**K. J. Somaiya College of Engineering, Mumbai-77**
**(Autonomous College Affiliated to University of Mumbai)**

**Batch: A3 Roll No.: 1911034**

**Experiment / assignment / tutorial No. 9**

**Grade: AA / AB / BB / BC / CC / CD /DD**

# 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
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:

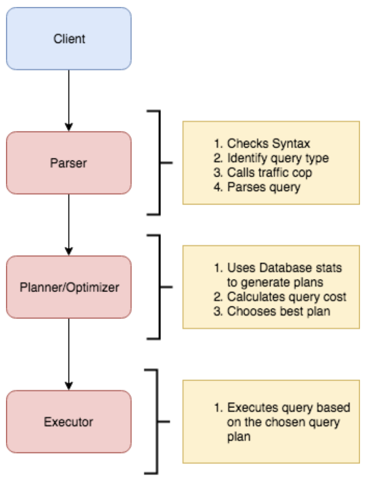
A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

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 -

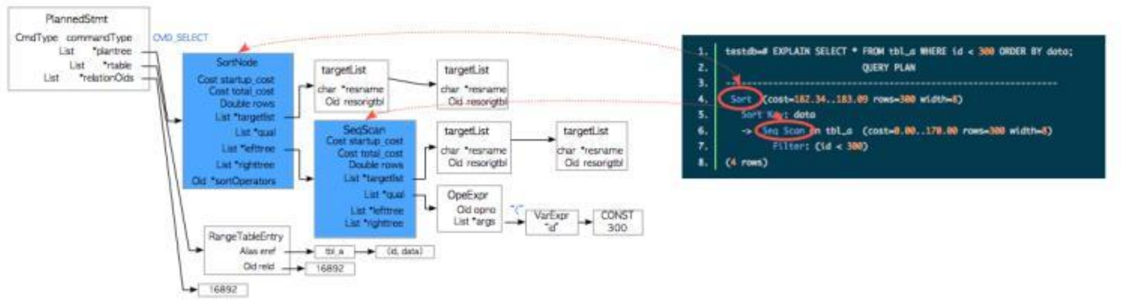
**EXPLAIN command:**

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. ---------------------------------------------------------------
4. Sort (cost=182.34..183.09 rows=300 width=8)
5. Sort Key: data
6. -> Seq Scan on tbl_a (cost=0.00..170.00 rows=300 width=8)
7. Filter: (id < 300)
8. (4 rows)

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



```
testdb=# EXPLAIN SELECT * FROM tbl_a WHERE id < 300 ORDER BY data;
                          QUERY PLAN
---------------------------------------------------------------
 Sort  (cost=182.34..183.09 rows=300 width=8)
   Sort Key: data
   ->  Seq Scan on tbl_a  (cost=0.00..170.00 rows=300 width=8)
         Filter: (id < 300)
(4 rows)
```

**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;

```
QUERY PLAN
— Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.012..61.524
rows=1000010 loops=1)
Total runtime: 90.944 ms
(2 rows)
```

The command displays the following additional parameters:

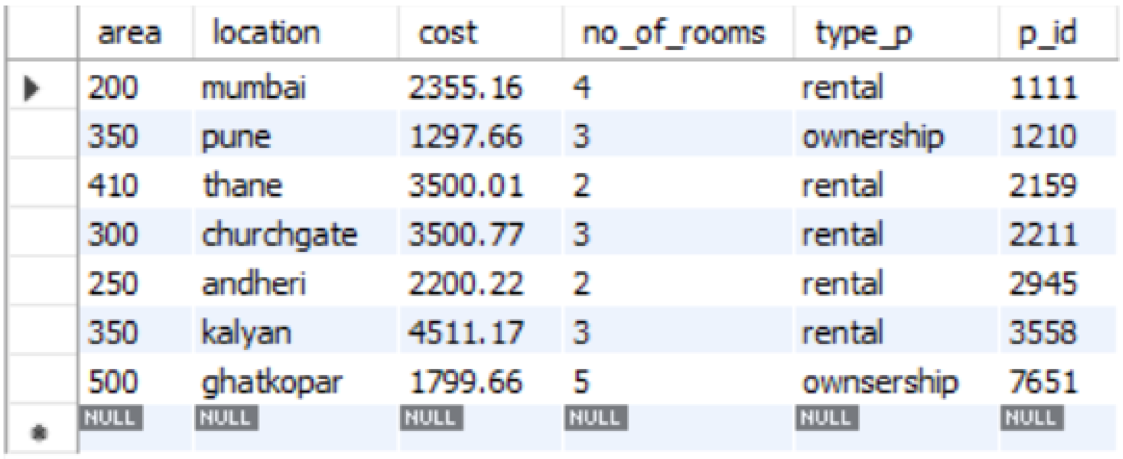
- **actual time** is the actual time in milliseconds spent to get the first row and all rows, respectively.
- **rows** is the actual number of rows received with Seq Scan.
- **loops** is the number of times the Seq Scan operation had to be performed.
- **Total runtime** is the total time of query execution.

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.

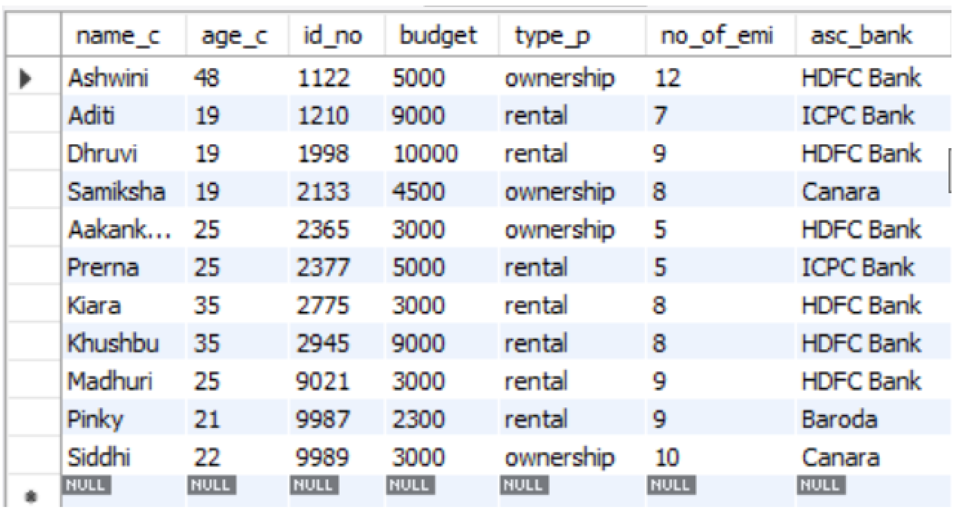
**Implementation Screenshots :**

**Property table:**



| | area | location | cost | no_of_rooms | type_p | p_id |
|---|---|---|---|---|---|---|
| ▶ | 200 | mumbai | 2355.16 | 4 | rental | 1111 |
| | 350 | pune | 1297.66 | 3 | ownership | 1210 |
| | 410 | thane | 3500.01 | 2 | rental | 2159 |
| | 300 | churchgate | 3500.77 | 3 | rental | 2211 |
| | 250 | andheri | 2200.22 | 2 | rental | 2945 |
| | 350 | kalyan | 4511.17 | 3 | rental | 3558 |
| | 500 | ghatkopar | 1799.66 | 5 | ownsership | 7651 |
| * | NULL | NULL | NULL | NULL | NULL | NULL |

**Customer table:**



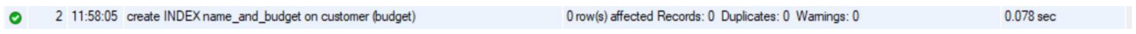
| name_c | age_c | id_no | budget | type_p | no_of_emi | asc_bank |
|---|---|---|---|---|---|---|
| Ashwini | 48 | 1122 | 5000 | ownership | 12 | HDFC Bank |
| Aditi | 19 | 1210 | 9000 | rental | 7 | ICPC Bank |
| Dhruvi | 19 | 1998 | 10000 | rental | 9 | HDFC Bank |
| Samiksha | 19 | 2133 | 4500 | ownership | 8 | Canara |
| Aakank... | 25 | 2365 | 3000 | ownership | 5 | HDFC Bank |
| Prerna | 25 | 2377 | 5000 | rental | 5 | ICPC Bank |
| Kiara | 35 | 2775 | 3000 | rental | 8 | HDFC Bank |
| Khushbu | 35 | 2945 | 9000 | rental | 8 | HDFC Bank |
| Madhuri | 25 | 9021 | 3000 | rental | 9 | HDFC Bank |
| Pinky | 21 | 9987 | 2300 | rental | 9 | Baroda |
| Siddhi | 22 | 9989 | 3000 | ownership | 10 | Canara |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

1. **Creating Index on 'Customer Table'**

   use propsys;

   create INDEX name_and_budget on customer (budget);

   **Output**



```
2  11:58:05  create INDEX name_and_budget on customer (budget)    0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0    0.078 sec
```

**Comprehend how indexes improves the performance of query applied for your database .**

Indexing makes columns faster to query by creating pointers to where data is stored within a database.

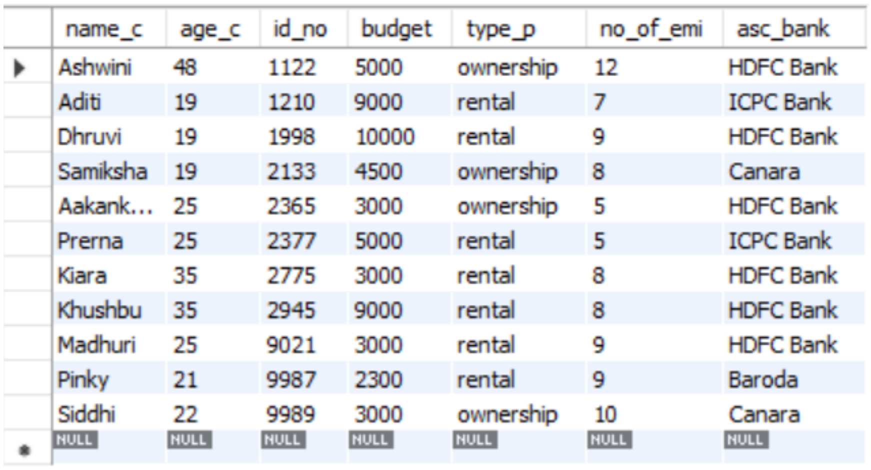
If we want to find a piece of information that is within a large database. To get this information out of the database the computer will look through every row until it finds it. If the data you are looking for is towards the very end, this query would take a long time to run.

If the table was ordered alphabetically, searching for a name could happen a lot faster because we could skip looking for the data in certain rows. If we wanted to search for "Zack" and we know the data is in alphabetical order we could jump down to halfway through the data to see if Zack comes before or after that row. We could then half the remaining rows and make the same comparison.

**Demonstrate for the following types of query on your database**

a. **Simple select query**

```
use propsys;
 select * from customer;
```



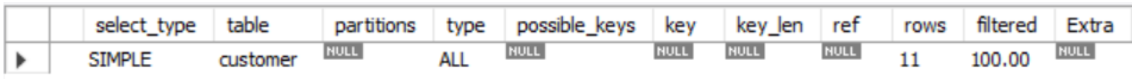
| name_c | age_c | id_no | budget | type_p | no_of_emi | asc_bank |
|---|---|---|---|---|---|---|
| Ashwini | 48 | 1122 | 5000 | ownership | 12 | HDFC Bank |
| Aditi | 19 | 1210 | 9000 | rental | 7 | ICPC Bank |
| Dhruvi | 19 | 1998 | 10000 | rental | 9 | HDFC Bank |
| Samiksha | 19 | 2133 | 4500 | ownership | 8 | Canara |
| Aakank... | 25 | 2365 | 3000 | ownership | 5 | HDFC Bank |
| Prerna | 25 | 2377 | 5000 | rental | 5 | ICPC Bank |
| Kiara | 35 | 2775 | 3000 | rental | 8 | HDFC Bank |
| Khushbu | 35 | 2945 | 9000 | rental | 8 | HDFC Bank |
| Madhuri | 25 | 9021 | 3000 | rental | 9 | HDFC Bank |
| Pinky | 21 | 9987 | 2300 | rental | 9 | Baroda |
| Siddhi | 22 | 9989 | 3000 | ownership | 10 | Canara |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

**Using explain command with simple select query**
**Code :**

```
use propsys;

EXPLAIN select * from customer;
```

**Output:**

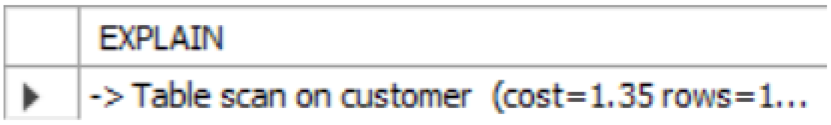


| select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|
| SIMPLE | customer | NULL | ALL | NULL | NULL | NULL | NULL | 11 | 100.00 | NULL |

**Using Explain Analyze with simple select query**
**Code:**

```
use propsys;
EXPLAIN analyze select * from customer;
```

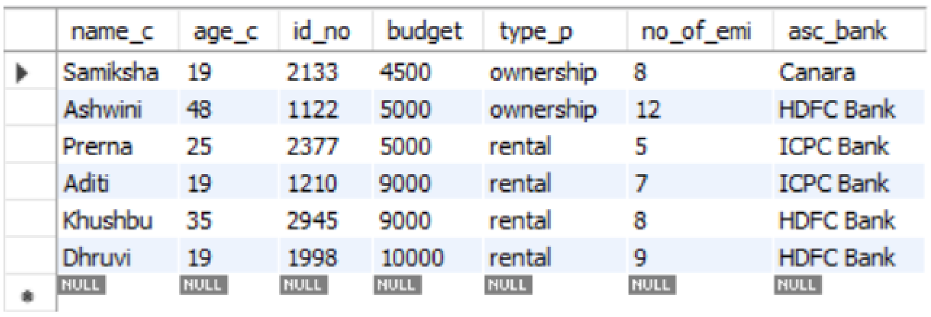
**Output:**



| EXPLAIN |
|---|
| -> Table scan on customer (cost=1.35 rows=1... |

b. **Select query with where clause**
**Code:**

```
use propsys;
 select * from customer where budget>4000;
```



| | name_c | age_c | id_no | budget | type_p | no_of_emi | asc_bank |
|---|---|---|---|---|---|---|---|
| ▶ | Samiksha | 19 | 2133 | 4500 | ownership | 8 | Canara |
| | Ashwini | 48 | 1122 | 5000 | ownership | 12 | HDFC Bank |
| | Prerna | 25 | 2377 | 5000 | rental | 5 | ICPC Bank |
| | Aditi | 19 | 1210 | 9000 | rental | 7 | ICPC Bank |
| | Khushbu | 35 | 2945 | 9000 | rental | 8 | HDFC Bank |
| | Dhruvi | 19 | 1998 | 10000 | rental | 9 | HDFC Bank |
| * | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

**Using explain command on select by where clause**

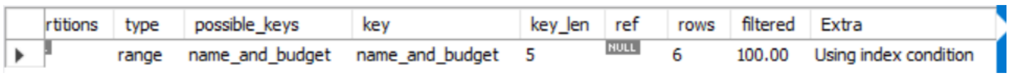
**Code:**

use propsys;

explain select * from customer where budget>4000;

(note we had created the name_and_budget index here and therefore the records are sorted by the budget for easy querying ordered by budget)

**Output:**



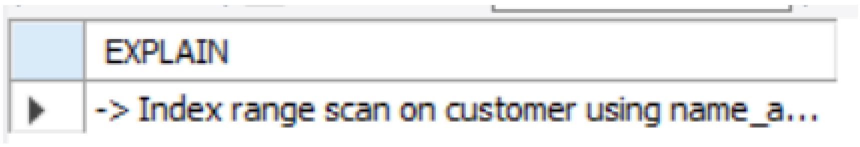
| | rtitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|
| ▶ | | range | name_and_budget | name_and_budget | 5 | NULL | 6 | 100.00 | Using index condition |

**Using Explain analyze command on select by where clause**

**Code:**

explain analyze select * from customer where budget>4000;

**Output:**



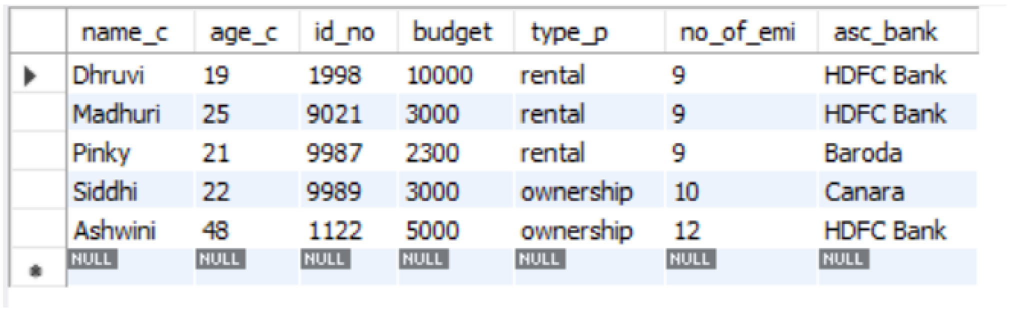
| | EXPLAIN |
|---|---|
| ▶ | -> Index range scan on customer using name_a... |

c. **Select query with order by query**

use propsys;

select * from customer where no_of_emi >8 order by no_of_emi;

**Output:**



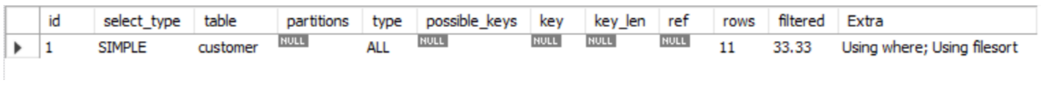
| | name_c | age_c | id_no | budget | type_p | no_of_emi | asc_bank |
|---|---|---|---|---|---|---|---|
| ▶ | Dhruvi | 19 | 1998 | 10000 | rental | 9 | HDFC Bank |
| | Madhuri | 25 | 9021 | 3000 | rental | 9 | HDFC Bank |
| | Pinky | 21 | 9987 | 2300 | rental | 9 | Baroda |
| | Siddhi | 22 | 9989 | 3000 | ownership | 10 | Canara |
| | Ashwini | 48 | 1122 | 5000 | ownership | 12 | HDFC Bank |
| * | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

**Using explain on select using order by clause**

**Code :**

explain select * from customer where no_of_emi >8 order by no_of_emi;

**Output:**



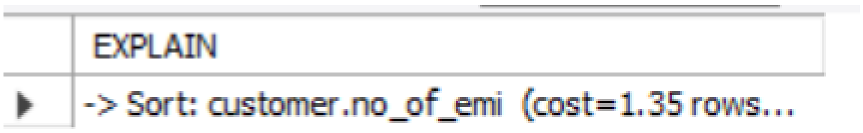
| | id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▶ | 1 | SIMPLE | customer | NULL | ALL | NULL | NULL | NULL | NULL | 11 | 33.33 | Using where; Using filesort |

**Using explain analyze on select using order by clause**

**Code:**

explain analyze select * from customer where no_of_emi >8 order by no_of_emi;

**Output**



| | EXPLAIN |
|---|---|
| ▶ | -> Sort: customer.no_of_emi (cost=1.35 rows... |

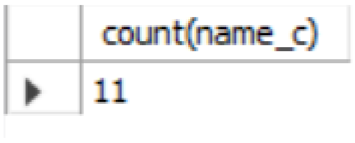
**d. Select query with aggregation**

**Code:**

use propsys;

select count(name_c) from customer; #selects the number of customers by counting with their names

**Output**:



| | count(name_c) |
|---|---|
| ▶ | 11 |

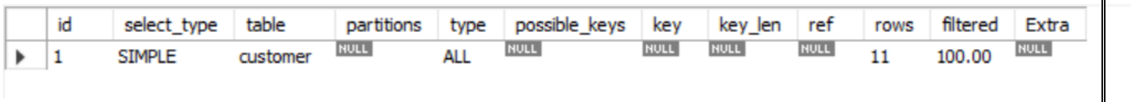
**Using Explain Statement**

**Code:**

use propsys;

explain select count(name_c) from customer;

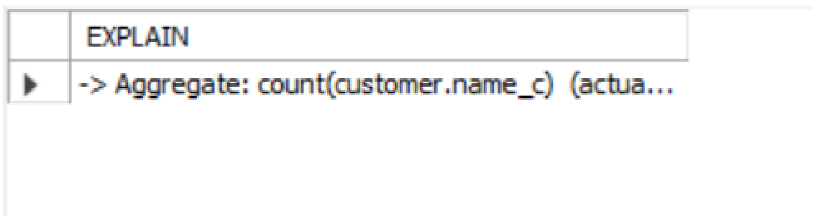
**Output**



| | id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▶ | 1 | SIMPLE | customer | NULL | ALL | NULL | NULL | NULL | NULL | 11 | 100.00 | NULL |

**Using Explain Analyze Statement:**

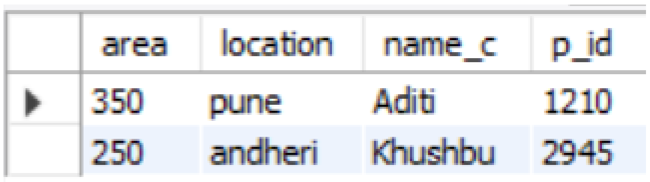
**Code :**



```
EXPLAIN
-> Aggregate: count(customer.name_c) (actua...
```

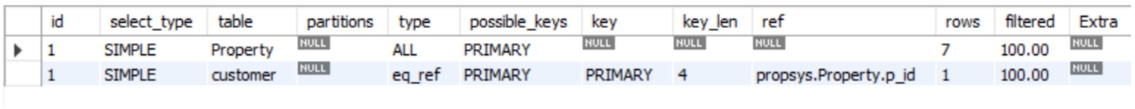
e. **Select query with JOIN clause.**
**Code:**
SELECT area, location, name_c, p_id FROM Property INNER JOIN customer ON Customer.id_no = property.p_id;

Output



| area | location | name_c | p_id |
|---|---|---|---|
| 350 | pune | Aditi | 1210 |
| 250 | andheri | Khushbu | 2945 |

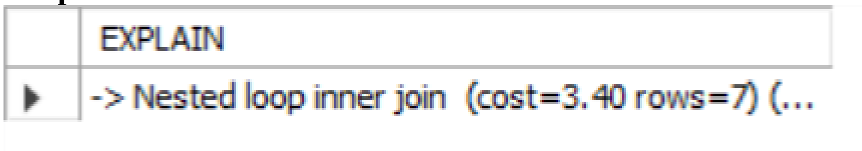
**Using Explain on Join Query:**



| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | Property | NULL | ALL | PRIMARY | NULL | NULL | NULL | 7 | 100.00 | NULL |
| 1 | SIMPLE | customer | NULL | eq_ref | PRIMARY | PRIMARY | 4 | propsys.Property.p_id | 1 | 100.00 | NULL |

**Using Explain Analyze on Join Query:**
**Code:**

explain analyze SELECT area, location, name_c, p_id FROM Property INNER JOIN customer ON Customer.id_no = property.p_id;
**Output:**



```
EXPLAIN
-> Nested loop inner join  (cost=3.40 rows=7) (...
```

**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 operations.
- Avoid cross-product operation since they result in very large-sized intermediate tables.

**Conclusion: In this experiment we have learnt about query optimization using indexing an have learnt how to implement the same . We have also learnt how to use the explain and explain analyze commands for query optimization.**