

# Batch: B1 Roll No.: 16010122104

**Experiment / assignment / tutorial No. 7 Grade: AA / AB / BB / BC / CC / CD**

**Title: Implementing indexing and query processing**

**Objective:** To understand Query Processing and implement indexing to improve query execution plans

# Expected Outcome of Experiment:

CO 3: Use SQL for relational database creation , maintenance and query processing



# Books/ Journals/ Websites referred:

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press
2. [www.db-book.com](http://www.db-book.com/)
3. Korth, Slberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw Hill
4. Elmasri and Navathe,”Fundamentals of database Systems”, 4th Edition,PEARSON Education.

**Resources used:** PostgreSQL/MySQL

**Theory:**

A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

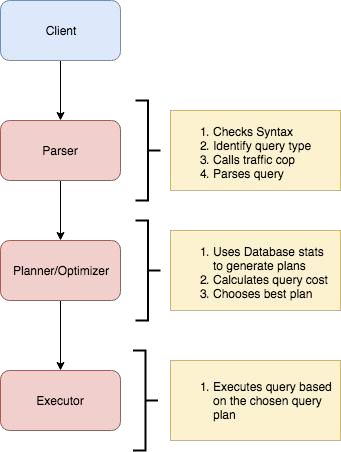
While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.

To add an index for a column or a set of columns, you use the CREATE INDEX statement as follows:



CREATE INDEX index\_name ON table\_name (column\_list)

Query life cycle



# Planner and Executor:

The planner receives a query tree from the rewriter and generates a (query) plan tree that can be processed by the executor most effectively.

The planner in Database is based on pure cost-based optimization -

# EXPLAIN command:

This command displays the execution plan that the PostgreSQL/MySQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc.

— and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

As in the other RDBMS, the EXPLAIN command in Database displays the plan tree itself. A specific example is shown below:-

Database: testdb=#

* 1. EXPLAIN SELECT \* FROM tbl\_a WHERE id < 300 ORDER BY data;
  2. QUERY PLAN

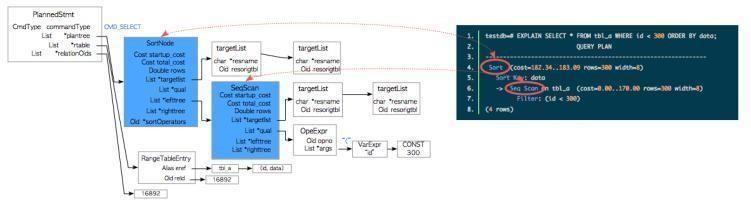
3. (cost=182.34..183.09 rows=300 width=8)

1. Sort Key: data
2. -> Seq Scan on tbl\_a (cost=0.00..170.00 rows=300 width=8)
3. Filter: (id < 300)
4. (4 rows)
5. Sort





# A simple plan tree and the relationship between the plan tree and the result of the EXPLAIN command in PostgreSQL.



**Nodes**

The first thing to understand is that each indented block with a preceeding “->” (along with the top line) is called a node. A node is a logical unit of work (a “step” if you will) with an associated cost and execution time. The costs and times presented at each node are cumulative and roll up all child nodes.

# Cost:

It is not the time but a concept designed to estimate the cost of an operation. The first number is start-up cost (cost to retrieve first record) and the second number is the cost incurred to process entire node (total cost from start to finish).

Cost is a combination of 5 work components used to estimate the work required: sequential fetch, non-sequential (random) fetch, processing of row, processing operator (function), and processing index entry.

**Rows** are the approximate number of rows returned when a specified operation is performed.

(In the case of select with where clause rows returned is Rows = cardinality of relation \* selectivity ) **Width** is an average size of one row in bytes**.**

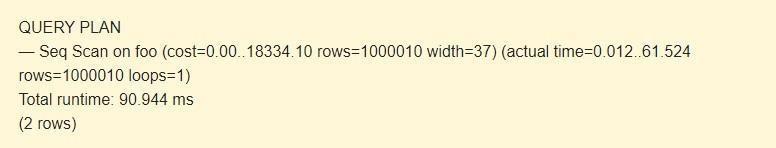
# Explain Analyze command:

The EXPLAIN ANALYZE option causes the statement to be actually executed, not only planned. Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality.

Ex: EXPLAIN (ANALYZE) SELECT \* FROM foo;







The command displays the following additional parameters:

* **actual time** is the actual time in milliseconds spent to get the first row and all rows, respectively.
* **rows** is the actual number of rows received with Seq Scan.
* **loops** is the number of times the Seq Scan operation had to be performed.
* **Total runtime** is the total time of query execution.

Query plans for select with where clause can be sequential scan, Index Scan, Index only Scan, Bitmap Index Scan etc.

Query plans for joins are Nested loop join, Hash join, Merge join etc.

# Implementation Screenshots :

**Comprehend how indexes improves the performance of query applied for your database . Demonstrate for the following types of query on your database**

# Simple select query: Creating table users, inserting a million values and selecting the table

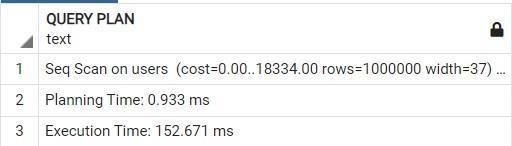
CREATE TABLE users(userID INTEGER, userName TEXT);

INSERT INTO users

SELECT i, md5(random()::TEXT) FROM generate\_series(1, 1000000) AS i;

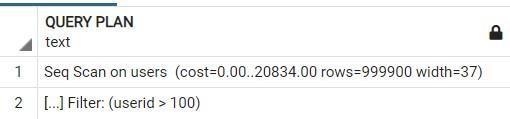
EXPLAIN SELECT \* FROM users;

EXPLAIN(ANALYSE) SELECT \* FROM users;

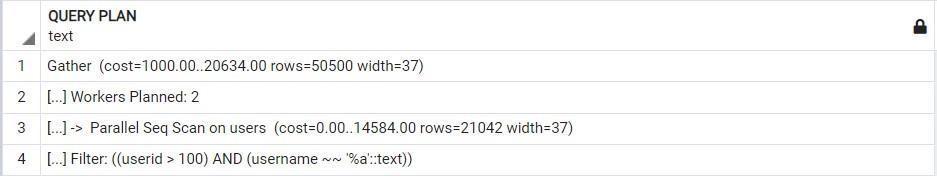




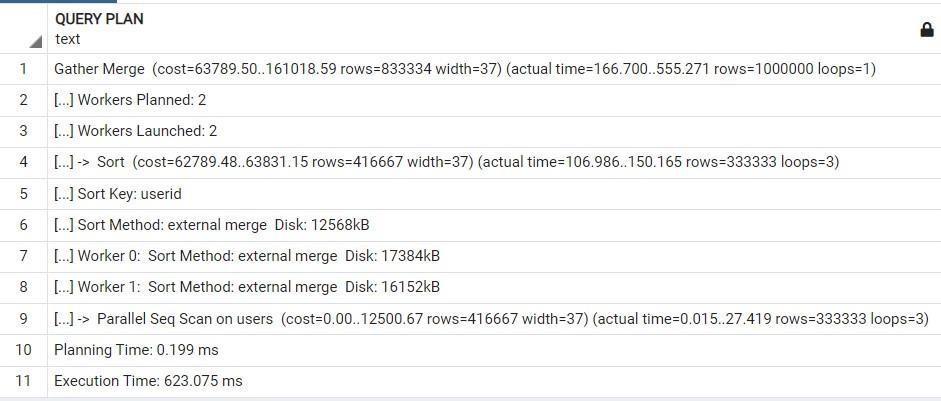
# Select query with where clause: Selecting users whose userID is greater than 100

EXPLAIN SELECT \* FROM users WHERE userID > 100;

EXPLAIN SELECT \* FROM users WHERE userID > 100 and userName LIKE '%a';



# Select query with order by query: Selecting all users by ordering on basis of UserID

EXPLAIN(ANALYSE) SELECT \* FROM users ORDER BY userID;

# Select query with JOIN: Selecting userID and flight name where the IDs of user and flight matches

CREATE TABLE users(userID INTEGER, userName TEXT);

CREATE TABLE flights(flightID INTEGER, flightName TEXT); INSERT INTO users

SELECT i, md5(random()::TEXT) FROM generate\_series(1, 1000000) AS i;

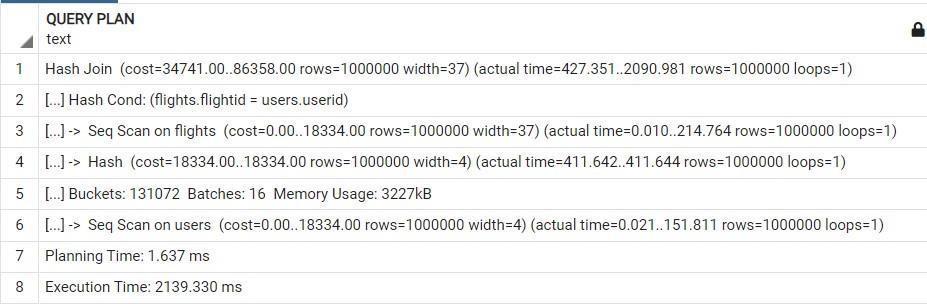




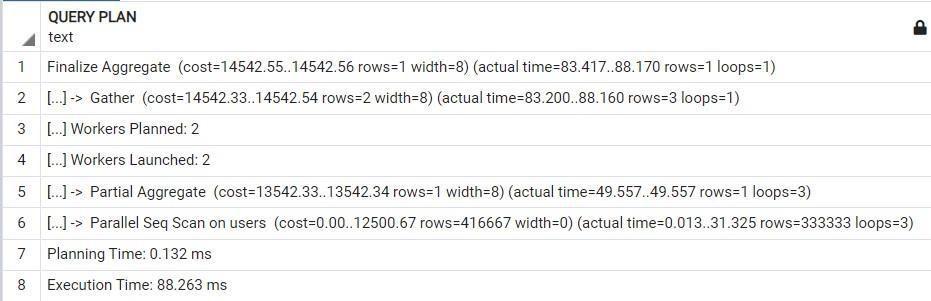
INSERT INTO flights

SELECT i, md5(random()::TEXT) FROM generate\_series(1, 1000000) AS i;

EXPLAIN(ANALYSE) SELECT userID, flightName FROM users INNER JOIN flights ON users.userID

= flights.flightID;

# Select query with aggregation : Selecting count of all the users

EXPLAIN(ANALYSE) SELECT COUNT(\*) FROM users;

# Adding Indexing on all the above queries

CREATE INDEX ON users(userID);

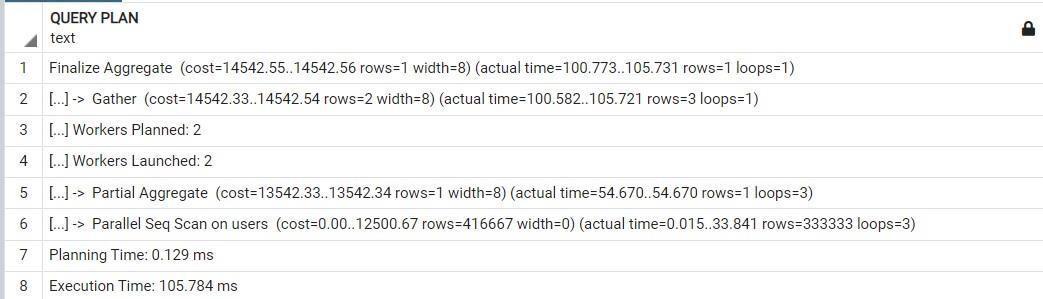
EXPLAIN(ANALYSE) SELECT \* FROM users WHERE userID > 500;



EXPLAIN(ANALYSE) SELECT \* FROM users ORDER BY userID;

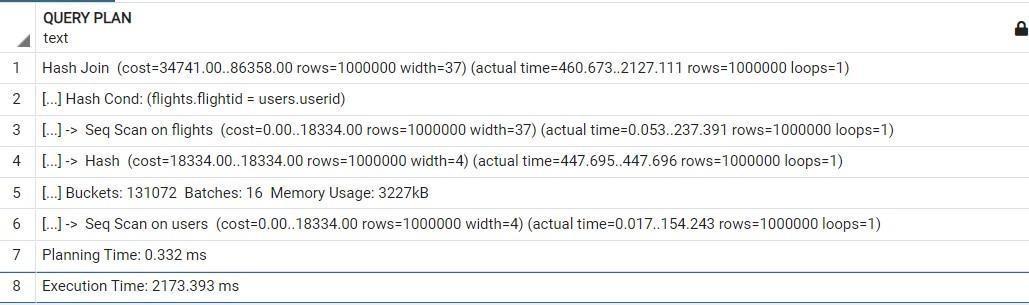




EXPLAIN(ANALYSE) SELECT COUNT(\*) FROM users;

EXPLAIN(ANALYSE) SELECT userID, flightName FROM users INNER JOIN flights ON users.userID

= flights.flightID;



# 

# Post Lab Question:

**1. Illustrate with an example Heuristic based query optimization with suitable example**

Heuristic based optimization uses rule-based optimization approaches for query optimization. These algorithms have polynomial time and space complexity, which is lower than the exponential complexity of exhaustive search-based algorithms. However, these algorithms do not necessarily produce the best query plan.

Some of the common heuristic rules are –

* Perform select and project operations before join operations. This is done by moving the select and project operations down the query tree. This reduces the number of tuples available for join.
* Perform the most restrictive select/project operations at first before the other operation
* Avoid cross-product operation since they result in very large-sized intermediate tables



# Conclusion:

Through this experiment we have successfully understood the concept and implementation of indexing and query processing.