Inf553 Project: Evaluation and Performance

a.k.a. database tuning

Pierre Bourhis, Benoît Groz, Ioana Manolescu

October 2019

- You will first write the queries below in SQL, using the schema of the database that was proposed in the first part of the project. The queries should be evaluated on the data given in Phase 1.
- Then, measure and analyze the performance of your queries.
- Then, select three of the queries and propose for each of them, some measures which are likely to improve their evaluation performance. Apply these measures, re-evaluate the queries, and analyze whether the desired impact was achieved.

1 Queries

Below we describe the queries you need to write. For each query, we also show you a few results, to clarify what the results should look like.

1. For each country, return the country name and the names of all artists who were born in this country. Return the results ordered first by the country name, then by the artist name.

```
    name     |         name
-------------+-----------------------
 Afghanistan | 143Band
 Afghanistan | Abdul Katrim Herati
 Afghanistan | Abdul Mazari
...
```

2. Find the countries with the largest number of artists, and return the name of these countries along with the names of their artists.

```
     name      |             name
---------------+------------------------------
 United States | Willie "Drive 'Em Down" Hall
 United States | Abraham Holzmann
 United States | Sean Brennan
...
```

3. Find the ids of artists who have released an album in a country whose name starts with an A.

```
                                          id
-------------------------------------------------------------------------------------------
 1
 4
 9
 ...
```

4. For each country id, return the number of different releases in this country. Order the results in descending order according to the number of releases.

```
 id  |   cc
-----+--------
 222 | 408526
 221 | 217909
  81 | 161777
 ...
```

5. For every artist and release such that the artist is a main contributor to this release, return the release ID and the artist ID.

```
 release | artist
---------+--------
       1 |     60
       2 |     60
       3 |     60
...
```

6. Return the triples (id_1, id_2, c) such that id_1, id_2 are ids of different artists that have collaborated for some releases, and c is the count of releases in which they have collaborated.

```
 artist  | artist  | count
---------+---------+-------
       1 |     101 |     2
       1 |     938 |     1
       1 |    1021 |     1
...
```

7. Return the pairs (country id, release id) such that the release has been made in that country and the release has at least two tracks.

```
 country | release
---------+---------
      81 |       1
      81 |       2
      81 |       3
...
```

8. Return the pairs (country id, release id) such that a release with this id has been made in this country, yet the country is not the first (in temporal order) in which a release of this id has been made.

```
 release | country
---------+---------
  459992 |     221
     760 |     202
 1968922 |     224
...
```

9. For each country, and each individual artist x (a Person) from that country, the number of artists born before x in that country and the number of artists born before x globally.

```
  country  |             artist              |  nb   | nb_global
-----------+---------------------------------+-------+-----------
 China     | Sima Xiang                      |     1 |         2
 Italy     | Gaius Vale                      |     1 |         3
 Europe    | Marcus Aur                      |     1 |         4
...
```

2 Query profiling

Measuring execution time In this subsection, the goal is to measure the performance of the query over the data.

First, you should measure the execution time. For that, start by executing each query of the previous section **five times**, measuring the time taken by each execution. Record the times, and add them to your report in some readable format (i.e., tables, figures, diagrams). If you see significant differences among the running times of the same query, explain why.

To measure query execution times in **milliseconds (ms)**, you can use for instance the `\timing` command in the interactive `psql` client.

Analyze the query plan

For each query, provide its **execution** plan and answer the following questions:

- Which part of the plan is the most costly to evaluate?
- Can the performance of the query be improved, and if yes, how?

You should compare and discuss the difference between the time estimated by the optimizer and the real time of the evaluation.

Postgres provides a command, `explain`, with two variants, for analyzing performance:

> `explain query` gives the plan that the system would use if asked to evaluate your query (but does not evaluate it). This plan is a *tree* made of *physical operators*. It is chosen based on *statistics* which are stored in the system before running any query.
>
> `explain analyze query` actually evaluates the query, and returns a description of the plan, together with some statistics about the data. These statistics are partially gathered while the plan runs, that is, they are not available to `explain query`.
>
> `explain (analyze, bufers) query` is a slight variation on the above. It also runs the query and displays information about the memory buffer management incurred during the execution.

More documentation about the `explain` command is available here:
`https://www.postgresql.org/docs/current/using-explain.html`

Statistics

PostgreSQL keeps statistic information about the data, in particular the following system tables:

- pg_class: you can find the attributes of this table in the following `https://docs.postgresql.fr/10/catalog-pg-class.html`
- pg_stats: you can find some statistics over the data stored in the table. The attributes can found in `https://www.postgresql.org/docs/10/view-pg-stats.html`.

These tables can be queried in SQL, lika any other table.
You can reevaluate the statistics by running the command vaccum analyze *relationname* (`https://www.postgresql.org/docs/9.3/sql-vacuum.html`, `https://www.postgresql.org/docs/9.3/sql-analyze.html`) or only analyze. The statistics could change each time that you run analyze as it is based on a sample.

Statistics can be improved (made more precise) as follows:

- You could change the target value x which sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram, by changing the configuration variable *default_statistics_target* for all the tables or for each column c of the table t, with the following command: ALTER TABLE t ALTER Column c SET STATISTICS x;.
- you can also update the values directly. For example, you can set the number of different values x of Table t and Column c by using the following command: ALTER TABLE t ALTER c name SET (n_distinct = x)

3 Performance improvement

Query performance can be improved in multiple ways : reformulating the query, creating indexes, materializing and reusing results, using parallel execution, increasing the size of the memory buffers, etc. In this project step, your task is to optimize queries by creating indexes and/or reformulating the query. We will not investigate in the other optimization techniques such as tuning hardware or database parameters.

Reformulation SQL is mostly *declarative*, meaning that the user specifies the information need, not how to evaluate it, and the system choses the evaluation method (or plan). In practice, however, this holds for queries with *relatively simple structure*, that is: (i) one block of select-project-join, possibly with grouping and aggregation, or (ii) more complex queries that Postgres is capable of rewriting into the structure (i). For the other queries, the optimizer starts by making a few "simple" decisions. Here is an example:

> If the query is $q_1 \cup q_2$, Postgres may decide to optimize q_1 separately, then optimize q_2 separately, then take the best plan for q_1 and the best plan for q_2, and union them.
> If q_1 and q_2 had an expensive, common sub-query q_3, it may have been more efficient to reuse the computation of q_3, i.e. execute it only once; the optimizer of Postgres will not see this opportunity.

Other similar (quite advanced) examples exist. For the above reasons, in practice, the way the query is formulated may have an impact on performance, especially outside of the "safe" spaces (i) and (ii). Here are some hints toward helping the system optimize the query.

> Try to avoid the use of functions when it is possible to express the query without them; they are more difficult to optimize and usually more costly to evaluate.
> Try to avoid redundant parts of the query; some form of query minimization (identifying and removing redundant query parts) may be available but it is better not to count on it as usually not all redundant fragments are identified.
> When checking that a value does not belongs to a relation, try to use `not exists` and `not in`. The engine has some particular operators to evaluate them efficiently (i.e., anti-join, semi-join operators) and using such syntax hints to the engine that it should use them.
> Try to avoid nested subqueries.
> Use `distinct` only if strictly necessary; the elimination of duplicates is a costly operation.
> You can use subqueries by using with ... as and creation of views as presented in the previous practice

Be careful: When reformulating the query, it is important not to change the semantics of the query with respect to the current schema. Sometimes a reformulated query gives the same answer on the database you are using, however, it may alter the query answer on another database. Query reformulation should preserve the query semantics (that is, give the same results on any database conforming to the schema and its constraints)!

Index In SQL, it is possible to create index on the demand. Moreover, Postgres creates automatically an index for the primary key of a relation (or any unique key actually: the indexes actually enable postgresql enforce those key constraints).

The command `create index myindex on test (a);` creates an index named myindex on the attribute a of the relation test.
It is also possible to drop an index via the commande `drop index`. You can find more information at: `https://www.postgresql.org/docs/9.6/static/indexes.html`.

Task

Pick 3 queries among the ones above for which you see avenues for improving performance.
For each query

- Show the original query, and the reformulated query (if you used query reformulation);
- Show the new indexes (if you created any);
- Provide the new execution plan;
- Measure the performance (as you have done before) and report the results.

You will endeavor to adopt varied approaches : not all queries should be improved through query rewriting, for instance.

4 Deliverables

Submit a report in PDF format (with name inf553_optim_firstname_lastname.pdf), covering all the elements that were asked above. Provide complete explanations, and justification of your choices.