DATA BASES PROJECT

ASKREDDIT (REDDIT MAY 2015) DATASET ANALYSIS

PHILIPPE MARIE (21601173) AND SERRAZ CLAIRE (21601145)

M2 D3S – TSE (APPRENTICES TRACK)

Contents

Introduction

This project has two goals. First, it aims to model and store a large dataset. By that, one means to create and populate a database, that is detailed in the first part of this report. The other objective is to provide an initial experience on facing big data problems and on performing analyses. This second objective is respected while exploring the database using SQL instructions, that what the second part of this report attempt to do.

This project focuses on a subselection of the AskReddit from May 2015 (on the social network *reddit).* As the original dataset includes more than 50 millions of rows, the dataset has been reduced to a little more than 4 millions of rows, which still is a large dataset. The dataset contains comments from users, and the characteristics of the comments (authors, score, parent, subreddit…). These characteristics will be detailed when creating the database.

For this project, the use of the software Python with the library sqlite3 will be needed. Visual Studio Code will be used to syntactically color the codes.

0. Preliminary Part

Before starting the project, several preliminary settings must be done. First, the needed library must be imported. Also, the connection to a database must be opened and established.

The following code shows the Python code to import sqlite3 and pandas. It also shows how to create a database, open, and establish a connection to this database.

```
#-- Import sqlite and pandas, two libraries needed for the analysis

import sqlite3
import pandas as pd

#-- Create a database and its connection

database_name='DBProject.db'
con = sqlite3.connect(database_name)
c=con.cursor()
```

1. Part 1: Creating and Populating the database

This part is dedicated to the creation and population of the database. It is divided into two parts: the creation and the population.

The relational scheme is given as follows:

- **AUTHOR**(author)
- **DISTINGUISHED**(distinguished)
- **CONTROVERSY**(controversiality)
- **REMOVAL**(removal_reason)
- **SCORE**(id#, score, ups, downs, score_hidden, gilded)
- **PARENT**(parent_id, link_id)
- **SUBREDDIT**(subreddit_id, subreddit)
- **COMMENT**(id, created_utc, name, body, edited, author_flair_css_class, author_flair_text, author#, controversiality#, subreddit_id#)
- **IS_DITINGUISHED**(id#, distinguished#)
- **REMOVED**(id#, removal_reason#)
- **DEPENDS**(id#, parent_id#)

An underlined word means that it is a primary key of the corresponding table, a word followed by a # means that is a foreign key of the corresponding table.

The entity diagram is as follows:

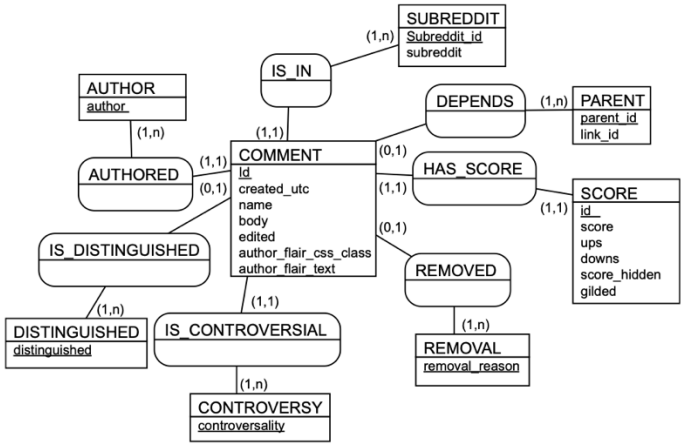

Figure 1: Entity/Relationship diagram of the database

1.1. Creating the database

Three steps are required to create our database:

- Drop the existing tables.
- Create the tables with their columns, adding constraints for the primary and foreign keys.
- Create unique indexes for each table.

1.1.1. Drop the existing tables

This step isn't always useful as it is just here to drop the existing tables, if any, in the database. The database is supposed to be empty as it has been created just before, however this step allows to make sure that the database will be formed as expected.

First, one can see that "comment" and "parent" are in blue. It is due to the fact that they are also SQL expressions however here the software will understand that it is the name of a potential table. It will be the case everywhere.

Second, as explained previously, the analysis has been done using Python. When using Python to write in SQL language, one needs to define it. The best is to store the SQL instructions in dictionary, and then, using a for loop, run each SQL instructions stored in the dictionary. Here, as the SQL instructions are used in Python, the c.execute (with c standing for cursor) is mandatory to run the SQL statement inside. The next image shows the same code as before but for Python language. After this step, all the potential existing tables (named author, distinguished, controversy, removal, score, parent, subreddit, comment, is_distinguished, removed, depends) have been dropped out.

```
#-- Create a dictionary for the drops instructions in SQL

sql_drops = {'drop_author': '''DROP TABLE IF EXISTS author''',
             'drop_distinguished': '''DROP TABLE IF EXISTS distinguished''',
             'drop_controversy':'''DROP TABLE IF EXISTS controversy''',
             'drop_removal':'''DROP TABLE IF EXISTS removal''',
             'drop_score':'''DROP TABLE IF EXISTS score''',
             'drop_parent':'''DROP TABLE IF EXISTS parent''',
             'drop_subreddit':'''DROP TABLE IF EXISTS subreddit''',
             'drop_comment':'''DROP TABLE IF EXISTS comment''',
             'drop_is_distinguished':'''DROP TABLE IF EXISTS is_distinguished''',
             'drop_removed': '''DROP TABLE IF EXISTS removed''',
             'drop_depends':'''DROP TABLE IF EXISTS depends'''}
```

1.1.2. Create the tables

After having dropped all possible existing tables, one need to create the tables needed and wanted for the upcoming analysis. These tables are specified in the relational scheme, and in the entity diagram.

In the statement to create a table, one needs to include all the variable names and their types, but also the different constraints, including the primary and foreign keys.

```
#-- Create a dictionary for the creates instricutions in SQL

sql_creates = {'create_author': '''CREATE TABLE author (author TEXT, CONSTRAINT
pk_author PRIMARY KEY (author))''',

             'create_distinguished': '''CREATE TABLE distinguished (distinguished
TEXT, CONSTRAINT pk_distinguished PRIMARY KEY (distinguished))''',
```

```
            'create_controversy':'''CREATE TABLE controversy (controversiality INT,
CONSTRAINT pk_controversy PRIMARY KEY (controversiality))''',

            'create_removal':'''CREATE TABLE removal (removal_reason TEXT,
CONSTRAINT pk_removal PRIMARY KEY (removal_reason))''',

            'create_score':'''CREATE TABLE score (id TEXT, score INT, ups INT,
downs INT, score_hidden BIT, gilded INT, CONSTRAINT pk_score PRIMARY KEY (id),
CONSTRAINT fk_score_comment FOREIGN KEY (id) REFERENCES comment (id))''',

            'create_parent':'''CREATE TABLE parent (parent_id TEXT, link_id INT,
CONSTRAINT pk_parent PRIMARY KEY (parent_id))''',

            'create_subreddit':'''CREATE TABLE subreddit (subreddit_id TEXT,
subreddit TEXT, CONSTRAINT pk_subreddit PRIMARY KEY (subreddit_id))''',

            'create_comment':'''CREATE TABLE comment (id TEXT, created_utc INT,
name TEXT, body TEXT, edited INT, author_flair_css_class TEXT, author_flair_text
TEXT, author TEXT, controversiality INT, subreddit_id INT, CONSTRAINT pk_comment
PRIMARY KEY (id), CONSTRAINT fk_comment_author FOREIGN KEY (author) REFERENCES
author (author), CONSTRAINT fk_comment_controversy FOREIGN KEY (controversiality)
REFERENCES controversy (controversiality), CONSTRAINT fk_comment_subreddit FOREIGN
KEY (subreddit_id) REFERENCES subreddit (subreddit_id))''',

            'create_is_distinguished':'''CREATE TABLE is_distinguished (id TEXT,
distinguished TEXT, CONSTRAINT pk_is_distinguished PRIMARY KEY (id, distinguished),
CONSTRAINT fk_id_comment FOREIGN KEY (id) REFERENCES comment (id), CONSTRAINT
fk_id_distinguished FOREIGN KEY (distinguished) REFERENCES distinguished
(distinguished))''',

            'create_removed': '''CREATE TABLE removed (id TEXT, removal_reason
TEXT, CONSTRAINT pk_removed PRIMARY KEY (id, removal_reason), CONSTRAINT
fk_removed_comment FOREIGN KEY (id) REFERENCES comment (id), CONSTRAINT
fk_removed_removal FOREIGN KEY (removal_reason) REFERENCES distinguished
(removal_reason))''',

            'create_depends':'''CREATE TABLE depends (id TEXT, parent_id TEXT,
CONSTRAINT pk_depends PRIMARY KEY (id, parent_id), CONSTRAINT fk_depends_comment
FOREIGN KEY (id) REFERENCES comment (id), CONSTRAINT fk_depends_parent FOREIGN KEY
(parent_id) REFERENCES parent (parent_id))'''}
```

```
#-- Create all the tables

for k in sql_creates:
    sql = sql_creates[k]
    c.execute(sql)
```

1.1.3. Create the unique indexes

Creating unique indexes is not a mandatory step, however it helps to decrease the query times, making a query or an execution way faster. It is mostly useful to large tables, but it has been decided to be done to all the tables.

```
#-- Create a dictionary for the unique index instructions in SQL

sql_unique_index = {'ui_author': '''CREATE UNIQUE INDEX unique_index_author ON
author (author)''',
            'ui_distinguished': '''CREATE UNIQUE INDEX unique_index_distinguished
ON distinguished (distinguished)''',
            'ui_controversy':'''CREATE UNIQUE INDEX unique_index_controversy ON
controversy (controversiality)''',
            'ui_removal':'''CREATE UNIQUE INDEX unique_index_removal ON removal
(removal_reason)''',
            'ui_score':'''CREATE UNIQUE INDEX unique_index_score ON score (id)''',
            'ui_parent':'''CREATE UNIQUE INDEX unique_index_parent ON parent
(parent_id)''',
            'ui_subreddit':'''CREATE UNIQUE INDEX unique_index_subreddit ON
subreddit (subreddit_id)''',
            'ui_comment':'''CREATE UNIQUE INDEX unique_index_comment ON comment
(id)''',
            'ui_is_distinguished':'''CREATE UNIQUE INDEX
unique_index_is_distinguished ON is_distinguished (id, distinguished)''',
            'ui_removed': '''CREATE UNIQUE INDEX unique_index_removed ON removed
(id, removal_reason)''',
            'ui_depends':'''CREATE UNIQUE INDEX unique_index_depends ON depends
(id, parent_id)'''}
```

```
#-- Create all the unique index

for k in sql_unique_index:
    sql = sql_unique_index[k]
    c.execute(sql)
```

1.2. Loading data into the database

The first step of this first part is dedicated to the creation of the database, with all the tables, being empty. The second step that is developed is dedicated to populating the database and load data into it. The data needed to be loaded in it are in the several csv files that need to be imported. Then, each of these files will be inserted in the corresponding already existing tables. Then, this session will be cut into four steps:

- Importing the CSV files
- Inserting these files into the existing tables
- Counting the number of rows per table
- Creating a subset of size 5MB

1.2.1. Importing the CSV files

This step needs to be done on Python. The following image shows the python code to do so, using the function *read_csv* from the *pandas* library.

```
#-- Import the csv

#-- Claire
#path = 'C:/DESKTOP/DB - WB/csv_askreddit_2015_split_dataset/'

#-- Marie
path = '/Users/marie/Desktop/Cours/S2/Data bases/Data/'

table_author = pd.read_csv(path + 'exp_author.csv')
table_distinguished = pd.read_csv(path + 'exp_distinguihshed.csv')
table_controversy = pd.read_csv(path + 'exp_controverse.csv')
table_removal = pd.read_csv(path + 'exp_removal.csv')
table_score = pd.read_csv(path + 'exp_score.csv')
table_parent = pd.read_csv(path + 'exp_parent.csv')
table_subreddit = pd.read_csv(path + 'exp_subreddit.csv')
table_comment = pd.read_csv(path + 'exp_comment.csv')
table_is_distinguished = pd.read_csv(path + 'exp_askreddit_is_distinguihshed.csv')
table_depends = pd.read_csv(path + 'exp_askreddit_depends.csv')
```

1.2.2. Populating the tables

Before populating the tables, one needs to transform the imported files into list. This step is still done in Python. The function *.itertuples(index=False, name=None)* is here to look inside the column, without looking at the name of the column and the index. The function *list()* is needed to transform the files into list.

This is how is it done using Python.

```
#-- Transform dataframe rows into list

list_author = list(table_author.itertuples(index=False, name=None))
list_distinguished = list(table_distinguished.itertuples(index=False, name=None))
list_controversy = list(table_controversy.itertuples(index=False, name=None))
list_removal = list(table_removal.itertuples(index=False, name=None))
list_score = list(table_score.itertuples(index=False, name=None))
list_parent = list(table_parent.itertuples(index=False, name=None))
list_subreddit = list(table_subreddit.itertuples(index=False, name=None))
list_comment = list(table_comment.itertuples(index=False, name=None))
list_is_distinguished = list(table_is_distinguished.itertuples(index=False,
name=None))
list_depends = list(table_depends.itertuples(index=False, name=None))
```

A dictionary is created to store the SQL instructions, and then we need the function *executemany* to apply the SQL instructions to each element of the previously defined lists. Finally, one needs to commit the cursor. The following image shows this reasoning on Python.

```
#-- Create a dictionary for the insert instructions in SQL

sql_insert = {'insert_author': '''INSERT INTO author (author) VALUES (?)''',
                  'insert_distinguished': '''INSERT INTO distinguished
(distinguished) VALUES (?)''',
                  'insert_controversy': '''INSERT INTO controversy
(controversiality) VALUES (?)''',
                  'insert_removal': '''INSERT INTO removal (removal_reason) VALUES
(?)''',
                  'insert_score': '''INSERT INTO score (id, score, ups, downs,
score_hidden, gilded) VALUES (?, ?, ?, ?, ?, ?)''',
                  'insert_parent': '''INSERT INTO parent (parent_id, link_id)
VALUES (?, ?)''',
                  'insert_subreddit': '''INSERT INTO subreddit (subreddit_id,
subreddit) VALUES (?, ?)''',
                  'insert_comment': '''INSERT INTO comment (id, created_utc, name,
body, edited, author_flair_css_class, author_flair_text, author, controversiality,
subreddit_id) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''',
                  'insert_is_distinguished': '''INSERT INTO is_distinguished (id,
distinguished) VALUES (?, ?)''',
                  'insert_removed': '''INSERT INTO removed (id, removal_reason)
VALUES (?, ?)''',
                  'insert_depends': '''INSERT INTO depends (id, parent_id) VALUES
(?, ?)'''}
```

```
#-- Execute the sql_insert instructions previously stored into a dictionary

c.executemany(sql_insert['insert_author'], list_author)
c.executemany(sql_insert['insert_distinguished'], list_distinguished)
c.executemany(sql_insert['insert_controversy'], list_controversy)
c.executemany(sql_insert['insert_removal'], list_removal)
c.executemany(sql_insert['insert_score'], list_score)
c.executemany(sql_insert['insert_parent'], list_parent)
c.executemany(sql_insert['insert_subreddit'], list_subreddit)
c.executemany(sql_insert['insert_comment'], list_comment)
c.executemany(sql_insert['insert_is_distinguished'], list_is_distinguished)
c.executemany(sql_insert['insert_depends'], list_depends)
```

1.2.3. Computing the size of each table

After having filled in the tables, it is interesting to know the size of each table. This can be done on Python, using a dictionary to story the SQL instructions, and then a for loop to run the sql instructions, to compute the length of the tables and store the lengths into another dictionary. The following shows the code used in Python.

```
#-- Show how many rows per table

sql_count = {'count_author': '''SELECT * FROM author''',
             'count_distinguished': '''SELECT * FROM distinguished''',
             'count_controversy': '''SELECT * FROM controversy''',
```

```
            'count_removal': '''SELECT * FROM removal''',
            'count_score': '''SELECT * FROM score''',
            'count_parent': '''SELECT * FROM parent''',
            'count_subreddit': '''SELECT * FROM subreddit''',
            'count_comment': '''SELECT * FROM comment''',
            'count_is_distinguished': '''SELECT * FROM is_distinguished''',
            'count_depends': '''SELECT * FROM depends''',
            'count_removed': '''SELECT * FROM removed''',}
n_rows={}

for k in sql_count:
    sql = sql_count[k]
    n_rows[k] = len(c.execute(sql).fetchall())

n_rows
```

The next table gives all the sizes:

Table	Number of Rows
score	4 234 970
comment	
is_distinguished	
depends	
parent	1 464 558
author	570 735
distinguished	2
controversy	
subreddit	1
removed	0
removal	

Table 1: Size of the tables

All the results are consistent with the entity relationship previously given. Some remarks on the above table:

- It makes sense that the table *comment* has the maximal (and expected) number of rows as it is the main table of interest. It also makes sense that the tables *score, is_distinguished, depends* have this same number of rows, as a given comment has a unique score, depends on a unique parent, and has unique distinguished status.
- It also makes sense that the tables *parent* and *author* have a little bit less of rows, as a parent can have several comments depending of it, and as an author can have written several comments.
- There are only two rows for the tables *distinguished* and *controversy* that are composed of either a Boolean (TRUE-FALSE) variable, or of either an integer taking 0 or 1 as values. Then, in both cases, there are only two modalities and that is why these tables only contain two rows each.
- All the comments come from the same subreddit and that is why there is only one line in the table *subreddit*.
- The fact that the tables *removed* and *removal* are empty means that no table has been removed so far. This is simply due to the selection of comments.

1.2.4. Creating a subset of size 5MB

Finally, the last step of this first part is to describe a possible sub selection of the data that would represent only about 5MB. This step is divided into several steps:

- Count the number of pages in our data base
- Compute the size of a page in bytes
- Count the number of rows in a page in our database
- Compute the size of the whole database in bytes
- Compute the size of the whole database in MB
- If superior to 5 MB (it is expected to be superior), compute the number of pages needed to have a database of size 5MB
- Compute the number of rows needed to have a database of size 5MB.

1.2.4.1. Counting the number of pages in our data base

```
#-- Count the number of page
c.execute('SELECT page_count FROM pragma_page_count();')
nb_page = c.fetchall()
print('There are', nb_page[0][0], 'pages in our database.')
```

The result is that there are 500 330 pages in our database.

1.2.4.2. Computing the size of a page in bytes

```
#-- Compute the size of a page, in bytes
c.execute('SELECT page_size FROM pragma_page_size();')
size_page = c.fetchall()
print('A page "measures"', size_page[0][0],'bytes.')
```

The result is that a page ‘measures’ 4096 bytes.

1.2.4.3. Counting the number of rows in a page in our database

Our database contains 4 234 970 rows and 500 330 pages.

```
#-- Compute the number of row in a page in our database.
c.execute('SELECT 4234970/page_count FROM pragma_page_count();')
page_row = c.fetchall()
print('There are', page_row[0][0], 'rows in a page in our database.')
```

The result is that there are 8 rows in a page in our database.

1.2.4.4. Computing the size of the whole database in bytes and in MB

```
#-- Compute the size of the whole database in bytes.
c.execute('SELECT page_count*page_size FROM pragma_page_count(),
pragma_page_size();')
```

```
size_db = c.fetchall()
print('The whole database "measures"', size_db[0][0], 'bytes.')

#-- 1 Byte = 0.000001 MB

#-- Compute the size of the whole database in MB.
c.execute('SELECT (page_count*page_size) * 0.000001 FROM pragma_page_count(),
pragma_page_size();')
size_db_MB = c.fetchall()
print('The whole database "measures"', round(size_db_MB[0][0], 4), 'MB.')
```

The results are that the whole database ‘measures’ 2049351680 bytes, i.e. 2049,3517 MB.

1.2.4.5. Computing the number of page and the number of rows needed to have a database of size 5MB

```
# Compute the number of pages needed to have a database of size 5MB.
c.execute('SELECT 5/0.000001/page_size FROM pragma_page_size();')
nb_pages_5MB = c.fetchall()
print('A sub database of size 5MB should have', round(nb_pages_5MB[0][0],4),
'pages.')
```

```
#-- Compute the number of rows needed to have a database of size 5MB.
c.execute('SELECT (5/0.000001/page_size)*(4234970/page_count) FROM
pragma_page_size(), pragma_page_count();')
nb_rows_5MB = c.fetchall()
print('A sub database of size 5MB should have', round(nb_rows_5MB[0][0],4),
'rows.')
```

The results are that a sub selection of size 5 MB should have 1220 pages and 9765 rows.

To create this subset of size 5 MB, one shall redo all the instructions above. By that, it is meant that one should:

- Drop all the tables
- Create new table, but this time using “ORDER BY random() LIMIT 1700” in the SQL code for the creation of the table comment.
- Show how many rows there are per table
- Insert pandas dataframes in the tables
- Extract the new files as csv. This step is mandatory as the function random() is used. To get the same results, a zip file is linked to this project, that allows to get, for the second part, exactly the same results originally got. Then, there isn’t any need to re run the first part: the results would differ from the original one. This also explains why this part of the code are comments in python.

The only codes that differ from previously are the creation of new tables and the extraction of files as csv, so here it is:

```
##-- Create the new tables
```

```
sql_new_tables = {'new_comment': '''CREATE TABLE comment2 AS
                                    SELECT * FROM comment ORDER BY random() LIMIT
1700''',

                  'new_author': '''CREATE TABLE author2 AS
                                    SELECT * FROM author AS a
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM comment2 AS c
                                        WHERE c.author = a.author)''',

                  'new_score': '''CREATE TABLE score2 AS
                                    SELECT * FROM score AS s
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM comment2 AS c
                                        WHERE c.id = s.id)''',

                  'new_controversy': '''CREATE TABLE controversy2 AS
                                    SELECT * FROM controversy AS cont
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM comment2 AS c
                                        WHERE c.controversiality =
cont.controversiality)''',

                  'new_depends': '''CREATE TABLE depends2 AS
                                    SELECT * FROM depends AS d
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM comment2 AS c
                                        WHERE c.id = d.id)''',

                  'new_parent': '''CREATE TABLE parent2 AS
                                    SELECT * FROM parent AS p
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM depends2 AS d
                                        WHERE d.parent_id = p.parent_id)''',

                  'new_is_distinguished': '''CREATE TABLE is_distinguished2 AS
                                    SELECT * FROM is_distinguished AS d
                                    WHERE EXISTS (
                                        SELECT *
                                        FROM comment2 AS c
                                        WHERE c.id = d.id)''',

                  'new_distinguished': '''CREATE TABLE distinguished2 AS
                                    SELECT * FROM distinguished AS d
                                    WHERE EXISTS (
                                        SELECT *
```

```
                                        FROM is_distinguished2 AS d2
                                        WHERE d2.distinguished =
d.distinguished)'''}
```

```
#-- Create a dictionary for the create instructions in SQL

for k in sql_new_tables:
    sql = sql_new_tables[k]
    c.execute(sql)
```

```
#-- Create pandas dataframes from the tables

comment2 = pd.read_sql('''SELECT * FROM comment2''', con)
author2 = pd.read_sql('''SELECT * FROM author2''', con)
score2 = pd.read_sql('''SELECT * FROM score2''', con)
controversy2 = pd.read_sql('''SELECT * FROM controversy2''', con)
depends2 = pd.read_sql('''SELECT * FROM depends2''', con)
parent2 = pd.read_sql('''SELECT * FROM parent2''', con)
is_distinguished2 = pd.read_sql('''SELECT * FROM is_distinguished2''', con)
distinguished2 = pd.read_sql('''SELECT * FROM distinguished2''', con)

#-- Extract the new files as csv

#-- Claire
#path = 'C:/DESKTOP/DB - WB/csv_askreddit_2015_split_dataset/'

#-- Marie
#path = '/Users/marie/Desktop/Cours/S2/Data bases/Data/New/'

comment2.to_csv(path + 'comment.csv', index = False)
author2.to_csv(path + 'author.csv', index = False)
score2.to_csv(path + 'score.csv', index = False)
controversy2.to_csv(path + 'controversy.csv', index = False)
depends2.to_csv(path + 'depends.csv', index = False)
parent2.to_csv(path + 'parent.csv', index = False)
is_distinguished2.to_csv(path + 'is_distinguished.csv', index = False)
distinguished2.to_csv(path + 'distinguished.csv', index = False)
```

2. Part 2: Exploring the database using SQL

2.0. Preliminary step

At the end of the first part, new files are created and exported as csv. To start this second part, one need to create the new tables and refill them with the new csv files.
Here are the steps:

- Creation of a database and its connection
- Drop all the tables

- Create all the new tables
- Import the csv files
- Transform dataframe rows into list
- Insert the dataframe in the table.

The instructions are the same as in the first part but applied on the subsamples created at the end of the first part. Some tables are not imported again (removal, subreddit and removed) because they don't bring any information.

2.1. Basic queries

2.1.1. Number of rows in each table

The number of rows in each table can be gotten by selection all the rows from all the tables.

```
#-- Count the number of rows in each table

sql_count3 = {'count_author': '''SELECT * FROM author''',
              'count_distinguished': '''SELECT * FROM distinguished''',
              'count_controversy': '''SELECT * FROM controversy''',
              'count_score': '''SELECT * FROM score''',
              'count_parent': '''SELECT * FROM parent''',
              'count_comment': '''SELECT * FROM comment''',
              'count_is_distinguished': '''SELECT * FROM is_distinguished''',
              'count_depends': '''SELECT * FROM depends'''}
n_rows3={}
sum_rows=0

for k in sql_count3:
    sql = sql_count3[k]
    n_rows3[k] = len(c.execute(sql).fetchall())
    sum_rows = sum_rows + n_rows3[k]

n_rows3
```

Table	Number of Rows
score	1700
depends	
is_distinguished	
comments	1701
parent	1618
author	1517
distinguished	1
controversy	2

Table 2: Size of the tables

The total number of rows is 9939. Thus, the database is know well on size 5MB.

2.1.2. Author statistics

```
#-- Minimum, maximum and average number of comments written by the same author

query_authors_1 = '''SELECT min(countcomment) AS "Minimum number of comments by
author",
                        max(countcomment) AS "Maximum number of comments by
author",
                        round(avg(countcomment),3) AS "Average number of comment
by author"
                        FROM (SELECT c.author, COUNT(c.id) AS "countcomment"
                        FROM comment AS c
                        GROUP BY c.author)'''

df_authors_1 = pd.read_sql(query_authors_1, con)
df_authors_1
```

The function min, max, and avg will be often used in this report. They enable to compute, respectively, the minimum, the maximum and the average. Here, these statistics are compute on the count of comment written by authors.

The python code above enables to compute the minimum, maximum and average number of comments written by one author.

Minimum	Maximum	Average
1	127	1.1

Table 3: Summary statistics on the author table

The maximum number of comments written by a single author is 127 whereas the minimum is 1. In average, it seems authors write only one comment. The average being so small and the maximum so high, it means that many authors write only one comment and very few a lot of comments.

If one looks at the number of comments written by author, for author having written more than one comment, one can see that only 31 authors over 1700 have written more than one comment.

This code enables to see that:

```
#-- Number of authors that have written more than 1 comment

query_authors_2 = '''SELECT c.author, COUNT(c.id) AS "number of comments"
                        FROM comment AS c
                        GROUP BY c.author
                        HAVING COUNT(c.id) > 1
                        ORDER BY count(c.id) DESC'''

df_authors_2 = pd.read_sql(query_authors_2, con)
df_authors_2
```

The function GROUP BY enables to put together all authors when they appear several time. If they appear more than once they are kept. They are then ordered by the author appearing the most to the least.

The following table shows the top 10 authors having written the most comments.

Author	Number of comments written
[deleted]	127
AutoModerator	18
Late_Night_Grumbler	7
-_-Equinox666-_-	5
suddenweightloss	3
roguetroll	3
bolognahole	3
walkingcarpet23	2
tinyhousebuilder	2
rotten_person	2

Table 4: Top 10 authors that have written the most comment

The maximum number of comments written, 127, corresponds to the comments that were deleted. The second author is AutoModerator. Thus, the "real" author that has written the most comment is Late_Night_Grumbler with 7 comments.

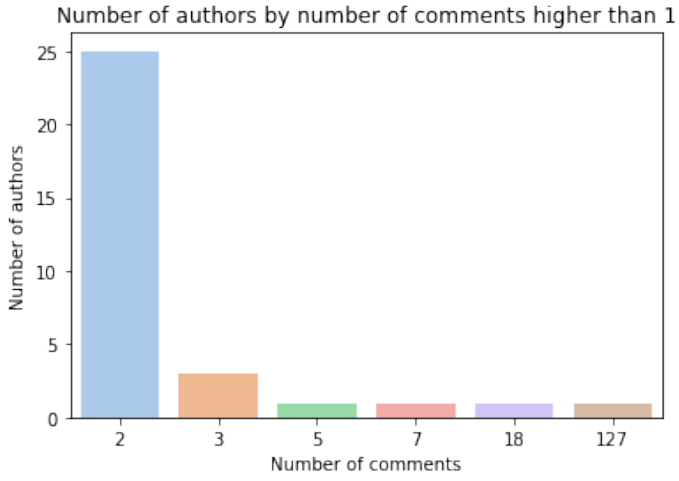

Figure 2: Bar plot number of authors (having written more than one comment) by number of comments

One can see on the bar plot that among the authors that have written more than one comments, most have written only 2.

2.1.3. Comment statistics

```
#-- First and last comments date
```

```
query_comment_1 = '''SELECT min(datetime(c.created_utc, 'unixepoch')) AS "Date
first comment",
                            max(datetime(c.created_utc, 'unixepoch')) AS "Date last
comment"
                            FROM comment AS c;'''

df_comment_1 = pd.read_sql(query_comment_1, con)
df_comment_1
```

To check if there are any missing values and the first and last day a comment was posted, the above code can be used. It gives us that
-the first date and hour is: 1970-01-01 00:00:00
-the last one is 2015-05-31 23:47:53.

The 1970-01-01 00:00:00 corresponds to a missing value and is replaced with the oldest date available. Once delete, one gets with the previous code that
-the first value is: 2015-05-01 00:02:28.
-the second is: 2015-05-31 23:47:53.

One now has only dates in May 2015 which is correct.
If one checks the number of comments in May with the following code, one can see that there are now 1700 number of comments.

```
#-- Month with the number of comments

query_comment_2 = '''SELECT strftime('%m', datetime(c.created_utc, 'unixepoch')) AS
"Month",
                            count(c.id) as "Number of comments"
                            FROM comment AS c
                            GROUP BY strftime('%m', datetime(c.created_utc,
'unixepoch'));'''

df_comment_2 = pd.read_sql(query_comment_2, con)
df_comment_2
```

This query just enables to check that the only month available is May.

Knowing the minimum, maximum and average number of comments posted by day is possible with the following code.

```
#-- Minimum, maximum and average number of comments by day

query_comment_3 = '''SELECT min(countcomment) AS "Minimum number of comments",
                            max(countcomment) AS "Maximum number of comments",
                            avg(countcomment) AS "Average number of comments"
                            FROM (SELECT strftime('%d', datetime(c.created_utc,
'unixepoch')) AS "Day",
                                        count(c.id) AS "countcomment"
                                        FROM comment AS c
                                        GROUP BY strftime('%d',
datetime(c.created_utc, 'unixepoch'))
```

```
                                        ORDER BY count(c.id) DESC);'''

df_comment_3 = pd.read_sql(query_comment_3, con)
df_comment_3
```

Minimum	Maximum	Average
32	89	55

Table 5: Summary statistics of the number of comments posted each day

The table shows that on average, the number of comments posted each day is 55. The day where the least comments were posted, 32 were posted. And the day where the post comments were posted, 89 were posted.

It can be interesting to see which one the days correspond to the minimum and maximum and in general the number of comments posted each day.

```
#-- Day with the most number of comments

query_comment_4 = '''SELECT strftime('%d', datetime(c.created_utc, 'unixepoch')) AS
"Day", count(c.id) AS "Number of comments"
                FROM comment AS c
                GROUP BY strftime('%d', datetime(c.created_utc, 'unixepoch'))
                ORDER BY count(c.id) DESC;'''

df_comment_4 = pd.read_sql(query_comment_4, con)
df_comment_4
```

The above code gives the following result. It computes for each day using a group by the number of times it appears, i.e. the number of times a comment is posted that day. The table below is ordered by day but the actual table given is ordered by number of comments posted.

Day	**Comments posted**	**Day**	**Comments posted**
1	63	17	51
2	50	18	54
3	54	19	54
4	47	20	41
5	54	21	44
6	89	22	64
7	58	23	32
8	47	24	48
9	40	25	49
10	40	26	63
11	58	27	53
12	62	28	84
13	59	29	83
14	54	30	47
15	63	31	47
16	48		

Table 6: Number of comments posted each day

The 6th was the day were the most comment was posted and the 23rd the day with the least posts. It can also be seen on the bar plot below.

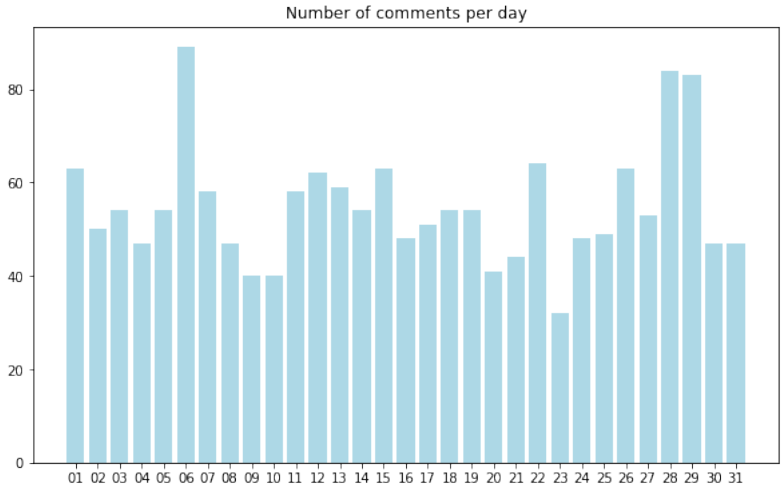

Figure 3: Bar plot of the number of comments each day

Each day corresponds to a weekday (Monday, Tuesday…). Hence, one looks at the number of comments posted each weekday.

The following code enable to do so:

```
#-- Associate weekday to day number

weekdays = [('monday', '("04", "11", "18", "25")'),
            ('tuesday', '("05", "12", "19", "26")'),
            ('wednesday', '("06", "13", "20", "27")'),
            ('thursday', '("07", "14", "21", "28")'),
            ('friday', '("01", "08", "15", "22", "29")'),
            ('saturday', '("02", "09", "16", "23", "30")'),
            ('sunday', '("03", "10", "17", "24", "31")')]
```

A list of weekdays is created that will be used in the query after. For each weekday, the query counts the number of times are day appears.

```
#-- Number of comments for each weekday

query_comment_6 = '''SELECT count(c.id) AS "Number of comments"
                        FROM comment AS c
                        WHERE strftime('%d',datetime(c.created_utc, 'unixepoch'))
IN {};'''

df_comment_6 = pd.DataFrame([], columns=['Weekday', 'Number of comments'])

for i in range(len(weekdays)):
    df_comment_6 = df_comment_6.append( {'Weekday': weekdays[i][0],
           'Number of comments':
pd.read_sql(query_comment_6.format(weekdays[i][1]),
```

```
                                                con)['Number of comments'][0]},
ignore_index=True)

df_comment_6
```

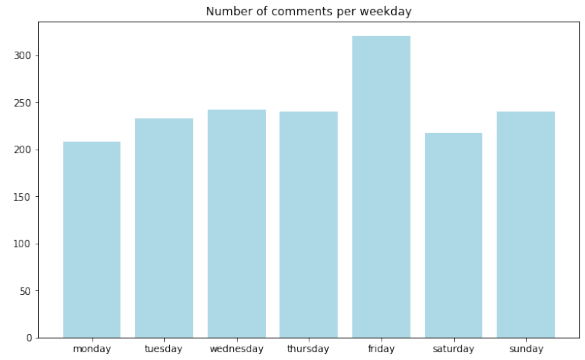

Figure 4: Bar plot of the number of comments each weekday

Weekday	Comments posted
Monday	208
Tuesday	233
Wednesday	242
Thursday	240
Friday	320
Saturday	217
Sunday	240

Table 7: Number of comments each weekday

One can see on the bar plot and table that Saturday is the day with the most post (320) whereas Monday is the day with the least (208). It makes sense since Saturday is during the weekend when people have more time than on Monday.

2.1.4. Parent statistics

A parent comment is a comment for which other comments are answers/replies. They are link with the link_id variable.

The minimum, maximum and average replies to a parent comment are gotten with the following code:

```
#-- Minimum, maximum and average number of replies

query_parent_1 = '''SELECT min(countcomment) AS "Minimum number of replies",
                           max(countcomment) AS "Maximum number of replies",
                           round(avg(countcomment),3) AS "Average number of
replies"
                    FROM (SELECT p.parent_id, COUNT(p.link_id) AS
"countcomment"
                    FROM parent AS p
                    GROUP BY p.link_id);'''

df_parent_1 = pd.read_sql(query_parent_1, con)
df_parent_1
```

Minimum	Maximum	Average
1	15	1

Table 8: Summary statistics of the number of replies

In average, a parent comment is linked to only 1 other comment, which means that only one person answered to the question/comment. The minimum is 1 as well but the maximum is 15.

206 parents have more than 1 answer.

The code below enables to get the id number of these parents and the number of answers. The number of answers is gotten with the GROUP BY function. Only parent_ids are kept if there are at least 1 answer, for this one uses HAVING COUNT. The parent_ids are then ordered by descending order of the number of answers they have.

```
#-- Number of comments that have more than one reply

query_parent_2 = '''SELECT p.parent_id, count(p.link_id) AS "number of comments"
                FROM parent AS p
                GROUP BY p.link_id
                HAVING count(p.link_id) > 1
                ORDER BY count(p.link_id) DESC;'''

df_parent_2 = pd.read_sql(query_parent_2, con)
df_parent_2
```

The 10 parents with the most answers are:

Parent_id	Number of answers
t1_crldt2w	15
t1_cron9zg	12
t1_crbbksr	12
t1_cr010xc	11
t1_cqzqzhn	11
t1_crh3vze	9
t1_crfvpep	9
t1_crazy6o	9
t1_cr7pha2	9
t1_cr6ej2j	9

Table 9: Top 10 parent comments with the most answers

Here is the code enabling to get this table. It is the same as before but with a limit and no constraint on the number of answers.

```
#-- Top 10 number of replies

query_parent_3 = '''SELECT p.parent_id, COUNT(p.link_id) AS "Number of replies"
                FROM parent AS p
                GROUP BY p.link_id
                ORDER BY COUNT(p.link_id) DESC
                LIMIT 10;'''

df_parent_3 = pd.read_sql(query_parent_3, con)
```

```
df_parent_3
```

The parent comment with the most answers is t1_crldt2w with 15 links.

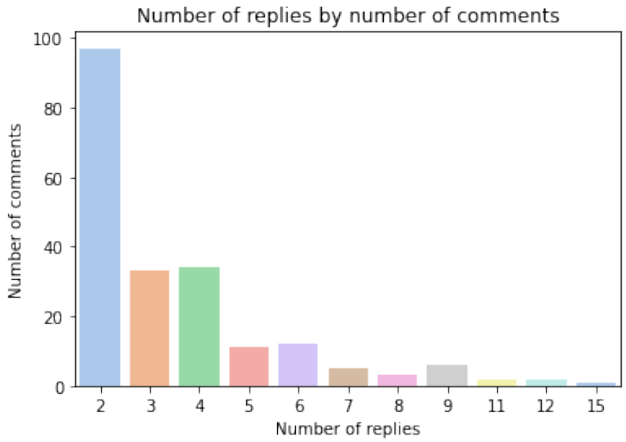

Figure 5: Number of replies by parent comments

One may see on the bar plot that almost 100 parent comments got only 2 answers, around 35 got 3 or 4 and very few got more.

2.1.5. Score statistics

The score table has 6 variables. However, ups is exactly equal to score, and downs, score_hidden and gilded are always equal to 0. Thus, one will focus on score.

```
#-- Mininum, maximum and average score

query_score_1 = '''SELECT avg(s.score) AS "Average score",
                          min(s.score) AS "Minimum score",
                          max(s.score) AS "Maximum score"
                          FROM score AS s;'''

df_score_1 = pd.read_sql(query_score_1, con)
df_score_1
```

First, one looks at the minimum, maximum and average score.

Minimum	Maximum	Average
-83	4466	21

Table 10: Summary statistics of score

The minimum is not going much below 0 since it of -83. One the contrary, the maximum is very high: 4466. In average a comment has a score of 21.

```
#-- 10 worst scores

query_score_2 = '''SELECT s.id, s.score
```

```
                        FROM score AS s
                        ORDER BY s.score
                        LIMIT 10;'''

df_score_2 = pd.read_sql(query_score_2, con)
df_score_2
```

```
#-- 10 best scores

query_score_3 = '''SELECT s.id, s.score
                        FROM score AS s
                        ORDER BY s.score DESC
                        LIMIT 10;'''

df_score_3 = pd.read_sql(query_score_3, con)
df_score_3
```

The codes above extract the 10 best and worst scores that can be found in the following tables.

Comment Id	Worst scores	Comment Id	Best scores
crklhcv	-83	cr4y7m7	4466
crktr4q	-34	cr8phcb	4397
crig3sx	-15	crmp2cq	3305
cquq9j5	-13	cr01luo	1713
cr759o9	-9	crl1czr	1413
cr0xhf4	-8	cqvt1ir	1278
cqzyqni	-7	cr5e1zm	1260
crcylp1	-7	cqxddni	1197
crbufl1	-6	cqy894x	1184
cqvbjbs	-5	crfc940	1155

Table 11: 10 best and words scores

Crklhcv is the comment with the worst score whereas cr4y7m7 is the comment with the best score.

It seems the negative score are rather close to 0, so let us see how many negative scores there are. One will also look at the number of scores equal to 0 and positive.

The next queries extract the count of scores based on some criteria, which is why there is an imbricated SELECT.

```
#-- Number of negative scores

query_score_4 = '''SELECT COUNT(neg.nbscore) AS "Number of negative scores"
                        FROM (SELECT s.id AS nbscore
                        FROM score AS s
                        WHERE s.score < 0) AS neg;'''

df_score_4 = pd.read_sql(query_score_4, con)
```

```
df_score_4
```

```
#-- Number of scores equal to 0

query_score_5 = '''SELECT COUNT(zero.nbscore) AS "Number of scores equal to 0"
                FROM (SELECT s.id AS nbscore
                FROM score AS s
                WHERE s.score = 0) AS zero;'''

df_score_5 = pd.read_sql(query_score_5, con)
df_score_5
```

```
#-- Number of positive scores

query_score_6 = '''SELECT COUNT(pos.nbscore) AS "Number of positive scores"
                FROM (SELECT s.id AS nbscore
                FROM score AS s
                WHERE s.score > 0) AS pos;'''

df_score_6 = pd.read_sql(query_score_6, con)
df_score_6
```

Negative	0	Positive
43	83	1574

Table 12: Number of positive and negative scores and scores equal to 0

There are clearly much more positive scores than negative ones or scores equal to 0.

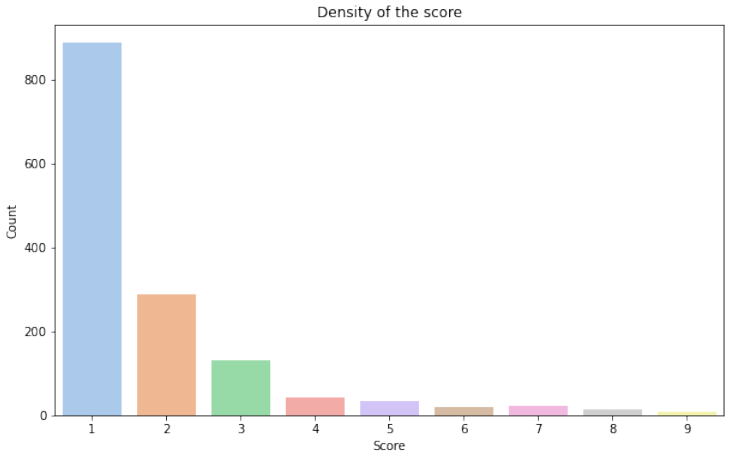

Figure 6: Bar plot of the score

When looking at scores between 1 and 9, which represents almost all the comments' scores, one may see most of the score are equal to 1 (around 900), a few (around 300) are equal to 3. A very few scores are strictly higher than 2.

2.2. Complex queries

2.2.1. Giving statistics about the scores according to the fact that a comment is deleted or not

This query aims to give some basic summary statistics on the scores based on the fact that a comment is deleted or not. This could help for the project of WebMining, to find some new features to predict the score. Two queries were created: one that gives statistics about the score for the comments that aren't deleted, and one that gives statistics about the score for the comments that are deleted.

The statistics given are in both cases are:

- The percentage of comments with a positive score
- The average score among the positive scores
- The percentage of comments with a negative score
- The average score among the negative scores
- The percentage of comments with a null score
- The number of comments
- The overall average score

Query on the not deleted comment:

This query has four subqueries:

- One allows to compute the number of comments and the average score for the positive score,
- One allows to compute the number of comments and the average score for the negative score,
- One allows to compute the number of comments that have a null score,
- One allows to compute the number of comments and the average score overall.

This then allows to compute the percentage.

```
#-- Statistics on comments not deleted

query_notdeleted ='''SELECT CAST(pos.NbCom AS float)/ CAST(overall.NbCom AS
float)*100 AS "Percentage of Comments not deleted with a positive score",
                            pos.AvgScore AS "Average Score among the positive
scores",
                            CAST(neg.NbCom AS float)/ CAST(overall.NbCom AS
float)*100 AS "Percentage of Comments not deleted with a negative score",
                            neg.AvgScore AS "Average Score among the negative
scores",
                            CAST(zero.NbCom AS float)/ CAST(overall.NbCom AS
float)*100 AS "Percentage of Comments not deleted with a null score",
                            ROUND(AVG(overall.AvgScore),2) AS "Overall Average
Score for the comments not deleted",
                            overall.NbCom AS "Number of comments not deleted"
```

```
                            FROM (SELECT COUNT(c.body) AS "NbCom",
                                         ROUND(AVG(s.score),2) AS "AvgScore"
                                         FROM comment AS c, score AS s
                                         WHERE c.id = s.id
                                             AND c.body NOT LIKE '%[deleted]%') AS
overall,

                                 (SELECT COUNT(c.body) AS "NbCom",
                                         ROUND(AVG(s.score),2) AS "AvgScore"
                                         FROM comment AS c, score AS s
                                         WHERE c.id = s.id AND s.score>0
                                             AND c.body NOT LIKE '%[deleted]%') AS
pos,

                                 (SELECT COUNT(c.body) AS "NbCom",
                                         ROUND(AVG(s.score),2) AS "AvgScore"
                                         FROM comment AS c, score AS s
                                         WHERE c.id = s.id
                                             AND s.score<0
                                             AND c.body NOT LIKE '%[deleted]%') AS
neg,

                                 (SELECT COUNT(c.body) AS "NbCom"
                                         FROM comment AS c, score AS s
                                         WHERE c.id = s.id
                                             AND s.score=0
                                             AND c.body NOT LIKE '%[deleted]%') AS
zero;'''

df_notdeleted = pd.read_sql(query_notdeleted, con)
df_notdeleted
```

Query on the deleted comments:

This query has four subqueries:

- One allows to compute the number of comments and the average score for the positive score,
- One allows to compute the number of comments and the average score for the negative score,
- One allows to compute the number of comments that have a null score,
- One allows to compute the number of comments and the average score overall.

This then allows to compute the percentage.

One can actually see that it is exactly the same as before but for deleted comments.

```
#-- Statistics on comments deleted

query_deleted ='''SELECT CAST(pos.NbCom AS float)/ CAST(overall.NbCom AS float)*100
AS "Percentage of Comments deleted with a positive score",
```

```
                        pos.AvgScore AS "Average Score among the positive scores",
                        CAST(neg.NbCom AS float)/ CAST(overall.NbCom AS float)*100
AS "Percentage of Comments deleted with a negative score",
                        neg.AvgScore AS "Average Score among the negative scores",
                        CAST(zero.NbCom AS float)/ CAST(overall.NbCom AS
float)*100 AS "Percentage of Comments deleted with a null score",
                        ROUND(AVG(overall.AvgScore),2) AS "Overall Average Score
for the comments deleted",
                        overall.NbCom AS "Number of comments deleted"

                        FROM   (SELECT COUNT(c.body) AS "NbCom",
ROUND(AVG(s.score),2) AS "AvgScore"
                                    FROM comment AS c, score AS s
                                    WHERE c.id = s.id  AND c.body LIKE
'%[deleted]%') AS overall,

                                (SELECT COUNT(c.body) AS "NbCom",
ROUND(AVG(s.score),2) AS "AvgScore"
                                    FROM comment AS c, score AS s
                                    WHERE c.id = s.id AND s.score>0 AND c.body LIKE
'%[deleted]%') AS pos,

                                (SELECT COUNT(c.body) AS "NbCom",
ROUND(AVG(s.score),2) AS "AvgScore"
                                    FROM comment AS c, score AS s
                                    WHERE c.id = s.id AND s.score<0 AND c.body LIKE
'%[deleted]%') AS neg,

                                (SELECT COUNT(c.body) AS "NbCom"
                                    FROM comment AS c, score AS s
                                    WHERE c.id = s.id AND s.score=0 AND c.body LIKE
'%[deleted]%') AS zero;'''

df_deleted = pd.read_sql(query_deleted, con)
df_deleted
```

The output of these two queries are the following tables:

Percentage of comments not deleted with a positive score	92,62%
Average score among the positive scores for the comments not deleted	25,32
Percentage of comments not deleted with a negative score	2,52%
Average score among the negative scores for the comments not deleted	-6,05
Percentage of comments not deleted with a null score	4,86%
Overall average score for the comments not deleted	23,3
Number of comments not deleted	1585

Table 13: Result tables with the statistics on the score for not deleted comments

Percentage of comments deleted with a positive score	92,62%
Average score among the positive scores for the comments deleted	2,94
Percentage of comments deleted with a negative score	2,61%
Average score among the negative scores for the comments deleted	-3
Percentage of comments deleted with a null score	5,22%
Overall average score for the comment deleted	2,63
Number of comments deleted	115

Table 13_bis: Result tables with the statistics on the score for deleted comments

This analysis allows to compare the deleted comments with the non-deleted comments. It allows to suggest why some comments have been deleted. One can see that the percentage of comments with a positive score is similar for the deleted comments and for the not deleted comments. Similarly, the percentage of comments with a negative score is similar for the deleted comments and for the not deleted comments. Finally, the percentage of comments with a null score is similar for the deleted comments and for the not deleted comments. Thus, even if there are much more not deleted comments than deleted comments, the proportions of comments according to the sign of the scores remains quite similar from one group to another.

However, one can see that the average score is much higher for the not deleted comments than for the deleted comments. As a comment cannot be scored after having been deleted, it means that they had this low score before being deleted, and that even may the reason why these comments were deleted.

2.2.2. Giving statistics about the scores according to the fact that a comment is edited or not

This query aims to give some basic summary statistics on the scores based on the fact that a comment is edited or not, and if so, on the date of edition. This could help for the project of WebMining, to find some new features to predict the score. Two queries were created: one that gives statistics about the score for the comments according to the edition of the comment, and one that gives statistics about the score for the edited comments according to the time of edition.

First Query:

The statistics given in the first query are:

- The number of comments that have been edited
- The average score for the comments that have been edited
- The minimum score for the comments that have been edited
- The maximum score for the comments that have been edited
- The number of comments that haven't been edited
- The average score for the comments that haven't been edited
- The minimum score for the comments that haven't been edited
- The maximum score for the comments that haven't been edited

This query has two subqueries:

- One that compute the number of comments and some statistics about the scores for the comments that have been edited,
- One that compute the number of comments and some statistics about the scores for the comments that haven't been edited.

This allows then to compare all these statistics together.

```
#-- Statistics on the comments according to the edition or not

query_edition = '''SELECT   Ed.NbCom AS "Number of comment that have been edited",
                            Ed.AvgScore AS "Average Score for the comments that
have been edited",
                            Ed.MinScore AS "Mininimum Score for the comments that
have been edited",
                            Ed.MaxScore AS "Maximum Score for the comments that
have been edited",
                            NotEd.NbCom AS "Number of comment that haven't been
edited",
                            NotEd.AvgScore AS "Average Score for the comments that
haven't been edited",
                            NotEd.MinScore AS "Mininimum Score for the comments
that haven't been edited",
                            NotEd.MaxScore AS "Maximum Score for the comments that
haven't been edited"

                        FROM (SELECT COUNT(*) AS "NbCom", ROUND(AVG(s.score),2) AS
AvgScore,
                                     MIN(s.score) AS MinScore,
                                     MAX(s.score) AS MaxScore

                                FROM comment AS C, score AS s WHERE s.id=c.ID AND
c.edited!=0) AS Ed,
                                    (SELECT COUNT(*) AS 'NbCom',
                                            ROUND(AVG(s.score),2) AS AvgScore,
                                            MIN(s.score) AS MinScore,
                                            MAX(s.score) AS MaxScore FROM comment
AS c,
                                            score AS s
                                            WHERE s.id=c.ID AND c.edited==0) AS
NotEd;'''

df_edition = pd.read_sql(query_edition, con)
df_edition
```

The output is the following table:

Number of comments that have been edited	29
Average score for the comments that have been edited	59,54
Minimum score for the comments that have been edited	-9
Maximum score for the comments that have been edited	1413
Number of comments that haven't been edited	1670
Average score for the comments that haven't been edited	21,27
Minimum score for the comments that haven't been edited	-83
Maximum score for the comments that haven't been edited	4466

Table 14: Result tables with the statistics on the score according to the edition or not

This analysis allows to see if the edition of a comment plays on the average score.

One can first see that there are much more not edited comments than edited comment. However, for the few comments that have been edited:

- The average score is more than twice higher than the average score for not edited comments.
- The minimum score is more than nine times higher than the minimum score for not edited comments.
- The maximum score is more than three times higher than the maximum score for not edited comments.

The study of the maximums and minimums allows to prove that the average is not influenced by noises and extreme values.

Editing a comment (for example, for grammatical correction) seems to lead to a higher score.

Second query:

The statistics given in the second query are:

- The number of comments that have been edited the same hour (and the same day) they were created,
- The average score for the comments that have been edited the same hour (and the same day) they were created,
- The minimum score for the comments that have been edited the same hour (and the same day) they were created,
- The maximum score for the comments that have been edited the same hour (and the same day) they were created,
- The number of comments that have been edited the same day they were created,
- The average score for the comments that have been edited the same day they were created,
- The minimum score for the comments that have been edited the same day they were created,

- The maximum score for the comments that have been edited the same day they were created,
- The number of comments that have been edited at least one day after they were created,
- The average score for the comments that have been edited at least one day after they were created,
- The minimum score for the comments that have been edited at least one day after they were created,
- The maximum score for the comments that have been edited at least one day after they were created.

This query has three subqueries:

- One that computes the number of comments and statistics about the scores for the comments that have been edited in the same day and in the same hour as the creation.
- One that computes the number of comments and statistics about the scores for the comments that have been edited in the same day as the creation.
- One that computes the number of comments and statistics about the scores for the comments that have been edited at least one day after the creation.

The limit of this query is that actually, if a comment has been created at 9:40am, the algorithm will state that it has been edited in the same hour only if it has been edited from 9:40pm to 9:59pm. Also, it will state that it has been edited in the same day only if it has been edited from 9:40pm to 11:59pm. Similarly, it will state that it has been edited at least one day after if it has been edited from 00:00am the next day. It doesn't take 1 full hour.

It gives the following code:

```
#-- Statistics on the comments according to the time/date of edition

query_edition2 = '''SELECT SameHour.NbCom AS "Number of comments that have been
edited the same hour they were created",
                           SameHour.AvgScore AS "Average Score of the comments that
have been edited the same hour they were created",
                           SameHour.MinScore AS "Mininimum Score for the comments
that have been edited the same hour they were created",
                           SameHour.MaxScore AS "Maximum Score for the comments
that have been edited the same hour they were created",

                           SameDay.NbCom AS "Number of comments that have been
edited the same day they were created",
                           SameDay.AvgScore AS "Average Score of the comments that
have been edited the same day they were created",
                           SameDay.MinScore AS "Mininimum Score for the comments
that have been edited the same day they were created",
                           SameDay.MaxScore AS "Maximum Score for the comments that
have been edited the same day they were created",

                           NotSameDay.NbCom AS "Number of comments that have been
edited at least one day after they were created",
```

```
                        NotSameDay.AvgScore AS "Average Score of the comments
that have been edited at least one day after they were created",
                        NotSameDay.MinScore AS "Mininimum Score for the comments
that have been edited at least one day after they were created",
                        NotSameDay.MaxScore AS "Maximum Score for the comments
that have been edited at least one day after they were created"

                        FROM (SELECT COUNT(*) AS "NbCom", ROUND(AVG(s.score),2)
AS AvgScore,
                                    MIN(s.score) AS MinScore,
                                    MAX(s.score) AS MaxScore,
                                    strftime('%H', datetime(c.edited,
'unixepoch')) AS "Hour_edited" ,
                                    strftime('%H', datetime(c.created_utc,
'unixepoch')) AS "Hour_created" ,
                                    strftime('%d', datetime(c.edited,
'unixepoch')) AS "Day_edited" ,
                                    strftime('%d', datetime(c.created_utc,
'unixepoch')) AS "Day_created"

                                    FROM comment AS c, score as s
                                    WHERE c.edited!=0
                                        AND Hour_edited=Hour_created
                                        AND Day_edited=Day_created
                                        AND c.id = s.id) AS SameHour,

                                (SELECT COUNT(*) AS "NbCom",
                                    ROUND(AVG(s.score),2) AS AvgScore,
                                    MIN(s.score) AS MinScore,
                                    MAX(s.score) AS MaxScore,
                                    strftime('%d', datetime(c.edited,
'unixepoch')) AS "Day_edited" ,
                                    strftime('%d', datetime(c.created_utc,
'unixepoch')) AS "Day_created"

                                    FROM comment AS c, score as s
                                    WHERE c.edited!=0
                                        AND Day_edited=Day_created
                                        AND c.id = s.id) AS SameDay,

                                (SELECT COUNT(*) AS "NbCom",
                                    ROUND(AVG(s.score),2) AS AvgScore,
                                    MIN(s.score) AS MinScore,
                                    MAX(s.score) AS MaxScore,
                                    strftime('%d', datetime(c.edited,
'unixepoch')) AS "Day_edited" ,
                                    strftime('%d', datetime(c.created_utc,
'unixepoch')) AS "Day_created"

                                    FROM comment AS c, score as s
                                    WHERE c.edited!=0
```

```
                                        AND Day_edited!=Day_created
                                        AND c.id = s.id) AS NotSameDay;'''

df_edition2= pd.read_sql(query_edition2, con)
df_edition2
```

The output is the following tables (One here shows three tables for visualization, but the output is only table):

Number of comments that have been edited the same hour (and the same day) they were created	Average score for the comments that have been edited the same hour (and the same day) they were created	Minimum score for the comments that have been edited the same hour (and the same day) they were created	Maximum score for the comments that have been edited the same hour (and the same day) they were created
13	109,15	-8	1413

Number of comments that have been edited the same day they were created	Average score for the comments that have been edited the same day they were created	Minimum score for the comments that have been edited the same day they were created	Maximum score for the comments that have been edited the same day they were created
23	71,78	-8	1413

Number of comments that have been edited at least one day after they were created	Average score for the comments that have been edited at least one day after they were created	Minimum score for the comments that have been edited at least one day after they were created	Maximum score for the comments that have been edited at least one day after they were created
6	11,17	-9	60

Table 15: Result tables with the statistics on the score according to the edition time/date

Actually, one can see that:

- The average score is very high for the comments that have been edited the same hour they were created, or even the same day they were created.
- The minimum score among the edited comments is lower than the minimum score corresponding to a comment that have been edited through the same hour and same day as its creation.
- The maximum score among the edited comments is the maximum score corresponding to a comment that have been edited through the same hour and same day as its creation.

It seems that the faster the comment is edited, the higher the score is.

2.2.3. Giving the Pearson coefficient between the length of the comment and the score

This last query aims to compute the Pearson Correlation Coefficient between the length of the comment and the score of the comment.

Here is the formula of this coefficient:

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

$$r = \frac{(\sum xy) - (\sum x)(\sum y)}{\sqrt{[\sum x^2 - (\sum x)^2][\sum y^2 - (\sum y)^2]}}$$

Where x is the length of the comment and y is the score of the comment.

We built a query that returns a table with two values: Numerator and Denominator. Actually, to compute the Pearson Correlation Coefficient, one need to, using Python language, to build a new column in the table that takes as value:

$$Coefficient = \frac{Numerator}{\sqrt{Denominator}}$$

Because no one could find a solution to compute a square root using SQLite3.

This query has only one subquery that allows to compute all the elements of the big equation.

```
#-- Compute Pearson coefficient

query_length = '''SELECT (tot_sum - (score_sum * comment_sum )/ _count) AS
'Numerator', (((score_sum_sq - score_sum*score_sum) / _count) * ((comment_sum_sq -
comment_sum*comment_sum) / _count)) AS "Denominator"
                        FROM (SELECT sum(s.score) AS score_sum,
                                     sum(length(c.body)) AS comment_sum,
                                     sum(s.score * s.score) AS score_sum_sq,
                                     sum(length(c.body) * length(c.body)) AS
comment_sum_sq,
                                     sum(s.score * length(c.body)) AS tot_sum,

                                     count(*) AS _count

                        FROM score AS s, comment AS c
                        WHERE s.id = c.id);'''
```

The output is the following table:

Numerator	Denominator	Pearson Correlation Coefficient
269664	25086040001466	0.05384

Table 16: Result tables of the Pearson coefficient

The Pearson Coefficient between the length of the body (without any precleaning) and the score is very low, equal to 0,05. It means that there isn't any correlation between these two variables.

2.2.4. Giving statistics about the words used in the comments

2.2.4.1 Occurrence of words

Before looking for statistics about the words used in the comment, these must be extracted. To do so a dataframe of only the comments is created. From this, the body of the comments can be cleaned using several steps:

- The text is converted to lowercase.
- The punctuation, as well as strings to go to the line (\n and \ r) are removed.
- The sentences are tokenized. It means the words are split.
- The defined english stopwords are removed.
- The comments are lemmatised (only the root of the words is kept).
- The comments are un-tokenized to have the sentence back.
- The url, hashtag, mentions, digits, html tags and co (all of these are specific to Reddit) are deleted.

It is then counted the times a word appears in the comments. Thus, the data frame has 2 columns: word which gives the words, and number which gives the number of occurrences.

A new table is then created from this dataframe.

```
# Delete table if already exists
drop_word = '''DROP TABLE IF EXISTS word'''
c.execute(drop_word)

# Create table word
create_word = '''CREATE TABLE word (word TEXT, number INT, CONSTRAINT pk_word
PRIMARY KEY (word, number))'''
c.execute(create_word)

# Insert values
insert_word = '''INSERT INTO word (word, number) VALUES (?, ?)'''
c.executemany(insert_word, list_words)
```

```
#-- Number of words in the table
pd.read_sql('''SELECT count(w.word) FROM word AS w''', con)
```

The last query gives the number of words, there are 5485 words.

```
#-- Ten words the most used
pd.read_sql('''SELECT * FROM word LIMIT 10''', con)
```

The table below gives the 10 most used words.

Word	Occurrence
like	209
one	176
people	159
would	152
get	151
time	136
know	125
deleted	116
thing	106
think	104

Table 17: 10 most used words

The word that appears the most is “like”.

```
#-- Minimum, maximum and average times a word appears

query_mma_words = '''SELECT avg(w.number) AS "Average number of reccurences",
                            min(w.number) AS "Minimum number of reccurences",
                            max(w.number) AS "Maximum number of reccurences"
                                    FROM word AS w;'''

df_query_mma_words = pd.read_sql(query_mma_words, con)
df_query_mma_words
```

Minimum	Maximum	Average
1	209	4

Table 18: Summary statistics of the number of occurrence of words

As one already knows it, the word that appears the most appears 209 times. Some words appear only once. In average, a word appears 4 times.

```
#-- Number of words that appear 1 time and 209 times

query_mm_number_words = ''' SELECT w1.count1 AS "Words reccurence = 1",
                                   w2.count2 AS "Words reccurence = 209"

                                    FROM

                                        (SELECT count(w.word) AS count1
                                    FROM word AS w
                                    WHERE w.number=1) AS w1,

                                        (SELECT count(w.word) AS count2
                                    FROM word AS w
                                    WHERE w.number=209) AS w2;'''
```

```
df_query_mm_number_words = pd.read_sql(query_mm_number_words, con)
df_query_mm_number_words
```

The query above enables us to say that one word appears 209 times and 3141 appear only once. Imbricated SELECTs are used because there are constraint on the count of words. One only wants counts of a certain value, here 1 and 209 which is why WHERE was used .

```
#-- Number of words that appear less and more than 4 times

query_number_words = ''' SELECT w1.count1 AS "Words reccurence <= 4 ",
                                    w2.count2 AS "Words reccurence > 4"

                                FROM

                                    (SELECT count(w.word) AS count1
                                FROM word AS w
                                WHERE w.number<=4) AS w1,

                                    (SELECT count(w.word) AS count2
                                FROM word AS w
                                WHERE w.number>4) AS w2;'''

df_query_number_words = pd.read_sql(query_number_words, con)
df_query_number_words
```

One can even be more precise with the query above and say that 4608 words appears 4 times or less and 877 appear more than 4 times. This query works as the previous one but with other conditions.

```
#-- Median reccurence

query_med_number_words = ''' SELECT w.number AS "Median"
                                FROM word AS w
                                ORDER BY w.number
                                LIMIT 1
                                OFFSET (SELECT count(*)
                                            FROM word) / 2;'''

df_query_med_number_words = pd.read_sql(query_med_number_words, con)
df_query_med_number_words
```

The median is 1 which proves once again that many words appear only once. The higher average means a few values are quite high.

2.2.4.2 Length of words

Now that one has looked at the occurrence, looking at the length of the words can also bring information.

```
#-- Minimum, maximum and average length of the words

query_mma_length_words = '''SELECT avg(length(w.word)) AS "Average length of
words",
                                  min(length(w.word)) AS "Miminum length of
words",
                                  max(length(w.word)) AS "Maximum length of words"
                                   FROM word AS w;'''

df_query_mma_length_words = pd.read_sql(query_mma_length_words, con)
df_query_mma_length_words
```

Minimum	Maximum	Average
1	96	7

Table 19: Summary statistics of the length of the words

The smallest word has only 1 (which could ‘r’ or ‘u’ for example used to say ‘are’ and ‘you’) letter whereas the longest one has 96. The longest word is probably not an existing word. In average, the words have 7 letters.

```
#-- Number of words of length smaller and higher than 10

query_number_length_words = ''' SELECT w1.count1 AS "Words of length < 10 ",
                                     w2.count2 AS "Words of length >= 10 "

                                  FROM

                                      (SELECT count(w.word) AS count1
                                  FROM word AS w
                                  WHERE length(w.word)<10) AS w1,

                                  (SELECT count(w.word) AS count2
                                  FROM word AS w
                                  WHERE length(w.word)>=10) AS w2;'''

df_query_number_length_words = pd.read_sql(query_number_length_words, con)
df_query_number_length_words
```

To see if there are more shorter or longer words, the above query is used and gives the number of words of length smaller and equal to above 10. It works at the ones previously used about conditions on the count of words. 4802 words have less than 10 characters whereas 683 have 10 or more. There are more “small” words. It is not surprising. Since so many words have a length under 10, let us look at each length individually.

The next query is quite long because several values are extracted, however it works as before. It just uses imbricated SELECTs to extract information on some conditions, here the length.

```
#-- Number of words of length 1 to 10

query_number_length_words2 = ''' SELECT w1.count1 AS "Words of length 1",
```

```
    w2.count2 AS "Words of length 2",
    w3.count3 AS "Words of length 3",
    w4.count4 AS "Words of length 4",
    w5.count5 AS "Words of length 5",
    w6.count6 AS "Words of length 6",
    w7.count7 AS "Words of length 7",
    w8.count8 AS "Words of length 8",
    w9.count9 AS "Words of length 9",
    w10.count10 AS "Words of length 10"

FROM

    (SELECT count(w.word) AS count1
FROM word AS w
WHERE length(w.word)=1) AS w1,

    (SELECT count(w.word) AS count2
FROM word AS w
WHERE length(w.word)=2) AS w2,

    (SELECT count(w.word) AS count3
FROM word AS w
WHERE length(w.word)=3) AS w3,

    (SELECT count(w.word) AS count4
FROM word AS w
WHERE length(w.word)=4) AS w4,

(SELECT count(w.word) AS count5
FROM word AS w
WHERE length(w.word)=5) AS w5,

    (SELECT count(w.word) AS count6
FROM word AS w
WHERE length(w.word)=6) AS w6,

    (SELECT count(w.word) AS count7
FROM word AS w
WHERE length(w.word)=7) AS w7,

    (SELECT count(w.word) AS count8
FROM word AS w
WHERE length(w.word)=8) AS w8,

(SELECT count(w.word) AS count9
FROM word AS w
WHERE length(w.word)=9) AS w9,

(SELECT count(w.word) AS count10
FROM word AS w
WHERE length(w.word)=10) AS w10;'''
```

```
df_query_number_length_words2 = pd.read_sql(query_number_length_words2, con)
df_query_number_length_words2
```

Length	Number of words
1	25
2	116
3	328
4	744
5	797
6	879
7	810
8	658
9	445
10	304

Table 20: Number of words of length 1 to 10

Most of these words, 879, have 6 characters. Only 25 words have only 1 character. It seems most of the words have between 4 and 7 characters.

Conclusion

The analysis of this database enabled to get information about different tables. One could establish with the basic queries the number of comments written by each author, the day and date the most comments were posted, the number of answers to comments and statistics about the score. The more complex queries gave one the possibility to look deeper in the score statistics by taking into account if the comment was deleted or not, edited or not and when the edition was made. An analysis on the words used in the comment could also be made. It gave some insight about the most common words, the occurrence of words and statistics about their length.