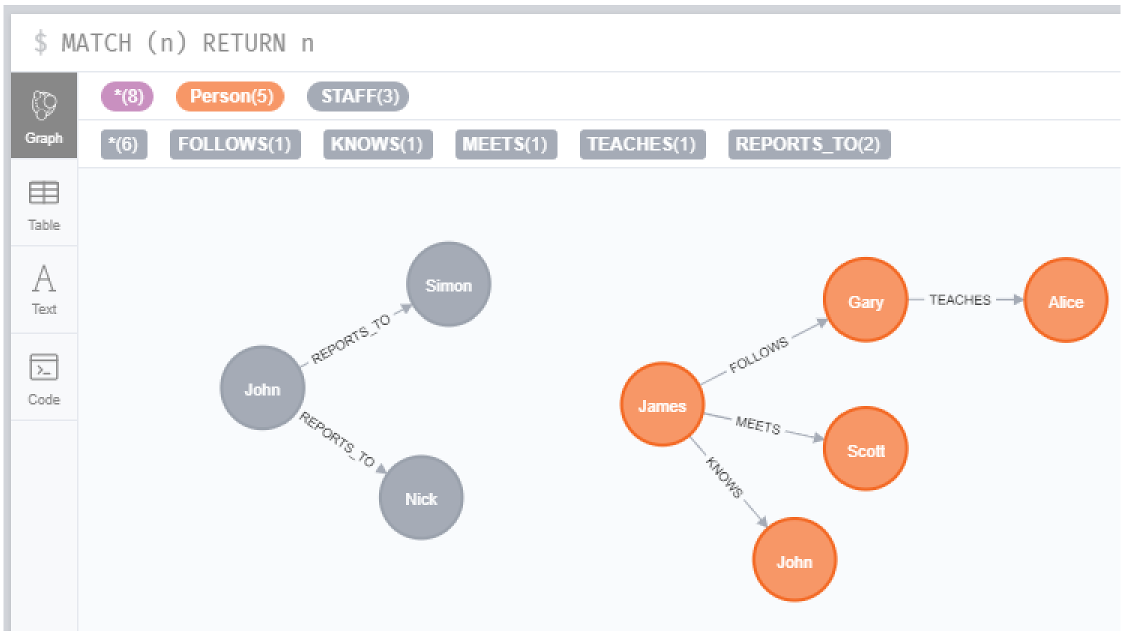
Using the provided model to implement it in Neo4j.

```
CREATE (james:Person{name:"James"})
CREATE (john:Person{name:"John"})
CREATE (scott:Person{name:"Scott"})
CREATE (gary:Person{name:"Gary"})
CREATE (alice:Person{name:"Alice"})
CREATE (james)-[:KNOWS]->(john)
CREATE (james)-[:MEETS]->(scott)
CREATE (james)-[:FOLLOWS]->(gary)
CREATE (gary)-[:TEACHES]->(alice)
```



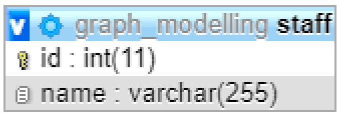
```
$ MATCH (n) RETURN n
```

```
CREATE (john:STAFF{name:"John"})
CREATE (simon:STAFF{name:"Simon"})
CREATE (nick:STAFF{name:"Nick"})
CREATE (john)-[:REPORTS_TO]→(simon)
CREATE (john)-[:REPORTS_TO]→(nick)
```

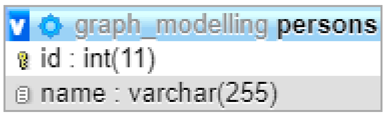


```
$ MATCH (n) RETURN n
```

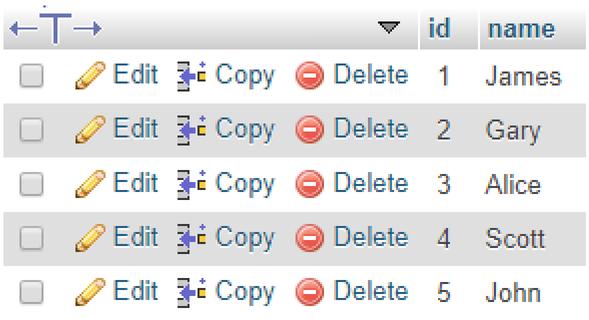
Similarly, staff and persons table was created in a relational database where the “Person” and “STAFF” data is added.



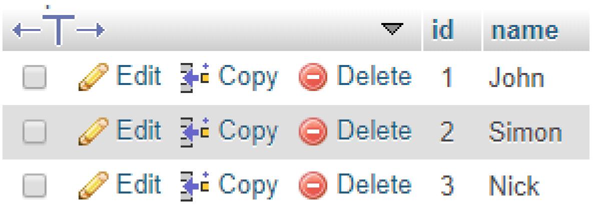
graph_modelling staff
- id : int(11)
- name : varchar(255)



graph_modelling persons
- id : int(11)
- name : varchar(255)



	id	name
Edit Copy Delete	1	James
Edit Copy Delete	2	Gary
Edit Copy Delete	3	Alice
Edit Copy Delete	4	Scott
Edit Copy Delete	5	John



	id	name
Edit Copy Delete	1	John
Edit Copy Delete	2	Simon
Edit Copy Delete	3	Nick

The graphical solution in Neo4j allows direct creation of nodes and relationships. The Person node, as mentioned in the rules to migrate to a graph database from an RDB, should be representing an entity and thus a table in the relational database. The same goes for staff node and table. Graph databases allow any type of relationship to be defined whereas in a relational database, to model this kind of situation, a recursive relationship is required where the type of relationship such as meets, knows, follows, teaches and reports to cannot be defined. This recursion induces further complexity. Also, if the staff entity and person entity both represented the same person, Neo4j allows to simply add Staff label to existing Person nodes who are also staffs using SET command. For a relational database, this should be done by introducing foreign keys and JOINs.

A link was provided to study and reflect on. The title of the page reads “7 Ways Your Data Is Telling You It’s a Graph”. The post mentions about how the first person loves relational databases and works with ERDs all day but agrees that they cannot be solutions to everything. They talk about the real-world scenario where people have a lot of different type of relationship between each other and designing a relational database for this would produce a very recursive database with a lot of self-referencing JOIN commands. They also mention how the lack of properties and labels is another disadvantage of relational database and that more than being about relationships, the relational databases deal more with constraints and data integrity. The first person mentions seven ways to determine whether the data is suitable to implement a graph database. It starts with the way people describe the data – if they describe it using words such as “network, tree, structure, ancestry, taxonomy”, they probably mean that relationship between the data is very important. Because graph databases deal with relationships a lot better than relational databases, which deal and focus on tables, they are the better choice. It can be understood that graph databases are very good under certain conditions mentioned in the page such as presence of recursion. The post ends with helpful links about learning and getting into graph databases and talking about how relationship between data can provide better understandings.

Transaction Management

This chapter discusses mostly about database transactions and maintenance of ACID principals by a DBMS. It was known that a transaction consists of many operations that are considered as a single block. Once any transaction is complete, it is committed using COMMIT and should still maintain a consistent state in the database.

Atomicity (all operations are reflected together but never only some operations that comprise a transaction), Consistency (consistent state to another consistent state), Isolation (partial effects of one transaction must not be visible to a parallel transaction), Durability (once committed, the changes should not be lost).

There are five stages of a database transaction. Active, partially committed, committed, failed and aborted.

The scheduling of transactions as part of transaction management was discussed to some level of detail. The two types of schedules are serial (operations of one transaction occurs consecutively instead of having interleaved) and non-serial. Some non-serial schedules can be converted to a serial schedule. This property of non-serial schedules is called serialisability.

Exercise:

W3(z) → R4(z) → W4(z) → R1(x) → R2(y) → W3(y) → R1(z) → W5(x) → R5(y) → R4(y) → W4(u) → R5(v).

The transactions:

T1: R1(x) R1(z)

T2: R2(y)

T3: W3(z) W3(y)

T4: R4(z) W4(z) R4(y) W4(u)

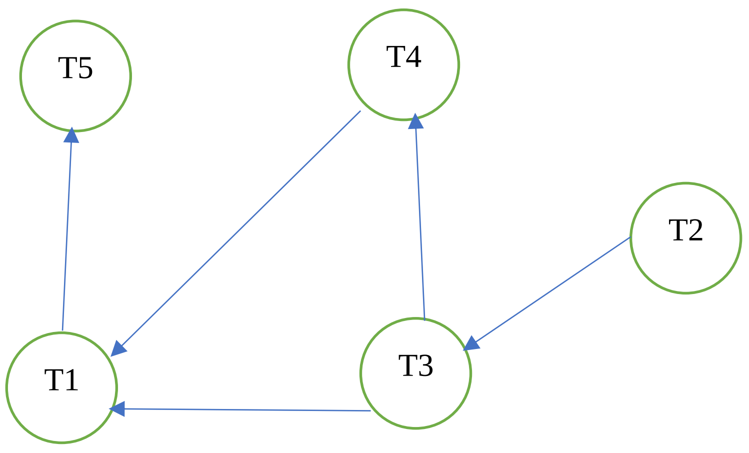
T5: W5(x) R5(y) R5(v).

T1	T2	T3	T4	T5
		W3(z)		
			R4(z)	
			W4(z)	
R1(x)				
	R2(y)			
		W3(y)		
R1(z)				
				W5(x)
				R5(y)
			R4(y)	
			W4(u)	
				R5(v)

Conflict pairs in the schedule:

< W3(z), R4(z)>, < W3(z), W4(z)>, < W3(z), R1(z)>, < W4(z), R1(z)>, < R1(x), W5(x) >, < R2(y), W3(y)>, < W3(y), R4(y)>

The precedence graph:


$$\begin{tikzcd}
T5 & & T4 \arrow[dll] & \\
 & & & T2 \arrow[dl] \\
T1 \arrow[uu] & & T3 \arrow[uu] \arrow[ll] &
\end{tikzcd}$$

Conflict Serialisability:

There is no cycle here thus, it is conflict serializable.

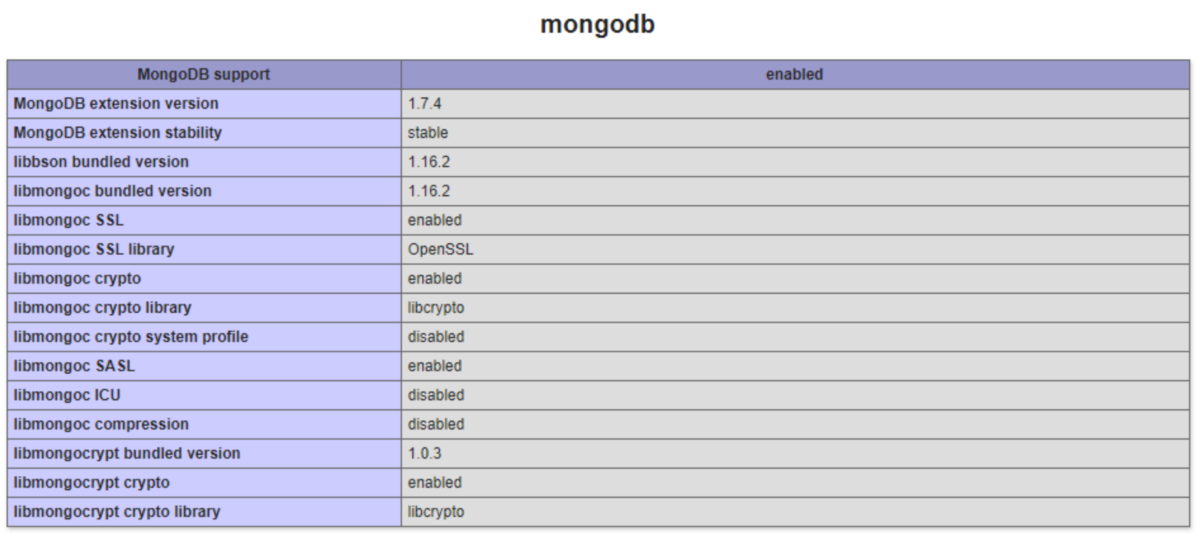
Example serializable schedule:

T2 – T3 – T4 – T1 – T5

MongoDB With PHP and Java

The activities from this week are included above as MongoDB exercises part II. This section will discuss about setting up and using MongoDB with PHP which was done practically as part of this chapter.

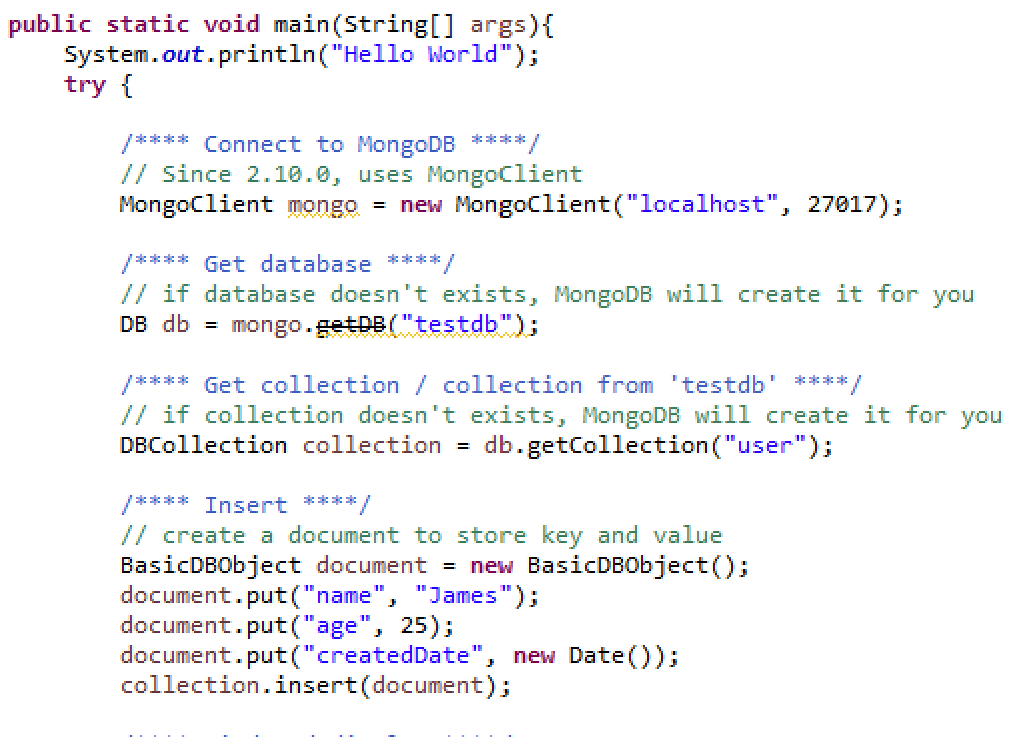
The MongoDB driver (DLL file) for PHP was downloaded and extracted to the php\ext folder. The phpinfo file was checked to ensure that MongoDB was set up correctly.



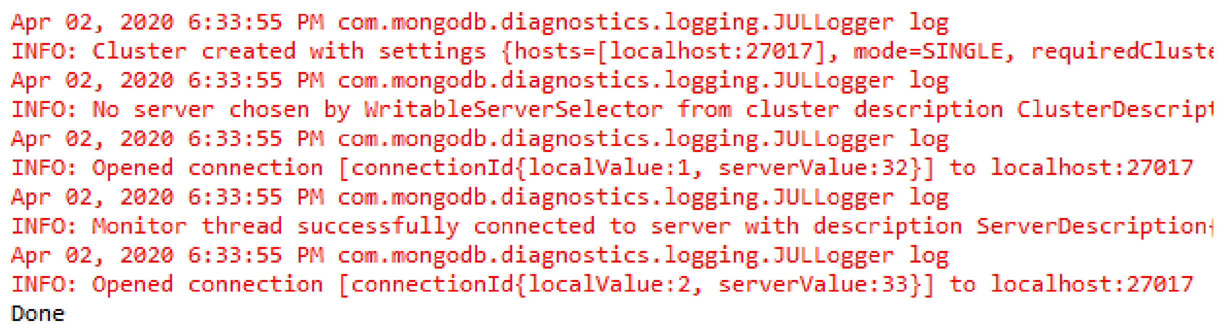
mongodb

MongoDB support	enabled
MongoDB extension version	1.7.4
MongoDB extension stability	stable
libbson bundled version	1.16.2
libmongoc bundled version	1.16.2
libmongoc SSL	enabled
libmongoc SSL library	OpenSSL
libmongoc crypto	enabled
libmongoc crypto library	libcrypto
libmongoc crypto system profile	disabled
libmongoc SASL	enabled
libmongoc ICU	disabled
libmongoc compression	disabled
libmongocrypt bundled version	1.0.3
libmongocrypt crypto	enabled
libmongocrypt crypto library	libcrypto

For Java setup, maven was not setup and was thus installed first. After setting up maven, the tutorial was followed, and the Java code was prepared into the main method. The first operation is for insertion.

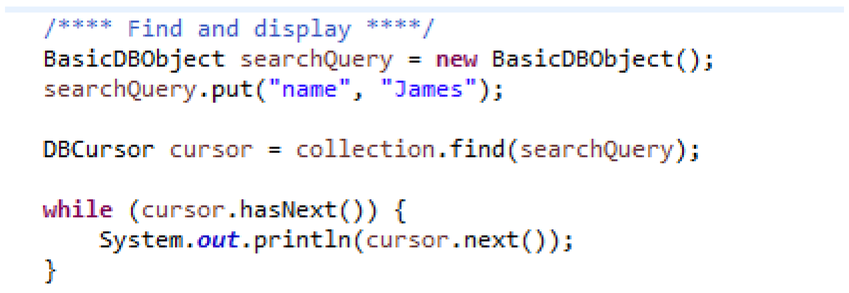


```
public static void main(String[] args){
    System.out.println("Hello World");
    try {

        /**** Connect to MongoDB ****/
        // Since 2.10.0, uses MongoClient
        MongoClient mongo = new MongoClient("localhost", 27017);

        /**** Get database ****/
        // if database doesn't exists, MongoDB will create it for you
        DB db = mongo.~~getDB~~("testdb");

        /**** Get collection / collection from 'testdb' ****/
        // if collection doesn't exists, MongoDB will create it for you
        DBCollection collection = db.getCollection("user");

        /**** Insert ****/
        // create a document to store key and value
        BasicDBObject document = new BasicDBObject();
        document.put("name", "James");
        document.put("age", 25);
        document.put("createdDate", new Date());
        collection.insert(document);
```



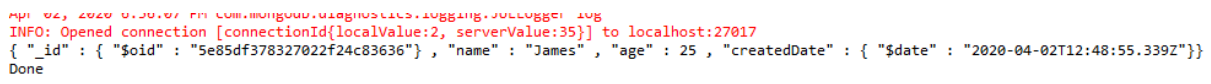
```
Apr 02, 2020 6:33:55 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Cluster created with settings {hosts=[localhost:27017], mode=SINGLE, requiredCluste
Apr 02, 2020 6:33:55 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: No server chosen by WritableServerSelector from cluster description ClusterDescript
Apr 02, 2020 6:33:55 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:1, serverValue:32}] to localhost:27017
Apr 02, 2020 6:33:55 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Monitor thread successfully connected to server with description ServerDescription{
Apr 02, 2020 6:33:55 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:2, serverValue:33}] to localhost:27017
Done
```

After running the code, “Done” was displayed in the command which means we successfully performed the insertion.



```java
/**** Find and display ****/
BasicDBObject searchQuery = new BasicDBObject();
searchQuery.put("name", "James");

DBCursor cursor = collection.find(searchQuery);

while (cursor.hasNext()) {
    System.out.println(cursor.next());
}
```

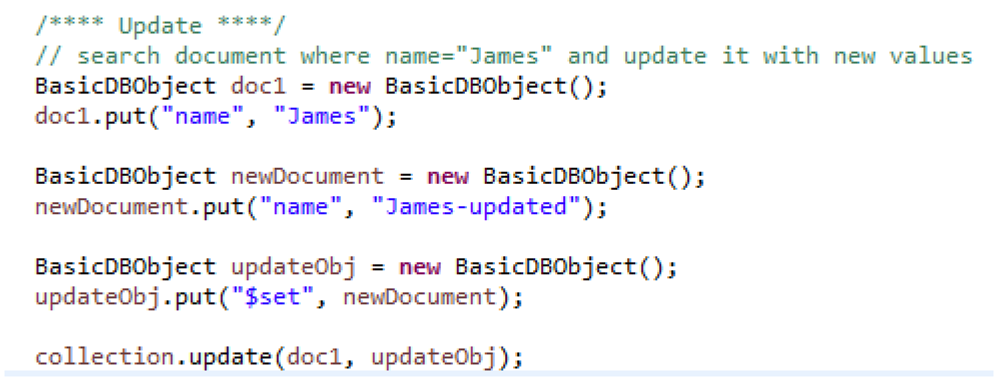
We can now run the code to display records.



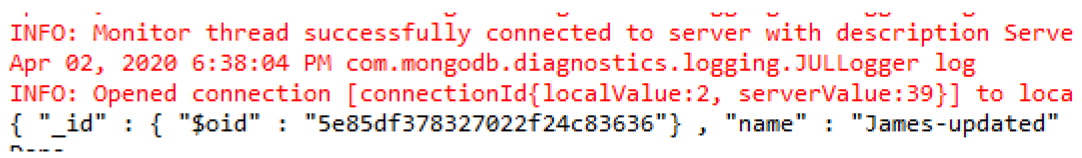
```
Apr 02, 2020 [illegible] PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:2, serverValue:35}] to localhost:27017
{ "_id" : { "$oid" : "5e85df378327022f24c83636"} , "name" : "James" , "age" : 25 , "createdDate" : { "$date" : "2020-04-02T12:48:55.339Z"}}
Done
```

The data inserted earlier is displayed successfully, followed by the “Done” message which means no exceptions occurred.

UPDATE

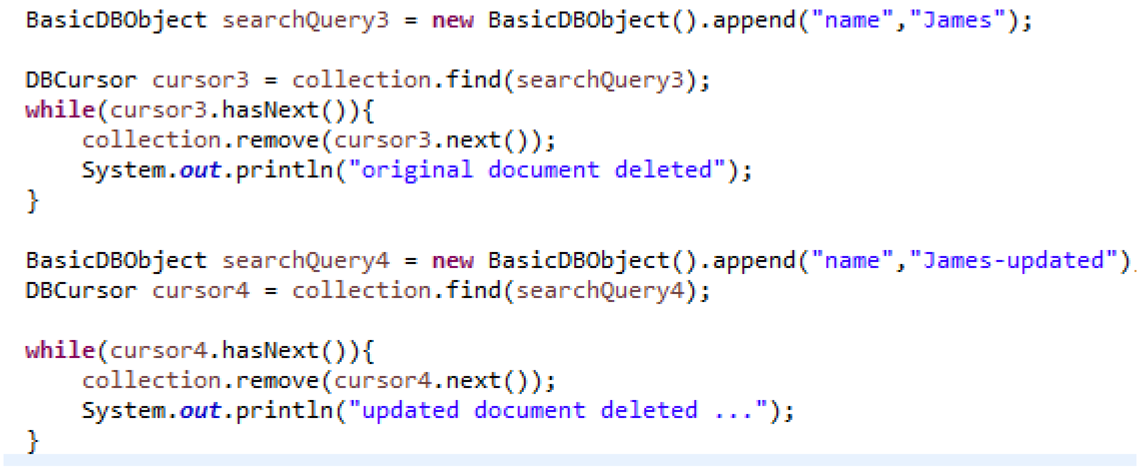


```java
/**** Update ****/
// search document where name="James" and update it with new values
BasicDBObject doc1 = new BasicDBObject();
doc1.put("name", "James");

BasicDBObject newDocument = new BasicDBObject();
newDocument.put("name", "James-updated");

BasicDBObject updateObj = new BasicDBObject();
updateObj.put("$set", newDocument);

collection.update(doc1, updateObj);
```

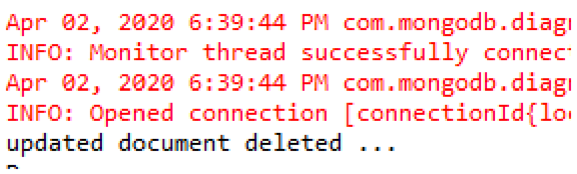


```
INFO: Monitor thread successfully connected to server with description Serve
Apr 02, 2020 6:38:04 PM com.mongodb.diagnostics.logging.JULLogger log
INFO: Opened connection [connectionId{localValue:2, serverValue:39}] to loca
{ "_id" : { "$oid" : "5e85df378327022f24c83636"} , "name" : "James-updated"
```

After updating, the name is now updated to James-updated. Similarly, deletion was performed at last



```
BasicDBObject searchQuery3 = new BasicDBObject().append("name","James");

DBCursor cursor3 = collection.find(searchQuery3);
while(cursor3.hasNext()){
    collection.remove(cursor3.next());
    System.out.println("original document deleted");
}

BasicDBObject searchQuery4 = new BasicDBObject().append("name","James-updated")
DBCursor cursor4 = collection.find(searchQuery4);

while(cursor4.hasNext()){
    collection.remove(cursor4.next());
    System.out.println("updated document deleted ...");
}
```



```
Apr 02, 2020 6:39:44 PM com.mongodb.diag
INFO: Monitor thread successfully connec
Apr 02, 2020 6:39:44 PM com.mongodb.diag
INFO: Opened connection [connectionId{lo
updated document deleted ...
```