**6.1 Return the number of transformers with a given primary voltage by locality:**

The main objective of using indexes is to reduce the query answering time.

In the first part we want to optimize the query that returns the number of transformers when is given a primary voltage grouping by locality. To facilitate our tests, let's assume that a primary voltage is 333.1V (pv = 333.1). Firstly, let’s use the command “SET enable seq\_scan = OFF;” to disable sequential scan when running a query. Then, we run:

‘’EXPLAIN ANALYZE SELECT locality, COUNT(\*) FROM transformer NATURAL JOIN substation WHERE pv = 333.1 GROUP BY locality;’’ . EXPLAIN ANALYZE is a Postgres command that accepts and executes the query and returns data regarding the query plan where we can see the approach that the planner took to execute the query and the planning and execution time as well. As we want to filter localities by the primary voltage of the transformer, the best choice here is to use an hash index. Analysing the given query, it would not make sense to use a b+tree index since we just want to filter by a line. An hash index will organise records into pages according to a hash function. This hash function will take as input the search key value (in this case, the primary voltage) and hashes it to know the container where the value is located. As we can see in the next two figures, there are differences in the cost, planning and execution time of the same query with and without index. It is not so noticeable the effect of the index since the given dataset is not so big as it should. Although, we can assist a decrease of the planning and execution time and the cost as well.

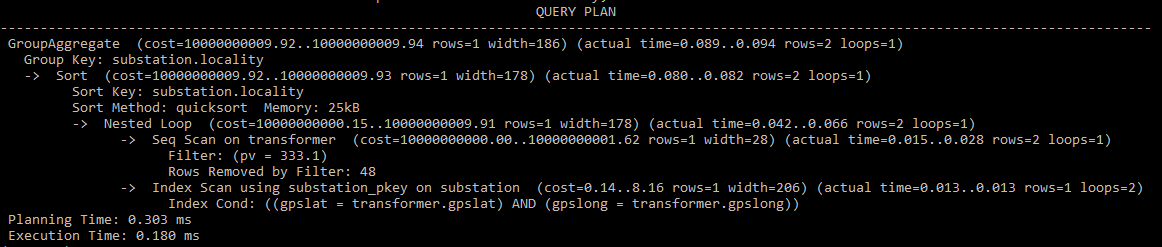


Figure 2: Query plan of the first query without using any index

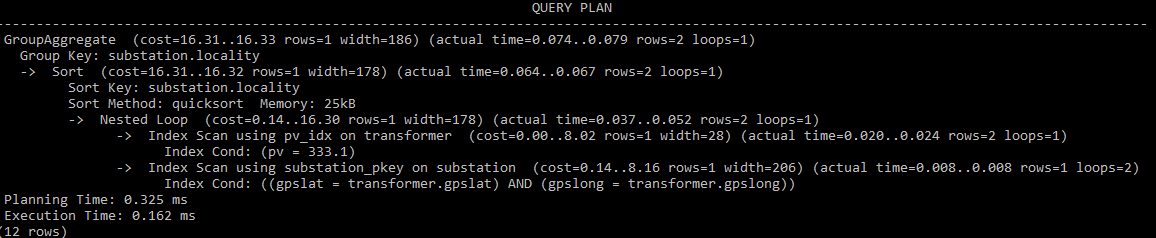
****

Figure 2: Query plan of the first query using ‘CREATE INDEX pv\_idx ON transformer USING HASH(pv);’

**6.2 List all descriptions of line incidents that start with a given prefix within two points in time:**

In the second part we want to optimize the query that lists all descriptions of line incidents starting and ending in two points in time. Let's assume that we want every description of line incident occurring in 2020. Again, let’s use the command “SET enable seq\_scan = OFF;” to disable sequential scan when running a query. Then, we run: EXPLAIN ANALYZE SELECT id, description FROM incident WHERE instant BETWEEN '2020-01-01 00:00:00.000' AND '2020-12-31 00:00:00.000' AND description LIKE 'Fall and Burn';. In these case, the best choice is doing a composite index that contains the incident and the description. The B+Tree is a balanced tree that auto-adapts with inserts and deletes to keep balanced ensuring the same depth for all nodes. Index entries are ordered by the search key value kept in a hierarchical search structure. This type of index support range queries and composite indexes oppositely to the hash index, used in the previous question. As we can see in the following figure, we have a bitmap heap scan on incident and it filters less (236) rows than when scanning by the primary key of the incident, so we can conclude that this method is the most efficient to reduce the query answering time.

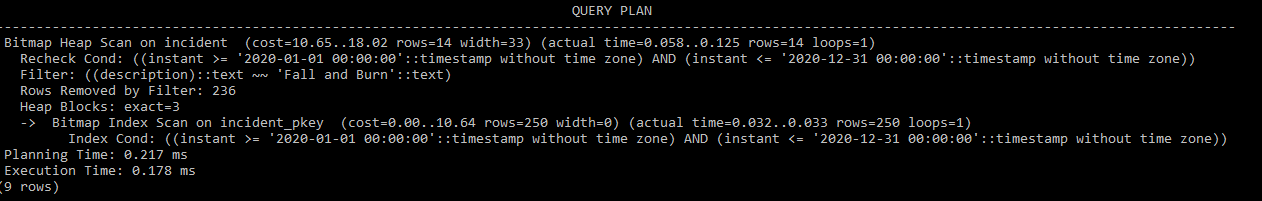
****

Figura 3: Query plan of the second query using ‘CREATE INDEX description\_of\_incidents\_idx ON incident USING BTREE(instant, description);’