Batch: A1 Roll No.: 16010123012
Experiment / assignment / tutorial No. 7
Grade: AA / AB / BB / BC / CC / CD /DD
Signature of the Staff In-charge with date

Title: Indexing - Create Index and observe the evaluation statistics of query execution with and without Index

Objective: Implement indexing to improve query execution plans

Expected Outcome of Experiment:
CO 4: Analyse Advanced Database Concepts like indexing, hashing, query processing, query optimization, normalization.

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.

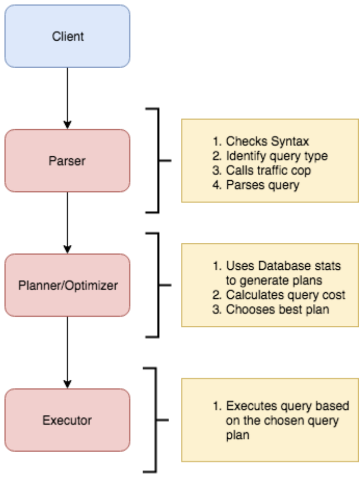
Resources used: PostgreSQL

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:

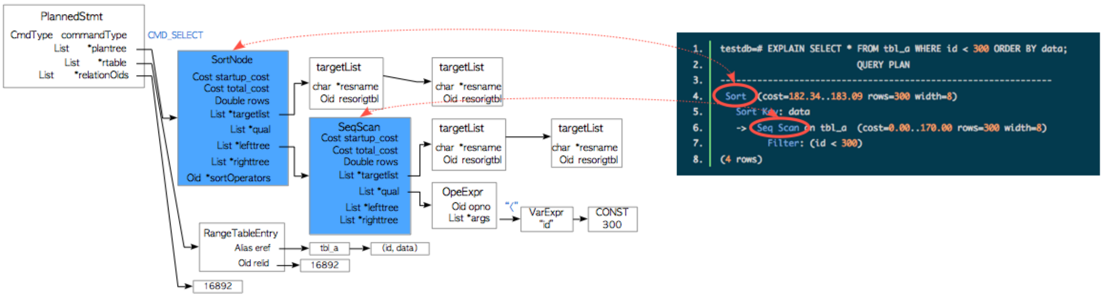
This command displays the execution plan that the PostgreSQL/MySQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms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



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

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
b. Select query with where clause
c. Select query with order by query
d. Select query with JOIN
e. Select query with aggregation

```
CREATE TABLE users (
   id SERIAL PRIMARY KEY,
   name TEXT,
   email TEXT
);
```

```
INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com'),
('Charlie', 'charlie@example.com');
```

EXPLAIN ANALYZE SELECT * FROM users WHERE name = 'Alice';

CREATE INDEX idx_users_name ON users (name);

EXPLAIN ANALYZE SELECT * FROM users WHERE name = 'Alice';

Index:

Query Query History

```sql
CREATE INDEX Idx ON employee (ssn);
```

Data Output Messages Notifications

```
CREATE INDEX

Query returned successfully in 216 msec.
```

Before Indexing:

Query Query History

```sql
EXPLAIN ANALYZE SELECT * FROM employee;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..10.90 rows=90 width=872) (actual time=0.043..0.045 rows=6 loops...
2	Planning Time: 0.107 ms
3	Execution Time: 0.074 ms

After Indexing:

Query Query History

```sql
EXPLAIN ANALYZE SELECT * From employee;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..1.06 rows=6 width=872) (actual time=0.036..0.038 rows=6 loops...
2	Planning Time: 0.393 ms
3	Execution Time: 0.052 ms

Before Indexing:

Query | Query History

```sql
EXPLAIN ANALYZE SELECT * FROM employee WHERE salary >= 60000;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..11.12 rows=30 width=872) (actual time=0.032..0.034 rows=4 loops...
2	Filter: (salary >= '60000'::numeric)
3	Rows Removed by Filter: 2
4	Planning Time: 0.321 ms
5	Execution Time: 0.069 ms

After Indexing:

Query | Query History

```sql
EXPLAIN ANALYZE SELECT * FROM employee WHERE salary >= 60000;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..1.07 rows=2 width=872) (actual time=0.020..0.022 rows=4 loops...
2	Filter: (salary >= '60000'::numeric)
3	Rows Removed by Filter: 2
4	Planning Time: 0.101 ms
5	Execution Time: 0.034 ms

Before Indexing:

Query   Query History

```sql
EXPLAIN ANALYZE SELECT * FROM employee WHERE salary >= 60000 ORDER BY salary DESC;
```

Data Output   Messages   Notifications

	QUERY PLAN text
1	Sort (cost=11.86..11.94 rows=30 width=872) (actual time=0.027..0.028 rows=4 loops=1)
2	Sort Key: salary DESC
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on employee (cost=0.00..11.12 rows=30 width=872) (actual time=0.017..0.019 rows=4 loop...
5	Filter: (salary >= '60000'::numeric)
6	Rows Removed by Filter: 2
7	Planning Time: 0.102 ms
8	Execution Time: 0.044 ms

After Indexing:

Query   Query History

```sql
EXPLAIN ANALYZE SELECT * FROM employee WHERE salary >= 60000 ORDER BY salary DESC;
```

Data Output   Messages   Notifications

	QUERY PLAN text
1	Sort (cost=1.08..1.09 rows=2 width=872) (actual time=0.027..0.027 rows=4 loops=1)
2	Sort Key: salary DESC
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on employee (cost=0.00..1.07 rows=2 width=872) (actual time=0.017..0.019 rows=4 loop...
5	Filter: (salary >= '60000'::numeric)
6	Rows Removed by Filter: 2
7	Planning Time: 0.108 ms
8	Execution Time: 0.041 ms

FOO-

Query Query History

```sql
CREATE TABLE foo (c1 integer, c2 text);
INSERT INTO foo
SELECT i, md5(random()::text)
FROM generate_series(1, 1000000) AS i;

```

Data Output Messages Notifications

```
INSERT 0 1000000

Query returned successfully in 7 secs 422 msec.
```

Query Query History

```sql
EXPLAIN SELECT * FROM foo;
```

Data Output Messages Notifications

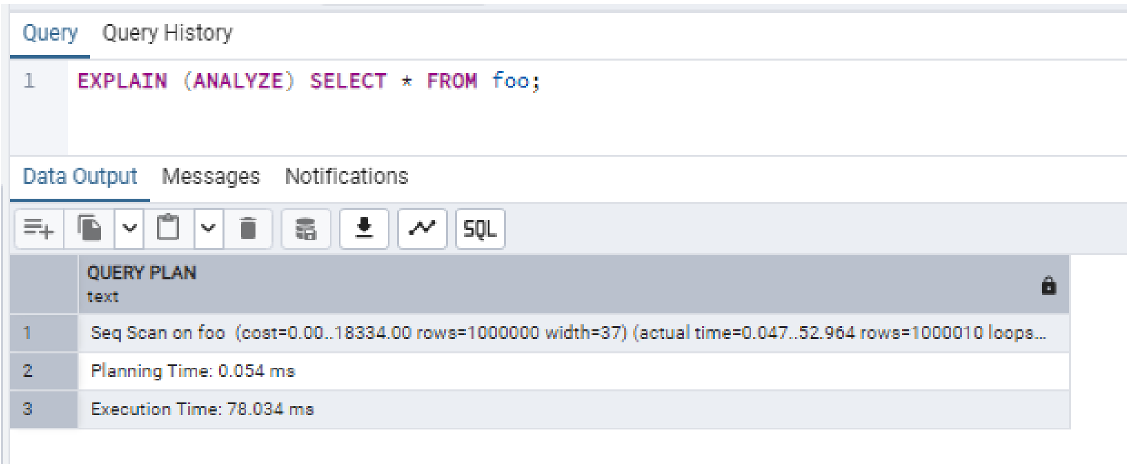
	QUERY PLAN text
1	Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=...

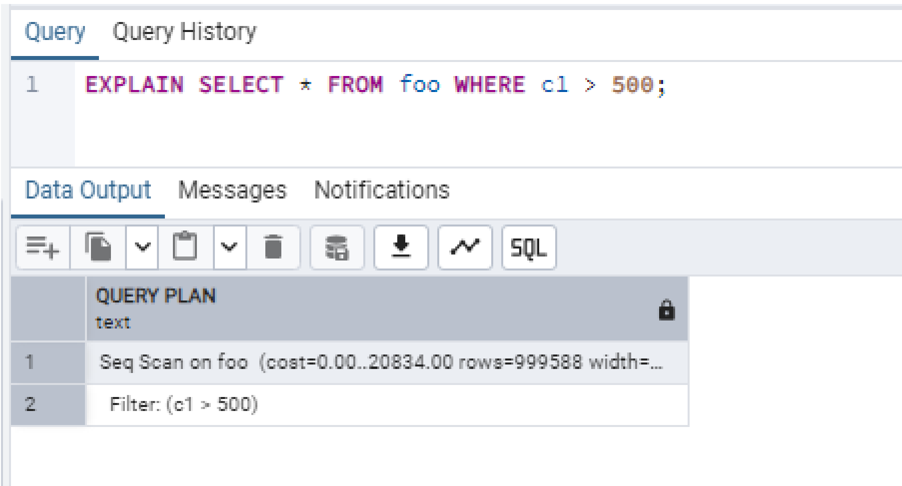
Query Query History

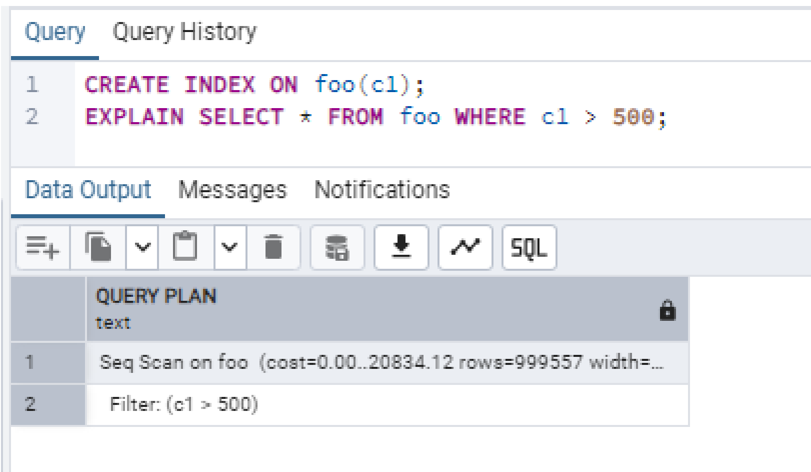
```sql
INSERT INTO foo
SELECT i, md5(random()::text)
FROM generate_series(1, 10) AS i;
EXPLAIN SELECT * FROM foo;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=...


Query
Query History
1 EXPLAIN (ANALYZE) SELECT * FROM foo;
Data Output
Messages
Notifications
SQL
QUERY PLAN
text
1 Seq Scan on foo (cost=0.00..18334.00 rows=1000000 width=37) (actual time=0.047..52.964 rows=1000010 loops...
2 Planning Time: 0.054 ms
3 Execution Time: 78.034 ms


Query
Query History
1 EXPLAIN SELECT * FROM foo WHERE c1 > 500;
Data Output
Messages
Notifications
SQL
QUERY PLAN
text
1 Seq Scan on foo (cost=0.00..20834.00 rows=999588 width=...
2 Filter: (c1 > 500)


Query
Query History
1 CREATE INDEX ON foo(c1);
2 EXPLAIN SELECT * FROM foo WHERE c1 > 500;
Data Output
Messages
Notifications
SQL
QUERY PLAN
text
1 Seq Scan on foo (cost=0.00..20834.12 rows=999557 width=...
2 Filter: (c1 > 500)

Query Query History

```
EXPLAIN SELECT * FROM foo WHERE c1 < 500;
```

Data Output Messages Notifications

	QUERY PLAN text
1	Index Scan using foo_c1_idx on foo (cost=0.42..23.34 rows=452 width=...
2	Index Cond: (c1 < 500)

Query Query History

```
EXPLAIN SELECT * FROM foo
WHERE c1 < 500 AND c2 LIKE 'abcd';
```

Data Output Messages Notifications

	QUERY PLAN text
1	Index Scan using foo_c1_idx on foo (cost=0.42..24.46 rows=1 width=...
2	Index Cond: (c1 < 500)
3	Filter: (c2 ~~ 'abcd'::text)

Query Query History

```
CREATE TABLE bar (c1 integer, c2 boolean);
INSERT INTO bar
SELECT i, i%2=1
FROM generate_series(1, 500000) AS i;
ANALYZE bar;
```

Data Output Messages Notifications

```
ANALYZE

Query returned successfully in 1 secs 275 msec.
```

Query | Query History

```sql
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Hash Join (cost=15417.00..60081.14 rows=500000 width=42) (actual time=114.963..801.182 rows=500010 loo...
2	Hash Cond: (foo.c1 = bar.c1)
3	-> Seq Scan on foo (cost=0.00..18334.10 rows=1000010 width=37) (actual time=0.020..52.945 rows=1000010...
4	-> Hash (cost=7213.00..7213.00 rows=500000 width=5) (actual time=114.458..114.459 rows=500000 loops=1)
5	Buckets: 262144 Batches: 4 Memory Usage: 6562kB
6	-> Seq Scan on bar (cost=0.00..7213.00 rows=500000 width=5) (actual time=0.025..40.061 rows=500000 l...
7	Planning Time: 0.583 ms
8	Execution Time: 815.085 ms

Query | Query History

```sql
CREATE INDEX ON bar(c1);
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Merge Join (cost=2.26..39845.63 rows=500000 width=42) (actual time=0.027..321.181 rows=500010 loops=1)
2	Merge Cond: (foo.c1 = bar.c1)
3	-> Index Scan using foo_c1_idx on foo (cost=0.42..34317.58 rows=1000010 width=37) (actual time=0.012..99.302 rows=500011 loop...
4	-> Index Scan using bar_c1_idx on bar (cost=0.42..15212.42 rows=500000 width=5) (actual time=0.012..94.733 rows=500010 loops=1)
5	Planning Time: 0.492 ms
6	Execution Time: 335.168 ms

Post Lab Question:

1. How does a B-tree differ from a B+-tree? Why is a B+-tree usually preferred as an access structure to a data file?

A B-tree and a B+-tree are both balanced tree structures used in databases and file systems for efficient data retrieval. In a B-tree, both internal and leaf nodes store keys and data, whereas in a B+-tree, only the leaf nodes hold the actual data, and internal nodes contain only keys used for navigation. Additionally, leaf nodes in a B+-tree are linked, enabling efficient sequential and range queries, unlike in a B-tree where leaf nodes are not connected. Search operations in a B-tree can end at any node, while in a B+-tree, they always reach the leaf level, making the search path uniform. Though B+-trees may be slightly taller, their smaller internal nodes fit more keys per disk block, reducing disk I/O. This structure also simplifies insertions and deletions, as changes mainly affect leaf nodes. Overall, a B+-tree is usually preferred as an access structure to a data file because it supports faster range queries, better disk access patterns, and more predictable performance, making it ideal for indexing large datasets.

Conclusion:

I have successfully implemented indexing techniques and analyzed their impact on query performance using SQL commands such as EXPLAIN ANALYZE. This experiment demonstrated how indexing optimizes query execution by minimizing data scan time and improving efficiency. By comparing query plans before and after indexing, I observed significant performance gains, especially in operations involving filtering and sorting.