Data Management Report

Master Degree in Data Science

Project 1: Querying a database

16 January 2020

- **Candidate 1: Christian Riccio P37000002**
- **Candidate 2: Giacomo Matrone P37000011**

Introduction

The following report comes from an analysis of mock-up database that was downloaded from Oracle.com. The data regards the operativity of a multinational firm that sells world wide computer parts and has different warehouses all over the world. Our analysis has been focused in discovering and exploiting actionable informations about the activity of the firm.

In particular, we discovered the following results:

- The average profit per unit by item category and warehouse's cities;
- Gross profit mean and variance.

We also created two classes `sql_parser` and `db_writer` which are specified below and are used to automate:

- the creation of the tables;
- the insertion of the data.

```
import os
import re
import cx_Oracle

class sql_parser():

    def __init__(self,path):

        self.path = path


    def sql_parse(self,sep,file):
        with open(self.path+sep+file,'r+') as file:
            data = file.read()
        queries = [re.findall('CREATE .*|Insert .*|ALTER .*|SET DEFINE .*',el)\
                   for el in re.sub('\n|\t| -- fk', ' ',data).split(';')]
        return queries



class db_writer():

    def __init__(self,host,sid,port,user,password):

        self.host = host
        self.sid = sid
        self.user = user
        self.port = port
        self.password = password

    def db_create_table(self,sql_statement):

        dsn = cx_Oracle.makedsn(self.host,self.port,self.sid)
        connection = cx_Oracle.connect(self.user,self.password, dsn, cx_Oracle.S
YSDBA)
        cursor = connection.cursor()
        if sql_statement.startswith('CREATE TABLE'):
            cursor.execute(sql_statement)
        else:
            raise Exception("Is not a Create table operation")
        connection.close()
```

In [1]:

```
import sys
sys.path.append("C:\\Users\\Win\\Desktop\\Riccio_Matrone\\script_python")
```

In [4]:

```
import cx_Oracle
import pandas as pd
import matplotlib.pyplot as plt
import os
import seaborn as sns
from Parser import sql_parser
from db_ops import db_writer
```

In [5]:

```
queries = {file: sql_parser("C:\\Users\\Win\\Desktop\\Riccio_Matrone\\SQL").sql_parse("
\\",file) for file in os.listdir("C:\\Users\\Win\\Desktop\\Riccio_Matrone\\SQL\\")}
```

In [8]:

```
queries.keys()
```

Out[8]:

```
dict_keys(['ot_data_new.sql', 'ot_schema.sql'])
```

Connection to the Database

In this section we define the variables used in order to set the connection to our SQL database, via cx_Oracle library. We also define a function in order to automate queries execution and we present the results in pandas dataframe format.

In [7]:

```
user = "SYS as SYSDBA"
password =
host = '127.0.0.1'
port = '1521'
sid = 'orcl'
conn_str = 'jdbc:oracle:thin:@{0}:{1}:{2}'.format(host,port,sid)
```

In [12]:

```
db_constructor = db_writer(host,sid,port,user,password)
#[db_constructor.db_create_table(sql[0]) for sql in queries["ot_schema.sql"] if len(sq
l)>0]
#[db_constructor.db_insert(sql[0]) for sql in queries["ot_data_new.sql"] if len(sql)>0]
```

In [13]:

```
conn_str
```

Out[13]:

```
'jdbc:oracle:thin:@127.0.0.1:1521:orcl'
```

In [14]:

```
dsn = cx_Oracle.makedsn(host,port,sid)
```

In [15]:

```
dsn
```

Out[15]:

```
'(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=127.0.0.1)(PORT=1521))(CONNECT_
DATA=(SID=orcl)))'
```

In [16]:

```
connection = cx_Oracle.connect(user,password,dsn,cx_Oracle.SYSDBA)
```

In [17]:

```
cursor = connection.cursor()
```

cx_Oracle trial

Here is performed an example query with cx_Oracle library.

In [18]:

```
query='SELECT * FROM REGIONS'
```

In [19]:

```
for row in cursor.execute(query):
    print(row)
```

```
(1, 'Europe')
(2, 'Americas')
(3, 'Asia')
(4, 'Middle East and Africa')
```

In [20]:

```
cursor.execute(query)
data = cursor.fetchall()
```

In [21]:

```
data
```

Out[21]:

```
[(1, 'Europe'), (2, 'Americas'), (3, 'Asia'), (4, 'Middle East and Afric
a')]
```

Metadata exploration

In order to understand metadata, we proceed in investigating `cursor.description` , which tells us different informations, like:

- Column name,
- Variable type,
- Eventual constraints.

In [22]:

```
cursor.description
```

Out[22]:

```
[('REGION_ID', cx_Oracle.NUMBER, 127, None, 0, -127, 0),
 ('REGION_NAME', cx_Oracle.STRING, 50, 50, None, None, 0)]
```

In [23]:

```
regions = pd.DataFrame(cursor.execute(query))
```

In [24]:

```
regions
```

Out[24]:

	0	1
0	1	Europe
1	2	Americas
2	3	Asia
3	4	Middle East and Africa

sql_data function

Our function allows us to perform fast queries whose results are presented in pandas dataframe. Column names are retrived from cursor metadata and are then inserted in the pandas dataframe. This function estabilishes a connection for each call to then close the same. This avoids service hanging.

Why pandas?

Pandas is a python library written in C that allows fast data munging and wrangling.

In [25]:

```
def sql_data(sql_statement,user,password,host,port,sid,col_names=True):
    dsn = cx_Oracle.makedsn(host,port,sid)
    connection = cx_Oracle.connect(user,password,dsn,cx_Oracle.SYSDBA)
    cursor = connection.cursor()

    if col_names:
        cursor.execute(sql_statement)
        col_names = [el[0] for el in cursor.description]
        df = pd.DataFrame(cursor.fetchall(), columns = col_names)
    else:
        df = pd.DataFrame(cursor.execute(sql_statement))
    connection.close()
    return df
```

A first example

Here we run a simple trial of our function and, as you can see, it produces a pandas dataframe with all the data in it. Data comes from our SQL DB and is presented in pandas format.

In [26]:

```
sql_data('SELECT * FROM REGIONS',user,password,host,port,sid)
```

Out[26]:

	REGION_ID	REGION_NAME
0	1	Europe
1	2	Americas
2	3	Asia
3	4	Middle East and Africa

Initializing variables

Here we define a list of all the tables' names and then we call SQL data first in a list comprehension and then in a dictionary comprehension. Those allow us to explore faster the database and has been chosen since the dimension of the DB is really small. The dictionary will have as keys the name of single tables of the DB and as values the pandas object obtained via `sql_function()` .

In [27]:

```
table_list = ['contacts', 'countries', 'employees', 'inventories', 'locations',
      'order_items', 'orders', 'products', 'warehouses']
```

In [28]:

```
all_data = [sql_data("SELECT * FROM %s"%(el),user,password,host,port,sid) for el in tab
le_list]
```

In [29]:

```
all_data[6]
```

Out[29]:

	ORDER_ID	CUSTOMER_ID	STATUS	SALESMAN_ID	ORDER_DATE
0	105	1	Pending	54.0	2016-11-17
1	44	2	Pending	55.0	2017-02-20
2	5	5	Canceled	56.0	2017-04-09
3	4	8	Shipped	59.0	2015-04-09
4	2	4	Shipped	NaN	2015-04-26
5	3	5	Shipped	NaN	2017-04-26
6	6	6	Shipped	NaN	2015-04-09
7	7	7	Shipped	NaN	2017-02-15
8	8	8	Shipped	NaN	2017-02-14
9	9	9	Shipped	NaN	2017-02-14
10	11	45	Shipped	NaN	2016-11-29
11	12	46	Shipped	NaN	2016-11-29
12	13	47	Shipped	NaN	2016-11-29
13	32	47	Shipped	NaN	2017-03-09
14	33	48	Shipped	NaN	2017-03-07
15	37	52	Shipped	NaN	2017-02-19
16	45	57	Shipped	64.0	2017-02-20
17	46	58	Pending	62.0	2017-02-20
18	69	44	Canceled	54.0	2017-03-17
19	70	45	Canceled	61.0	2017-02-21
20	71	46	Shipped	54.0	2017-02-21
21	72	47	Shipped	64.0	2016-02-17
22	73	48	Shipped	NaN	2016-02-17
23	74	49	Shipped	64.0	2017-02-10
24	75	16	Shipped	NaN	2017-02-10
25	76	17	Shipped	55.0	2017-02-10
26	86	5	Pending	60.0	2016-11-30
27	88	6	Shipped	61.0	2017-11-01
28	98	48	Shipped	55.0	2017-03-18
29	103	17	Pending	64.0	2016-02-08
30	104	18	Shipped	60.0	2017-02-01

Missing values dedection

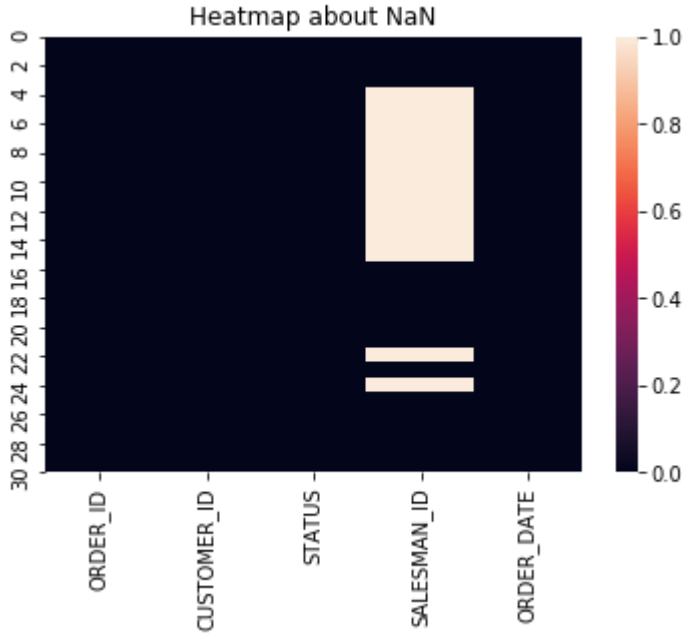
From an insight of some dataframes, it is possibile to note that some missing values appear (i.e `NaN`). We belive that these values are imputable to:

- Missing data
- Erroneous transcription of the DB

As result of what noticed, we had decided to identify, for the `orders` table, all of these values. The result is presented in the following heatmap, where the white area represents NaN presence:

In [30]:

```
n = all_data[6]
plt.figure()
sns.heatmap(n.isnull())
plt.title("Heatmap about NaN")
plt.show()
```



Also, by the execution of the following query, we perform an analysis for counting how many values, inside the orders table, are not affected by this alteration. Obviously, the following query is completely generalizable for each dataframe of the DB.

```
SELECT DISTINCT SALESMAN_ID
    COUNT(*)
        FROM orders
            GROUP BY SALESMAN_ID
                HAVING SALESMAN_ID IS NOT NULL
```

In [31]:

```
sql_data("SELECT DISTINCT SALESMAN_ID, COUNT(*) FROM orders GROUP BY SALESMAN_ID HAVING
SALESMAN_ID IS NOT NULL" ,user,password,host,port,sid)
```

Out[31]:

	SALESMAN_ID	COUNT(*)
0	64	4
1	59	1
2	62	1
3	54	3
4	56	1
5	55	3
6	60	2
7	61	2

In [32]:

```
all_data_dict = {el:sql_data("SELECT * FROM %s"%(el),user,password,host,port,sid) for e
l in table_list}
```

In [33]:

```
all_data_dict.keys()
```

Out[33]:

```
dict_keys(['contacts', 'countries', 'employees', 'inventories', 'location
s', 'order_items', 'orders', 'products', 'warehouses'])
```

In [34]:

```
all_data_dict[list(all_data_dict.keys())[1]]
```

Out[34]:

	COUNTRY_ID	COUNTRY_NAME	REGION_ID
0	AR	Argentina	2
1	AU	Australia	3
2	BE	Belgium	1
3	BR	Brazil	2
4	CA	Canada	2
5	CH	Switzerland	1
6	CN	China	3
7	DE	Germany	1
8	DK	Denmark	1
9	EG	Egypt	4
10	FR	France	1
11	IL	Israel	4
12	IN	India	3
13	IT	Italy	1
14	JP	Japan	3
15	KW	Kuwait	4
16	ML	Malaysia	3
17	MX	Mexico	2
18	NG	Nigeria	4
19	NL	Netherlands	1
20	SG	Singapore	3
21	UK	United Kingdom	1
22	US	United States of America	2
23	ZM	Zambia	4
24	ZW	Zimbabwe	4

PRODUCT_NAME summary

We used `pandas.Series().value_counts()` method to understand how many times the single products repeat in the data frame resulting from the following query:

```
SELECT *
   FROM products
       INNER JOIN inventories
           ON products.PRODUCT_ID = inventories.PRODUCT_ID
```

As we can see the result gives us the absolute frequency of the single product name repetition.

In [35]:

```
sql_data("SELECT * FROM products INNER JOIN inventories ON products.PRODUCT_ID = invent
ories.PRODUCT_ID ",user,password,host,port,sid).loc[:,"PRODUCT_NAME"].value_counts()
```

Out[35]:

```
G.Skill Ripjaws V Series           58
Corsair Vengeance LPX              40
G.Skill Trident Z                  38
Corsair Dominator Platinum         28
Kingston                           20
                                   ..
MSI X299 GAMING PRO CARBON AC       1
SanDisk SDSSDHII-480G-G25           1
MSI X99A XPOWER GAMING TITANIUM     1
ASRock EP2C612 WS                   1
Western Digital WDS256G1X0C         1
Name: PRODUCT_NAME, Length: 173, dtype: int64
```

Joining data

We then joined data in order to obtained a bigger picture of the avaiable DB, by executing the query:

```
SELECT *
    FROM products
        INNER JOIN inventories
            ON products.PRODUCT_ID = inventories.PRODUCT_ID
                WHERE PRODUCT_NAME
                    LIKE '%Xeon%'
```

With this query we combined informations from products with the inventories. This allows us to link further up to location and countries, as it will be seen in the next steps. Please notice that we restricted, for this example purpose, the result only to Xenon processors.

In [36]:

```
sql_data("SELECT * FROM products INNER JOIN inventories ON products.PRODUCT_ID = invent
ories.PRODUCT_ID WHERE PRODUCT_NAME LIKE '%Xeon%'",user,password,host,port,sid).head()
```

Out[36]:

	PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	STANDARD_COST	LIST_
0	228	Intel Xeon E5-2699 V3 (OEM/Tray)	Speed:2.3GHz,Cores:18,TDP:145W	2867.51	
1	228	Intel Xeon E5-2699 V3 (OEM/Tray)	Speed:2.3GHz,Cores:18,TDP:145W	2867.51	
2	2	Intel Xeon E5-2697 V4	Speed:2.3GHz,Cores:18,TDP:145W	2144.40	
3	2	Intel Xeon E5-2697 V4	Speed:2.3GHz,Cores:18,TDP:145W	2144.40	
4	2	Intel Xeon E5-2697 V4	Speed:2.3GHz,Cores:18,TDP:145W	2144.40	

Null value search

We have also performed a brief search about `NULL` values in employees table, in phone column. Our discovery was pretty obvious. As you might have guessed, there are no missing values.

In [37]:

```
sql_data(" SELECT * FROM employees WHERE PHONE IS NULL",user,password,host,port,sid)
```

Out[37]:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE	HIRE_DATE	MANAGER_ID	JC

Job title mapping

Here is presented the list of unique job titles avaiable in employees table.

In [38]:

```
sql_data("SELECT DISTINCT JOB_TITLE FROM employees ORDER BY JOB_TITLE DESC",user,passwo
rd,host,port,sid,col_names=True)
```

Out[38]:

	JOB_TITLE
0	Stock Manager
1	Stock Clerk
2	Shipping Clerk
3	Sales Representative
4	Sales Manager
5	Purchasing Clerk
6	Programmer
7	Marketing Manager
8	Accountant

General analysis

From this point onward, we present our analysis on the database. We, as illustrated below, create first a unique dataframe whose columns give us informations about:

- The product (PRODUCT_ID, PRODUCT_NAME, DESCRIPTION, etc.),
- The inventory (units in storage or price etc.),
- The warehouse,
- The geographical dislocation.

The query launched is the following:

```
SELECT *
    FROM products
        INNER JOIN inventories
            ON products.PRODUCT_ID = inventories.PRODUCT_ID
        INNER JOIN warehouses
            ON inventories.WAREHOUSE_ID=warehouses.WAREHOUSE_ID
        INNER JOIN locations
            ON warehouses.LOCATION_ID=locations.LOCATION_ID
        INNER JOIN countries
            ON locations.COUNTRY_ID = countries.COUNTRY_ID
```

In [39]:

```
sql_data("SELECT * FROM products INNER JOIN inventories ON products.PRODUCT_ID = invent
ories.PRODUCT_ID INNER JOIN warehouses ON inventories.WAREHOUSE_ID=warehouses.WAREHOUSE
_ID INNER JOIN locations ON warehouses.LOCATION_ID=locations.LOCATION_ID INNER JOIN cou
ntries ON locations.COUNTRY_ID = countries.COUNTRY_ID",user,password,host,port,sid)
```

Out[39]:

	PRODUCT_ID	PRODUCT_NAME	DESCRIPTION	STANDARD_COST	L
0	210	Intel Core i9-7900X	Speed:3.3GHz,Cores:10,TDP:140W	855.82	
1	211	Intel Xeon E5-2650	Speed:2.0GHz,Cores:8,TDP:95W	869.03	
2	212	Intel Xeon E5-2680 V4	Speed:2.4GHz,Cores:14,TDP:120W	1365.13	
3	214	Intel Core i7-5960X	Speed:3.0GHz,Cores:8,TDP:140W	865.59	
4	216	MSI GTX 1080 TI LIGHTNING X	Chipset:GeForce GTX 1080 Ti,Memory:11GBCore Cl...	742.94	
...	...	...	...	...	
1107	201	Kingston	Speed:DDR3-1600,Type:240-pin DIMM,CAS:11Module...	566.98	
1108	203	Kingston	Speed:DDR3-1333,Type:240-pin DIMM,CAS:9Module:...	556.84	
1109	204	G.Skill Ripjaws V Series	Speed:DDR4-3200,Type:288-pin DIMM,CAS:15Module...	546.64	
1110	205	G.Skill Trident X	Speed:DDR3-3100,Type:240-pin DIMM,CAS:12Module...	507.32	
1111	207	PNY VCQM6000-PB	Chipset:Quadro M6000,Memory:12GBCore Clock:988MHz	2505.04	

1112 rows × 21 columns

In the following example, we explore the managed orders from warehouses, in order to obtain details about their status.

```
SELECT WAREHOUSE_NAME,
    COUNT(STATUS)
        AS number_orders,
       STATUS
      FROM
 (SELECT *
     FROM (
         orders
             INNER JOIN order_items
                 ON orders.ORDER_ID = order_items.ORDER_ID
             INNER JOIN inventories
                 ON order_items.PRODUCT_ID=inventories.PRODUCT_ID
             INNER JOIN warehouses
                 ON inventories.WAREHOUSE_ID=warehouses.WAREHOUSE_ID
             INNER JOIN locations
                 ON warehouses.LOCATION_ID=locations.LOCATION_ID
           ))
           WHERE
               STATUS='Shipped'
           GROUP BY
               (WAREHOUSE_NAME,STATUS)
```

In [41]:

```
sql_data(" SELECT WAREHOUSE_NAME, COUNT(STATUS) AS number_orders, STATUS FROM( SELECT *
FROM (orders INNER JOIN order_items ON orders.ORDER_ID = order_items.ORDER_ID INNER JOI
N inventories ON order_items.PRODUCT_ID=inventories.PRODUCT_ID INNER JOIN warehouses ON
inventories.WAREHOUSE_ID=warehouses.WAREHOUSE_ID INNER JOIN locations ON warehouses.LOC
ATION_ID=locations.LOCATION_ID))  WHERE STATUS='Shipped'  GROUP BY(WAREHOUSE_NAME,STATU
S)" ,user,password,host,port,sid)
```

Out[41]:

	WAREHOUSE_NAME	NUMBER_ORDERS	STATUS
0	Southlake, Texas	15	Shipped
1	Toronto	49	Shipped
2	Sydney	108	Shipped
3	Mexico City	45	Shipped
4	Bombay	71	Shipped
5	Beijing	95	Shipped
6	San Francisco	88	Shipped
7	New Jersey	21	Shipped
8	Seattle, Washington	59	Shipped

Average profit calculation

We grouped all the data retrived from the former query, in order to calculate the average profit by `CITY` and `CATEGORY_ID` . The query we launched is the following:

```
SELECT CITY,CATEGORY_ID, AVG(LIST_PRICE-STANDARD_COST) AS avg_profit
    FROM
    (SELECT *
         FROM products
             INNER JOIN inventories
                 ON products.PRODUCT_ID = inventories.PRODUCT_ID
             INNER JOIN warehouses
                 ON inventories.WAREHOUSE_ID=warehouses.WAREHOUSE_ID
             INNER JOIN locations
                 ON warehouses.LOCATION_ID=locations.LOCATION_ID
             INNER JOIN countries
                 ON locations.COUNTRY_ID = countries.COUNTRY_ID)
    GROUP BY (CITY, CATEGORY_ID)
```

We finally use the `pandas.Series().replace()` function in order to map the CATEGORY_ID column from integer to a string containing an acronymic description.

In [42]:

```
sql_data("SELECT CITY,CATEGORY_ID, AVG(LIST_PRICE-STANDARD_COST) AS avg_profit FROM (SE
LECT * FROM products INNER JOIN inventories ON products.PRODUCT_ID = inventories.PRODUC
T_ID INNER JOIN warehouses ON inventories.WAREHOUSE_ID=warehouses.WAREHOUSE_ID INNER JO
IN locations ON warehouses.LOCATION_ID=locations.LOCATION_ID INNER JOIN countries ON lo
cations.COUNTRY_ID = countries.COUNTRY_ID) GROUP BY (CITY, CATEGORY_ID) ",user,password
,host,port,sid).replace({1:"CPU",2:"GPU",4:"MoBo",5:"HDD/RAM"})
```

Out[42]:

	CITY	CATEGORY_ID	AVG_PROFIT
0	Beijing	MoBo	81.213750
1	South San Francisco	MoBo	79.647308
2	Bombay	GPU	296.841176
3	Seattle	GPU	296.841176
4	Toronto	HDD/RAM	104.290000
5	Bombay	MoBo	107.395625
6	South San Francisco	CPU	178.253103
7	Toronto	CPU	320.910000
8	Sydney	HDD/RAM	118.736023
9	Sydney	MoBo	80.598780
10	Mexico City	MoBo	55.031875
11	Beijing	GPU	296.841176
12	Bombay	CPU	222.014500
13	Bombay	HDD/RAM	146.770526
14	South San Francisco	GPU	285.349737
15	South San Francisco	HDD/RAM	123.507381
16	South Brunswick	GPU	296.841176
17	Seattle	HDD/RAM	146.770526
18	Seattle	CPU	165.228214
19	Mexico City	CPU	320.910000
20	Southlake	GPU	296.841176
21	Mexico City	GPU	296.841176
22	Mexico City	HDD/RAM	89.927857
23	Beijing	CPU	222.014500
24	Southlake	CPU	119.275000
25	Seattle	MoBo	122.572222
26	Toronto	GPU	285.349737
27	Sydney	CPU	229.842195
28	Sydney	GPU	285.349737
29	Beijing	HDD/RAM	116.928125
30	South Brunswick	CPU	320.910000
31	Toronto	MoBo	55.031875

Bonus analysis

In the end, we give an insight about gross income distribution by category, by executing the following query:

```
SELECT CATEGORY_ID, avg(QUANTITY*UNIT_PRICE) AS avg_gross_income, variance(QUANT
ITY*UNIT_PRICE) AS var_gross_income
    FROM
    (SELECT *
        FROM ORDER_ITEMS
            LEFT JOIN products
                ON order_items.PRODUCT_ID=products.PRODUCT_ID )
    GROUP BY CATEGORY_ID
```

As it is possible to notice, the CPU and GPU show the greatest mean gross income and has the greatest variability. The query relates `order_items` with `products` tables.

In [43]:

```
sql_data("SELECT CATEGORY_ID, avg(QUANTITY*UNIT_PRICE) AS avg_gross_income, variance(QU
ANTITY*UNIT_PRICE) AS var_gross_income FROM (SELECT * FROM ORDER_ITEMS LEFT JOIN produc
ts ON order_items.PRODUCT_ID=products.PRODUCT_ID ) GROUP BY CATEGORY_ID",user,password,
host,port,sid).replace({1:"CPU",2:"GPU",4:"MoBo",5:"HDD/RAM"})
```

Out[43]:

	CATEGORY_ID	AVG_GROSS_INCOME	VAR_GROSS_INCOME
0	CPU	126050.270629	6.126451e+09
1	GPU	131309.720190	1.106244e+10
2	MoBo	35640.711232	4.530642e+08
3	HDD/RAM	53209.737034	3.150839e+09

Here, the above query has been modified in order to obtain informations about the maximum income due by each product. The request pass trough the following query:

```
SELECT CATEGORY_ID, max(QUANTITY*UNIT_PRICE) AS max_gross_income
    FROM
    (SELECT *
        FROM ORDER_ITEMS
            LEFT JOIN products
                ON order_items.PRODUCT_ID=products.PRODUCT_ID )
    GROUP BY CATEGORY_ID
```

In [44]:

```
sql_data("SELECT CATEGORY_ID, max(QUANTITY*UNIT_PRICE) AS max_gross_income FROM (SELECT
* FROM ORDER_ITEMS LEFT JOIN products ON order_items.PRODUCT_ID=products.PRODUCT_ID ) G
ROUP BY CATEGORY_ID",user,password,host,port,sid).replace({1:"CPU",2:"GPU",4:"MoBo",5:
"HDD/RAM"})
```

Out[44]:

	CATEGORY_ID	MAX_GROSS_INCOME
0	CPU	328038.42
1	GPU	600155.00
2	MoBo	120521.73
3	HDD/RAM	603023.32

In [ ]:

```

```