Athens University of Economics and Business
Department: Computer Science
Course: Databases
Academic Year: 2019-2020
Instructor: Vasilis Vassalos

6th Assignment: Query Optimization and Triggers

Goal:
This assignment consists of two parts. In the first part we will add triggers to ensure that specific properties of the database are retained while in the latter we will improve query execution time by adding indexes.

Inputs:
Airbnb database in 5th assignment state.

Desired Outputs:

Part A: Triggers

- Write a trigger function which updates the host_listings_count field of the Host table in corresponding entry/entries, when a record is added/deleted in the Listing table. Check that your function works properly by adding/deleting an entry in the Listing table and see if the field (could have multiple entries) host_listings_count of this host has been updated correctly.
- Implement your own trigger function for some table/tables in your database.

Place the trigger code in a file named triggers.sql. For your own trigger function add a brief description of what it does.

EXPLAIN ANALYZE- Example

As SQL is a declarative language, a query describes the result we want, regardless of which stepsthe database must follow to reach this result. Therefore, a DataBase Management system can execute a query in N different ways and by choosing from a set of algorithms for each operator. The EXPLAIN ANALYZE command allows us to see the execution plan followed by the DBMS to execute a query. Suppose we run EXPLAIN ANALYZE command in the following query:

EXPLAIN ANALYZE SELECT Listing.city, Count(Listing.id) AS listings
FROM Listing GROUP BY (Listing.city) ORDER BY listings;

The query plan will be the following:

```
QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------
"Sort                  (cost=2992.89..2993.17          rows=109       width=15)                                (actual
time=17.315..17.323 rows=109 loops=1)"
"     Sort Key:  (count(id))"
"     Sort Method: quicksort  Memory: 32kB"
"      -> HashAggregate       (cost=2988.11..2989.20 rows=109 width=15)                                (actual
time=17.226..17.246  rows=109  loops=1)"
"            Group Key: city"
"              -> Seq Scan on listing              (cost=0.00..2930.41       rows=11541                          width=11)
(actual time=0.009..5.396 rows=11541 loops=1)"
"Planning Time: 1.249 ms"
"Execution Time: 17.391 ms"
----------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------
```

The query plan is depicted as a tree structure with the last nodes depicting the leaves of the tree, which return records from the tables. So let's start reading the plan bottom up. Each node, except the root of the tree, starts with -> and represents the execution of an action. For some actions there may be additional information such as the field in which the classification was made (e.g. Sort key: count(id))). Such information, with the exception of the root node, does not start with -> and this helps us to separate them from the action nodes.

Let's look at the actions performed in this plan one by one.

- Seq scan on "Listing": a table scan on Listing table.
- HashAggregate: This algorithm uses a hash table to group and aggregate records. This node is executed on the records that resulted from the table scan on the Listing table.
- Sort: Sorting of the result obtained from the HashAggregate node.

We notice that in the execution plan some measurements are mentioned. For example the cost results from a cost function used by its Postgres query optimizer to estimate the execution cost of each node, and is based on statistics of tables, such as the number of records or the number of blocks on the disk. Actual time parameter is the actual time it took a node to execute, while rows refer to lines accessed by a node. Finally, the planning and execution time refer to the query plan's optimization and execution time.

Part B: Indexes

- Suppose we have the following query:

  SELECT "Host".host_id, COUNT(*) FROM "Listing", "Host" WHERE "Host".host_id="Listing".host_id GROUP BY "Host".host_id;

  Use the EXPLAIN ANALYZE command to print the plan and time execution of the query. Then add a suitable index that will speed up the execution time of the above query and run the EXPLAIN ANALYZE command again to see the new execution time.
- Suppose we have the following query:

  SELECT id, price FROM "Listing", "Price" WHERE guests_included > 5 AND price > 40;

  Use the EXPLAIN ANALYZE command to print the plan and execution time of the query. Then experiment with adding indexes to one or more fields and check if the execution time improves. Would adding an index in the price field help? Justify.
- For each of the aggregate queries in 5th assignment, add an appropriate index that will speed up its execution. Run EXPLAIN ANALYZE command as in the previous queries, to find out if the execution time is indeed reduced. Explain why you chose this index. If for a query that you can not find an index to speed it up, write down which indexes you tried to add and explain why you think that this query cannot be improved by adding an index.

Place the execution plans for each query before and after adding indexes to a file named **plans.txt.** Before each plan write the EXPLAIN ANALYZE command from which it resulted. For each query to which you added an index, write a comment in the form:

/* Query 1: w/out index: 110 ms; w/index: 80 ms */

In the same comment add your justification for the improvement or not in the execution time of the query, that results from adding an index.

Place the commands you ran to create all the indexes in a file named **query_indexes.sql**

Tools you need:

- Postgres psql or/ and PgAdmin

Hints:

- Run the VACUUM ANALYZE command on all the tables in your database before you start printing any execution plans.
- Run **set enable_seqscan=off;** to disable table scan when testing a candidate index.
- Especially for triggers, before you start performing them, it is good to make one copy of the tables that are affected by the trigger function. If something goes wrong and it's difficult to return to the previous state that your table was, you can start working with its copy. In case you make such copies, however, when you have completed the Task, please delete all copies.

Useful Links:

- VACUUM command: https://www.postgresql.org/docs/9.6/sql-vacuum.html
- EXPLAIN command: https://www.postgresql.org/docs/9.6/using-explain.html
- Multicolumn index: https://www.postgresql.org/docs/9.6/indexes-multicolumn.html
- Partial index: https://www.postgresql.org/docs/9.6/indexes-partial.html
- Triggers: http://www.postgresqltutorial.com/creating-first-trigger-postgresql/