

Batch: E-2 Roll No.: 16010123325

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:

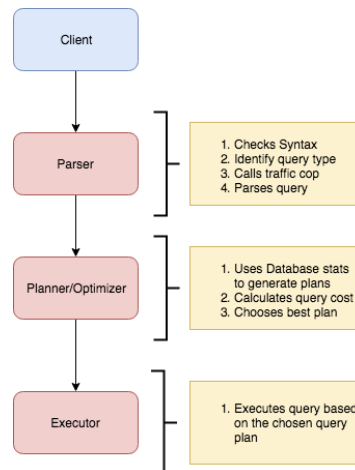
A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

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 -

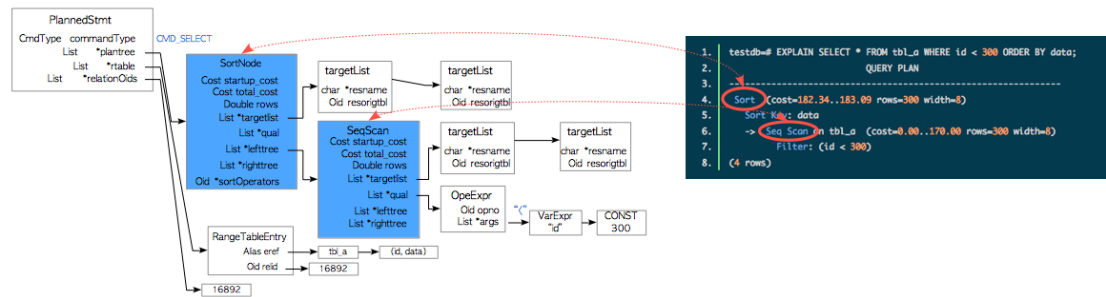
EXPLAIN command:

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 preceding “->” (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
);
```

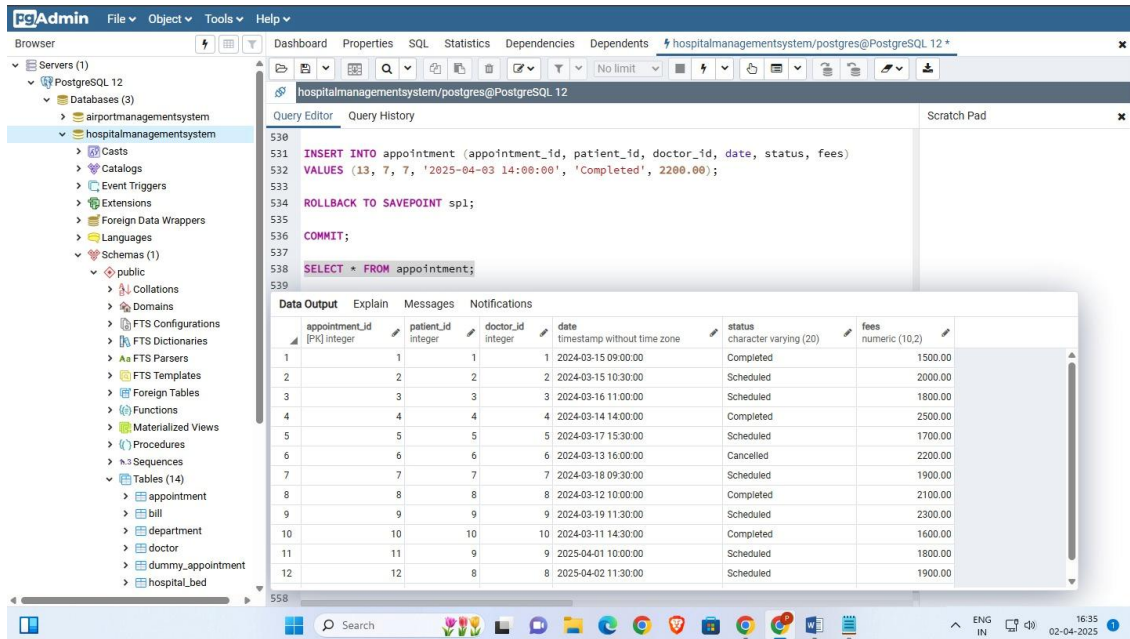
```
INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com'),
('Charlie', 'charlie@example.com');
```

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

```
CREATE INDEX idx_users_name ON users (name);
```

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

### a. Simple select query

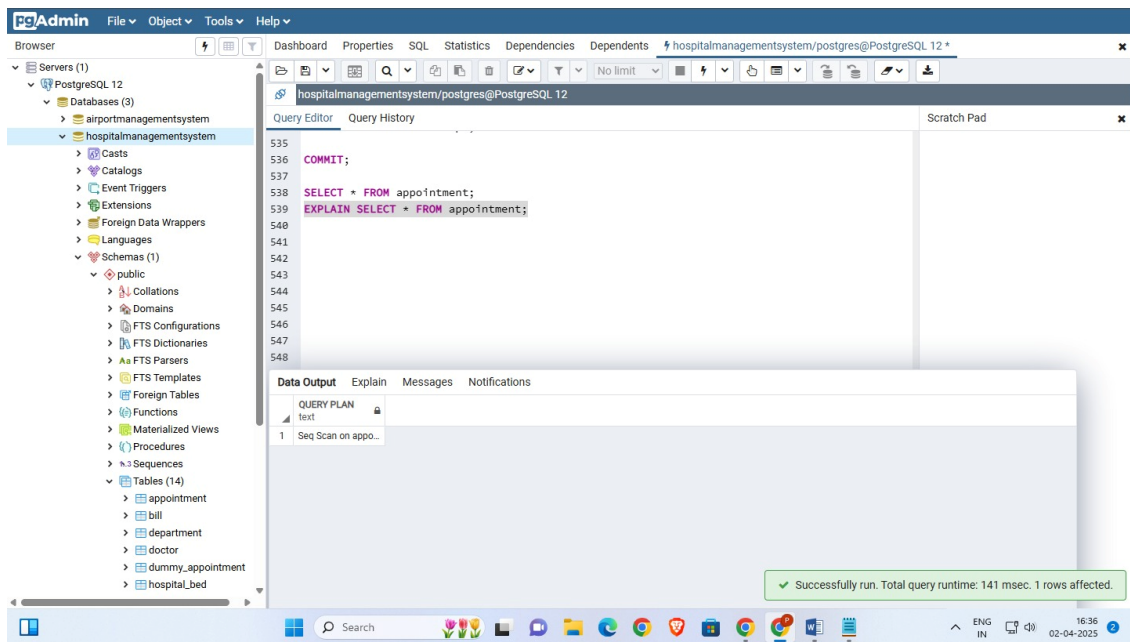


The screenshot shows the pgAdmin interface with the following SQL query in the Query Editor:

```
530
531 INSERT INTO appointment (appointment_id, patient_id, doctor_id, date, status, fees)
532 VALUES (13, 7, 7, '2025-04-03 14:00:00', 'Completed', 2200.00);
533
534 ROLLBACK TO SAVEPOINT sp1;
535
536 COMMIT;
537
538 SELECT * FROM appointment;
539
```

The Data Output pane displays the following table:

| appointment_id | patient_id | doctor_id | date                | status    | fees    |
|----------------|------------|-----------|---------------------|-----------|---------|
| 1              | 1          | 1         | 2024-03-15 09:00:00 | Completed | 1500.00 |
| 2              | 2          | 2         | 2024-03-15 10:30:00 | Scheduled | 2000.00 |
| 3              | 3          | 3         | 2024-03-16 11:00:00 | Scheduled | 1800.00 |
| 4              | 4          | 4         | 2024-03-14 14:00:00 | Completed | 2500.00 |
| 5              | 5          | 5         | 2024-03-17 15:30:00 | Scheduled | 1700.00 |
| 6              | 6          | 6         | 2024-03-13 16:00:00 | Cancelled | 2200.00 |
| 7              | 7          | 7         | 2024-03-18 09:30:00 | Scheduled | 1900.00 |
| 8              | 8          | 8         | 2024-03-12 10:00:00 | Completed | 2100.00 |
| 9              | 9          | 9         | 2024-03-19 11:30:00 | Scheduled | 2300.00 |
| 10             | 10         | 10        | 2024-03-11 14:30:00 | Completed | 1600.00 |
| 11             | 11         | 9         | 2025-04-01 10:00:00 | Scheduled | 1800.00 |
| 12             | 12         | 8         | 2025-04-02 11:30:00 | Scheduled | 1900.00 |



The screenshot shows the pgAdmin interface with the following SQL query in the Query Editor:

```
535
536 COMMIT;
537
538 SELECT * FROM appointment;
539 EXPLAIN SELECT * FROM appointment;
540
541
542
543
544
545
546
547
548
```

The Data Output pane displays the following table:

| QUERY PLAN            |
|-----------------------|
| 1 Seq Scan on appo... |

A green status bar at the bottom indicates: "Successfully run. Total query runtime: 141 msec. 1 rows affected."

The screenshot shows the pgAdmin 4 web interface. On the left, the 'Servers' tree is expanded to show 'PostgreSQL 12' > 'Databases (3)' > 'hospitalmanagementsystem'. The 'Schemas (1)' section is also expanded, showing the 'public' schema with various objects like 'Collations', 'Domains', 'FTS Configurations', etc. The main pane displays the 'Query Editor' for the 'hospitalmanagementsystem/postgres@PostgreSQL 12' connection. The query text is as follows:

```
535
536 COMMIT;
537
538 SELECT * FROM appointment;
539 EXPLAIN SELECT * FROM appointment;
540 EXPLAIN ANALYZE SELECT * FROM appointment;
541
542
543
544
545
546
547
548
```

Below the query editor, the 'Data Output' tab is active, showing a 'QUERY PLAN' table:

| QUERY PLAN              |
|-------------------------|
| text                    |
| 1 Seq Scan on appo...   |
| 2 Planning Time: 0.0... |
| 3 Execution Time: 0...  |

The screenshot shows the pgAdmin 4 web interface for the 'Airline Reservation System'. The 'Servers' tree is expanded to show 'PostgreSQL 17' > 'Databases (2)' > 'Airline Reser...'. The 'Schemas (1)' section is expanded, showing the 'public' schema. The main pane displays the 'Query Editor' for the 'Airline Reservation System/postgres@PostgreSQL 17' connection. The query text is as follows:

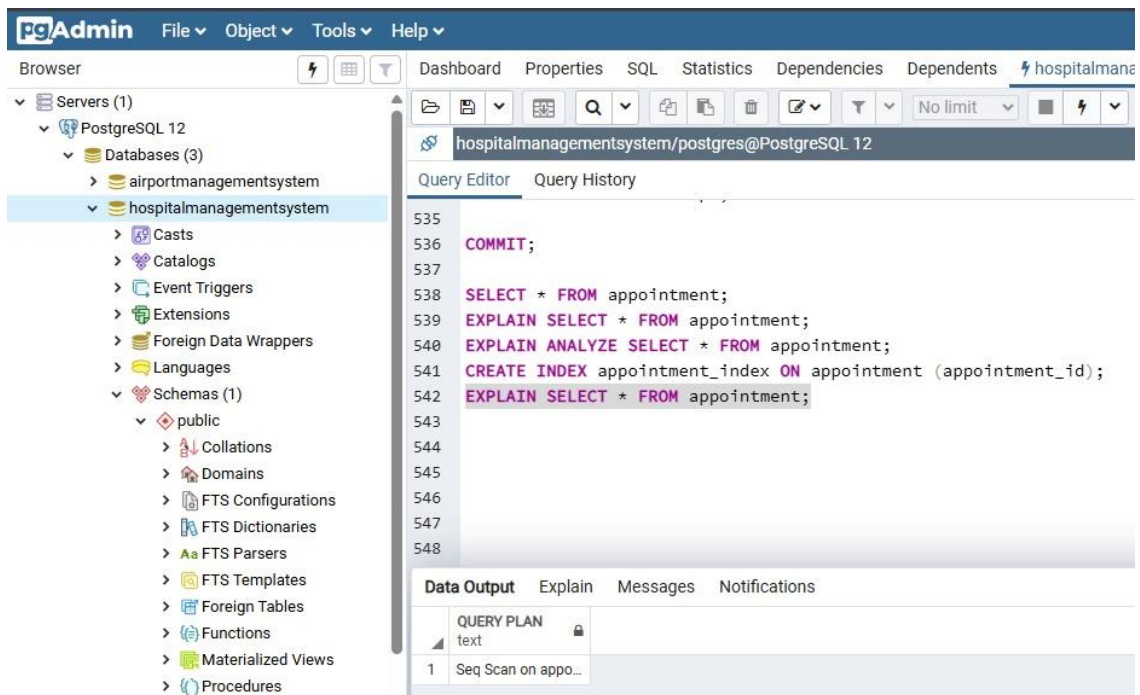
```
1 SELECT * FROM customer;
2
3 explain select * from customer;
4
5 explain analyze select * from customer;
6
7 create index c_index on customer (u_id);
8
9 explain select * from customer;
10
11
12
13
14
```

Below the query editor, the 'Data Output' tab is active, showing a 'CREATE INDEX' message:

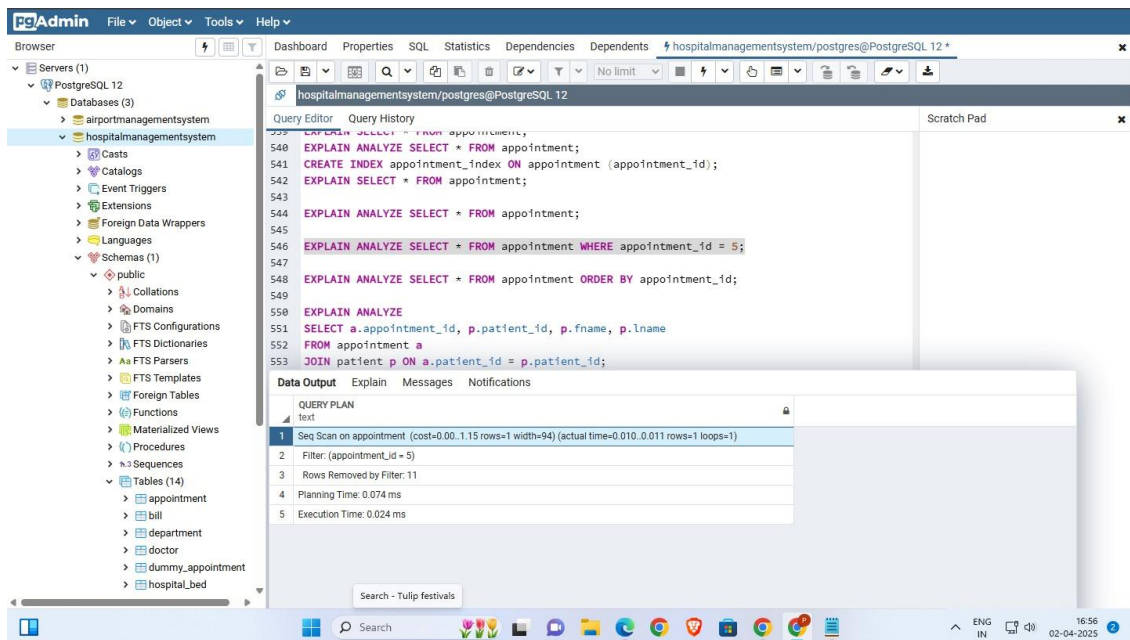
```
CREATE INDEX
Query returned successfully in 119 msec.
```

At the bottom of the interface, a status bar indicates 'Total rows: 3 Query complete 00:00:00.119' and a green message box says 'Query returned successfully in 119 msec.'.





**b. Select query with where clause**





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

The screenshot shows the PGAdmin interface with a SQL query in the Query Editor. The query is as follows:

```
539 CREATE INDEX ON appointment;
540 EXPLAIN ANALYZE SELECT * FROM appointment;
541 CREATE INDEX appointment_index ON appointment (appointment_id);
542 EXPLAIN SELECT * FROM appointment;
543
544 EXPLAIN ANALYZE SELECT * FROM appointment;
545
546 EXPLAIN ANALYZE SELECT * FROM appointment WHERE appointment_id = 5;
547
548 EXPLAIN ANALYZE SELECT * FROM appointment ORDER BY appointment_id;
549
550 EXPLAIN ANALYZE
551 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
552 FROM appointment a
553 JOIN patient p ON a.patient_id = p.patient_id;
```

The Data Output pane shows the execution plan for the query with `ORDER BY appointment_id`:

| Step | Operation                   | Cost                              | Actual Time | Actual Rows | Actual Loops   |
|------|-----------------------------|-----------------------------------|-------------|-------------|----------------|
| 1    | Seq Scan on appointment     | (cost=0.00..1.15 rows=1 width=94) | 0.010       | 0.011       | rows=1 loops=1 |
| 2    | Filter (appointment_id = 5) |                                   |             |             |                |
| 3    | Rows Removed by Filter: 11  |                                   |             |             |                |
| 4    | Planning Time               |                                   | 0.074 ms    |             |                |
| 5    | Execution Time              |                                   | 0.024 ms    |             |                |

**d. Select query with JOIN**

The screenshot shows the PGAdmin interface with a SQL query in the Query Editor. The query is as follows:

```
556 EXPLAIN ANALYZE SELECT AVG(fees) FROM appointment WHERE status = 'Scheduled';
557
558
559 EXPLAIN ANALYZE
560 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
561 FROM appointment a
562 JOIN patient p ON a.patient_id = p.patient_id;
563
564 CREATE INDEX idx_appointment_patient_id ON appointment (patient_id);
565
566 EXPLAIN ANALYZE
567 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
568 FROM appointment a
569 JOIN patient p ON a.patient_id = p.patient_id;
```

The Data Output pane shows the execution plan for the JOIN query:

| Step | Operation                                  | Cost                                 | Actual Time | Actual Rows | Actual Loops    |
|------|--------------------------------------------|--------------------------------------|-------------|-------------|-----------------|
| 1    | Hash Join                                  | (cost=1.27..12.49 rows=12 width=244) | 0.101       | 0.110       | rows=12 loops=1 |
| 2    | Hash Cond: (p.patient_id = a.patient_id)   |                                      |             |             |                 |
| 3    | Seq Scan on patient p                      | (cost=0.00..10.80 rows=80 width=240) | 0.058       | 0.059       | rows=10 loops=1 |
| 4    | Hash                                       | (cost=1.12..1.12 rows=12 width=8)    | 0.028       | 0.028       | rows=12 loops=1 |
| 5    | Buckets: 1024 Batches: 1 Memory Usage: 9KB |                                      |             |             |                 |
| 6    | Seq Scan on appointment a                  | (cost=0.00..1.12 rows=12 width=8)    | 0.018       | 0.020       | rows=12 loops=1 |
| 7    | Planning Time                              |                                      | 0.388 ms    |             |                 |
| 8    | Execution Time                             |                                      | 0.393 ms    |             |                 |

A green message bar at the bottom indicates: "Successfully run. Total query runtime: 72 msec. 8 rows affected."

The image displays two screenshots of the pgAdmin 4 interface, demonstrating the execution of a SQL query and the resulting query plan.

**Top Screenshot:** The Query Editor shows the following SQL code:

```
556 EXPLAIN ANALYZE SELECT AVG(fees) FROM appointment WHERE status = 'Scheduled';
557
558
559 EXPLAIN ANALYZE
560 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
561 FROM appointment a
562 JOIN patient p ON a.patient_id = p.patient_id;
563
564 CREATE INDEX idx_appointment_patient_id ON appointment (patient_id);
565
566 EXPLAIN ANALYZE
567 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
568 FROM appointment a
569 JOIN patient p ON a.patient_id = p.patient_id;
```

The Messages pane shows the output: "CREATE INDEX" and "Query returned successfully in 165 msec."

**Bottom Screenshot:** The same SQL code is shown, but the Data Output pane displays the Query Plan:

```
561 FROM appointment a
562 JOIN patient p ON a.patient_id = p.patient_id;
563
564 CREATE INDEX idx_appointment_patient_id ON appointment (patient_id);
565
566 EXPLAIN ANALYZE
567 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
568 FROM appointment a
569 JOIN patient p ON a.patient_id = p.patient_id;
```

The Data Output pane shows the following query plan details:

| Step | Plan                                                                                                   |
|------|--------------------------------------------------------------------------------------------------------|
| 1    | Hash Join (cost=1.27..12.49 rows=12 width=244) (actual time=0.034..0.037 rows=12 loops=1)              |
| 2    | Hash Cond: (p.patient_id = a.patient_id)                                                               |
| 3    | Seq Scan on patient p (cost=0.00..10.80 rows=80 width=240) (actual time=0.011..0.011 rows=10 loops=1)  |
| 4    | Hash (cost=1.12..1.12 rows=12 width=8) (actual time=0.014..0.014 rows=12 loops=1)                      |
| 5    | Buckets: 1024 Batches: 1 Memory Usage: 9kB                                                             |
| 6    | Seq Scan on appointment a (cost=0.00..1.12 rows=12 width=8) (actual time=0.009..0.010 rows=12 loops=1) |
| 7    | Planning Time: 0.387 ms                                                                                |
| 8    | Execution Time: 0.054 ms                                                                               |

A green status bar at the bottom indicates: "Successfully run. Total query runtime: 74 msec. 8 rows affected."

**e. Select query with aggregation**

The screenshot shows the PgAdmin interface with a SQL query editor. The query is as follows:

```
549
550 EXPLAIN ANALYZE
551 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
552 FROM appointment a
553 JOIN patient p ON a.patient_id = p.patient_id;
554
555 EXPLAIN ANALYZE SELECT COUNT(*) FROM appointment;
556
557 EXPLAIN ANALYZE SELECT AVG(fees) FROM appointment WHERE status = 'Scheduled';
558
559
560
561
562
```

The query is executed successfully, and the results are displayed in the Data Output pane. The query plan shows the following steps:

| Step | Operation               | Cost                              | Rows | Width | Actual Time | Actual Rows | Actual Width | Loops |
|------|-------------------------|-----------------------------------|------|-------|-------------|-------------|--------------|-------|
| 1    | Aggregate               | (cost=1.15..1.16 rows=1 width=8)  | 1    | 8     | 0.014       | 0.014       | 8            | 1     |
| 2    | Seq Scan on appointment | (cost=0.00..1.12 rows=12 width=0) | 12   | 0     | 0.009       | 0.010       | 0            | 1     |
| 3    | Planning Time           |                                   |      |       | 0.085 ms    |             |              |       |
| 4    | Execution Time          |                                   |      |       | 0.034 ms    |             |              |       |

The status bar indicates: Successfully run. Total query runtime: 64 msec. 4 rows affected.

The screenshot shows the PgAdmin interface with a SQL query editor. The query is as follows:

```
549
550 EXPLAIN ANALYZE
551 SELECT a.appointment_id, p.patient_id, p.fname, p.lname
552 FROM appointment a
553 JOIN patient p ON a.patient_id = p.patient_id;
554
555 EXPLAIN ANALYZE SELECT COUNT(*) FROM appointment;
556
557 EXPLAIN ANALYZE SELECT AVG(fees) FROM appointment WHERE status = 'Scheduled';
558
559
560
561
562
```

The query is executed successfully, and the results are displayed in the Data Output pane. The query plan shows the following steps:

| Step | Operation                                  | Cost                              | Rows | Width | Actual Time | Actual Rows | Actual Width | Loops |
|------|--------------------------------------------|-----------------------------------|------|-------|-------------|-------------|--------------|-------|
| 1    | Aggregate                                  | (cost=1.15..1.16 rows=1 width=32) | 1    | 32    | 0.020       | 0.020       | 32           | 1     |
| 2    | Seq Scan on appointment                    | (cost=0.00..1.15 rows=1 width=16) | 1    | 16    | 0.011       | 0.012       | 16           | 7     |
| 3    | Filter: ((status).text = 'Scheduled').text |                                   |      |       |             |             |              |       |
| 4    | Rows Removed by Filter: 5                  |                                   |      |       |             |             |              |       |
| 5    | Planning Time                              |                                   |      |       | 0.181 ms    |             |              |       |
| 6    | Execution Time                             |                                   |      |       | 0.033 ms    |             |              |       |

The status bar indicates: Successfully run. Total query runtime: 92 msec. 6 rows affected.

### **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** stores keys and data in both internal and leaf nodes, leading to varied search paths. In contrast, a **B+-tree** keeps data only in leaf nodes, while internal nodes store keys for navigation, ensuring uniform access time. Additionally, B+-trees have linked leaf nodes, enabling faster range queries. Due to efficient search performance and optimized disk I/O, B+-trees are preferred for database indexing and file systems.

Preferred because-

- **Efficient Range Queries:** Since leaf nodes are linked, B+-trees allow sequential access to records, making them ideal for range queries (e.g., retrieving records in a sorted order).
- **Uniform Access Time:** In B+-trees, all searches fetch data from leaf nodes, leading to more predictable and optimized disk I/O.
- **Better Space Utilization:** Internal nodes only store keys, making them more compact and reducing the height of the tree, leading to faster searches.
- **Efficient Insertion & Deletion:** Since modifications mostly happen at the leaf level, restructuring is minimized, leading to better performance in dynamic databases.

### **Conclusion:**

The above experiment deals with creation of relevant indexing for different tables in the database, and analysing the statistics with and without use of indexing.