**Batch:C1 Roll-No:16010122221**

**Experiment / assignment / tutorial No. 7**

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

**Signature of the Staff In-charge with date**

|  |
| --- |
| **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:** MYSQL

**Theory:**

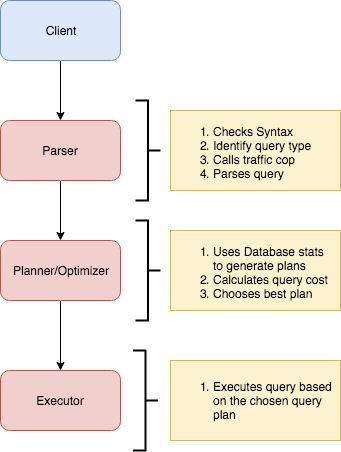
A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random look-ups and efficient ordering of access to records.

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 rewrite 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:**

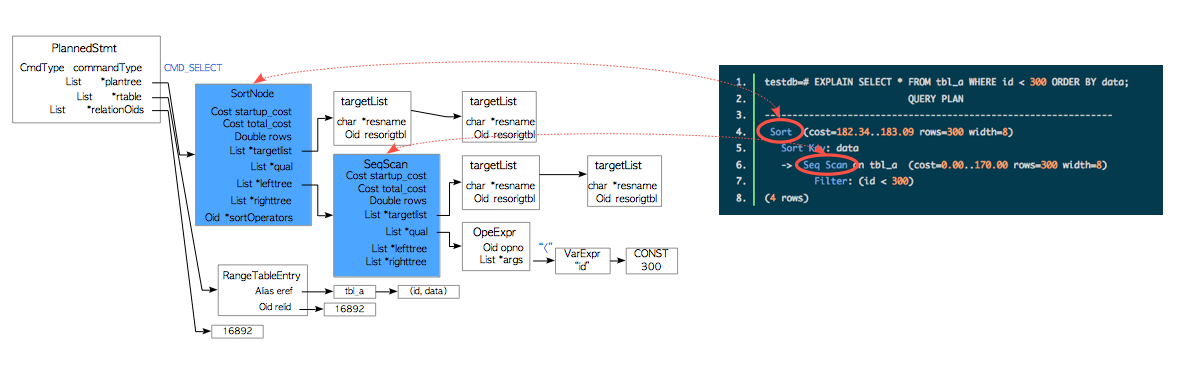
This command displays the execution plan that the PostgreSQL/MySQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms 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 MySQL.**



**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

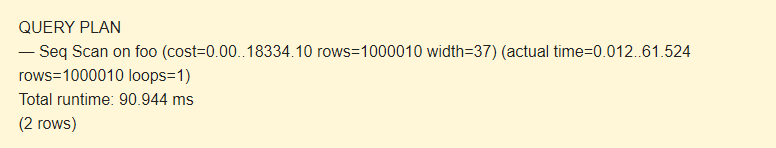
Rows = cardinality of relation \* selectivity )

**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 :**

**create database indexing;**

**use indexing;**

**CREATE TABLE tbl\_a (**

**id INT,**

**data TEXT**

**);**

**INSERT INTO tbl\_a (id, data) VALUES**

**(100, 'Data 1'),**

**(200, 'Data 2'),**

**(300, 'Data 3'),**

**(400, 'Data 4');**

**SELECT \* FROM tbl\_a;**

**SELECT \* FROM tbl\_a WHERE id = 100;**

**SELECT \* FROM tbl\_a ORDER BY data;**

**CREATE TABLE tbl\_b (**

**id INT,**

**other\_data TEXT**

**);**

**INSERT INTO tbl\_b (id, other\_data) VALUES**

**(100, 'Other Data 1'),**

**(200, 'Other Data 2'),**

**(300, 'Other Data 3'),**

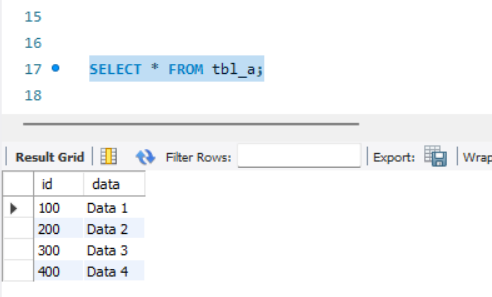
**(400, 'Other Data 4');**

**SELECT \* FROM tbl\_a JOIN tbl\_b ON tbl\_a.id = tbl\_b.id;**

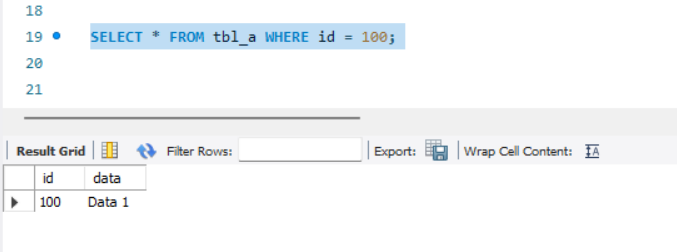
**SELECT COUNT(\*) FROM tbl\_a WHERE id < 300;**

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

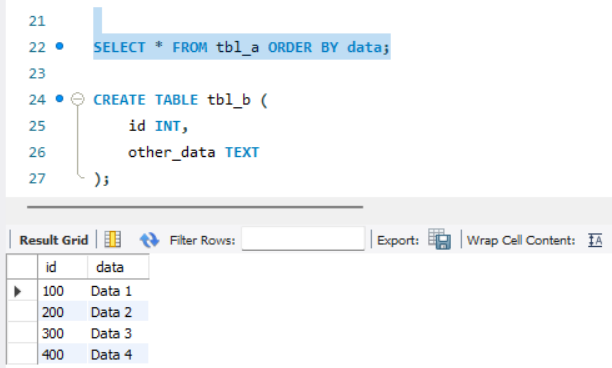
* 1. **Simple select query**



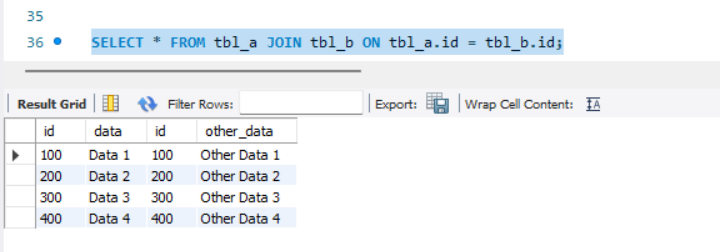
* 1. **Select query with where clause**



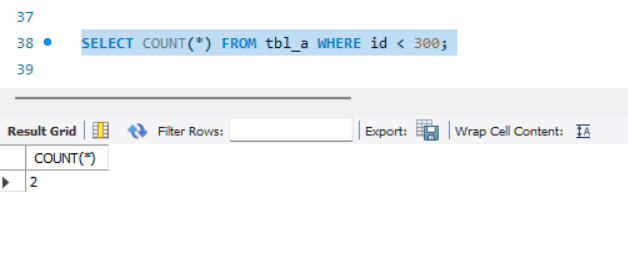
* 1. **Select query with order by query**



* 1. **Select query with JOIN**



* 1. **Select query with aggregation**



**Post Lab Question:**

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

The examples provided illustrate heuristic-based query optimization, which involves rewriting queries based on rules or guidelines to potentially improve performance. Let's break down each example:

1. Optimizing Filter Order:

Original Query:

```sql

SELECT \* FROM customers WHERE country = 'USA' AND age > 18;

```

Heuristic-based optimization:

```sql

SELECT \* FROM customers WHERE age > 18 AND country = 'USA';

```

Explanation: By filtering on `age > 18` first, which is likely to be more restrictive, the optimizer reduces the number of rows processed by the subsequent filter on `country = 'USA'`, potentially improving performance.

2. Optimizing Join Order and Index Usage:

Original Query:

```sql

SELECT \* FROM customers JOIN orders ON customers.id = orders.customer\_id;

```

Heuristic-based optimization:

```sql

SELECT \* FROM customers JOIN orders ON customers.id = orders.customer\_id;

```

Explanation: In this case, the optimizer chooses to join the tables in the order of their cardinality, meaning the smaller table (`customers`) is joined to the larger table (`orders`). This reduces the number of rows processed during the join operation, potentially improving performance. Additionally, the optimizer might utilize indexes to speed up the join operation.

Heuristic-based query optimization relies on heuristics or rules of thumb to rewrite queries in a way that's likely to improve performance. While it doesn't guarantee the optimal query execution plan, it can often lead to significant performance improvements by reducing the number of rows processed or optimizing the order of operations.

# Conclusion:

Learned Query Processing and successfully implemented indexing to improve query execution plans .