DM556 - Principle of Database Systems
Project 2: NoSQL

Jonas I. Sørensen <joso216>
Simon D. Jørgensen <simjo16>
Jeff Gyldenbrand <jegyl16>

Supervisor: Jacob A. Mikkelsen

July 20, 2018

Contents

1 Introduction

The objective of the assignment is to implement the storage layer for a *NoSQL* database application of optional choice. This could be; *Cassandra*, *Firebase*, *MongoDB*, *Neo4j*, or any other *NoSQL* application.

More specific the task is to make a database interface that deals with a rating app for gin and tonics. These methods should include ways to handle the creating of a non existing gin, tonic or garnish and a search method of said components. Furthermore a given user must be able to put a rating and an evaluation for a combination of a gin and a tonic, potentially with a garnish too. This could be denoted as a drink.

From a given gin, tonic or garnish, it should be possible to retrieve a list of ratings / comments, an average rating, and number of users having rated said drink combinations.

Lastly, it should be possible to mark a rating as helpful, and sort the ratings after the number of helpful marks. These are the problems sought to be solved. In the next section, the design choices of the problems will be explained.

2 Design

Because relations between items is an important factor in this project, an ideal approach would have been to use a relational database, such as *PostgreSQL*. As such, per requirement of the assignment, a *NoSQL* database system handling relations is required.

As per requirement of the assignment, the database is to have 3 unique lists of items, namely gin, tonic and garnish. These 3 lists are to be used for the combinations of the drinks, which can be as large as:

$$|gin| \cdot |tonic| \cdot (|garnish| + 1) \tag{1}$$

However, the real number of relevant combinations are far below that, since the odds of having all possible combinations for display, would be extremely low.

2.1 NoSQL choice

For this design, *Neo4j* was a perfect choice. It handles relations by itself and can show relations between users, depending on their interest in the same item. Furthermore, for future planning, a recommendation system was discussed and brainstormed. The idea behind it, was to enable a way to show a user a recommendation of a drink, based on a relation between that user and others with same rating on the same drink. Using a graph based database system, such as *Neo4j*, allows for such an recomendation system.

2.2 Database design

In designing this database, it is important to realise, that the database is graph based. This means that objects has to be thought of as nodes and relations as edges.
When categorising the items in a graph database, items are marked with one or more labels. Thus achieving an ability to differentiate between nodes, the same holds true for edges.

In the example of this assignment, one could create the labels: Gin, Tonic, Garnish and User. These are the four main labels, and how they interact is the design process.
When creating the drinks, a naive approach would be, to have edges between the users and the gins/tonics. This however handicaps the ability to tell the difference between edges. However because drinks require edges to users and ratings, it has to be a node. The same approach can be taken when determining if a rating should be a edge or a node. Does a rating have a edge, if yes then it is a node, otherwise it is a edge. The helpful mark, however has no edge at any point, as such it shall be a edge.

To summarise the relations in the graph:

- A user U relates to N ratings R such that : $U \rightarrow R_n$
- A user U has M helpful relations H which maps to a given rating R : $U \xrightarrow{H_m} R$
- A rating R relates to 1 drink D : $R \rightarrow D$
- A drink D relates to 1 gin G : $D \rightarrow G$
- A drink D relates to 1 tonic T : $D \rightarrow T$
- A drink D possibly relates to 1 garnish A : $D? \rightarrow A$

An example for this graph can be seen at figure 8.3

3 Implementation

It was chosen to implement queries, that fulfil the assignment requirements, as simple as possible. For this, the programming language *python* was found most ideal, as it manages string manipulation fairly efficient. When writing the code, it is important to remember the structure of a node in the *Neo4j* database system. Any node is given by a number of labels, and a dictionary holding the information of the node, such as name and age for users. This dictionary is, for simplicity sake, used to search for a given node. An example for this could be the merge function in the program:

```
1    def merge(self,*label,**dict):
2        l = "".join(":{}".format(l) for l in label)
3        self.transaction("Merge ({l} {d})",l=l,d = dict)
4        return dict
```

This also means that most of the variables in the program, such as a user or gin, is a dictionary.

3.1 Queries

The queries that fulfil the assignment requirements, can be found in section 8.4, they will also be explained in this section. To fulfil the first requirement, a query which searches the database for an item fitting the description, needs to be created. *Neo4j* already support such a feature with the *Merge* statement. The *Merge* statement searches the database for a specific pattern, if it doesn't find it, it creates said pattern.
This means that the *Merge* statement can be used to great effect in creating unique items in the database. This means that a query for creating a unique gin could be: *"Merge (:Gin{name =...})"*, or any unique element for that matter.

As stated before, it is pessimal, to construct all combinations. Therefore a combination, should only be created if it has a rating attached. Thus it is only required to create a combination during rating. However it might occur that a combination exists, hereby requiring the use of the *Merge* function.

3.2 Performance

When searching for patterns in the database, performance, although not essential is still vital. As discussed previously, a combination should only be created if it has a rating attached, to avoid a Cartesian product.
However, this will be for naught, if the queries searching for the combination perform Cartesian products. To prevent this, it is possible to index a search. Such that, the run time for a search between disconnected patterns, becomes significantly faster. In *Neo4j*, this is achieved by separating the match statements.

4 Testing

Seven tests have been developed to show that everything works as intended. Initially the test-script clears the database before any test is run. All test outputs are found in section 7.2 as figure 1-7. When refered to code, see section 7.4 under test.py.

The first test outputs everything that is in the database. Before test 1 is run, three users, one gin, one tonic and one garnish is created. Users have the attributes; name, age and gender. Gin has; name, producer, volume and alcohol percentage. Tonics has; name, producer, volume. Finally, garnish has only name as attribute. See code, line 11-20. Output of test 1 is shown in figure 1, where *nodeid* is the index of the hash-table in which *Neo4j* searches for a node in a graph, and *label* is the name of the table. *Properties* is the dictionary of values related to a given node.

The second test shows ratings by users on given drinks. See code, line 31-41, where e.g., line 34, a user "jeff" rates a given gin and tonic with the value 4, and comment "Needs more pickles." If no such rating exists beforehand, it will be created, else if it exist, it returns the rating. See figure 2 for output on test 2.

The third test shows when a user finds a given rating helpful. See code, line 44-55. On line 49, user "Simon" rates a drink, and on line 50, user "Jeff" set this rating as helpful. See figure 3 for output on test 3.

The fourth test shows a search in the database on everything with label "Gin", every percentage equal to "50" and every percentage in range 50-99, using regular expressions. The two latter wont return anything as output

because they do not exists in the database. Furthermore the test searches for everything from the producer "schweppes", every garnish, and finally everything from user "Jonas". See code, line 57-66. For output of test 4, see figure 4.

The fifth test is to show the possibility to search for rating on drinks. See code, line 73, where the search query is for all gin and tonics. As shown in figure 5, the output is the rating that fulfils the query, and the ratings attributes: one list with; count of ratings, the average of all ratings, and one list with; comment and a rating, and finally the count of helpful marks. On code line 74, the same search is given, but outputs it ordered by helpful marks from users in descending order. Because there is to few combination of drinks created in the database, its not possible to show an example with an output with descending order, hence the output appears to be printed double. Feel free to add more drinks, users, ratings and helpful marks, to see output with descending order.

The sixth tests simply returns all ratings from a given user. See code, line 85-87, where a query on "Jonas", "Simon" and "Jeff" is sent to the database. See figure 6 for output, where both comment and rating, and number of helpful-marks is shown, for a given user.

The seventh and final test outputs all ratings, and shows that its possible to sort of different parameters in ratings. See figure 7 for output.

5 Conclusion

The storage layer has been successfully implemented for a *NoSQL* database, where the choice fell on *Neo4j* with the programming language python. It is possible to create gins, tonics, garnishes, users and set their relations to form drinks. Furthermore it is possible to rate drinks with scores between 1 and 5, and comment them. It is also possible to retrieve a list of ratings and comments, an get the average of a rating, and the number of users who rated a drink. Lastly it is possible to mark a rating as helpful, and sort the ratings after the number of helpful marks. All requirements are met.

6 Division of Labor

The group work have been unified for both code, test and the report.
Jeff initially proposed *Firebase* as *NoSQL* database, as he has experience with this, but based on a conversation Jonas and Simon had with the supervisor, the decision fell on *Neo4j*.

Jonas is the most experienced programmer in the group, and did the major part of the programming. However, Simon and Jeff have been included during the coding from start to finish and made suggestions for implementation.

The report itself is made together, where each section has been discussed and written together.

7 Evaluation

Due to the free nature of the project, it was found to be more enjoyable and educational, then previous projects. The practiality of the project also felt more business oriented.

8 Appendix

8.1 How to install and run the application

Note: the script only works with Ubuntu Linux. It is required that python3 and pip3 is installed.

First install and run the *Neo4j* database. Then compile the inter-script and then run the test-script. This will output all tests made. Its possible to see a visual representation of the database and relations in a web-browser.

- Terminal: "sudo ./install_neo4j.sh"
- Terminal: "neo4j start"
- Terminal: "python3 inter.py"
- Terminal: "python3 test.py"
- Web-browser: "localhost:7474/browser/"
- Neo4j browser: "$ match (u) return u"

8.2 Test output

Figure 1: Result output of test 1:

```
TEST  1
(<Node id=543 labels={'User'} properties={'gender': 'Male', 'age': 21, 'name': 'Jonas Ingerslev Sørensen'}>,)
(<Node id=544 labels={'User'} properties={'gender': 'Apache Attack Helicopter', 'age': 33, 'name': 'Jeff Gyldenbrand'}>,)
(<Node id=545 labels={'User'} properties={'gender': 'Emu', 'age': 26, 'name': 'Simon Dradrach Jørgensen'}>,)
(<Node id=546 labels={'Gin'} properties={'volume': 70, 'producer': 'BOMBAY SPIRITS', 'percentage': 40, 'name': 'BOMBAY SAPPHIRE'}>,)
(<Node id=547 labels={'Tonic'} properties={'volume': 33, 'producer': 'SCHWEPPES', 'name': 'TONIC WATER'}>,)
(<Node id=548 labels={'Garnish'} properties={'type': 'LIME'}>,)
```

Figure 2: Result output of test 2:

```
TEST  2
(<Node id=556 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, <Node id=555 labels=set() properties={}>)
(<Node id=557 labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, <Node id=555 labels=set() properties={}>)
(<Node id=559 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, <Node id=558 labels=set() properties={}>)
```

Figure 3: Result output of test 3:

```
TEST  3
(<Relationship id=475 start=561 end=570 type='Helpfull' properties={}>,)
```

Figure 4: Result output of test 4:

```
TEST  4
(<Node id=574 labels={'Gin'} properties={'volume': 70, 'producer': 'BOMBAY SPIRITS', 'percentage': 40, 'name': 'BOMBAY SAPPHIRE'}>,)
(<Node id=575 labels={'Tonic'} properties={'volume': 33, 'producer': 'SCHWEPPES', 'name': 'TONIC WATER'}>,)
(<Node id=576 labels={'Garnish'} properties={'type': 'LIME'}>,)
```

Figure 5: Result output of test 5:

```
TEST  5
[(2, 2.5)]
[(<Node id=741 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0), (<Node id=742
labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)]

[(2, 2.5)]
[(<Node id=741 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0), (<Node id=742
labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)]

[(1, 3.0)]
[(<Node id=744 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)]

[(1, 3.0)]
[(<Node id=744 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)]
```

Figure 6: Result output of test 6:

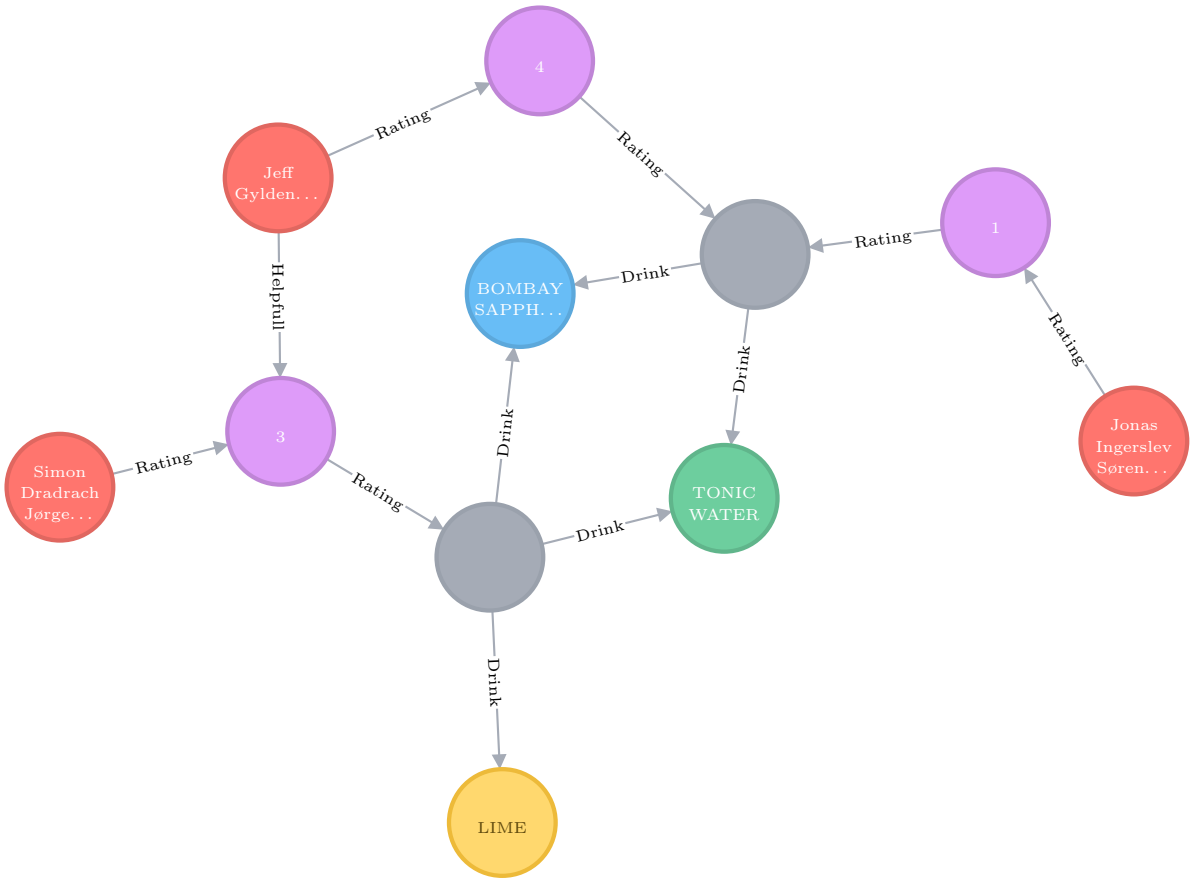
```
TEST  6
(<Node id=601 labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)
(<Node id=603 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)
(<Node id=600 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0)
```

Figure 7: Result output of test 7:

```
TEST  7
(<Node id=606 labels={'User'} properties={'gender': 'Emu', 'age': 26, 'name': 'Simon Dradrach Jørgensen'}>,
<Node id=614 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)

(<Node id=604 labels={'User'} properties={'gender': 'Male', 'age': 21, 'name': 'Jonas Ingerslev Sørensen'}>,
<Node id=612 labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)

(<Node id=605 labels={'User'} properties={'gender': 'Apache Attack Helicopter', 'age': 33, 'name': 'Jeff
Gyldenbrand'}>, <Node id=611 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0)

(<Node id=605 labels={'User'} properties={'gender': 'Apache Attack Helicopter', 'age': 33, 'name': 'Jeff
Gyldenbrand'}>, <Node id=611 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0)

(<Node id=606 labels={'User'} properties={'gender': 'Emu', 'age': 26, 'name': 'Simon Dradrach Jørgensen'}>,
<Node id=614 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)

(<Node id=604 labels={'User'} properties={'gender': 'Male', 'age': 21, 'name': 'Jonas Ingerslev Sørensen'}>,
<Node id=612 labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)

(<Node id=606 labels={'User'} properties={'gender': 'Emu', 'age': 26, 'name': 'Simon Dradrach Jørgensen'}>,
<Node id=614 labels={'Rating'} properties={'comment': 'Tastes like Australian tears', 'rating': 3}>, 1)

(<Node id=604 labels={'User'} properties={'gender': 'Male', 'age': 21, 'name': 'Jonas Ingerslev Sørensen'}>,
<Node id=612 labels={'Rating'} properties={'comment': "It's a shit!", 'rating': 1}>, 0)

(<Node id=605 labels={'User'} properties={'gender': 'Apache Attack Helicopter', 'age': 33, 'name': 'Jeff
Gyldenbrand'}>, <Node id=611 labels={'Rating'} properties={'comment': 'Needs more pickles.', 'rating': 4}>, 0)
```

8.3 Figures and illustrations

Figure 8: Database with relations



8.4 Source-code

inter.py

```
from neo4j.v1 import GraphDatabase
import sys

#Python puts quotes around strings, neo4j wont accept strings around dictionary keys.
class QueObj(dict):
    @staticmethod
    def isdict(d):
        t = type(d)
        if issubclass(t,dict) and t is not QueObj:
            return QueObj(d)
        return d

    def __init__(self,*arg,**kwarg):
        f = lambda x : {k : QueObj.isdict(x[k]) for k in x}
        arg = [f(l) for l in arg]
        kwarg = f(kwarg)
        super(QueObj,self).__init__(*arg,**kwarg)

    def __repr__(self):
        return "{{{}}}".format (
            ", " .join( "{} : {!r}".format(k,self[k]) for k in self)
        )

class Neo4jDB:
    def __init__ (self, uri = "bolt://localhost:7687", user = "neo4j", password = "password"):
        self.driver = GraphDatabase.driver(uri,auth=(user,password))

    def __exit__ (self,exc_type, exc_value, traceback):
        self.driver.close()

    # To prevent faulty queries.
    def transaction(self,query,**dict):
        a = QueObj(dict)
        #print(query.format(**a))
        with self.driver.session() as session :
            with session.begin_transaction() as tx:
                return tx.run(query.format(**a))


    #Create a node if it does not exists.
    def merge(self,*label,**dict):
```

```
        l = "".join(":{}".format(l) for l in label)
        self.transaction("Merge ({l} {d})",l=l,d = dict)
        return dict

    #Create a wrapper function around merge for a gin.
    def new_gin(self,name,producer,volume,percentage):
        return self.merge("Gin",
            name = name.upper(),
            producer = producer.upper(),
            volume = volume,
            percentage = percentage
        )
    #Create a wrapper function around merge for a tonic.
    def new_tonic(self,name,producer,volume):
        return self.merge("Tonic",
            name = name.upper(),
            producer = producer.upper(),
            volume = volume
        )
    #Create a wrapper function around merge for a garnish.
    def new_garnish(self,type):
        return self.merge("Garnish",
            type = type.upper()
        )

    #Create a user if it doesn't exists.
    def new_user(self, name, age, gender):
        return self.merge("User",
            name = name,
            age = age,
            gender = gender
        )


    # Searches the graph for a drink, or creates it if it doesn't exist. It then rates the drink.
    def drink(self, user, gin, tonic, garnish="", rating = None, comment = None):

        dict = {}
        if rating and 0 <= rating <= 5 :
            dict['rating'] = rating
        if comment :
            dict['comment'] = comment

        if garnish:
            garnish = "Match (a:Garnish{}) Merge (a) <- [:Drink] -(d)".format(QueObj(garnish))

```

```
88          self.transaction( """
89              Match (g:Gin{gin})
90              Match (t:Tonic{tonic})
91              Match (u:User {user})
92              {garnish}
93              Merge (g) <- [:Drink] -(d)
94              Merge (t)  <- [:Drink] -(d)
95              Merge (u) - [:Rating] -> (r:Rating) - [:Rating] -> (
    d)
96              Set r += {dict}
97          """,
98              garnish = garnish,
99              gin = gin,
100             tonic = tonic,
101             user = user,
102             dict = dict
103         )
104         return dict
105
106     #Searches for a node with the given labels, and where the
    nodex matches the regular expression given from dictionary
    such that for (K:V) n.k = regex(V)
107     def search(self,*labels,**regexdict):
108         where = " and ".join("a.{} =~ {!r}".format(k,str(v)) for
     k,v in regexdict.items())
109         if where:
110             where = "Where " + where
111         label = "".join(":{}".format(l) for l in labels)
112         return self.transaction("Match (a{l}) {w} return a",l =
    label, w = where)
113
114     #Searches for all relations given a gin, tonic and possibly
    a garnish. If sort is set to true, the list will be sorted by
     number of helpfull marks.
115     def search_drink(self, gin, tonic, garnish = None, order="")
    :
116         if garnish:
117             garnish = "Match (garnish:Garnish{}) <- [:Drink] - (
    drink)".format(QueObj(garnish))
118         else:
119             garnish = "Where not (:Garnish) <- [:Drink] - (drink
    )"
120
121         if order:
122             order = "Order by {} DESC".format(order)
123
124         r = self.transaction("""
125             Match (gin:Gin{gin}) <- [:Drink] - (drink)
126             Match (tonic:Tonic{tonic}) <- [:Drink] - (drink)
```

```
127             Match (rating:Rating) - [:Rating] -> (drink)
128             {garnish}
129             Optional Match(rating:Rating) <- [helpfull:Helpfull]
     -()
130             return rating, count(helpfull)
131             {order}
132         """,gin=gin, tonic=tonic, garnish=garnish, order = order
    )
133 
134         avg = self.transaction("""
135             Match (gin:Gin{gin}) <- [:Drink] - (drink)
136             Match (tonic:Tonic{tonic}) <- [:Drink] - (drink)
137             Match (rating:Rating) - [:Rating] -> (drink)
138             {garnish}
139             return count(rating), avg(rating.rating)
140         """,gin=gin, tonic=tonic, garnish=garnish)
141         return (avg,r)
142 
143     #Marks a rating helpfull.
144     def helpfull(self,user,rating):
145         return self.transaction("""
146             Match (rating:Rating{rating}), (user:User{user})
147             Merge (user) - [h:Helpfull] -> (rating)
148         """,user=user,rating=rating)
149 
150     #Returns all ratings from a user.
151     def my_ratings(self,user,order = "count(helpfull)"):
152         if order:
153             order = "Order by {} DESC".format(order)
154         return self.transaction("""
155             Match (user:User{user}) - [:Rating] -> (rating:
    Rating) - [:Rating] -> (drink)
156             Optional Match (rating:Rating) <- [helpfull:Helpfull
    ] - ()
157             return rating, count(helpfull)
158             {order}
159         """,user=user, order = order)
160 
161     #Returns all ratings, with the order being, per default, by
    the number of helpfull marks.
162     def ratings(self, order = "count(helpfull)"):
163         if order:
164             order = "Order by {} DESC".format(order)
165 
166         return self.transaction("""
167             Match (user:User) - [:Rating] -> (rating:Rating)
168             Optional Match (:User) - [helpfull:Helpfull] -> (
    rating:Rating)
169             return user,rating,count(helpfull)
```

```
170             {order}
171         """, order = order)
172 
173     #Clear the database.
174     def clear(self):
175         return self.transaction("""
176             Match (u) detach delete u
177         """)
```

test.py

```
def testlist(db,lf):
    for n,f in enumerate(lf):
        if n > 0 :
            input("\n\nPress enter for new test...")
        print("TEST ",n + 1)
        print(linelist(f(db)))

def linelist(l):
    return "\n".join(str(i.values()) for i in l)

def real_db(db):
    return dict(
        Jonas = db.new_user("Jonas Ingerslev S rensen", 21 , "Male"),
        Jeff = db.new_user("Jeff Gyldenbrand", 33, "Apache Attack Helicopter"),
        Simon = db.new_user("Simon Dradrach J rgensen", 26, "Emu"),

        gin = db.new_gin("Bombay Sapphire", "Bombay spirits", 70, 40),
        tonic = db.new_tonic("Tonic water", "Schweppes", 33),
        garnish = db.new_garnish("Lime")
    )

def db_output(db):
    db.clear()
    real_db(db)
    result = db.transaction("""
        match (u) return u
    """)
    return list(result)


def ratings(db):
    db.clear()
    def f(Jeff,Jonas,Simon,gin,tonic,garnish):
        db.drink(Jeff,gin,tonic, rating = 4, comment = "Needs more pickles.")
        db.drink(Jonas,gin,tonic, rating = 1, comment = "It's a shit!")
        db.drink(Simon,gin,tonic,garnish, 3, "Tastes like Australian tears")
        return list(db.transaction("""
            match (rating:Rating) - [:Rating] -> (drink)
            return rating, drink
        """))
    return f(**real_db(db))


def helpfull(db):
    db.clear()
    def f(Jeff,Jonas,Simon,gin,tonic,garnish):
        db.drink(Jeff,gin,tonic,rating=4, comment = "Needs more pickles.")
```

```
48          db.drink(Jonas,gin,tonic,rating=1, comment = "It's a shit!")
49          rating = db.drink(Simon,gin,tonic,garnish, 3, "Tastes like Australian tears"
        )
50          db.helpfull(Jeff,rating)
51          return list(db.transaction("""
52              match ()-[u:Helpfull]->() return u
53          """))
54
55      return f(**real_db(db))
56
57  def search(db):
58      helpfull(db)
59      l = [
60          db.search("Gin"),
61          db.search(percentage = 50),
62          db.search(percentage = "^([5-9]|[0-9])"),
63          db.search(producer = "(?i)Schweppes"),
64          db.search("Garnish"),
65          db.search(name = "(?i)Jonas*")
66      ]
67      return [y for x in l for y in x]
68
69  def search_drinks(db):
70      helpfull(db)
71      def f(gin,tonic,garnish,**_):
72          l =  [
73              db.search_drink(gin,tonic),
74              db.search_drink(gin,tonic,order="count(helpfull)"),
75              db.search_drink(gin,tonic,garnish),
76              db.search_drink(gin,tonic,garnish,order="count(helpfull)")
77          ]
78          return [y for x in l for y in x]
79      return f(**real_db(db))
80
81  def my_ratings(db):
82      helpfull(db)
83      def f(Jonas,Jeff,Simon,**_):
84          l = [
85              db.my_ratings(Jonas),
86              db.my_ratings(Simon),
87              db.my_ratings(Jeff)
88          ]
89          return [y for x in l for y in x]
90      return f(**real_db(db))
91
92  def all_rating(db):
93      helpfull(db)
94      l=[
95          db.ratings(),
```

```
96          db.ratings("rating.rating"),
97          db.ratings("user.name")
98      ]
99      return [y for x in l for y in x]
100 if __name__ == '__main__':
101     from inter import *
102 
103     db = Neo4jDB()
104     testlist(db,[db_output,ratings,helpfull,search,search_drinks,my_ratings,
        all_rating])
```