02807 Computational Tools for Big Data
Assignment 2: Databases and Streaming

Anonymous authors

October 26th 2015

5 Week 05: SQL and NoSQL

5.1 Exercise 5.1

The connection to the database, given by the exercise, is accomplished through the use of the sqlite3 module - a library that provides a disk-based database without requiring a separate server process.

Seeing since all the sub-exercises of exercise 5 take place within the same database connection the Counter is imported here, however it is not used until later-on. The 'con'-variable is set to be the connecting to the database, and then set sqlite3 to return bytestrings with the 'text_factory'.

```
import sqlite3 as lite
from collections import Counter

con = lite.connect('northwind.db')
con.text_factory = str
```

In order to test if everything works properly, the connection is used to fetch all products from the 'Products'-list. A for-loop runs through the gathered products and prints them.

```
with con:

    cur = con.cursor()

    cur.execute('select ProductName from Products')
    allProducts = cur.fetchall()

    for i in allProducts:
        print i[0]
```

A section of the printed list of products can be seen in Figure 1. As the sole purpose of fetching and printing these products is to demonstrate that the connection works, only the first 10 products of the larger list are shown in the image.

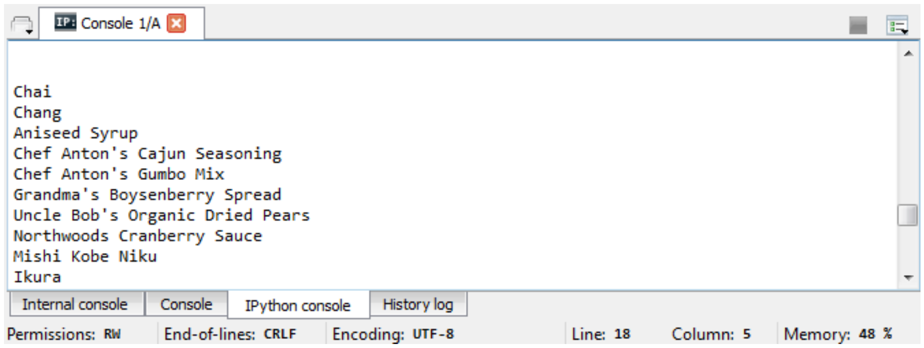


Figure 1: The first 10 products of the larger printed list

5.2 Exercise 5.2

The task of querying for, and returning, all orders made by ALFKI, and the products they contain, is accomplished by fetching all the orderIDs relating to the customerID "ALFKI" and a double for-loop then runs through the products within each order and prints them.

```
cur.execute('select OrderID from Orders where CustomerID
    ="ALFKI"')

OrderIDs = cur.fetchall()

for i in OrderIDs:
    print "OrderID: %d"%(i[0])
    cur.execute('select ProductID from [Order Details]
        where OrderID=%d'%(i[0]))

    ProductIDs = cur.fetchall()

    for h in ProductIDs:
        cur.execute('select ProductName from Products
            where ProductID=%d'%(h[0]))
        data = cur.fetchall()

        for j in data:

            print "Product: %s"%(j[0])
    print " "
```

The orders, as well as the products within each order, related to the customer 'ALFKI' can be seen in Figure 2.

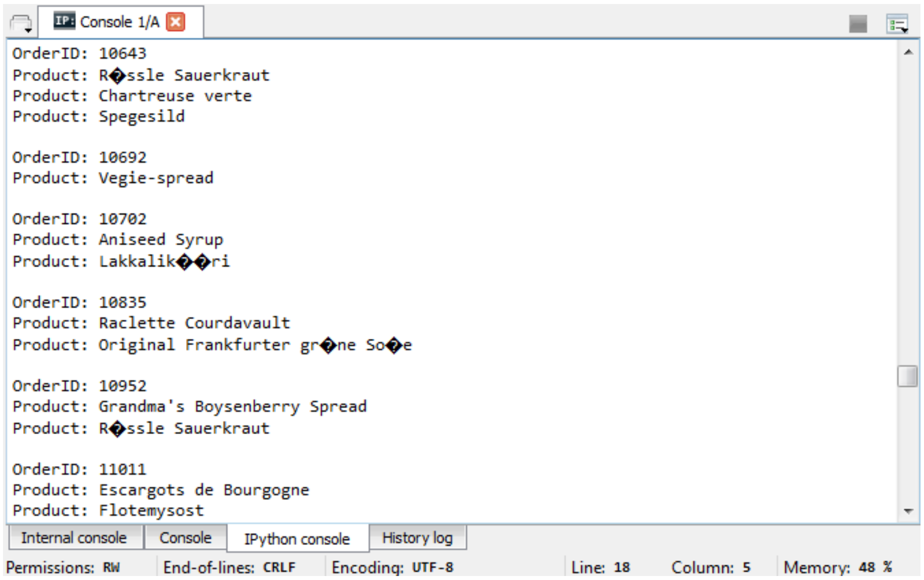


Figure 2: Orders and products related to customer 'ALFKI'

5.3 Exercise 5.3

Just like exercise 5.2 this task, regarding obtaining all orders (with products) made by ALFKI that contain at least 2 product types, requires the use of a double for-loop. However, within the first for-loop an if-statement will be implemented to check for whether or not the length ('len') has a value higher than 2.

```
cur.execute('select OrderID from Orders where CustomerID
    ="ALFKI"')

OrderIDs = cur.fetchall()

for i in OrderIDs:
    cur.execute('select ProductID from [Order Details]
        where OrderID=%d'%(i[0]))
    ProductIDs = cur.fetchall()
    if len(ProductIDs) >= 2:
        print "OrderID: %d"%(i[0])

        for h in ProductIDs:
            cur.execute('select ProductName from
                Products where ProductID=%d'%(h[0]))
            data = cur.fetchall()

            for j in data:
```

```
            print "Product: %s"%(j[0])
    print " "
```

The orders, related to the customer 'ALFKI', with at least of 2 product types can be seen in Figure 3.

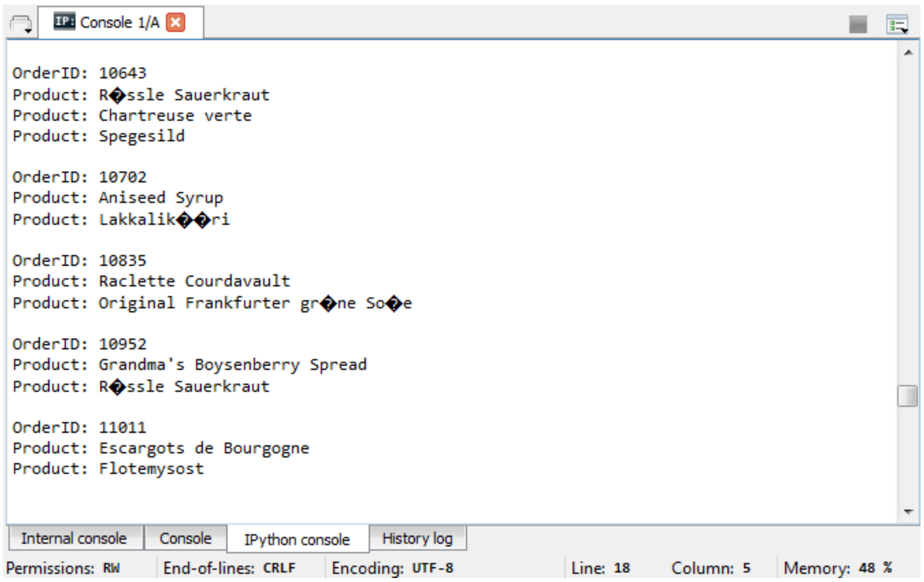


Figure 3: Orders related to customer 'ALFKI' with at least 2 products

5.4 Exercise 5.4

In order to determine how many, and who, ordered "Uncle Bob's Organic Dried Pears" (productID 7) the double for-loop is set up with the inner loop creating a list. The list is appended each time a productID 7 order is registered, thereby creating the desired list of customers. The created list is printed followed by the length of said list.

```
countedCustomers = []

    cur.execute('select OrderID from [Order Details] where
        ProductID=7')
    Order7 = cur.fetchall()

    for i in Order7:

        cur.execute('select CustomerID from Orders where
            OrderID=%d'%(i[0]))
        Customers = cur.fetchall()

        for h in Customers:

            countedCustomers.append(h[0])

    print Counter(countedCustomers)
    print len(Counter(countedCustomers))
```

The list of customers registered for having ordered "Uncle Bob's Organic Dried Pears" (productID 7) can be seen in Figure 4.

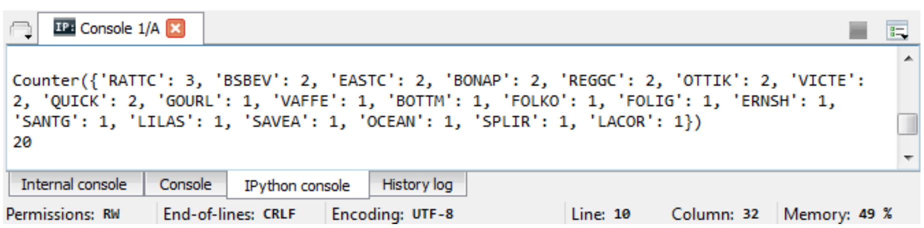


Figure 4: Customers who ordered productID 7, followed by their total count

5.5 Exercise 5.5

For the sake of determining how many different products, and which, have been ordered by customers who have also ordered “Uncle Bob’s Organic Dried Pears”, all customers related to productID 7 are fetched from the database. The multiple loops then run through each product for each order for each customer that is registered on the list for productID 7.

```
cur.execute('select OrderID from [Order Details] where
    ProductID=7')
    Order7 = cur.fetchall()
    totalList = []

    for k in Order7:
        cur.execute('select CustomerID from Orders where
            OrderID=%d'%(k[0]))
        Customers = cur.fetchall()
        Customers = sorted(set(Customers))

        for i in Customers:
            cur.execute('select OrderID from Orders where
                CustomerID="%s"'%(i[0]))
            Orders = cur.fetchall()

            for h in Orders:
                cur.execute('select ProductID from [Order
                    Details] where OrderID="%d"'%(h[0]))
                Products = cur.fetchall()

                for j in Products:
                    cur.execute('select ProductName from
                        Products where ProductID="%d"'%(j[0])
                        )
                    ProductNames = cur.fetchall()

                    for l in ProductNames:
                        totalList.append(l)

    totalList = sorted(set(totalList))
    for p in totalList:
        print "Product: %s"%(p[0])

    print " "

    print "Total Amount Of Products: %s"%(len(totalList))

    print "______________________________________________"
```

The list of how many different products that have been ordered by customers who have also ordered “Uncle Bob’s Organic Dried Pears” (productID 7) and what these products are, can be seen in Figure 5.

```
Product: Alice Mutton
Product: Aniseed Syrup
Product: Boston Crab Meat
Product: Camembert Pierrot
Product: Carnarvon Tigers
Product: Chai
Product: Chang
Product: Chartreuse verte
Product: Chef Anton's Cajun Seasoning
Product: Chef Anton's Gumbo Mix
Product: Chocolade
Product: C�te de Blaye
Product: Escargots de Bourgogne
Product: Filo Mix
Product: Flotemysost
Product: Geitost
Product: Genen Shouyu
Product: Gnocchi di nonna Alice
Product: Gorgonzola Telino
Product: Grandma's Boysenberry Spread
Product: Gravad lax
Product: Guaran� Fant�stica
Product: Gudbrandsdalsost
Product: Gula Malacca
Product: Gumb�r Gummib�rchen
Product: Gustaf's Kn�ckebr�d
Product: Ikura
Product: Inlagd Sill
Product: Ipoh Coffee
Product: Jack's New England Clam Chowder
Product: Konbu
Product: Lakkalik��ri
Product: Longlife Tofu
Product: Louisiana Fiery Hot Pepper Sauce
Product: Louisiana Hot Spiced Okra
Product: Manjimup Dried Apples
Product: Mascarpone Fabioli
Product: Maxilaku
Product: Mishi Kobe Niku
Product: Mozzarella di Giovanni
Product: Nord-Ost Matjeshering
Product: Northwoods Cranberry Sauce
Product: NuNuCa Nu�-Nougat-Creme
Product: Original Frankfurter gr�ne So�e
Product: Outback Lager
Product: Pavlova
Product: Perth Pasties
Product: P�t� chinois
Product: Queso Cabrales
Product: Queso Manchego La Pastora
Product: Raclette Courdavault
Product: Ravioli Angelo
Product: Rh�nbr�u Klosterbier
Product: Rogede sild
Product: R�d Kaviar
Product: R�ssle Sauerkraut
Product: Sasquatch Ale
Product: Schoggi Schokolade
Product: Scottish Longbreads
Product: Singaporean Hokkien Fried Mee
Product: Sir Rodney's Marmalade
Product: Sir Rodney's Scones
Product: Sirop d'�rable
Product: Spegesild
Product: Steeleye Stout
Product: Tarte au sucre
Product: Teatime Chocolate Biscuits
Product: Th�ringer Rostbratwurst
Product: Tofu
Product: Tourti�re
Product: Tunnbr�d
Product: Uncle Bob's Organic Dried Pears
Product: Valkoinen suklaa
Product: Vegie-spread
Product: Wimmers gute Semmelkn�del
Product: Zaanse koeken

Total Amount Of Products: 76
```

Figure 5: A list of different products ordered by people who registered as having ordered productID 7, and a total count

5.6 Exercise 5.6

Determining which product was ordered the most frequently, in terms of the group of customers that also ordered “Uncle Bob’s Organic Dried Pears”, is accomplished by modifying the looping slightly. Through the five loops the code will check product name of each product of each order of each customer of each productID 7 (“Uncle Bob’s Organic Dried Pears”) order.

```
for k in Order7:
```

```
cur.execute('select CustomerID from Orders where
    OrderID=%d'%(k[0]))
Customers = cur.fetchall()

for i in Customers:
    print "Customer: %s"%(i[0])
    cur.execute('select OrderID from Orders where
        CustomerID="%s"'%(i[0]))
    Orders = cur.fetchall()
    productAmount = 0

    for h in Orders:
        cur.execute('select ProductID from [Order
            Details] where OrderID="%d"'%(h[0]))
        Products = cur.fetchall()

        for j in Products:
            cur.execute('select ProductName from
                Products where ProductID="%d"'%(j[0])
                )
            ProductNames = cur.fetchall()

            for l in ProductNames:
                productAmount = productAmount + 1

print "Product Amount: %s"%(productAmount)
print " "
```

The list of products that were ordered most frequently by customers who have also ordered “Uncle Bob’s Organic Dried Pears” (productID 7) can be seen in Figure 6.

Customer: RATTC
Product Amount: 18

Customer: SPLIR
Product Amount: 9

Customer: VICTE
Product Amount: 10

Customer: BSBEV
Product Amount: 10

Customer: BONAP
Product Amount: 17

Customer: VICTE
Product Amount: 10

Customer: VAFFE
Product Amount: 11

Customer: SAVEA
Product Amount: 31

Customer: FOLIG
Product Amount: 5

Customer: QUICK
Product Amount: 28

Customer: OTTIK
Product Amount: 10

Customer: GOURL
Product Amount: 9

Customer: OTTIK
Product Amount: 10

Customer: LACOR
Product Amount: 4

Customer: REGGC
Product Amount: 12

Customer: SANTG
Product Amount: 6

Customer: BONAP
Product Amount: 17

Customer: OCEAN
Product Amount: 5

Customer: QUICK
Product Amount: 28

Customer: ERNSH
Product Amount: 30

Customer: BOTTM
Product Amount: 14

Customer: EASTC
Product Amount: 8

Customer: RATTC
Product Amount: 18

Customer: FOLKO
Product Amount: 19

Customer: REGGC
Product Amount: 12

Customer: BSBEV
Product Amount: 10

Customer: EASTC
Product Amount: 8

Customer: LILAS
Product Amount: 14

Customer: RATTC
Product Amount: 18

Figure 6: The list of products that were ordered most frequently by people who registered as having ordered productID 7

5.7 Exercise 5.7

In order to determine what other customers bought most of the same products as customerID ALFKI all orderIDs registered to ALFKI are fetched. Next, the multiple loop runs through every customerID of every product name of every productID of every orderID and increments each customerID's value (same product order as ALFKI) by one. Whichever customer has the highest value is the one with the most identical list of ordered products.

```
cur.execute('select OrderID from Orders where CustomerID="
    ALFKI"')
    OrderIDs = cur.fetchall()
    customerIdCounter = []
```

```
for i in OrderIDs:
    cur.execute('select ProductID from [Order Details]
        where OrderID=%d'%(i[0]))
    ProductIDs = cur.fetchall()
    if len(ProductIDs) >= 2:

        for h in ProductIDs:
            cur.execute('select ProductName from
                Products where ProductID=%d'%(h[0]))
            data = cur.fetchall()
            for j in data:
                print "Product: %s"%(j[0])

            cur.execute('select OrderID from [Order
                Details] where ProductID=%d'%(h[0]))
            totalOrderIDs = cur.fetchall()
            for k in totalOrderIDs:
                cur.execute('select CustomerID from
                    Orders where OrderID = %d'%(k[0]))
                totalCustomerIds = cur.fetchall()
                for l in totalCustomerIds:
                    customerIdCounter.append(l[0])
            print " "

print Counter(customerIdCounter)
```

The resulting list of customers, whose list of ordered products resemble that of customerID ALFKI, can be seen in Figure 7.

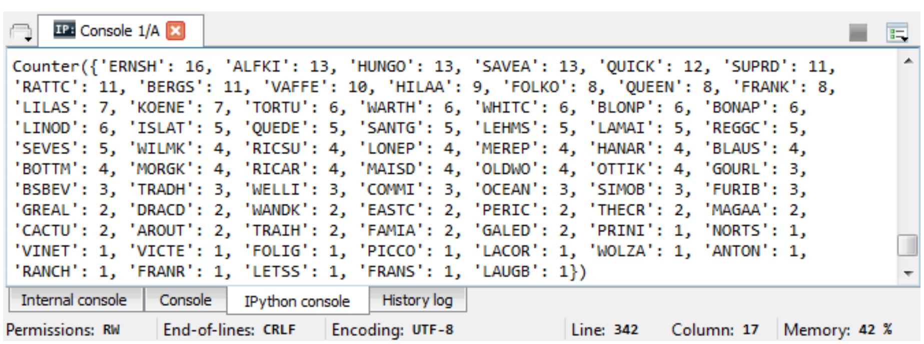


Figure 7: List of customers with, gradually declining, similarity of product lists to customerID ALFKI

6 Week 06: Graph Databases

6.1 Exercise 6.1

In exercise 6.1 the online ide neo4j and the graph-generator is to be used. To get started, an account at graph story is made, after creating an account a server is assigned. This server is the one computing and storing the data. after getting the server assigned, the online interface is opened. This is done by pushing on the actions button, and click on Web UI. see figure 9

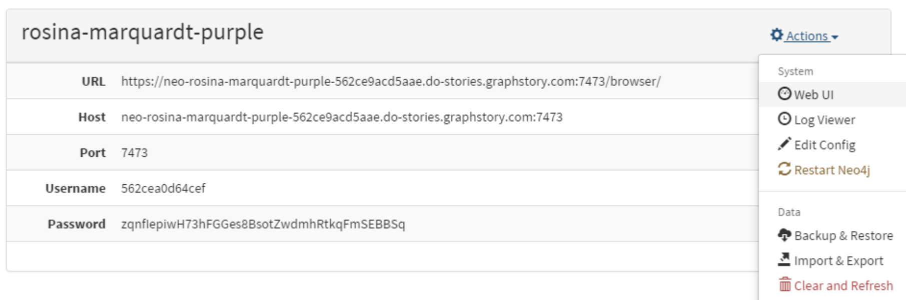


Figure 8: The server instance and the action-bar

Before adding the data from the server provided by the course, a cleanup of the server have to be done. This can either be done from Graph-Story homepage, or by writing a piece of code. With a clean server, the data can begin to be added. This is done by writing the chunks of code, which drags all the data from another server.

6.2 Exercise 6.2

The assignement is about writing a piece of code, that returns all the orders made by ALFKI while at the same time show all the products. The piece of code is as follows.

```
MATCH (Customer { customerID:"ALFKI" })--(Order)
RETURN  Order
```

To find the orders with a relationship to ALFKI, we start by using MATCH, which locates the place we want to search. We start by searching for alfki inside customer instances. This is done using curlie brackets after writing which nodes that should be searched in. Two lines are used to write that we want to have the relation between the Customer node with the customerID alfki, and the orders. Afterwareds is the Order returned figure 9. This resulted in a neat graph figure 10, showing all the links and the relationships.

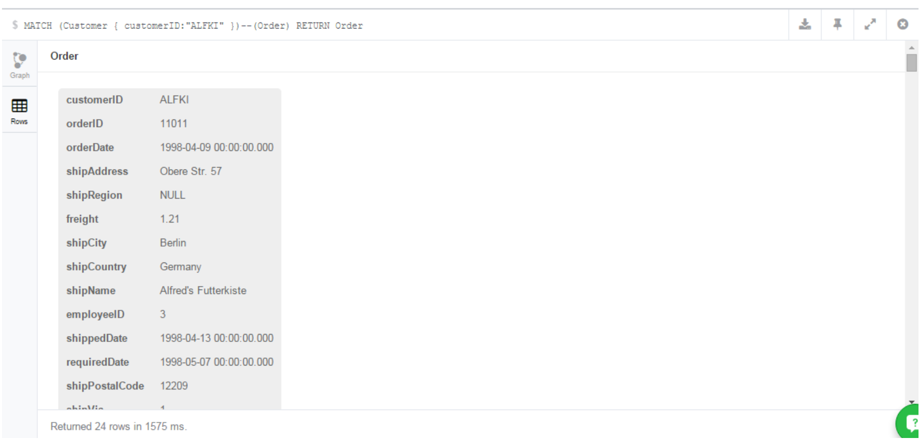


Figure 9: A list of alle the nodes, inside the graph.

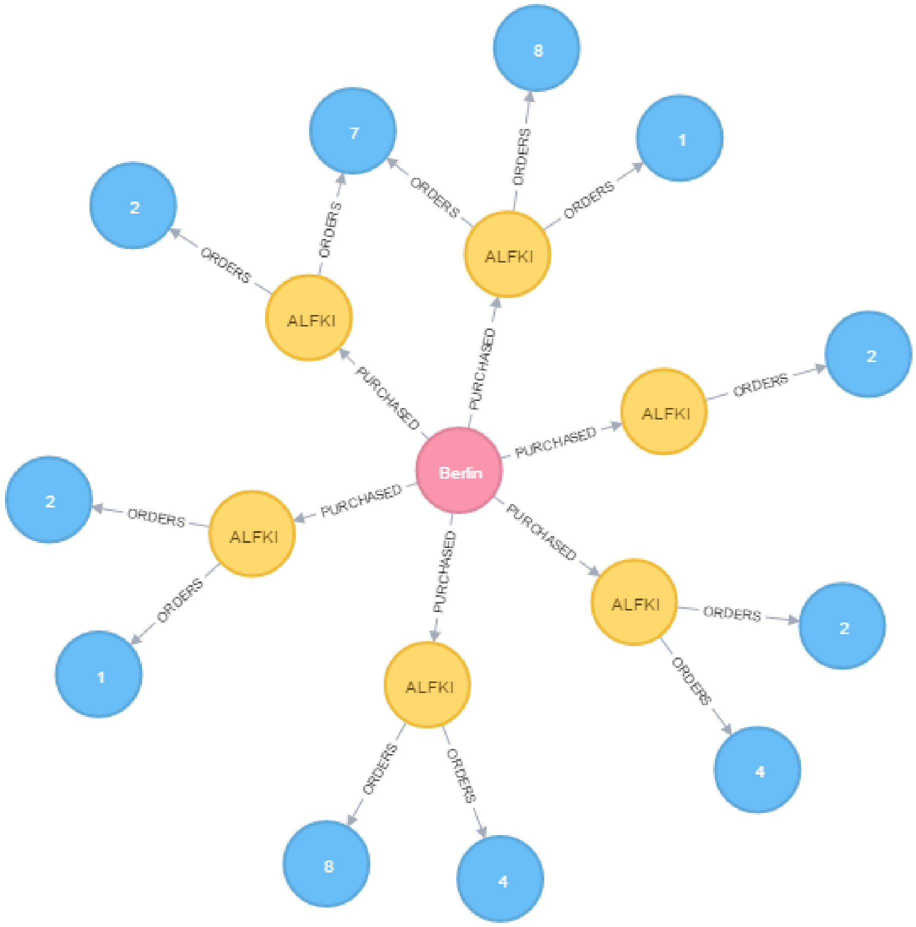


Figure 10: The graph

6.3 Exercise 6.3

To filter the orders, we need to store all the products, bought by ALFKI, and then count the relationships of each order. After that is done, the orders are saved into a variable called OrderID, and the counted relationships of each order is saved in amount. Amount is afterwards used as filter, that is checked if it is higher than 2 at each node. the amount of relations and the orderID is returned, and afterwards sorted. the result can be seen in figure 11

```
MATCH (Customer { customerID:"ALFKI" })--(Product)
WITH Product AS hej
MATCH (Customer { customerID:"ALFKI" })-[r:ORDERS]-(hej)
WITH r.orderID AS OrderID, Count(r) AS Amount
WHERE Amount > 1
```

```
RETURN Amount,  OrderID
ORDER BY Amount DESC
```

$ MATCH (Customer { customerID:"ALFKI" })--(Product) WITH Product AS hej MATCH (Customer { customerID:"ALFKI" })-[r:ORDERS]-(hej) WI...

Amount	OrderID
4	10643
3	10952
2	10835
2	11011
2	10702

Returned 5 rows in 1023 ms.

Figure 11: The the List of filtered products

6.4 Exercise 6.4

To find how many have bought Uncle Bob's Organic Dried Pears, the same procedure as in exercise 6.2 is followed. This makes graph story generate the relation by itself, and linking the nodes together.

To get a nice list of all the different customers, who have ordered Uncle Bob's Organic Dried Pears the code.

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
RETURN distinct Customer.customerID
```

It is totally same procedure as in 6.2, with changed variables. One small change is the .notation Customer.customerID. This notation creates a list of a specific property of the nodes. Distinct is added to filtrate the output, so it only shows a node one time. The result looks like this see. figure 12

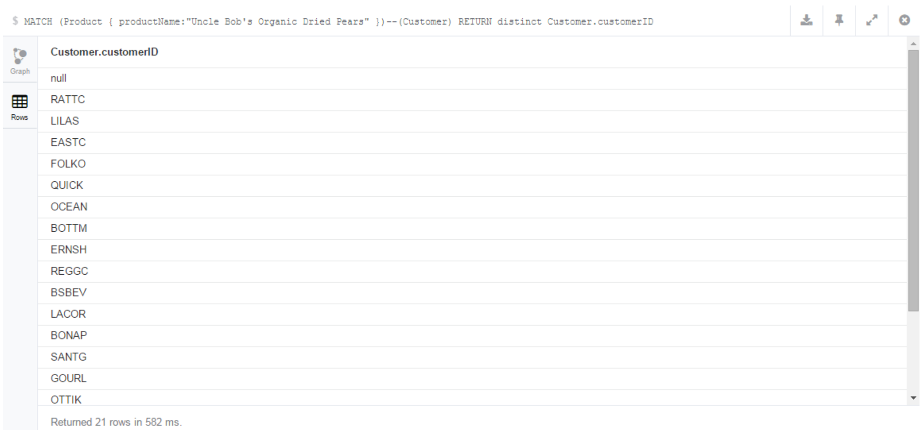


Figure 12: The list of customerIDs of all the customers who have bought the pears

The list show all the different customers who have ordered the pears. To get the number of times the pears have been ordered, a function called count() is used. This use a list of nodes as input, and returns the amount of nodes.

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
RETURN count(Customer)
```

Which in total is 31, this means that there are 31 orders in total. To get how many different customers who have ordered the the pears. A new variable is made, by using as. Distinct is a keyword that filtrates duplicates, which now is minimised to only uniques.

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
```

```
WITH distinct Customer.customerID as totalDistinct
RETURN count(totalDistinct)
```

This result in 20 different customers in total as seen in figure 13

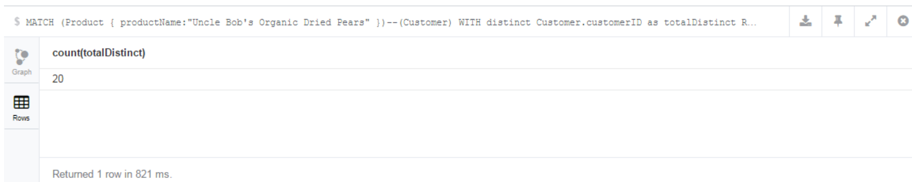


Figure 13: The the total amount of different customers

6.5 Exercise 6.5

To make a list of products that have been ordered, by customers who also have ordered "Uncle Bob's Organic Dried Pears" two times match are used. This is a way of specifying the perspective of the products we want to find. We start in the first match, to find the customers who have ordered "Uncle Bob's Organic Dried Pears", and defines that with the perspective of the Customer.customerID we want to find n. n is in this case eveything, but by writing n.productName, we specify that we want all things that have a relationship to the customer, and that have a productName, which are the products they have ordered. The keyword distinct is used, to avoid duplicates in the list. In figure 14 the results can be seen.

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
WITH Customer.customerID AS hej
MATCH (n)
RETURN distinct n.productName
```

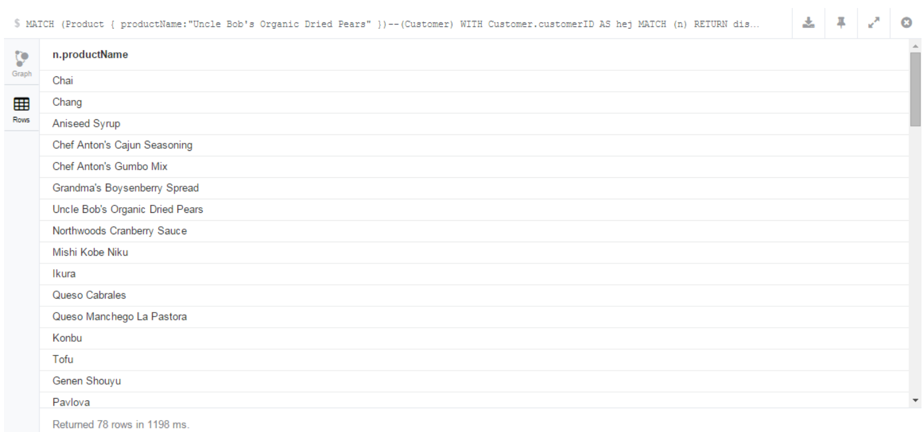
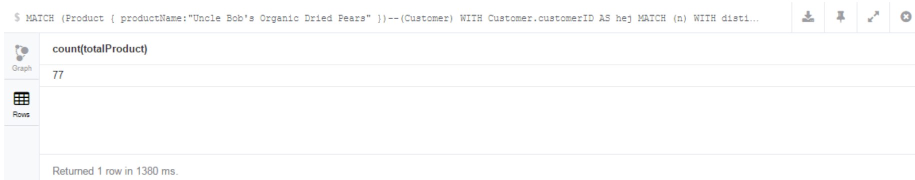


Figure 14: The list of all the different products also ordered

To get the total number of different products that have been ordered, yet another with is made, to create a new variable that is possible to count. The results of the code can be seen in Figure 15

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
WITH Customer.customerID AS hej
MATCH (n)
WITH distinct n.productName as totalProduct
RETURN count(totalProduct)
```



$ MATCH (Product { productName:"Uncle Bob's Organic Dried Pears" })--(Customer) WITH Customer.customerID AS hej MATCH (n) WITH disti...

count(totalProduct)
77

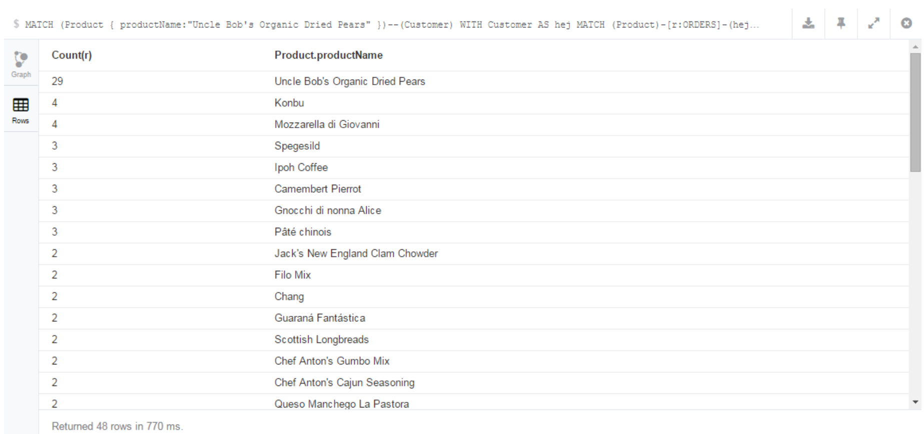
Returned 1 row in 1380 ms.

Figure 15: The the total amount of different products ordered.

6.6 Exercise 6.6

In exercise 6.6 we want to find the product that have been ordered the most, of the different products ordered by the ones who also have ordered "Uncle Bob's Organic Dried Pears". To do this we start by finding all the customers who have ordered "Uncle Bob's Organic Dried Pears", who is saved in the variable hej, that is transferred in the second match. to find the products of hej we need to find the relationship between the customer and the product. The relationship ORDERS describe the relations between one product, and all the customers. we are therfore counting how many relationships to the customers that each product have, and orders it descending. In figure 16 the descending order can be seen, and the most ordered product, next after the "Uncle Bob's Organic Dried Pears" is "Konbu" or "Mozzarella di Giovanni" which both have 4 orders.

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
WITH Customer AS hej
MATCH (Product)-[r:ORDERS]-(hej)
RETURN Count(r), Product.productName
ORDER BY Count(r) DESC
```



$ MATCH (Product { productName:"Uncle Bob's Organic Dried Pears" })--(Customer) WITH Customer AS hej MATCH (Product)-[r:ORDERS]-(hej...

Count(r)	Product.productName
29	Uncle Bob's Organic Dried Pears
4	Konbu
4	Mozzarella di Giovanni
3	Spegesild
3	Ipoh Coffee
3	Camembert Pierrot
3	Gnocchi di nonna Alice
3	Pâté chinois
2	Jack's New England Clam Chowder
2	Filo Mix
2	Chang
2	Guaraná Fantástica
2	Scottish Longbreads
2	Chef Anton's Gumbo Mix
2	Chef Anton's Cajun Seasoning
2	Queso Manchego La Pastora

Returned 48 rows in 770 ms.

Figure 16: The list of products, that is ordered with the dried Pears

To see the relationships between products, and the customers. Another code is used. Figure 17 is just showing all the relationships that is counted in figure 16

```
MATCH (Product { productName:"Uncle Bob's Organic Dried
    Pears" })--(Customer)
WITH Customer AS hej
MATCH (Product)-[r:ORDERS]-(hej)
RETURN r
```

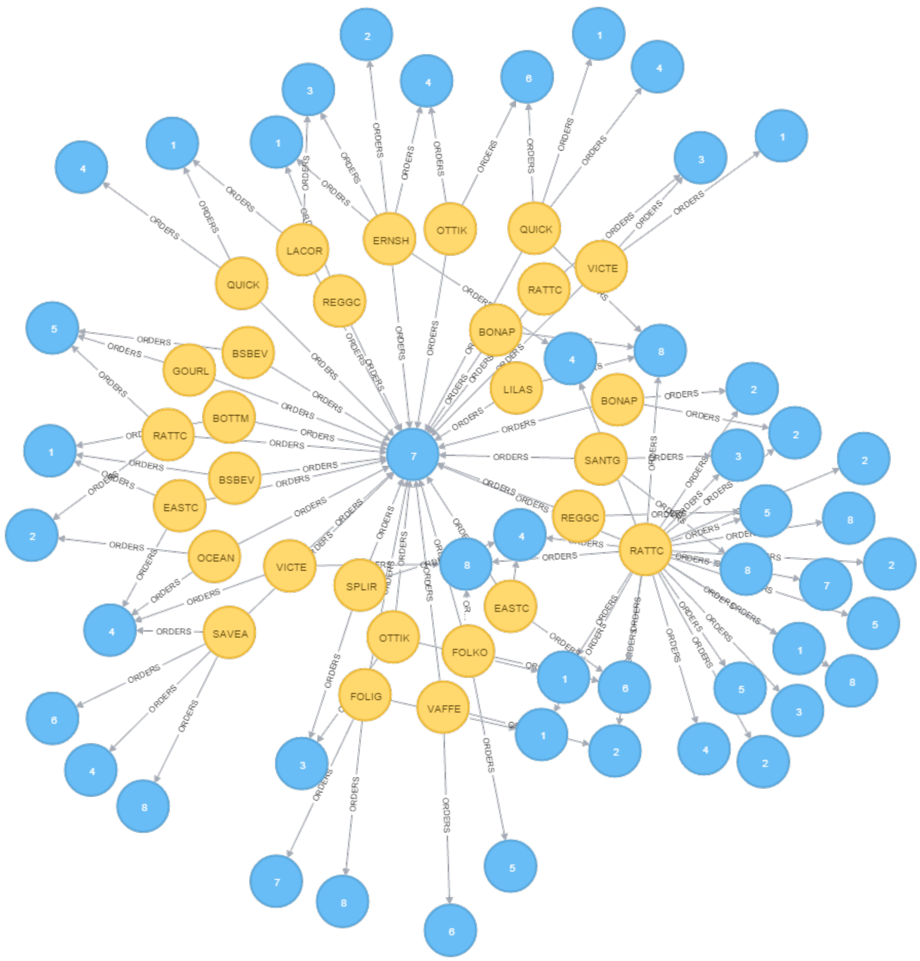


Figure 17: The Graph of products, that is ordered with the dried Pears

6.7 Exercise 6.7

This assignment is the totally same procedure used, as in exercise 6.6, instead of Product it is the costumer AFLKI. The result of this, is outputted both as a list figure 18, and as a graph figure 19. The list shows that the costumer who have bought most of the same products as ALFKI, is ERNSH, who actually have bought more of the same products than ALFKI. Which in total is 18 products of the same products as ALFKI

```
MATCH (Customer { customerID:"ALFKI" })--(Product)
WITH Product AS hej
MATCH (Customer)-[r:ORDERS]-(hej)
RETURN Count(r), Customer.customerID
ORDER BY Count(r) DESC
```

$ MATCH (Customer { customerID:"ALFKI" })--(Product) WITH Product AS hej MATCH (Customer)-[r:ORDERS]-(hej) RETURN Count(r), Customer...

Count(r)	Customer.customerID
18	ERNSH
14	SAVEA
14	HUNGO
14	ALFKI
12	QUICK
12	BERGS
12	null
11	VAFFE
11	SUPRD
11	RATTC
9	HILAA
9	FOLKO
8	FRANK
8	QUEEN
7	TORTU
7	WHITC

Returned 77 rows in 627 ms.

Figure 18: The List of customer, buying the same as alfki

To get the relations between the products, and the customers, the relationships are returned. This results in the beautiful figure 19

```
MATCH (Customer { customerID:"ALFKI" })--(Product)
WITH Product AS hej
MATCH (Customer)-[r:ORDERS]-(hej)
RETURN r
```

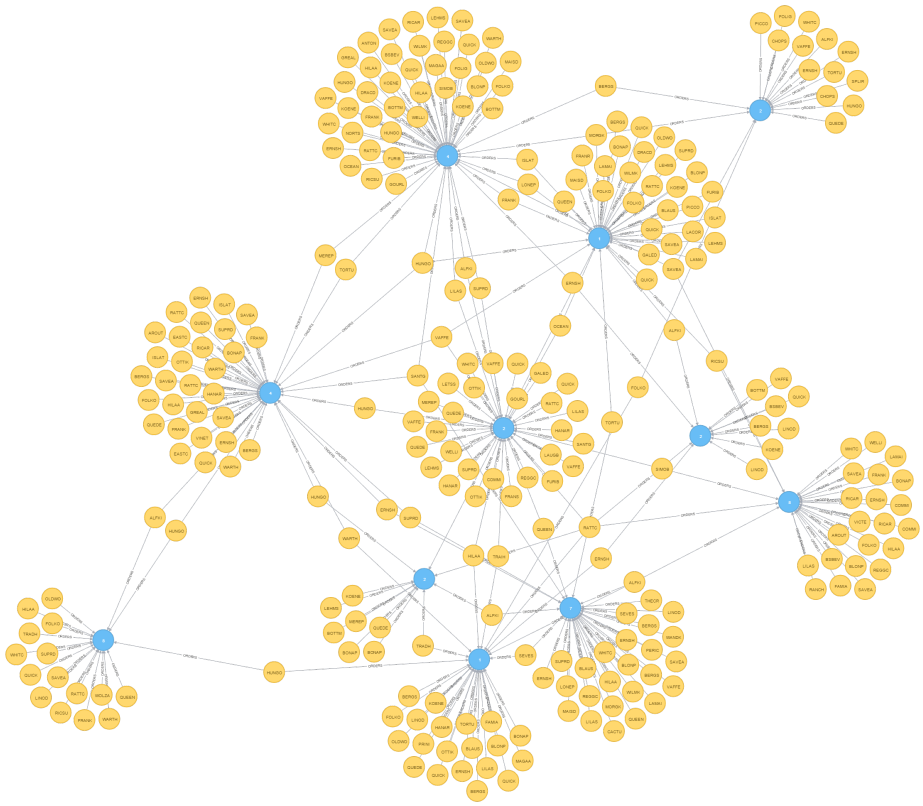


Figure 19: The Graph of customer, buying the same as alfki

7 Week 07: Streaming Algorithms

7.1 Exercise 7.1

The Bloom filter is constructed with three functions. The first of these is the initiator function which is responsible for initializing the whole class. The 'self' parameter refers to the instance whose method was called. The 'size' parameter refers to the size of the bit_array that will be made, meaning an array of bits with a value of either 0 or 1. The 'hash_count' is how many times each added string should be hashed, meaning the higher the count, the more precise results, while also taking longer time to calculate. The calculation concerning how high it should be is in the file the class is called from. In the 'init' the bit_array is made with the size requested by the exercise and all bits are set equal to 0.

In the 'add' function the string, being words from the text-file, is added to the bit_array. For the desired amount of hash functions (225) the word is hashed, which then gives an bit_array-index which is then changed to 1 instead of 0.

The 'lookup' function is pretty much the reverse of the 'add' function. It checks whether a given word either positively is not in the text or most likely is among the words already added. The word searched for is once again hashed in order to obtain an index in the bit_array. If the bit_array is 0 in the found index then the word is positively not there. If the bit_array is 1 in the index then the word is most likely there.

```
from bitarray import bitarray

class BloomFilter:

    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)

    def add(self, string):
        for seed in xrange(self.hash_count):
            result = hash(string) % self.size
            self.bit_array[result] = 1

    def lookup(self, string):
        for seed in xrange(self.hash_count):
            result = hash(string) % self.size
            if self.bit_array[result] == 0:
                return False
        return True
```

Having completed the Bloom filter class, the next thing to do is to apply it to the Shakespeare-file and compare the difference in speed between the filter and simply looping over the dictionary for each word.

Before the two approaches, to searching through the text-file, are applied to the file, the following adjustments are performed on the text:

- All characters are set to lower-case
- Words are split up in a list
- Each word is stemmed
- All extra characters are removed

The value for how many unique words that have been found is printed. The dictionary is then read into the code and the whole process is repeated, this time using the dictionary.

The Bloom filter is then instantiated with 1000000 bits, lists are setup to hold the words that are not in the dictionary, and lastly, the speed test between the Bloom filter and the method of simply looping over the dictionary for each word in the text-file is set to run. The Bloom filter turns out to be many times faster than the simple looping method, as seen in Figure 20 illustrating the amount of unique words (3082) followed by the time in seconds it took the

Bloom filter followed by the time it took the simple dictionary method. The words reported to be in the dictionary without actually being in there can be seen in Figure 21.

```
from BloomFilter import BloomFilter
from stemming.porter2 import stem
import datetime

#Read in the shakespeare file
with open('shakespeare.txt', 'rb') as in_file:
    shakespeare = in_file.read()

#Set all characters to lower
shakespeare = shakespeare.lower()
#Split words up in a list
shakespeare = shakespeare.split()
#Stem each word
shakespeare = [stem(word) for word in shakespeare]
#Remove extra characters
shakespeare = [''.join(e for e in word if e.isalnum()) for
    word in shakespeare]

#Print amount of unique values (3082 unique words)
print len(set(shakespeare))

#Read in dictionary
with open('dict', 'rb') as in_file:
    dictionary = in_file.read()

#Same as before
dictionary = dictionary.lower()
dictionary = dictionary.split()
dictionary = [stem(word) for word in dictionary]

#Instantiate bloom filter with 1000000 bits
#m/n * ln(2) = 225 calculated with m = 1000000 and n = 3082
bf = BloomFilter(1000000,225)

#Lists to see which words are not in dictionary
bloom_words = []
loop_words = []
combined_words = []

#Speed test the method
start = datetime.datetime.now()
for word in dictionary:
    bf.add(word)

for word in shakespeare:
    if bf.lookup(word):
```

```
        bloom_words.append(word)
finish = datetime.datetime.now()
print (finish-start).total_seconds()
#19.028 seconds to finish

start = datetime.datetime.now()
for word in shakespeare:
    for dic in dictionary:
        if word == dic:
            loop_words.append(word)
            break
finish = datetime.datetime.now()
print (finish-start).total_seconds()
#222.953 seconds to finish

for word in bloom_words:
    if word not in dictionary:
        combined_words.append(word)

print set(combined_words)
```

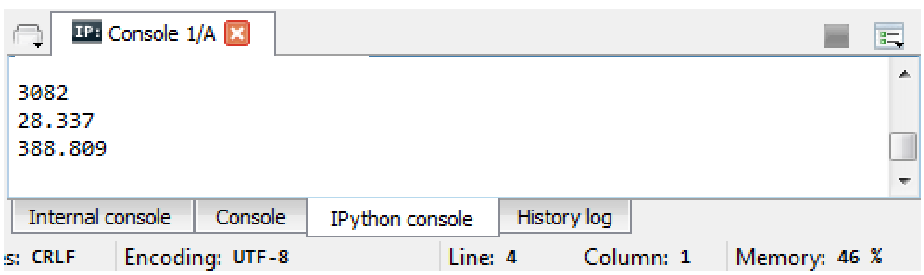


Figure 20: The amount of unique words followed by the time in seconds it took the Bloom filter and the time it took the simple dictionary looping method

```
set(['transformd', 'jointure', 'dance', 'indeed', 'charles', 'quarrels', 'pointdevic', 'railing', 'hornbeasts',
'unbanded', 'addressd', 'happiness', 'talking', 'feeding', 'praises', 'unhappy', 'forms', 'baser', 'wags',
'invocation', 'jars', 'swans', 'atomies', 'means', 'femalewhich', 'truly', 'rings', 'words', 'ambition', 'beards',
'similes', 'hereafter', 'fields', 'try', 'miles', 'assembly', 'laughing', 'offenders', 'adversity', 'says',
'wedlockhymn', 'swears', 'cleanliest', 'challenger', 'lips', 'befell', 'peacemaker', 'troilus', 'savage',
'executioner', 'overcame', 'entertainment', 'since', 'please', 'jolly', 'strippd', 'outside', 'messenger', 'tellst',
'covetousness', 'youll', 'body', 'meanings', 'inseparable', 'brooks', 'butterwomen', 'bringest', 'diest', 'intot',
'simples', 'greasy', 'pity', 'goods', 'contemplation', 'goest', 'ribs', 'experience', 'comingon', 'pleasures',
'tongue', 'pleaseth', 'duty', 'shelamb', 'unbuttoned', 'vii', 'ganymede', 'antiquity', 'feet', 'whate', 'names',
'brotherno', 'wormeaten', 'illroast', 'story', 'libertine', 'traveller', 'entreated', 'france', 'outlaws', 'merrier',
'raild', 'halfpenc', 'injury', 'happy', 'shoulders', 'religiously', 'believe', 'butchery', 'blessing', 'nobler', 'keyhole',
'doth', 'iii', 'removedbear', 'begone', 'loved', 'circumstantial', 'bled', 'grieve', 'bitterness', 'believes',
'severally', 'making', 'promised', 'tongues', 'vi', 'scrippage', 'prized', 'flouting', 'exile', 'fortunes', 'overheard',
'revelry', 'purity', 'foiled', 'ready', 'wisely', 'any', 'counterfeited', 'moves', 'began', 'offerst', 'tis', 'pilgrimage',
'mightst', 'greatest', 'bottle', 'voyage', 'dies', 'prunest', 'accustomd', 'device', 'stanzos', 'deny', 'pulpiter',
'cheerly', 'simpering', 'stronger', 'bathed', 'filld', 'sayst', 'jaqu', 'qualities', 'pleasure', 'fittest'])
```

Figure 21: Words reported to be in the dictionary without actually being in there

7.2 Exercise 7.2

To start things off with the Flajolet-Martin class an initiator is constructed to initiate and fill the size of each group, the same for each function and also to keep track through the use of a max-count.

Next, the function for adding data is constructed, which, for each function within each grouping, does the following:
- Sets the given function equal to the hash-function, which is used to quickly compare dictionary keys during a dictionary look-up
- Hashes the current entry
- Finds the amount of trailing zeroes in the current hashed string
- Updates the max-count if the new count is larger than the previous.

The function for finding trailing zeroes is then set up, which takes 'self' and 'num' as input parameters. The 'num' parameter is the value for counting up the number of trailing 0 bits. The final part of the function is a simple counter algorithm for counting up these trailing bits.

The last function needed for this class is the estimate-function which is responsible for estimating the amount of unique entries (distinct words). The function is started out with having two for-loops, one for looping through groups ad another for looping through the elements within each group. The second for-loop will then take each element and derive the bias that is the magic constant, which the Flajolet-Martin procedure predictably introduces. The group is the appended with the estimate with the two for-loops finished the cardinality, which is the approximate amount of unique entries, is calculated and returned.

To wrap things up, the Shakespeare-file is loaded, the class is called to run with the parameters (10, 10), meaning the class will with 10 hash-groups, each containing 10 hashes. The words found by the search are registered, and placed into a list, using the regular expressions module. And lastly, the estimate-value calculated with the Flajolet-Martin algorithm is printed, as seen in Figure 22.

```
import re
import random

class FlajoletMartin():

    #Initiater for the class
    def __init__(self, size, size_hashes):
        #Initiate and fill the size of each group
        self.groups = []
        [self.groups.append(size_hashes) for p in range(size
            )]
        #Do the same for
        self.functions = []
        [self.functions.append(random.randint(1,10000)) for
            p in range(size_hashes)]
        #Set the max count to keep track
        self.max_count = 0

    #Function for adding new data
    def add(self, string):
        for j in self.groups:
            for k in self.functions:
                #hash function
                func = lambda x: (hash(x) % 2**20) + 1
                #hash the current entry
                h = func(string)
                #find amount of trailing zeroes in current
                    hashed string
                count = self.trailing_zeroes(h)
                #Update max count if new count is larger
                if count > self.max_count:
                    self.max_count = count

    #Function for finding amount of trailing zeroes
    def trailing_zeroes(self, num):
     """Counts the number of trailing 0 bits in num."""
    #Simply count trailing zeroes with bit operations
```

```
        if num == 0:
            return 32
        p = 0
        while (num >> p) & 1 == 0:
            p += 1
        return p

    #Function for estimating the amount of unique entries
    def estimate(self):
        #for loop per group
        estimate_pr_group = []
        for j in self.groups:
            #loop per element in group
            estimate_in_group = []
            for k in self.functions:
                #Append 2 to the power of the max count
                    times the magic number
                estimate_in_group.append(2 ** self.max_count
                     * 0.77351)
            #Append the group
            estimate_pr_group.append(estimate_in_group)
        #The cardinality is the approximate amount of unique
             entries
        cardinality = 1.0*((sum(sum(x) for x in
            estimate_pr_group)) / len(estimate_pr_group))/10

        return cardinality

with open('shakespeare.txt', 'rb') as in_file:
    shakespeare = in_file.read()
shakespeare = shakespeare.split()

fm = FlajoletMartin(10, 10)

for word in shakespeare:
    word = re.sub(r'[^\w\s]','',word.lower())
    fm.add(word)

#Running the function gives a unique count of 3168.29696
print fm.estimate()
```

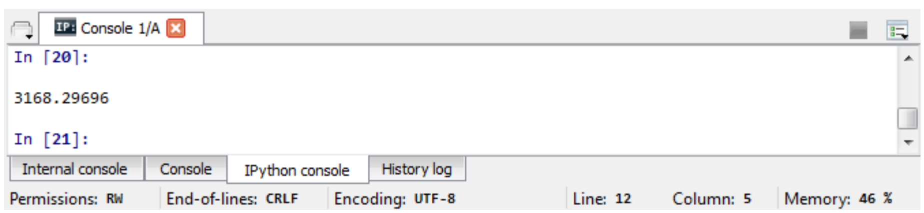


Figure 22: The determined number of distinct words within the Shakespeare-file