Batch: B-2 Roll No.: 16010122151

Experiment / assignment / tutorial No. 7
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:

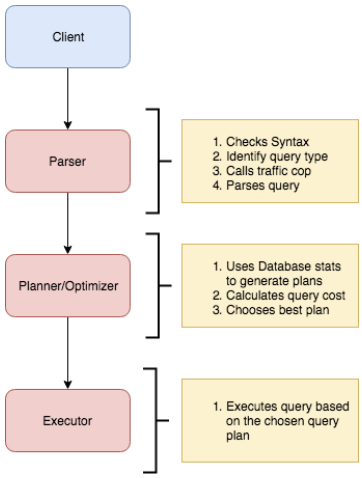
```
CREATE INDEX index_name ON table_name (column_list)
```

Query life cycle

Batch: B-2 Roll No.: 16010122151

Experiment / assignment / tutorial No. 7

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



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:-

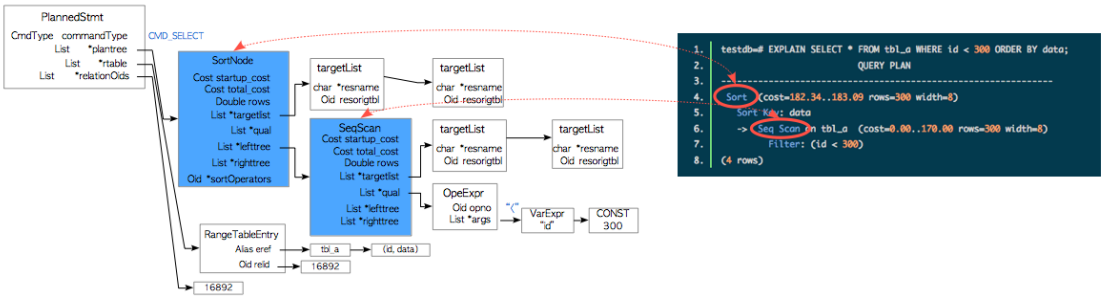
Database: testdb=#

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.

Batch: B-2 Roll No.: 16010122151

l



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.

Indexing: `CREATE INDEX` constructs an index on the specified column(s) of the specified relation, which can be a table or a materialized view. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

Syntax

```
 CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ]
name ] ON [ ONLY ] table_name [ USING method ]

    ( { column_name | ( expression ) } [ COLLATE collation ] [
opclass [ ( opclass_parameter = value [, ... ] ) ] ] [ ASC |
DESC ] [ NULLS { FIRST | LAST } ] [, ...] )

    [ INCLUDE ( column_name [, ...] ) ]

    [ NULLS [ NOT ] DISTINCT ]

    [ WITH ( storage_parameter [= value] [, ... ] ) ]

    [ TABLESPACE tablespace_name ]

    [ WHERE predicate ]
```

example

Batch: B-2 Roll No.: 16010122151

tutorial

CC / CD

explain Analyse select * from std2 where branch ='ext'

create index dept on std2(Branch)

select * from std2

explain Analyse select * from std2 where branch ='ext'

drop index dept

Implementation Screenshots :

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

a. Simple select query

Without Index

```
1  EXPLAIN ANALYSE SELECT cust_id FROM bank;
2
3
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Seq Scan on bank (cost=0.00..1.04 rows=4 width=4) (actual time=0.009..0.009 rows=4 loops=1)
2	Planning Time: 0.151 ms
3	Execution Time: 0.017 ms

With index

```
4  EXPLAIN ANALYSE SELECT cust_id from bank;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Seq Scan on bank (cost=0.00..1.04 rows=4 width=4) (actual time=0.010..0.011 rows=4 loops=1)
2	Planning Time: 0.054 ms
3	Execution Time: 0.020 ms

b. Select query with where clause

Batch: B-2 Roll No.: 16010122151

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

```
1  EXPLAIN ANALYSE SELECT emp_name FROM bank WHERE  emp_id = 1;
2
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Seq Scan on bank (cost=0.00..1.05 rows=1 width=58) (actual time=0.011..0.012 rows=1 loops=1)
2	Filter: (emp_id = 1)
3	Rows Removed by Filter: 3
4	Planning Time: 0.052 ms
5	Execution Time: 0.023 ms

With index

```
3  EXPLAIN ANALYSE SELECT emp_name FROM bank WHERE  emp_id = 1;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Seq Scan on bank (cost=0.00..1.05 rows=1 width=58) (actual time=0.023..0.025 rows=1 loops=1)
2	Filter: (emp_id = 1)
3	Rows Removed by Filter: 3
4	Planning Time: 0.116 ms
5	Execution Time: 0.044 ms

c. Select query with order by query

Batch: B-2 Roll No.: 16010122151

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

```
1  EXPLAIN ANALYSE SELECT cust_name FROM bank ORDER BY cust_name;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Sort (cost=1.08..1.09 rows=4 width=58) (actual time=0.024..0.024 rows=4 loops=1)
2	Sort Key: cust_name
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on bank (cost=0.00..1.04 rows=4 width=58) (actual time=0.010..0.011 rows=4 loops=1)
5	Planning Time: 0.061 ms
6	Execution Time: 0.035 ms

With index

```
3  EXPLAIN ANALYSE SELECT cust_name FROM bank ORDER BY cust_name;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Sort (cost=1.08..1.09 rows=4 width=58) (actual time=0.023..0.023 rows=4 loops=1)
2	Sort Key: cust_name
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on bank (cost=0.00..1.04 rows=4 width=58) (actual time=0.010..0.011 rows=4 loops=1)
5	Planning Time: 0.069 ms
6	Execution Time: 0.035 ms

d. Select query with JOIN

Batch: B-2 Roll No.: 16010122151

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

Without index

```
1  EXPLAIN ANALYSE SELECT employ.emp_id,bank.cust_name,bank.branch_id
2  FROM employ INNER JOIN bank ON bank.cust_id = employ.emp_id;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Hash Join (cost=1.09..15.53 rows=4 width=66) (actual time=0.027..0.029 rows=4 loops=1)
2	Hash Cond: (employ.emp_id = bank.cust_id)
3	-> Seq Scan on employ (cost=0.00..13.20 rows=320 width=4) (actual time=0.011..0.011 rows=8 loops=1)
4	-> Hash (cost=1.04..1.04 rows=4 width=66) (actual time=0.011..0.011 rows=4 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Seq Scan on bank (cost=0.00..1.04 rows=4 width=66) (actual time=0.006..0.008 rows=4 loops=1)
7	Planning Time: 0.120 ms
8	Execution Time: 0.046 ms

With index

```
1  EXPLAIN ANALYSE SELECT employ.emp_id,bank.cust_name,bank.branch_id
2  FROM employ INNER JOIN bank ON bank.cust_id = employ.emp_id;
3
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Hash Join (cost=1.09..2.24 rows=4 width=66) (actual time=0.027..0.029 rows=4 loops=1)
2	Hash Cond: (employ.emp_id = bank.cust_id)
3	-> Seq Scan on employ (cost=0.00..1.08 rows=8 width=4) (actual time=0.011..0.012 rows=8 loops=1)
4	-> Hash (cost=1.04..1.04 rows=4 width=66) (actual time=0.011..0.011 rows=4 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Seq Scan on bank (cost=0.00..1.04 rows=4 width=66) (actual time=0.006..0.007 rows=4 loops=1)
7	Planning Time: 0.154 ms
8	Execution Time: 0.053 ms

Batch: B-2 Roll No.: 16010122151

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

e. Select query with aggregation

Without index

```
1  EXPLAIN ANALYSE SELECT AVG(emp_salary) FROM employ;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Aggregate (cost=25.00..25.01 rows=1 width=32) (actual time=0.023..0.023 ro...
2	-> Seq Scan on employ (cost=0.00..22.00 rows=1200 width=4) (actual time=0...
3	Planning Time: 0.077 ms
4	Execution Time: 0.053 ms

Batch: B-2 Roll No.: 16010122151

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

With index

```
1  EXPLAIN ANALYSE SELECT AVG(emp_salary) FROM employ;
```

Explain | Data Output | Notifications | Messages

	QUERY PLAN text
1	Aggregate (cost=1.05..1.06 rows=1 width=32) (actual time=0.022..0.022 rows...
2	-> Seq Scan on employ (cost=0.00..1.04 rows=4 width=4) (actual time=0.013....
3	Planning Time: 0.087 ms
4	Execution Time: 0.051 ms

Batch: B-2 Roll No.: 16010122151

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

Post Lab Question:

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

Heuristic-based query optimization involves using rules of thumb or common-sense strategies to optimize query performance without exhaustively exploring all possible execution plans. Here's an example:

Consider a simple database table called Employees with columns EmployeeID, Name, Department, Salary, and YearsOfExperience. Let's say we want to retrieve the names of employees who belong to the IT department and have a salary greater than $50,000. The SQL query for this might look like:

```
SELECT Name
FROM Employees
WHERE Department = 'IT' AND Salary > 50000;
```

Now, let's say our database system supports indexing on the Department and Salary columns. We have two potential execution plans for this query:

Using Indexes: We could use indexes on both Department and Salary columns to filter rows efficiently.

Code :

```
SELECT Name
FROM Employees
WHERE Department = 'IT' AND Salary > 50000;
```

Batch: B-2 Roll No.: 16010122151

Experiment / assignment / tutorial No. 7

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

Full Table Scan: We could simply scan the entire table and filter out the rows that meet the condition.

SELECT Name

FROM Employees

WHERE Department = 'IT' AND Salary > 50000;

In this scenario, if the table is small or if a large percentage of employees belong to the IT department with high salaries, using a full table scan might be more efficient than using indexes, as it avoids the overhead of index lookups.

So, the heuristic-based optimization might prioritize a full table scan over using indexes based on factors like table size, data distribution, and selectivity of the conditions.

However, if the table is large and the selectivity of the conditions is high (i.e., only a small percentage of rows satisfy the condition), using indexes might be more efficient.

In real-world scenarios, the choice between these two plans might depend on various factors such as table statistics, hardware configurations, and the workload of the database system. A heuristic-based approach would weigh these factors and make an educated guess on the optimal plan without exhaustively exploring all possibilities.

Conclusion:

Using Indexing we can increase the speed of data processing.