Danmarks Tekniske Universitet

02807 Computational Tools for Big Data

Assignment

WEEK 04: DBSCAN
WEEK 05: SQL AND NOSQL

Teacher and teaching assistant:

David Kofoed Wind,
Finn Årup Nielsen
(Rasmus) Malthe Jørgensen
Ulf Aslak Jensen

CONTENTS

Week 4 DBSCAN

In this section, our mission is to actualize a clustering algorithm. And the key problems we face are to find a container to load the matrix, to calculate Jaccard-distance between two points, to find out how many other points there are in the neighboring region of a given point, to expand a clustering correctly and rapidly, and at last, how to apply the algorithm to the system with the given parameters. In order to solve these problems, we develop several functions in our algorithm, thus making code of our method having a clear structure.

1. Load the matrix

```
import pickle as pk
from sklearn.metrics import jaccard_similarity_score
f=open('data_1000points_1000dims.dat','rb')
mat=pk.load(f,encoding='latin1')
num=mat.shape[0]
points=[]
for i in range(num):
    points.append(mat.getrow(i).todense())
status=[0 for i in range(num)]
```

In this part, we import the package of *pickle* and *jaccard_similarity_score* preparing for the following steps. After opening the file, we use the function *pickle.load()* to read in the matrix and store it in *mat*. Since *mat* is of a special variable type, sparse matrix, several functions like *shape() getrow()* and *todense()* can help us to extract each row of the matrix and form a list of these points. The list *status* implies the property of each point. For example, 0 means ‘unvisited’, -1 means ‘outlier’ and each positive integer (1, 2, 3 …) indicates a cluster. Obviously all points are unvisited at the beginning.

2. Calculate Jaccard-distance

```
def Jaccard_Distance(point1, point2):
    index=jaccard_similarity_score(point1,point2)
    distance=1-index
    return distance
```

Thanks to the function *jaccard_similarity_score()* of the package *sklearn.metrics*, it would be very easy to get the Jaccard-index of two points, and thus the Jaccard-distance.

3. Find neighboring points

```
def Region_Query(current_point, eps):
    temp=[]
    for i in range(num):
        if Jaccard_Distance(current_point, points[i])<=eps:
            temp.append(i)
    return temp
```

This part is a bit easy as well. For a given point, we check all points in matrix for their Jaccard-distance and append points close enough to a temporary list. Finally we return this list.

4. Expand Cluster

```
def Expand_Cluster(pt, neighbor, cluster, eps, Minpts):
    status[pt]=cluster
    for newpt in neighbor:
        if status[newpt]==0:
            new_neighbor=Region_Query(points[newpt], eps)
            if len(new_neighbor)>=Minpts:
                neighbor.extend(new_neighbor)
        if status[newpt]<=0:
            status[newpt]=cluster
```

First let's get familiar with the meaning of parameters. The variable *pt* is the starting point of expansion, *neighbor* is the list of neighboring points mentioned in Step 3, cluster is the index of the cluster, *eps* and *Minpts* are two given parameters. Main idea of this function is as follows. For each unvisited point or outlier in the list *neighbor*, we mark it with the same index of status with the starting point. And for those points with adequate neighbors, we add them to the list so that we could check them later. As you can see, the cluster is expanding gradually.

5. Main function

```
def DBSCAN(eps, Minpts):
    C=0
    for i in range(num):
        if status[i]!=0:
            continue
        NeighborPts=Region_Query(points[i], eps)
        if len(NeighborPts)<Minpts:
            status[i]=-1
        else:
            C=C+1
            Expand_Cluster(i, NeighborPts, C, eps, Minpts)
```

This is the major structure of the whole process. For each points in the list, if it has been visited (*status*≠0), we don't need to care about it since it has been processed. We only need to pay attention to unvisited points. Once we catch such a point, first we found its neighbouring points using function *Region_Query()*. If its neighbours are too few, we label it as outlier. Otherwise, it would be the starting point of a new cluster, whose index is the index of former cluster plus one. Now we can expand this cluster using function Expand_Cluster().

6. Application and output

```
DBSCAN(0.15, 2)
num_cluster=max(status)+1
print ("There are %d clusters." %(num_cluster))
```

Here we use the example of the third input file and its parameters. Just apply the main function with given parameters. Since we use positive integers (1, 2, 3…) to index clusters, the biggest index number is just the quantity of clusters. However we need to add it by one because we regard all outliers as an individual cluster. Finally we could print our result.

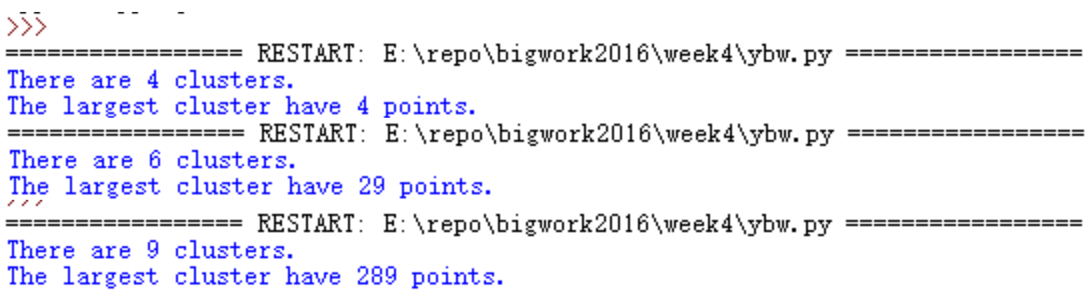
7. Figure out the size of the largest cluster

```
result=status
result.remove(-1)
from collections import Counter
pivot=Counter(result)
print ("The largest cluster have %d points." %(pivot.most_common()[0][1]))
```

In this part we mainly use the function *Counter()* from package *collections* to find out how many points there are in the largest cluster. Before that we need to remove all -1 representing outliers in the list since the collection of outliers is not a real cluster. We use the variable *pivot* to restore the result of counting and the biggest cluster size that we want is *pivot.most_common()[0][1]*. Now we can print it.

Result

Here are some screenshots of the result. They are identical to the answers given.



```
>>>
================= RESTART: E:\repo\bigwork2016\week4\ybw.py =================
There are 4 clusters.
The largest cluster have 4 points.
================= RESTART: E:\repo\bigwork2016\week4\ybw.py =================
There are 6 clusters.
The largest cluster have 29 points.
================= RESTART: E:\repo\bigwork2016\week4\ybw.py =================
There are 9 clusters.
The largest cluster have 289 points.
```

Week 5 SQL AND NOSQL(SQLite Part)

This week, we will take a look at two of the most common tools for storing data records: *relational* and *NoSQL* databases.

As is required, we will play around with querying and returning data from first a SQLite, and then a Mongo database. In order to show the two solution lines in a clear way, we divide the exercises into 3 parts.

1. Using a SQLite database system to solve the problems
2. Using a Mongo database system to solve the problems
3. Making comparison between the two database systems

//Part1. SQLite Part

Exercise 5.1

For SQLite: Establish connection to this database in Python (use the sqlite3 module). Document the connection by making some simple queries.

Formulation:

The exercise contains how to establish connection to the database ‘northwind.db’. We will also show some examples of querying the database.

SQLite Solution:

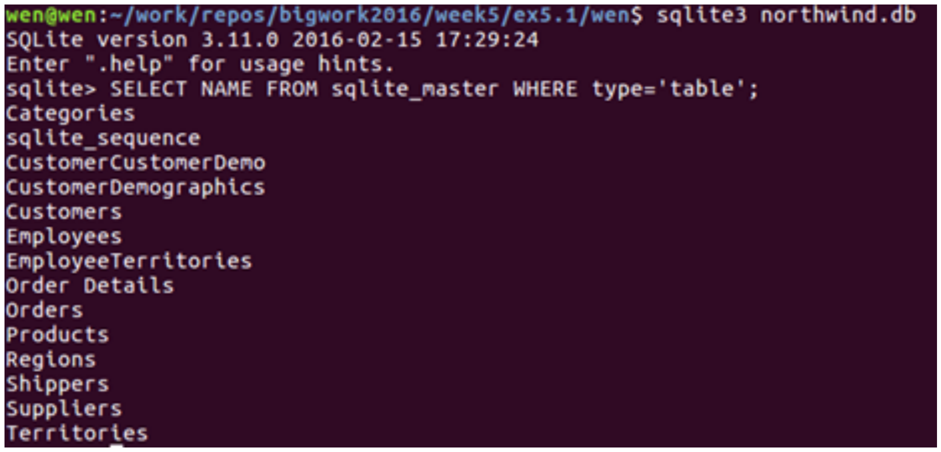
The following paragraphs will show how we made connection to the database.
Two query examples followed will validate the connection. The first one is to list all tables of the database. The second one is to list the CustomerID from Orders table.

1. List all tables

```
$ sqlite3 northwind.db
Sqlite> SELECT NAME FROM sqlite_master WHERE type='table';
```

The query would list all the tables from ‘northwind.db’. The query will put out the name of the all tables in the database. The SELECT is used to query the database.

Result:



```
wen@wen:~/work/repos/bigwork2016/week5/ex5.1/wen$ sqlite3 northwind.db
SQLite version 3.11.0 2016-02-15 17:29:24
Enter ".help" for usage hints.
sqlite> SELECT NAME FROM sqlite_master WHERE type='table';
Categories
sqlite_sequence
CustomerCustomerDemo
CustomerDemographics
Customers
Employees
EmployeeTerritories
Order Details
Orders
Products
Regions
Shippers
Suppliers
Territories
```

Figure 5.1.1 the output of the query

Figure 5.1.1 shows the output result of the query which shows all lists. The result could be validated by look up the ‘Northwind’ database.

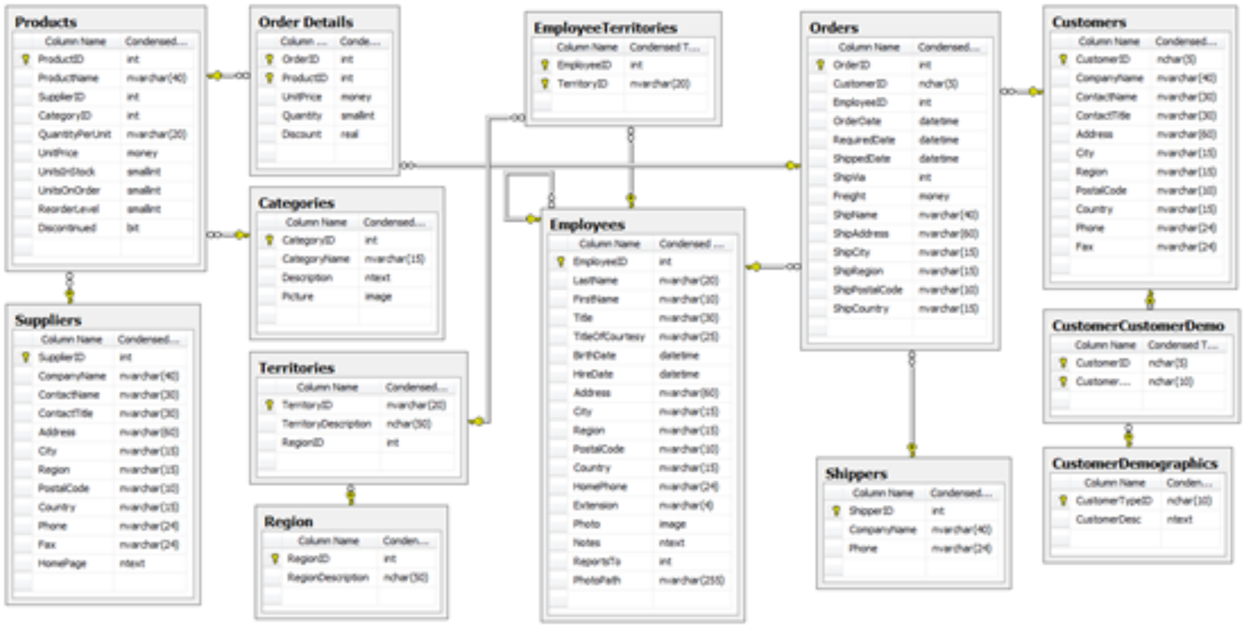


Figure 5.1.2 all the tables of the database

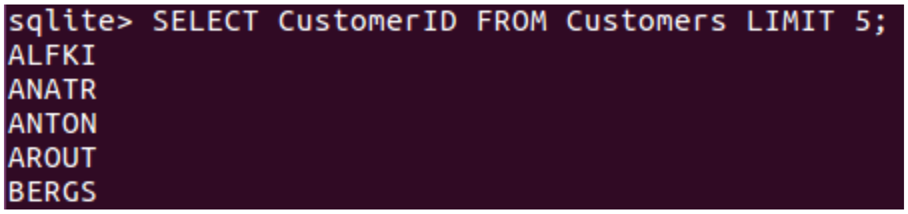
Figure 5.1.2 show the image of 'Northwind' database. The original image could be view on https://northwinddatabase.codeplex.com/. This validates the output of the query is correct.

2. List Customer IDs

```
sqlite> SELECT CustomerID FROM Customers LIMIT 5;
```

The query would list out 5 CustomerIDs from the 'Customers' table.

Result:



```
sqlite> SELECT CustomerID FROM Customers LIMIT 5;
ALFKI
ANATR
ANTON
AROUT
BERGS
```

Figure 5.1.3 part of the CustomerID

Figure 5.1.3 shows by this query the terminal outputs a limited number of the CustomerID from the 'Customers' table.

Exercise 5.2:

The customer with customerID ALFKI has made a number of orders containing some products. Query for, and return, all orders made by ALFKI and the products they contain.

Formulation:

The exercise will show database queries of orders made by 'ALFKI'. We will also show the all the products that the orders contain.

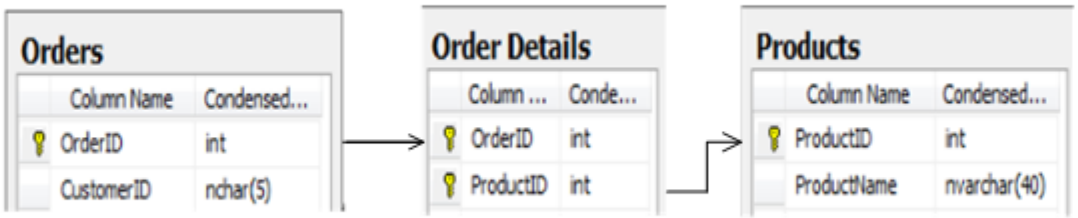


Figure 5.2.1 the relations between the 'CustomerID', 'OrderID', 'ProductID'

Figure 5.2.1 shows the relations between the keys. The exercise can be solved by using query of subqueries method.

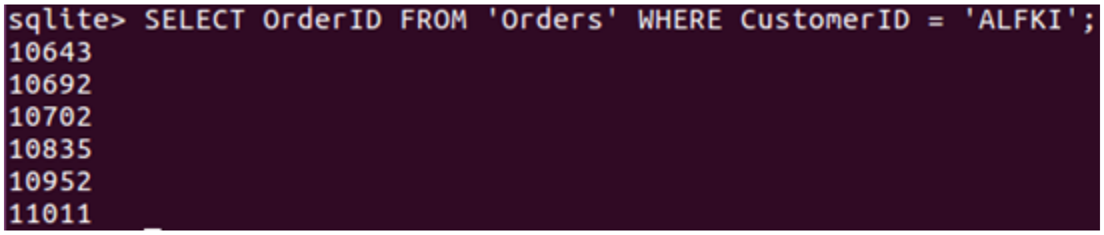
SQLite Solution:

1. Queries of all Orders made by 'ALFKI'

```
sqlite> SELECT OrderID FROM 'Orders' WHERE CustomerID = 'ALFKI';
```

The query will select 'OrderID' from the table 'Orders' where the 'CustomerID' is 'ALFKI'.

SQLite Result:



```
sqlite> SELECT OrderID FROM 'Orders' WHERE CustomerID = 'ALFKI';
10643
10692
10702
10835
10952
11011
```

Figure 5.2.2 the query and result of 'Orders'

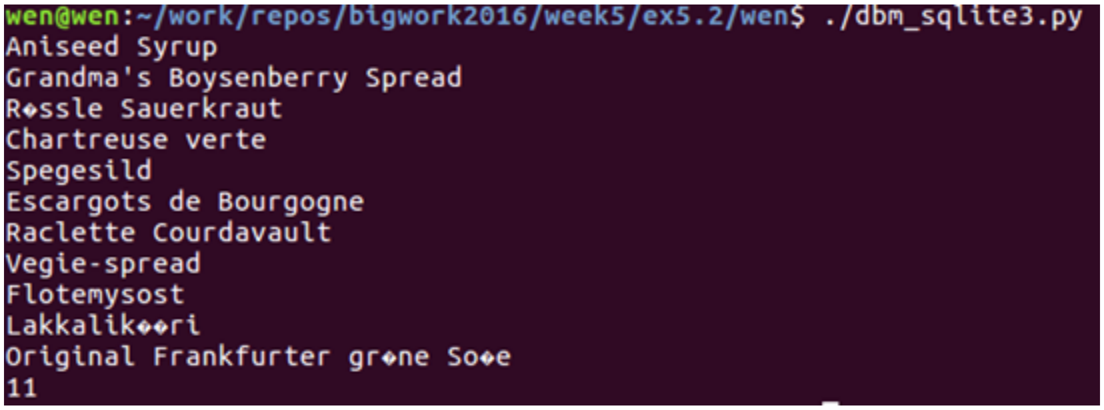
Figure 5.2.2 shows the single query and result of 'Orders' made by 'ALFKI'. We can see that there are 6 orders in total. Single query is quite convenient to input into the SQLite command line. The following paragraph would be a complicate single query. For the documenting purpose those will be written in python 2.7.

2. List the products made from the orders of 'ALFKI'

```
    cur.execute(
      "SELECT ProductName FROM 'Products' WHERE ProductID IN(      \
        SELECT ProductID FROM 'Order Details' WHERE OrderID IN(    \
        SELECT OrderID FROM 'Orders' WHERE CustomerID = 'ALFKI'))")
```

The ‘Cursor’ object and execute() method is used to perform the SQL command. The command is a query of ‘ProductName’ from ‘Products’ table where the ‘ProductID’ is defined by subquery. The ‘ProductID’ is defined by another subquery which is the given condition ‘CustomerID’ = ‘ALFKI’

SQLite Result:



```
wen@wen:~/work/repos/bigwork2016/week5/ex5.2/wen$ ./dbm_sqlite3.py
Aniseed Syrup
Grandma's Boysenberry Spread
R◆ssle Sauerkraut
Chartreuse verte
Spegesild
Escargots de Bourgogne
Raclette Courdavault
Vegie-spread
Flotemysost
Lakkalik◆◆ri
Original Frankfurter gr◆ne So◆e
11
```

Figure 5.2.3 all unique kinds of products ordered by ‘ALFKI’

Figure 5.2.3 shows all of the products ordered by ‘ALFKI’ and also shows the total number of the different products which is eleven.

Exercise 5.3:

Get all orders (with products) made by ALFKI that contain at least 2 product types.
The exercise will show the query of all orders made by 'ALFKI' which has at least two types.

Formulation:

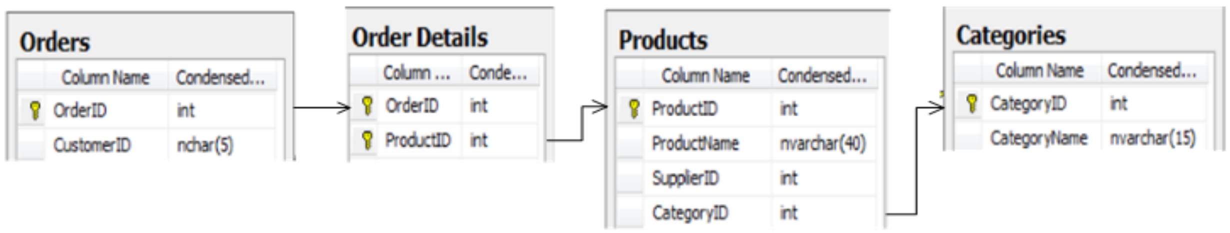


Figure 5.3.1 the relations between 'OrdersID', 'ProductID' and 'CategoryID'

Figure 5.3.1 shows the relations between the keys. The exercise can be solved by 'SELECT', 'GROUP', 'HAVING' queries.

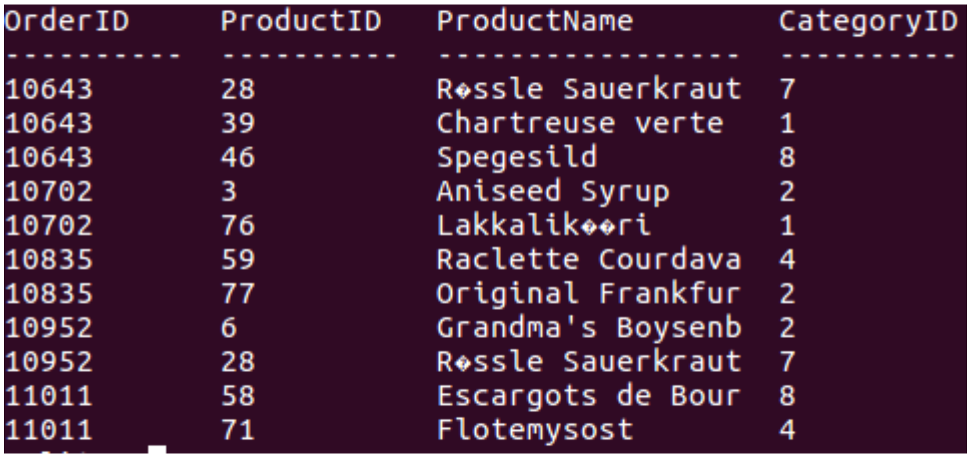
SQLite Solution:

List 'OrderID', 'ProductID', 'ProductName', 'CategoryID' of orders which have at least two types of products.

```
cur.execute("                                                         \
SELECT Orders.OrderID,Products.ProductID,ProductName,CategoryID \
FROM Orders,'Order Details',Products                                  \
WHERE Orders.OrderID='Order Details'.OrderID AND                      \
      'Order Details'.ProductID=Products.ProductID AND                 \
      Orders.OrderID IN (                                              \
      SELECT Orders.OrderID FROM Orders,'Order Details',Products\
      WHERE Orders.OrderID='Order Details'.OrderID AND                 \
           'Order Details'.ProductID=Products.ProductID AND            \
            CustomerID='ALFKI'                                         \
            GROUP BY Orders.OrderID                                    \
            HAVING count(CategoryID)>1)")
```

The SQL commands query to the 'OrderID', 'ProductID', 'ProductName', 'CategoryID. The 'OrderID' is constrained so that it should have at least two types.

SQLite Result:



```
OrderID     ProductID   ProductName        CategoryID
----------  ----------  -----------------  ----------
10643       28          R�ssle Sauerkraut  7
10643       39          Chartreuse verte   1
10643       46          Spegesild          8
10702       3           Aniseed Syrup      2
10702       76          Lakkalik��ri       1
10835       59          Raclette Courdava  4
10835       77          Original Frankfur  2
10952       6           Grandma's Boysenb  2
10952       28          R�ssle Sauerkraut  7
11011       58          Escargots de Bour  8
11011       71          Flotemysost        4
```

Figure 5.3.2 outputs of ProductID made by ‘ALFKI’

Figure 5.3.2 shows all orders made by ‘ALFKI’ which contains at least two types of products. We can see that there are 11 orders in total.

Exercise 5.4:

Determine how many and who ordered “Uncle Bob’s Organic Dried Pears” (productID 7).

Formulation:

The exercise will show the query to the ‘ContactName’ who ordered “Uncle Bob’s Organic Dried Pears”. Since the ‘ProductID’ of the ‘ProductName’ is given as 7, the ‘ProductID’ can be used.

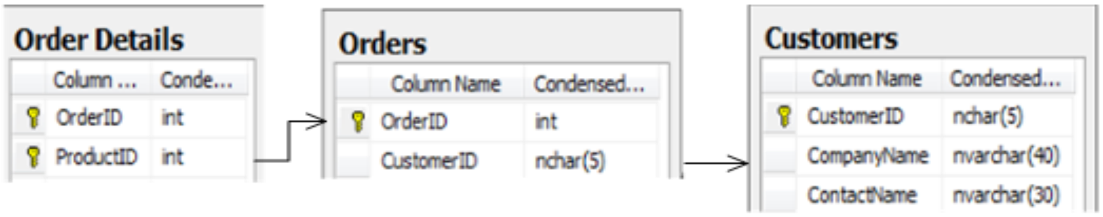


Figure 5.4.1 the relations between ‘ProductID’, ‘OrderID’, ‘ContactName’

Figure 5.4.1 shows the relations between the keys. As for this exercise, a query of cascaded sub queries could be used to solve the problem.

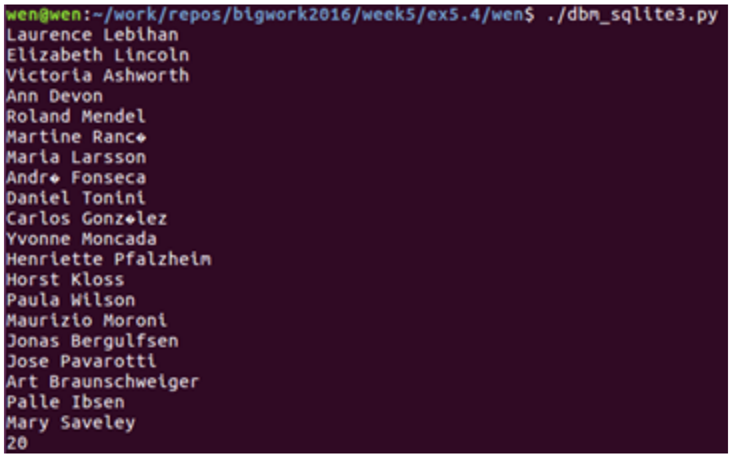
SQLite Solution:

List of all ‘ContactName’ who ordered “Uncle Bob’s Organic Dried Pears” (productID 7).

```
cur.execute("
      SELECT ContactName From 'Customers' WHERE CustomerID IN(\
      SELECT CustomerID FROM 'Orders' WHERE OrderID IN(       \
      SELECT OrderID FROM 'Order Details' WHERE ProductID=7))")
```

The SQL command will query ‘ContactName’ From ‘Customers’ where ‘CustomerID’ is defined by subquery from the table ‘Orders’. The cascaded ‘SELECT’ and ‘WHERE’ clause will output the value defined by the destination table which contains ‘Product = 7’.

SQLite Result:



```
wen@wen:~/work/repos/bigwork2016/week5/ex5.4/wen$ ./dbm_sqlite3.py
Laurence Lebihan
Elizabeth Lincoln
Victoria Ashworth
Ann Devon
Roland Mendel
Martine Ranc◆
Maria Larsson
Andr◆ Fonseca
Daniel Tonini
Carlos Gonz◆lez
Yvonne Moncada
Henriette Pfalzheim
Horst Kloss
Paula Wilson
Maurizio Moroni
Jonas Bergulfsen
Jose Pavarotti
Art Braunschweiger
Palle Ibsen
Mary Saveley
20
```

Figure 5.4.2 all the ‘ContactName’ and number of the Customers

Exercise 5.5:

How many different and which products have been ordered by customers who have also ordered "Uncle Bob's Organic Dried Pears"?

Formulation:

This exercise will show the query to the different product names and numbers by given condition.

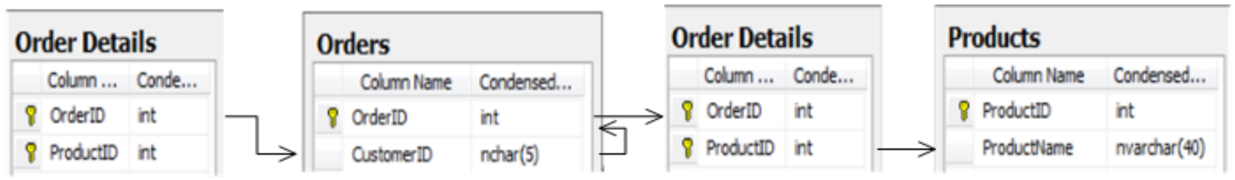


Figure 5.5.1 the relations between keys

Figure 5.5.1 shows the realaitons between keys. The exercise is a bit different other than 5.3, 5.4. Because of the 'Order Details' has been looked up twice. The reason is the exercise needs to find al l 'CustomerID' by the primary condition. To the exercise, cascaded 'SELECT' and 'WHERE' clause could be used.

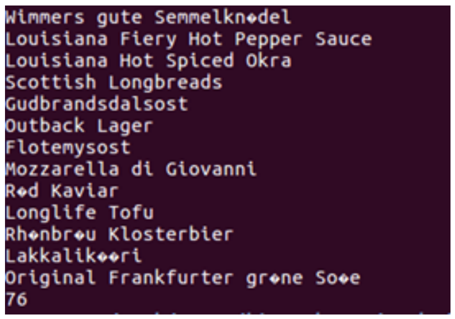
SQLite Solution:

List all different 'ProductName' as the exercise requested

```
cur.execute("
  SELECT ProductName FROM 'Products' WHERE ProductID IN(          \
  SELECT DISTINCT ProductID FROM 'Order Details' WHERE OrderID IN(\
  SELECT OrderID FROM 'Orders' WHERE CustomerID IN(               \
  SELECT CustomerID FROM 'Orders' WHERE OrderID IN(               \
  SELECT OrderID FROM 'Order Details' WHERE ProductID=7)))))")
```

The SQL command is query to 'ProductName' where 'Product ID is defined by subquery. Since the requirement is to find unique products, a 'DISTINCT' keyword is used to limit 'ProductID'. The rest of the 'SELECT' and 'WHERE' clauses look up all the way to the primary condition. And also as we mentioned before, there is a forth and back on the key 'OrderID'.

SQLite Result:



```
Wimmers gute Semmelkn◆del
Louisiana Fiery Hot Pepper Sauce
Louisiana Hot Spiced Okra
Scottish Longbreads
Gudbrandsdalsost
Outback Lager
Flotemysost
Mozzarella di Giovanni
R◆d Kaviar
Longlife Tofu
Rh◆nbr◆u Klosterbier
Lakkalik◆◆ri
Original Frankfurter gr◆ne So◆e
76
```

Figure 5.5.2 part of the 76 different products

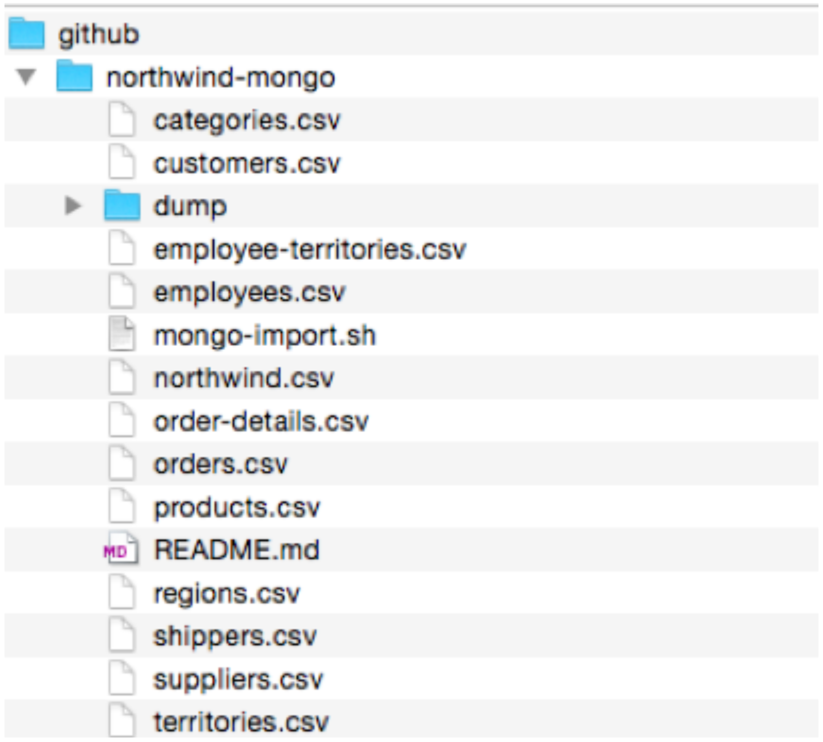
//Part2. Mongo Part

Exercise 5.1:

For Mongo: To get started, clone this repository into your working directory. Start a running instance of MongoDB* (on command-line: *mongod*), then run the .sh file in a terminal. This should create a live Mongo database named 'Northwind' that you can connect to in Python. Document the connection by making some simple queries.

SOLUTION:

1. First, we cloned the repository *https://github.com/tmcnab/northwind-mongo* into our working directory.
The commands and methods used here are the same as those in Ex 1.5, so we won't go into more details.



```
github
▼ northwind-mongo
    categories.csv
    customers.csv
  ▶ dump
    employee-territories.csv
    employees.csv
    mongo-import.sh
    northwind.csv
    order-details.csv
    orders.csv
    products.csv
    README.md
    regions.csv
    shippers.csv
    suppliers.csv
    territories.csv
```

Figure. A Quick View of Our Working Directory

2. Then after installing MongoDB, we started a running instance of MongoDB*.
According to the tutorial, we did the following things.
1) Install **brew**.
   ***:~ ***$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
2) Then check if brew is already up-to-date.
   ***:~ ***$ brew update

Already up-to-date.

3) Then install mongob.

***:~ ***$ brew install mongodb

The output showed that installation had been successful.

4) Then start mongodb using brew.

***:~ ***$ brew services start mongodb

***:~ ***$ mongo

3. Then ran the .sh file in a terminal.

***:Desktop ***$ cd northwind

***:northwind ***$ cd github

***:github ***$ cd northwind-mongo

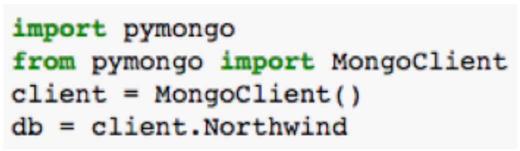
***:northwind-mongo ***$ bash mongo-import.sh

Then the live Mongo database named 'Northwind' that we can connect to in Python is created successfully.

4. We next document the connection by making some queries.

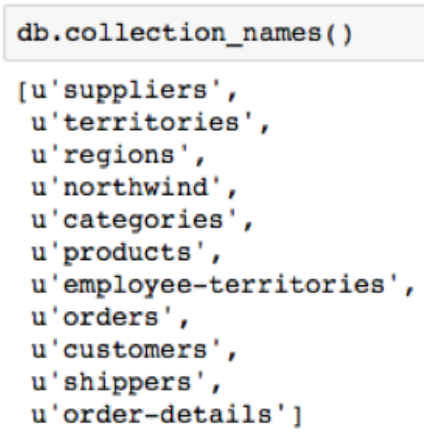
1) The first step when working with PyMongo is to create a MongoClient to the running mongod instance. The following code will connect on the default host and port.

2) Then we accessed databases using attribute style access on MongoClient instances.



```
import pymongo
from pymongo import MongoClient
client = MongoClient()
db = client.Northwind
```

3) Then we got all the names of the collections using the *db.collection_names()*



```
db.collection_names()
```

```
[u'suppliers',
 u'territories',
 u'regions',
 u'northwind',
 u'categories',
 u'products',
 u'employee-territories',
 u'orders',
 u'customers',
 u'shippers',
 u'order-details']
```

4) We took a look at one piece of information in the form "suppliers" using *find_one()*

```
db.suppliers.find_one()
```

```
{u'Address': u'49 Gilbert St.',
 u'City': u'London',
 u'CompanyName': u'Exotic Liquids',
 u'ContactName': u'Charlotte Cooper',
 u'ContactTitle': u'Purchasing Manager',
 u'Country': u'UK',
 u'Fax': u'NULL',
 u'HomePage': u'NULL',
 u'Phone': u'(171) 555-2222',
 u'PostalCode': u'EC1 4SD',
 u'Region': u'NULL',
 u'SupplierID': 1,
 u'_id': ObjectId('57ecd557e5e4f52d7a31b150')}
```

5) We counted the total number of documents in the form “suppliers” using *db.suppliers.find().count()*

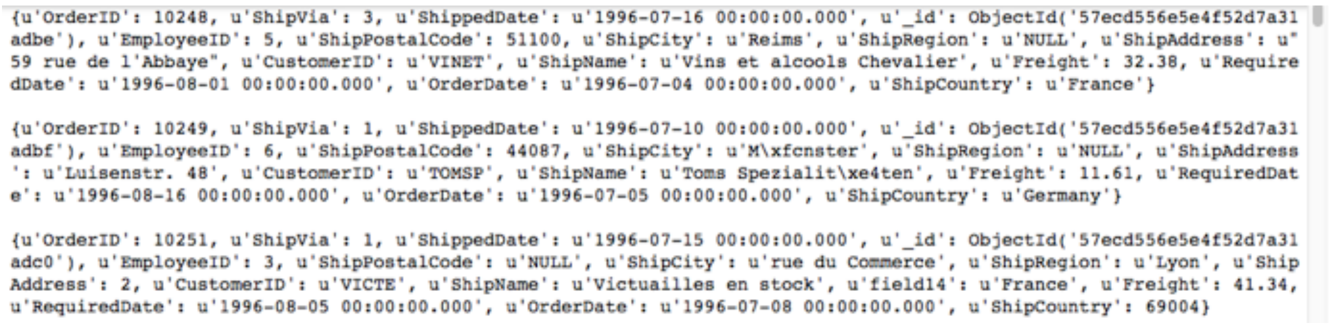
```
db.suppliers.find().count()
```

```
29
```

6) We printed out all the items in the form orders using *find()* and print.

```
for item in db.orders.find():
    print item
    print
```

This is the sample output:



```
{u'OrderID': 10248, u'ShipVia': 3, u'ShippedDate': u'1996-07-16 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31
adbe'), u'EmployeeID': 5, u'ShipPostalCode': 51100, u'ShipCity': u'Reims', u'ShipRegion': u'NULL', u'ShipAddress': u"
59 rue de l'Abbaye", u'CustomerID': u'VINET', u'ShipName': u'Vins et alcools Chevalier', u'Freight': 32.38, u'Require
dDate': u'1996-08-01 00:00:00.000', u'OrderDate': u'1996-07-04 00:00:00.000', u'ShipCountry': u'France'}

{u'OrderID': 10249, u'ShipVia': 1, u'ShippedDate': u'1996-07-10 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31
adbf'), u'EmployeeID': 6, u'ShipPostalCode': 44087, u'ShipCity': u'M\xfcnster', u'ShipRegion': u'NULL', u'ShipAddress
': u'Luisenstr. 48', u'CustomerID': u'TOMSP', u'ShipName': u'Toms Spezialit\xe4ten', u'Freight': 11.61, u'RequiredDat
e': u'1996-08-16 00:00:00.000', u'OrderDate': u'1996-07-05 00:00:00.000', u'ShipCountry': u'Germany'}

{u'OrderID': 10251, u'ShipVia': 1, u'ShippedDate': u'1996-07-15 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31
adc0'), u'EmployeeID': 3, u'ShipPostalCode': u'NULL', u'ShipCity': u'rue du Commerce', u'ShipRegion': u'Lyon', u'Ship
Address': 2, u'CustomerID': u'VICTE', u'ShipName': u'Victuailles en stock', u'field14': u'France', u'Freight': 41.34,
u'RequiredDate': u'1996-08-05 00:00:00.000', u'OrderDate': u'1996-07-08 00:00:00.000', u'ShipCountry': 69004}
```

Exercise 5.2:

The customer with customerID ALFKI has made a number of orders containing some products. Query for, and return, all orders made by ALFKI and the products they contain.

SOLUTION:

We want to get those documents in the form “orders” whose customerID is “ALFKI”. Thus, we use the function *db.orders.find({"CustomerID":"ALFKI"})*

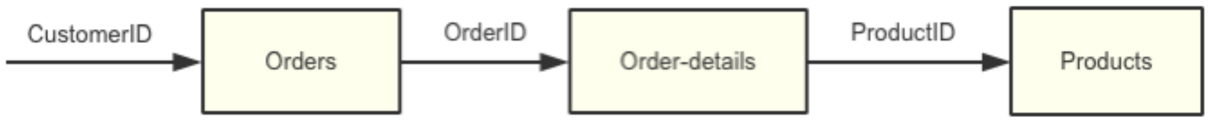
- Check out the total number of orders made by ALFKI:

```
db.orders.find({"CustomerID":"ALFKI"}).count()
```

```
6
```

- Print out all the orders made by ALFKI and the products they contain.

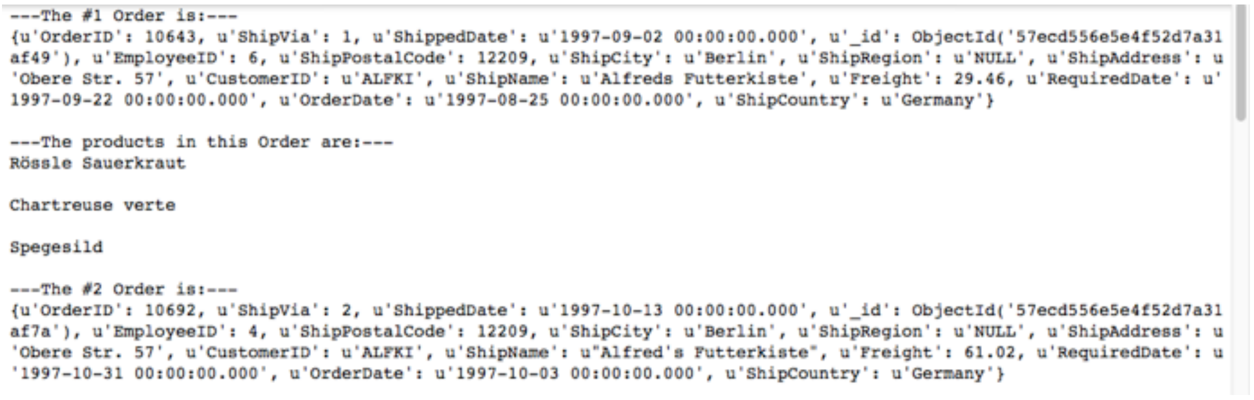
First, we analyze the relationship of the three form “orders”, “order-details” and “products” by printing the documents of them.



So we first got the orders made by ALFKI from the form ”orders”, then printed it out. Then we got documents in form “order-details” using “OrderID” in orders made by ALFKI. Then we got the products from the form “products” using “ProductID” got from the “order-details”.

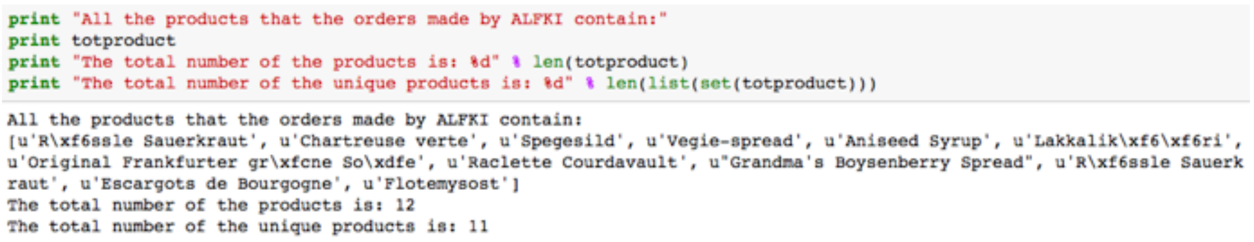
```
i = 1
totproduct = []
for item in db.orders.find({"CustomerID":"ALFKI"}):
    print "---The #%d Order is:---" % i
    print item
    print
    i += 1
    print "---The products in this Order are:---"
    for details in db['order-details'].find({"OrderID":item["OrderID"]}):
        for prod in db.products.find({"ProductID":details["ProductID"]}):
            print prod["ProductName"]
            print
            totproduct.append(prod["ProductName"])
```

And here is the sample output.



```
---The #1 Order is:---
{u'OrderID': 10643, u'ShipVia': 1, u'ShippedDate': u'1997-09-02 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31
af49'), u'EmployeeID': 6, u'ShipPostalCode': 12209, u'ShipCity': u'Berlin', u'ShipRegion': u'NULL', u'ShipAddress': u
'Obere Str. 57', u'CustomerID': u'ALFKI', u'ShipName': u'Alfreds Futterkiste', u'Freight': 29.46, u'RequiredDate': u'
1997-09-22 00:00:00.000', u'OrderDate': u'1997-08-25 00:00:00.000', u'ShipCountry': u'Germany'}

---The products in this Order are:---
Rössle Sauerkraut

Chartreuse verte

Spegesild

---The #2 Order is:---
{u'OrderID': 10692, u'ShipVia': 2, u'ShippedDate': u'1997-10-13 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31
af7a'), u'EmployeeID': 4, u'ShipPostalCode': 12209, u'ShipCity': u'Berlin', u'ShipRegion': u'NULL', u'ShipAddress': u
'Obere Str. 57', u'CustomerID': u'ALFKI', u'ShipName': u"Alfred's Futterkiste", u'Freight': 61.02, u'RequiredDate': u
'1997-10-31 00:00:00.000', u'OrderDate': u'1997-10-03 00:00:00.000', u'ShipCountry': u'Germany'}
```

As for the total products in all these orders made by ALFKI, we used a list “totproduct” to store the names of these products. And we got:



```
print "All the products that the orders made by ALFKI contain:"
print totproduct
print "The total number of the products is: %d" % len(totproduct)
print "The total number of the unique products is: %d" % len(list(set(totproduct)))
```

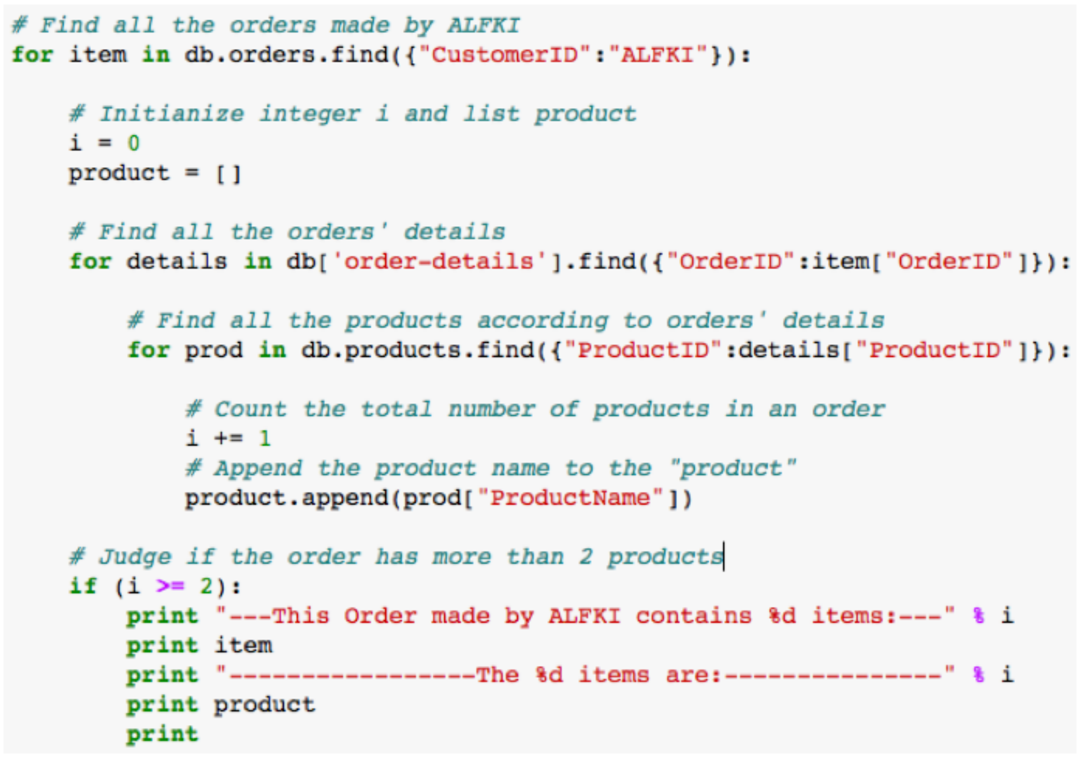
```
All the products that the orders made by ALFKI contain:
[u'R\xf6ssle Sauerkraut', u'Chartreuse verte', u'Spegesild', u'Vegie-spread', u'Aniseed Syrup', u'Lakkalik\xf6\xf6ri',
u'Original Frankfurter gr\xfcne So\xdfe', u'Raclette Courdavault', u"Grandma's Boysenberry Spread", u'R\xf6ssle Sauerk
raut', u'Escargots de Bourgogne', u'Flotemysost']
The total number of the products is: 12
The total number of the unique products is: 11
```

And to check out if this result is correct, we found that in this part there were 6 orders. By manually checking the first order, for example, and compare that with the result got in SQLite results, we concluded that we were getting the right results.

Exercise 5.3:

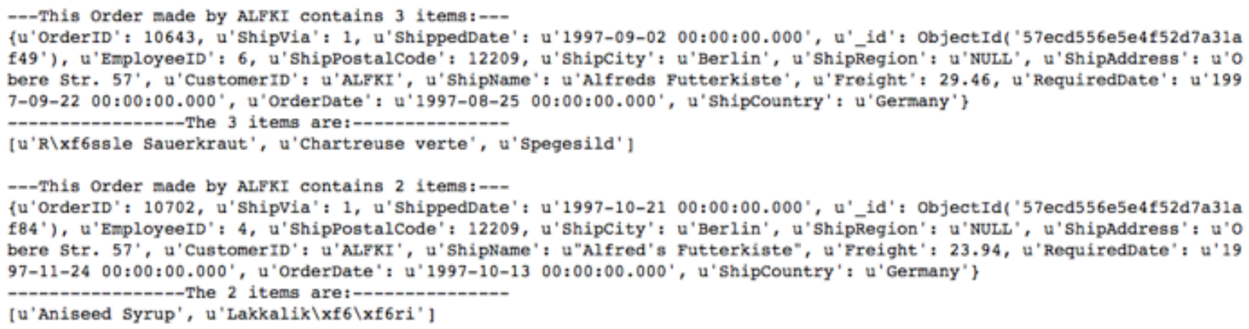
Get all orders (with products) made by ALFKI that contain at least 2 product types.

SOLUTION:
We first got all the orders made by ALFKI, and then find the total number of the products in each order. We extracted the orders that contain more than 2 items, printed the order, and printed the products.



```python
# Find all the orders made by ALFKI
for item in db.orders.find({"CustomerID":"ALFKI"}):

    # Initianize integer i and list product
    i = 0
    product = []

    # Find all the orders' details
    for details in db['order-details'].find({"OrderID":item["OrderID"]}):

        # Find all the products according to orders' details
        for prod in db.products.find({"ProductID":details["ProductID"]}):

            # Count the total number of products in an order
            i += 1
            # Append the product name to the "product"
            product.append(prod["ProductName"])

    # Judge if the order has more than 2 products
    if (i >= 2):
        print "---This Order made by ALFKI contains %d items:---" % i
        print item
        print "-----------------The %d items are:---------------" % i
        print product
        print
```

And then we got the sample putout.



```
---This Order made by ALFKI contains 3 items:---
{u'OrderID': 10643, u'ShipVia': 1, u'ShippedDate': u'1997-09-02 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31a
f49'), u'EmployeeID': 6, u'ShipPostalCode': 12209, u'ShipCity': u'Berlin', u'ShipRegion': u'NULL', u'ShipAddress': u'O
bere Str. 57', u'CustomerID': u'ALFKI', u'ShipName': u'Alfreds Futterkiste', u'Freight': 29.46, u'RequiredDate': u'199
7-09-22 00:00:00.000', u'OrderDate': u'1997-08-25 00:00:00.000', u'ShipCountry': u'Germany'}
-----------------The 3 items are:---------------
[u'R\xf6ssle Sauerkraut', u'Chartreuse verte', u'Spegesild']

---This Order made by ALFKI contains 2 items:---
{u'OrderID': 10702, u'ShipVia': 1, u'ShippedDate': u'1997-10-21 00:00:00.000', u'_id': ObjectId('57ecd556e5e4f52d7a31a
f84'), u'EmployeeID': 4, u'ShipPostalCode': 12209, u'ShipCity': u'Berlin', u'ShipRegion': u'NULL', u'ShipAddress': u'O
bere Str. 57', u'CustomerID': u'ALFKI', u'ShipName': u"Alfred's Futterkiste", u'Freight': 23.94, u'RequiredDate': u'19
97-11-24 00:00:00.000', u'OrderDate': u'1997-10-13 00:00:00.000', u'ShipCountry': u'Germany'}
-----------------The 2 items are:---------------
[u'Aniseed Syrup', u'Lakkalik\xf6\xf6ri']
```

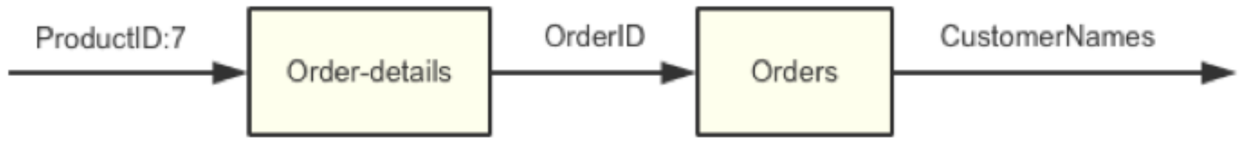
By comparing the integer i with the printed products list, we concluded that we got the right result.

Exercise 5.4:

Determine how many and who ordered “Uncle Bob’s Organic Dried Pears” (productID 7).

SOLUTION:

We use the following graph to represent the relationships of the three forms.



Here is the code and the attached texts denote what we’re doing.

```
# Initialize the list "customer"
customer = []

# Find all the orders whose product ID is 7
for details in db['order-details'].find({"ProductID":7}):

    # Append the names of the customers who made these orders
    for item in db.orders.find({"OrderID":details["OrderID"]}):
        customer.append(item["CustomerID"])

# Leave out the reduplicate names
ucustomer = list(set(customer))

# Print the customers who ordered “Uncle Bob’s Organic Dried Pears” (productID 7)
print "Below are customers who ordered “Uncle Bob’s Organic Dried Pears” (productID 7):"
print ucustomer
print

# Print the number of those customers
print "Below is the number of customers who ordered “Uncle Bob’s Organic Dried Pears” (productID 7):"
print len(ucustomer)
```

Here is the output, which can be proved correct by comparing that with SQLite.

```
Below are customers who ordered “Uncle Bob’s Organic Dried Pears” (productID 7):
[u'GOURL', u'VAFFE', u'BOTTM', u'FOLKO', u'BSBEV', u'EASTC', u'FOLIG', u'ERNSH', u'BONAP', u'SANTG', u'REGGC', u'OTTIK
', u'RATTC', u'VICTE', u'LILAS', u'QUICK', u'SAVEA', u'OCEAN', u'SPLIR', u'LACOR']

Below is the number of customers who ordered “Uncle Bob’s Organic Dried Pears” (productID 7):
20
```

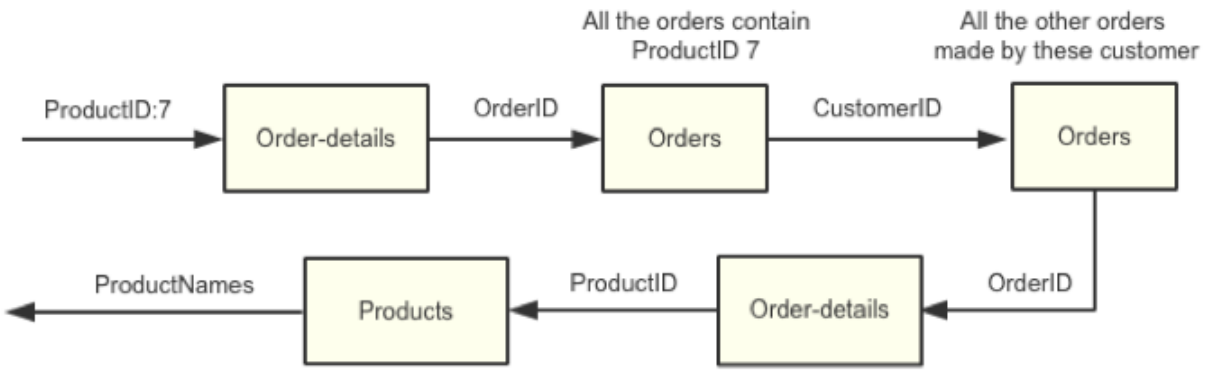
Exercise 5.5:

How many different and which products have been ordered by customers who have also ordered “Uncle Bob’s Organic Dried Pears”?

SOLUTION:

Here we first found the order details which contain ProductID7, then we found all the customers who made these orders. Using CustomerID, we were able to find all the other orders made by these customers. Then we could find out all the other products.

Below is a flowchart which demonstrate the steps.

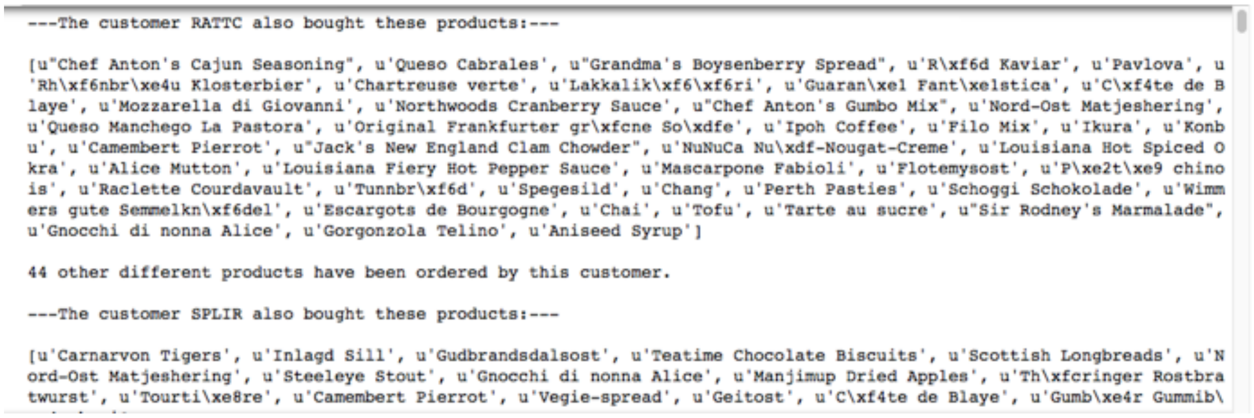


These are the codes, with texts explaining them.

```
# Initialize list difproduct for total different products purchased by all the customers
difproduct = []

# Find all the order details that have “Uncle Bob’s Organic Dried Pears” (productID 7)
for details in db['order-details'].find({"ProductID":7}):

    # Find all the orders of those details according to OrderID
    for item in db.orders.find({"OrderID":details["OrderID"]}):
        print "---The customer " + item["CustomerID"] + " also bought these products:---"
        print

        # Initialize list product for different products purchased by each customer
        product = []

        # According to the CustomerID, find all the orders made by those customers who also
        # purchased productID 7
        for item2 in db.orders.find({"CustomerID":item["CustomerID"]}):

            # Find the details of the orders to aquire information of the products in those order
            for details2 in db['order-details'].find({"OrderID":item2["OrderID"]}):
                for prod in db.products.find({"ProductID":details2["ProductID"]}):

                    # append the product of the orders to the list "product"
                    product.append(prod["ProductName"])

        # Remove the repeated products
        uproduct = list(set(product))
```

```
# Remove the ProductID 7, because we need OTHER products
if "Uncle Bob's Organic Dried Pears" in uproduct:
    uproduct.remove("Uncle Bob's Organic Dried Pears")
print uproduct
print

print "%d other different products have been ordered by this customer." % len(uproduct)
print

# Append each customer's different product to this list of lists
difproduct.append(uproduct)
```

Here is the sample output.



```
---The customer RATTC also bought these products:---

[u"Chef Anton's Cajun Seasoning", u'Queso Cabrales', u"Grandma's Boysenberry Spread", u'R\xf6d Kaviar', u'Pavlova', u
'Rh\xf6nbr\xe4u Klosterbier', u'Chartreuse verte', u'Lakkalik\xf6\xf6ri', u'Guaran\xe1 Fant\xe1stica', u'C\xf4te de B
laye', u'Mozzarella di Giovanni', u'Northwoods Cranberry Sauce', u"Chef Anton's Gumbo Mix", u'Nord-Ost Matjeshering',
u'Queso Manchego La Pastora', u'Original Frankfurter gr\xfcne So\xdfe', u'Ipoh Coffee', u'Filo Mix', u'Ikura', u'Konb
u', u'Camembert Pierrot', u"Jack's New England Clam Chowder", u'NuNuCa Nu\xdf-Nougat-Creme', u'Louisiana Hot Spiced O
kra', u'Alice Mutton', u'Louisiana Fiery Hot Pepper Sauce', u'Mascarpone Fabioli', u'Flotemysost', u'P\xe2t\xe9 chino
is', u'Raclette Courdavault', u'Tunnbr\xf6d', u'Spegesild', u'Chang', u'Perth Pasties', u'Schoggi Schokolade', u'Wimm
ers gute Semmelkn\xf6del', u'Escargots de Bourgogne', u'Chai', u'Tofu', u'Tarte au sucre', u"Sir Rodney's Marmalade",
u'Gnocchi di nonna Alice', u'Gorgonzola Telino', u'Aniseed Syrup']

44 other different products have been ordered by this customer.

---The customer SPLIR also bought these products:---

[u'Carnarvon Tigers', u'Inlagd Sill', u'Gudbrandsdalsost', u'Teatime Chocolate Biscuits', u'Scottish Longbreads', u'N
ord-Ost Matjeshering', u'Steeleye Stout', u'Gnocchi di nonna Alice', u'Manjimup Dried Apples', u'Th\xfcringer Rostbra
twurst', u'Tourti\xe8re', u'Camembert Pierrot', u'Vegie-spread', u'Geitost', u'C\xf4te de Blaye', u'Gumb\xe4r Gummib\
```

We finally got the number of all the different products --- 75.

Be careful this is not 76 because we excluded the product "Uncle Bob's Organic Dried Pears", because the question requires to output “how many DIFFERENT and which”. :)

```
# Flatten the list of lists of all the different products
flat = [item for sublist in difproduct for item in sublist]

# The length of "flat" is the number of total diffrent items
print len(list(set(flat)))
```

```
75
```

//Part3. Comparison Part

MongoDB is one of the most popular document stores, meanwhile SQLite is a kind of Widely used in-process RDBMS.

- In SQLite, we use SQL queries. SQL is a lightweight declarative language. It's deceptively powerful, and has become an international standard, although most systems implement subtly different syntaxes.
  The SQL language grammar is quite like English. And all of its commands are in the upper case.
  To use it, first we create a connection to the database. Then from the connection, we get the cursor object. And after getting the version of the SQLite database, we are able to conduct execute() on the database.
  It has fixed schema, but do not have the same data structure as MongoDB.
- In MongoDB, we use python. A single instance of MongoDB can support multiple independent databases.
  The first step when working with PyMongo is to create a MongoClient to the running mongod instance. After making this connection, we can get the database .
  Then we get collections, which is equivalent to tables in a relational database.
  Data in MongoDB is represented (and stored) using JSON-style documents. In PyMongo we use dictionaries to represent documents. So the queries and other executions are similar to that of json and dictionaries.