Massive Data Sets
Assignment No. 1

Note 1 **This assignment is to be done individually**

Note 2 Working with other people is prohibited.

Note 2 Sharing queries or files with other people is prohibited.

A note on Academic Integrity and Plagiarism

Please review the following documents:

- Standards for Professional Behaviour, Faculty of Engineering:
  `https://www.uvic.ca/engineering/assets/docs/professional-behaviour.pdf`
- Policies Academic Integrity, UVic:
  `https://www.uvic.ca/students/academics/academic-integrity/`
- Uvic's Calendar section on Plagirism:
  `https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk_0xsM_V`

Note specifically:

> Plagiarism
> Single or multiple instances of inadequate attribution of sources should result in a failing grade for the work. A largely or fully plagiarized piece of work should result in a grade of F for the course.

Submissions will be screened for plagiarism **at the end of the term**.
You are responsible for your own submission, but you could also be responsible if somebody plagiarizes your submission.

Objectives

After completing this assignment, you will have experience:

- Understanding how the DBMS evaluates a query
- Doing statistical analysis using a DBMS
- Operating on JSON data using a DBMS
- 500-level students only: loading CSV data into a DBMS

IMDB

The IMDB makes available a lot of information about movies. See `https://www.imdb.com/interfaces/` for a description of this information.

A. I have created a database called *imdb* that corresponds to this information. You should be able to read its relations. Look at their names and schemas, and their foreign key constraints. These are some specifics about the data:

   - Productions. Correspond to all types of productions. Use the field *productiontype* to determine the type of production (movies, tvSeries, tvSpecial, tvEpisodes, etc).
   - Persons. People who have been involved making movies. Referenced by Roles and Crew.
   - Roles. This relation contains the most important actor/actresses of each movie (e.g. *major roles*). See Persons.
   - Crew. contains the 'director', 'writer', 'cinematographer', etc of each production.
   - Episodes. A production type 'tvEpisode' is linked to a placeholder for the entire series. For example, the tvSeries called 'Detectorists' has the id *tt4082744*. However, each of its episodes has a different tuple in productions, each of type 'tvEpisode' (e.g. its first 3 episodes are *tt4088818, tt4095606, and tt4095612*). The relation Episodes links the episodes to their corresponding 'tvSeries' entry in productions.
   - Ratings. The rating of productions. Note that a tvSeries has an "overall" rating and each episode has its own rating.
   - Genres. The genres of each production.

B. To connect to the database:

   - You need to login to one of the Linux computers in the faculty. For example: linux.csc.uvic.ca
   - To connect to the database you should use: host *studentdb.csc.uvic.ca*, database name *imdb*, your username and your password has been made available to you via a git repo in gitlab (see `http://gitlab.csc.uvic.ca/`). You will find a file called account.txt. For example, your database user id might be user120
   - Connect to the DBMS using psql using the following command (replace user120 with your db userid):

     ```
     psql -h studentdb.csc.uvic.ca -U user120 imdb
     ```

   - In psql, you can:
     - Change the password. **DO NOT REUSE a password**. Passwords in postgresql are not secure, therefore do not use a sensitive one.
     - See the documentation of `.pgpass` to make your life a bit easier.
     - List the relations using `\d`
     - List the schema of a relation using `\d relationName`
     - You can execute a query inside a file using `\i filename`.
     - You can output the results of a query using `\o filename`. This file is saved in the computer where you are running psql from. You stop such output using `\o`
     - Read the manual for psql for more information.
   - For this assignment, use only relations that are type *relation* (output of `\d`)

Your task, should you choose to accept it

Part I: queries

Most queries can be rewritten in more than one way. Sometimes the DBMS will evaluate both queries with exactly the same evaluation path, sometimes it will not. You need to answer the following question:

How many productions have the same person as a director and in a major role?

1. First, write a query that answers this question without cross-products or joins. Hint: use the operator **IN** in the selection.
2. Second, write this query again, using a join.
3. For each query:
   - Write the relational algebra expressions that correspond to each of both queries.
   - Using explain, draw an evaluation plan tree that explains how the DBMS answers each query.
4. Which query would you use to answer the question and why? Use the evaluation plans to support your answer.

By the way, the answer to the question is 3.

What to submit (see submission section for details):

- 2 queries
- 1 document with your relational algebra and your answers to the rest of the questions.

Part II: cost estimation

Understanding how the cost of a query is estimated will make you aware of the challenges of processing big data.

Before the DBMS computes a query it needs to estimate the number of tuples that will be returned. For a select of one table with a WHERE clause, this is done by computing the probability that any tuple t satisfies this clause. This probability is known as the selectivity of the clause. The selectivity of the clause is then multiplied by the number of tuples in the relation and the result is the expected number of tuples that the query returns. For example, given the query:

```
select * from productions where year = 2010;
```

`EXPLAIN` returns the following information:

```
# explain  select  * from productions where year = 2010;
                                      QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..153899.96 rows=212374 width=72)
   Workers Planned: 2
   ->  Parallel Seq Scan on productions  (cost=0.00..131662.56 rows=88489 width=72)
         Filter: (year = 2010)
(4 rows)
```

Note the number of rows: 212374. This number is computed by multiplying the selectivity of the clause (year = 2010) by the number of tuples in the relation. The selectivity of the clause is 0.0423667, and there are 7.6947e+06 tuples in this relation (this information is collected in a relation called pg_class). The result of multiplying these values is 212,374 (rounded to closest integer, the number of rows to return).

To understand how to compute the selectivity of a clause you have to read Postgresql's manual, specifically section 68.1: `https://www.postgresql.org/docs/10/row-estimation-examples.html`
As of version 10, postgresql uses 100 buckets, not 10 (the documentation is incorrect); also, postgresql stores 100 most frequent values, not 10. Everything else is correct (as far as I can tell).

When a query is composed of more than one comparison, the selectivities of each comparison predicate are considered mutually exclusive (even if they are not). Thus:

$$selectivity(A \wedge B) = selectivity(A) * selectivity(B)$$

Given the following query:

```
select * from productions
where year > 1994 and year <= 1996 and productiontype = 'short';
```

1 Using EXPLAIN, record the number of tuples that postgres estimates each query will return and the actual number of tuples returned.

2 Using **only the information in pg_stats and pg_class** compute the selectivity of the WHERE clause of this query.

3 using this selectivity, compute the expected number of matching tuples of this query.

Show all your work.

My calculated result is 126377.775750797, which is exactly the same the DBMS returns. This should be your target.

What to submit (see submission section for details):

- 1 document with your calculations and final result.

Part III: percentiles

Big data analysis usually requires us to group values into percentiles and study correlations between different factors. For this question you will study the correlation between number of votes and the rating of a movie.

For the purpose of this part, we are only concerned with movies that have at least one vote (productions of type 'movie', there are 259,661 in our database)[1].

We are going to divide the set of these movies into subsets based on percentiles of their number of votes. These sets will be: all movies (0th percentile), the top 75% (25th percentile), the top 50% (50th percentile), the top 25% (75th percentile), the top 5% (95th percentile) and the top 1% (99th percentile). In other words, if we order the movies based on their number of votes, descending: the first set will be all tuples, the second the top 75% tuples, the next the top 50% etc. For your reference, these are the number of movies in each subset:

[1]The minimum number of votes for any production is 5.

```
| Percentile |   count |
|----------- |--------+
|          0 | 259661 |
|         25 | 198122 |
|         50 | 130616 |
|         75 |  65004 |
|         95 |  12985 |
|         99 |   2597 |
```

Write a single SQL query that:

1. For each of these subsets, compute:

   - Count (number of movies in the subset)
   - Minimum number of votes
   - Median number of votes
   - Average number of votes
   - Median of the average ratings
   - Average of the average rating
   - Pearson correlation between the average rating and the number of votes of a movie (use the `corr` function)

   Your result should have the following schema and the following rows (below I am only showing all attributes in the first tuple, your result should include all attributes).

   ```
   |  count | minimumvotes | medianvotes |         averagevotes | medianavgratings |         avgavgratings |               corr |
   |--------+-------------+------------+----------------------+------------------+----------------------+--------------------|
   | 259661 |            5 |          50 | 3402.0517944550779670 |              6.3 | 3402.0517944550779670 | 0.0706852428617908 |
   | 198122 | ...
   | 130616 | ...
   |  65004 | ...
   |  12985 | ...
   |   2597 | ...
   ```

2. What conclusion do you draw from the answer to this question?

Hints:

1. The median of a set of values is its 50th percentile.

2. To compute the percentile of a set of values you can use the function `percentile_cont` (see Postgresql documentation). For example:

   ```
   with
       r(a,b) as (values (1,10),(2,20),(3,30),(4,40),(5,50),(6,60),(7,70),(8,80),(200,100))
       select avg(a),percentile_cont(0.5) as median within group (order by a) from r;

   |                  avg | median |
   |---------------------+--------|
   | 26.2222222222222222 |      5 |

   with
       r(a,b) as (values (1,10),(2,20),(3,30),(4,40),(5,50),(6,60),(7,70),(8,80),(200,100))
   ```

```
    select unnest(percentile_cont(array[0, 0.25, 0.5, 0.75, 1.00])
         within group (order by a)) as perc from r;
```

```
| perc |
|------|
|    1 |
|    3 |
|    5 |
|    7 |
|  200 |
```

3. Find the number of votes in each percentile and do a theta join of the movies with the same number of votes or larger (for each of these percentiles). Then aggregate.

What to submit (see submission section for details):

- 1 query
- 1 document with the result of your query and your conclusions

Part IV: JSON

JSON data is pervasive. This is one of the reasons that the SQL standard has support for JSON. One of the advantages of using a relational DBMS is that we can transcode the JSON data into relational data and then use relational operators to do computation, and, optionally, generate JSON for output.

The Nobel Prize committee publishes a dataset of Nobel recipients. Its structure is fairly simple. It is composed of a array of records like the one below. Note that the field `laureates` is an array of one or more subrecords with the name and info of each co-recipient.

```
{
  "year": "2021",
  "category": "chemistry",
  "laureates": [
    {
      "id": "1002",
      "firstname": "Benjamin",
      "surname": "List",
      "motivation": "\"for the development of asymmetric organocatalysis\"",
      "share": "2"
    },
    {
      "id": "1003",
      "firstname": "David",
      "surname": "MacMillan",
      "motivation": "\"for the development of asymmetric organocatalysis\"",
      "share": "2"
    }
  ]
}
```

I have created a database called nobel. It contains one relation called **prizes**. This relation has one attribute only (`tuple`) which is of type json.

Write a query that lists the recipients that have received more than one prize. Your result should be a set of tuples where each tuple has one attribute only (in json format), identical to the one below. Note that in each tuple the third JSON field, awards, is an array; this array contains two fields: year, and category (ordered by year). They represent the awards won by each person.

```
| r                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------|
| {"surname":"Curie","firstname":"Marie","awards":[{"year":"1903","category":"physics"}, {"year":"1911","category":"chemistry"}]} |
| {"surname":"Pauling","firstname":"Linus","awards":[{"year":"1954","category":"chemistry"}, {"year":"1962","category":"peace"}]} |
```

Hints:

- These are Postgresql's JSON functions :
  `https://www.postgresql.org/docs/10/functions-json.html`
- This tutorial introduces the basis of Postgresql's JSON features:
  `https://www.postgresqltutorial.com/postgresql-json/`
- You will find the following functions useful:
  - To convert a JSON array to a set of tuples (with attributes type JSON) use `json_array_elements`. Try this query:

    ```
    select tuple->'year',tuple->'category',
           json_array_elements(tuple->'laureates')
    from prizes limit 5;
    ```
  - To convert a relational tuple to a JSON record use `row_to_json`
  - To convert an list of JSON records to a JSON array use `json_agg`
- For your query: convert the JSON data to relational first, then find the persons in the result; finally convert back to JSON.

What to submit (see submission section for details):

- 1 query

Part V: CSV files. Only 500 level students

This part should be completed by 500 level students only.

Find a dataset in CSV format that has at least 1000 tuples. It should not be too large (less than 10,000 tuples). There are literally thousands of datasets in this format. Kaggle is a good source: `https://www.kaggle.com/datasets`.

I have created a database for you. Its name is `db_<number>` where `<number>` is the same as in your userid. For example, if you userid is `user020` then your personal database is called `db_020` In your personal database:

- Design and create a database to store this dataset. It can include one or more relations.

- Load this dataset into the database. There are many ways to do this. One is to write a small python program to do it. Run this program from linux.csc.uvic.ca.

Identify a question that can be answered with this data. Answer it in a single SQL query. The question should have some properties:

- The question should be relevant to the dataset.
- It should be about the general characteristics of the entire dataset (not a single entity in this dataset)
- It should contain the following operations, each at least once: aggregation, join (of any type, including cross product).

Part I and Part IV are examples of such datasets and queries.

What to submit (see submission section for details):

- 1 query
- 1 document describing the schema of your database and the processed followed to load this data. Include any scripts/programs you used in this document.

Other information and restrictions

1. The queries you have to write are not trivial. I recommend that you build them incrementally. Break the problem into smaller ones, and use common table expressions (`WITH`). If you have never used them, read about them first. They are going to make this assignment much, much easier and enjoyable. All my solution queries use `WITH`.
2. **You cannot use LIMIT in any of your queries**
3. In postgresql, usernames are case sensitive.
4. Once you are logged in, you need to type the semicolon at the end or every query. If you don't, psql will not process your query and way for more input. If your prompt is different from `<username>=`, then your query is not yet complete.
5. You can always use `Ctrl-c` to stop your query (either you are in the middle of typing it, or it is being processed.
6. To exist psql type `Ctrl-d`. It is bad practice to simply kill the terminal, since the dbms will still hold resources for a while until it can detect that the client is dead.

What to submit

Via brightspace download the template for submitting your answers. This template is a zip file that will create the files that you need to populate with your answers. Specifically:

- Every query should be placed in its corresponding .sql file. We will test your queries by running them against the database. You can easily test your query as follows. Given a file called `query.sql`:

```
cat query.sql | psql -h studentdb.csc.uvic.ca <rest of parameters>
```

- **You must submit your answers to non-query questions in PDFs or plain text files. Any other file formats will not be graded.**
- Compress the files into a zip file (similar to the one you downloaded) and upload it to brightspace in its corresponding assignment entry.
- Submit **a single** zip file.