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:

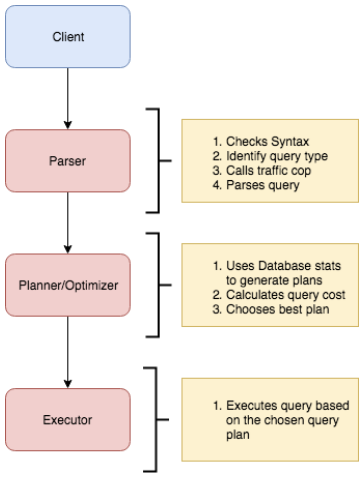
A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

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:

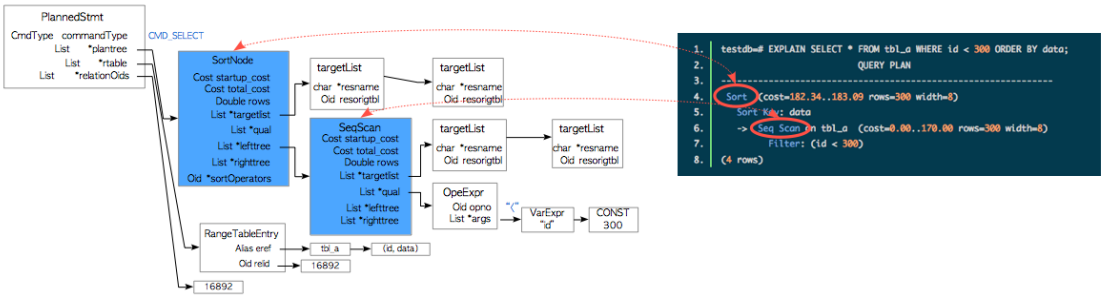
This command displays the execution plan that the PostgreSQL/MySQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms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

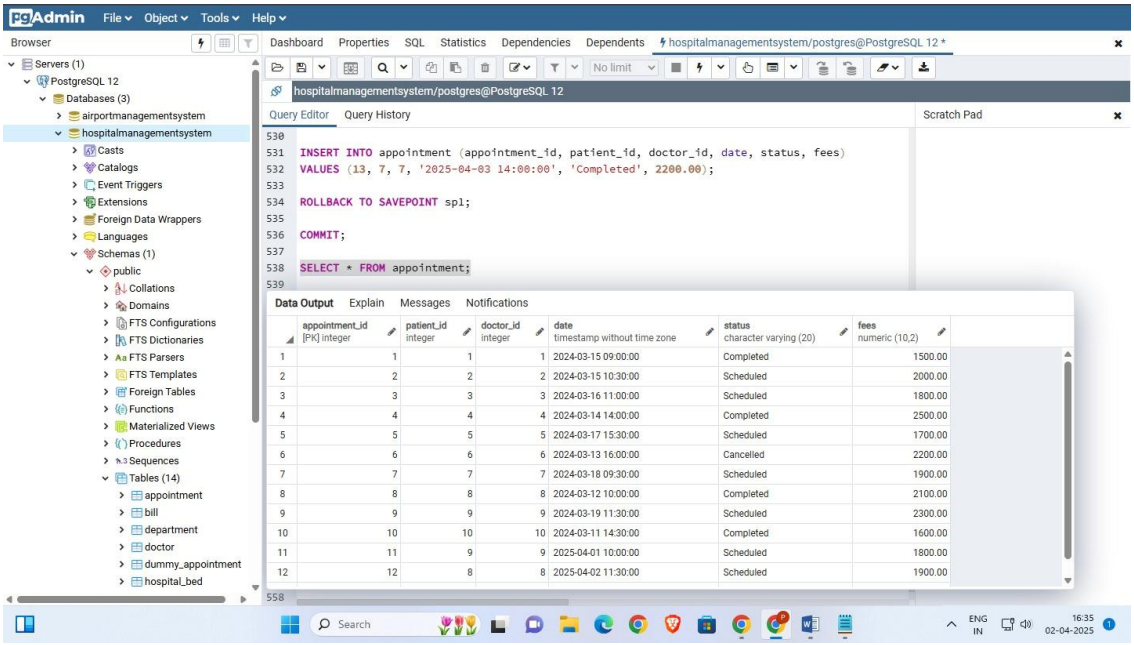
```
INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com'),
('Charlie', 'charlie@example.com');
```

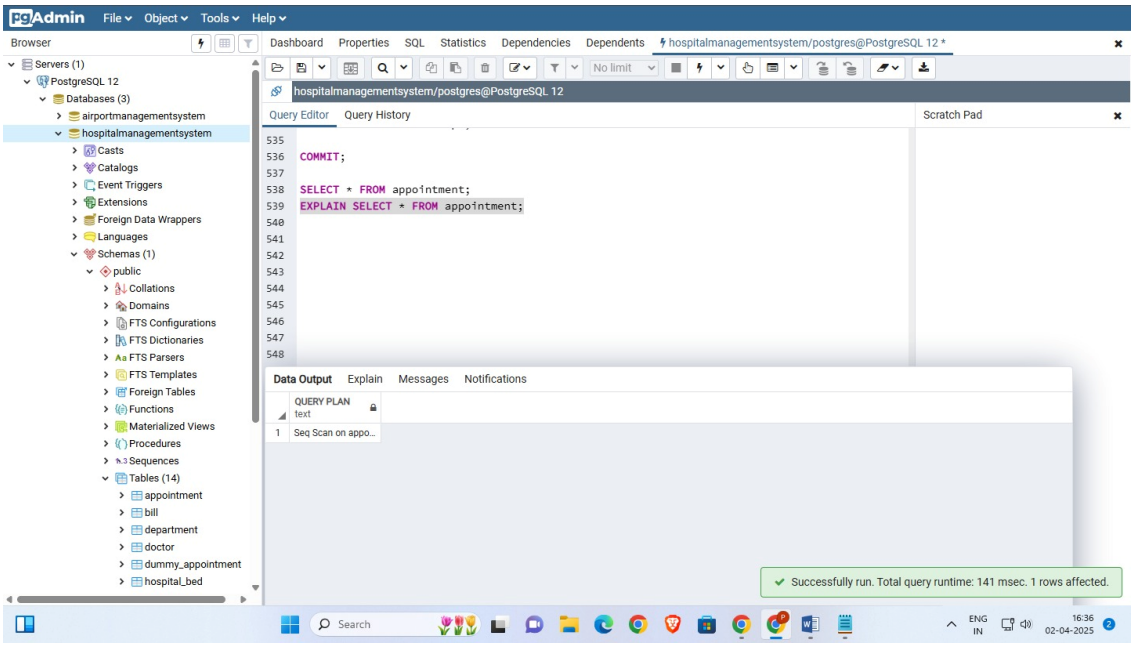
```
EXPLAIN ANALYZE  SELECT * FROM users WHERE name = 'Alice';
```

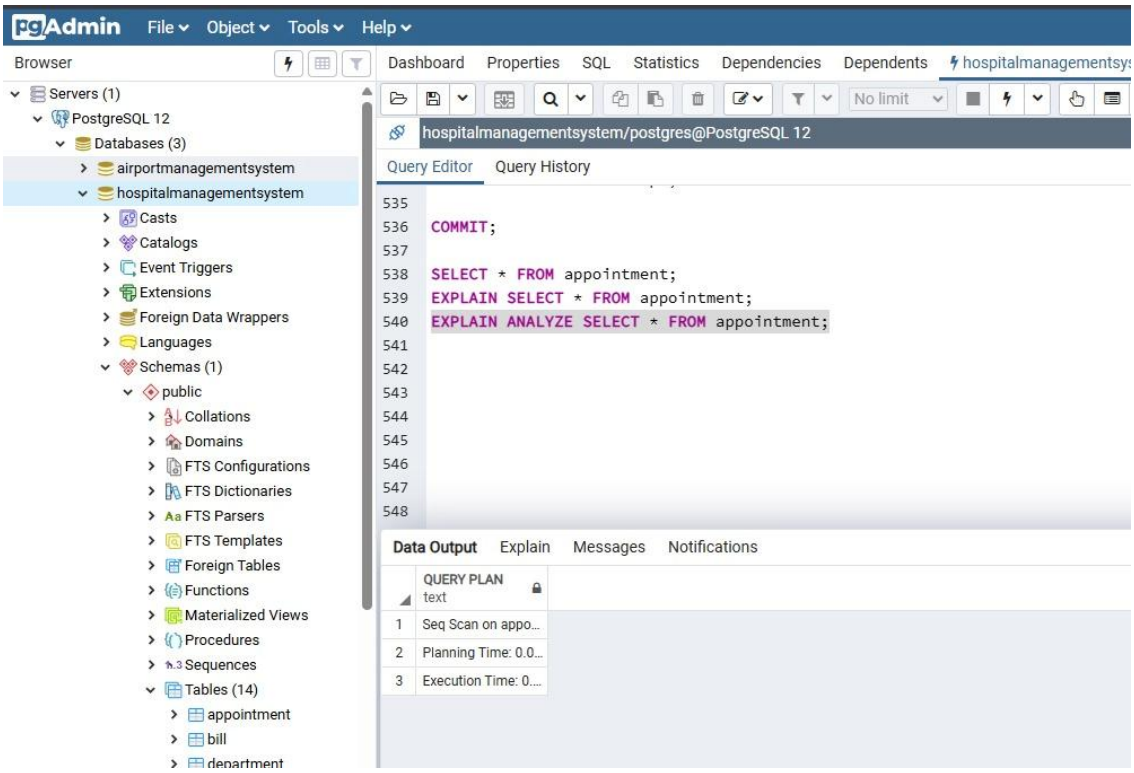
```
CREATE INDEX idx_users_name ON users (name);
```

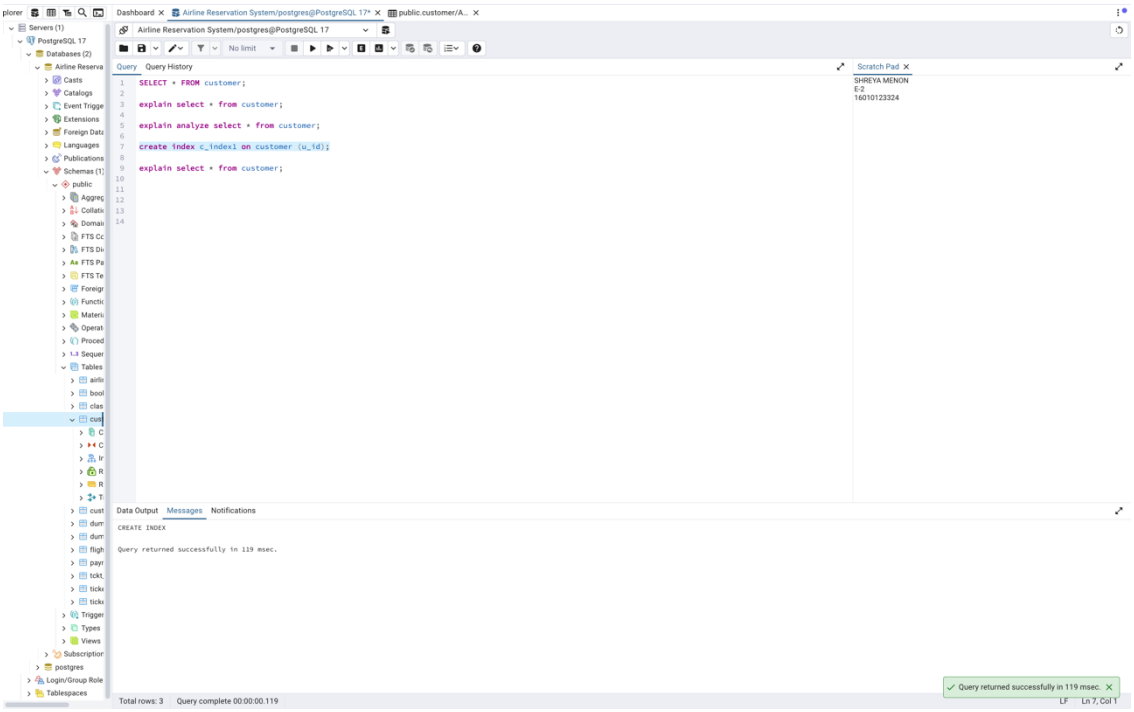
```
EXPLAIN ANALYZE SELECT * FROM users WHERE name = 'Alice';
```

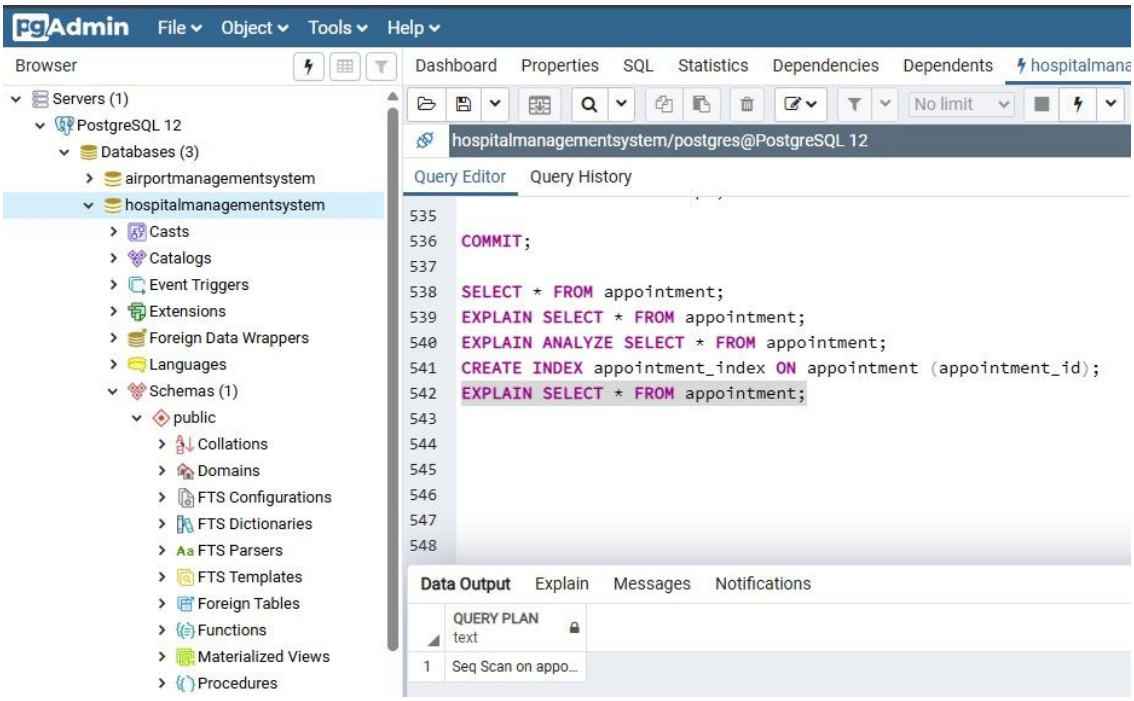
a. Simple select query



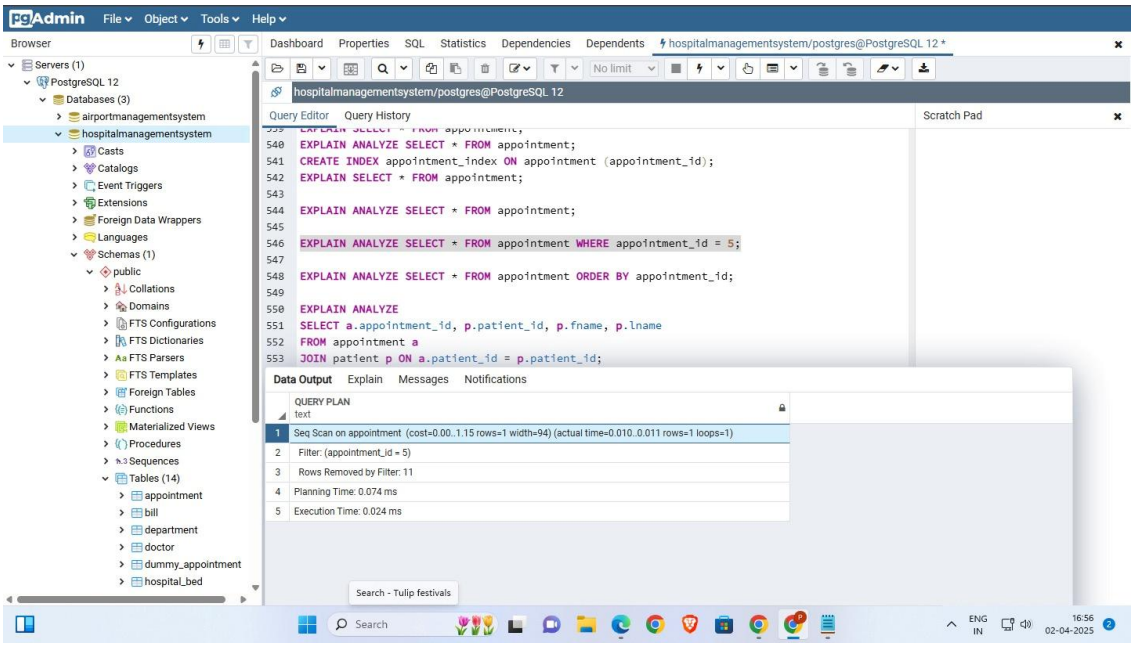



pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
hospitalmanagementsys
Servers (1)
PostgreSQL 12
Databases (3)
airportmanagementsystem
hospitalmanagementsystem
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Schemas (1)
public
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Procedures
Sequences
Tables (14)
appointment
bill
department
No limit
hospitalmanagementsystem/postgres@PostgreSQL 12
Query Editor
Query History
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
Data Output
Explain
Messages
Notifications
QUERY PLAN
text
1 Seq Scan on appo...
2 Planning Time: 0.0...
3 Execution Time: 0....

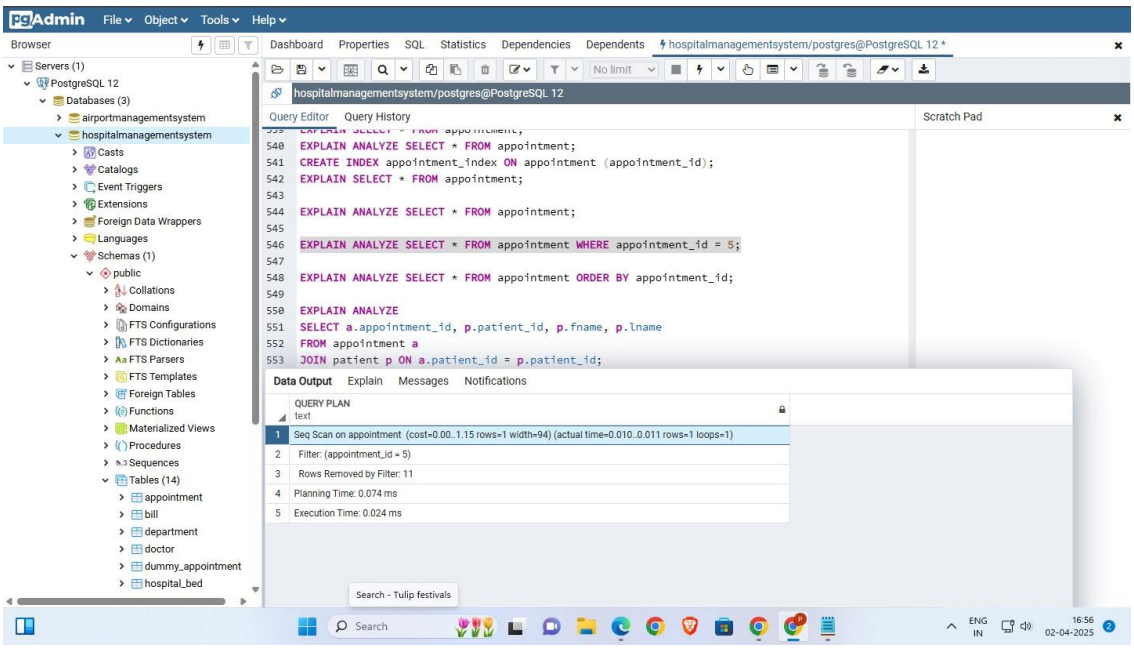

Airline Reservation System/postgres@PostgreSQL 17
Query
Query History
SELECT * FROM customer;
explain select * from customer;
explain analyze select * from customer;
create index c_index1 on customer (u_id);
explain select * from customer;
Scratch Pad
SHREYA MENON
E-2
16010123324
Data Output
Messages
Notifications
CREATE INDEX
Query returned successfully in 119 msec.
Query returned successfully in 119 msec.
Total rows: 3
Query complete 00:00:00.119



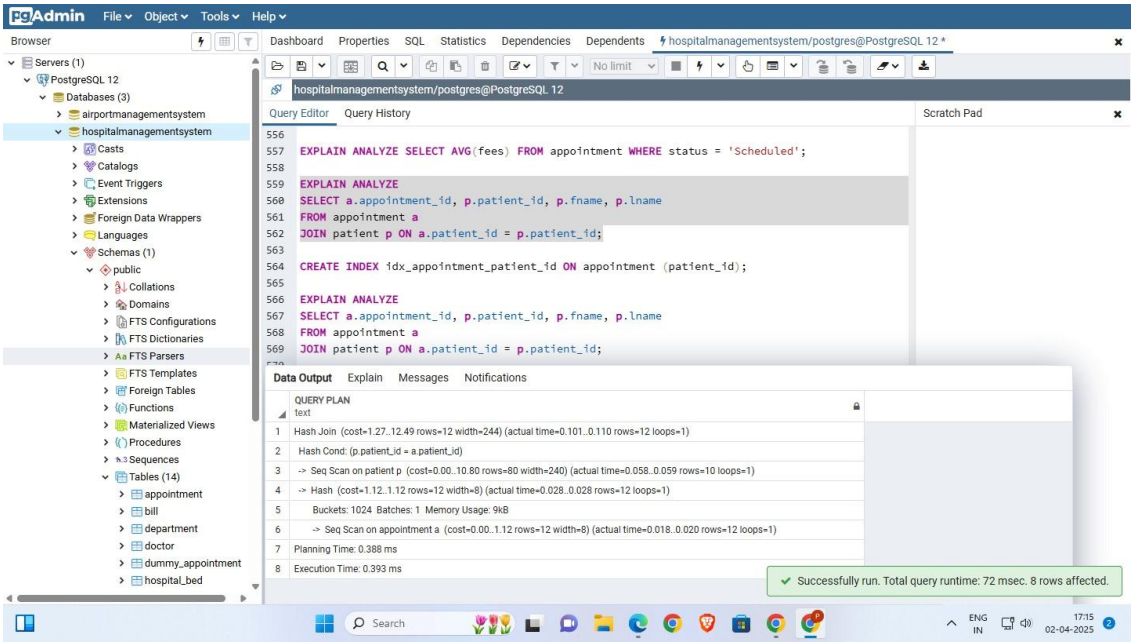
b. Select query with where clause

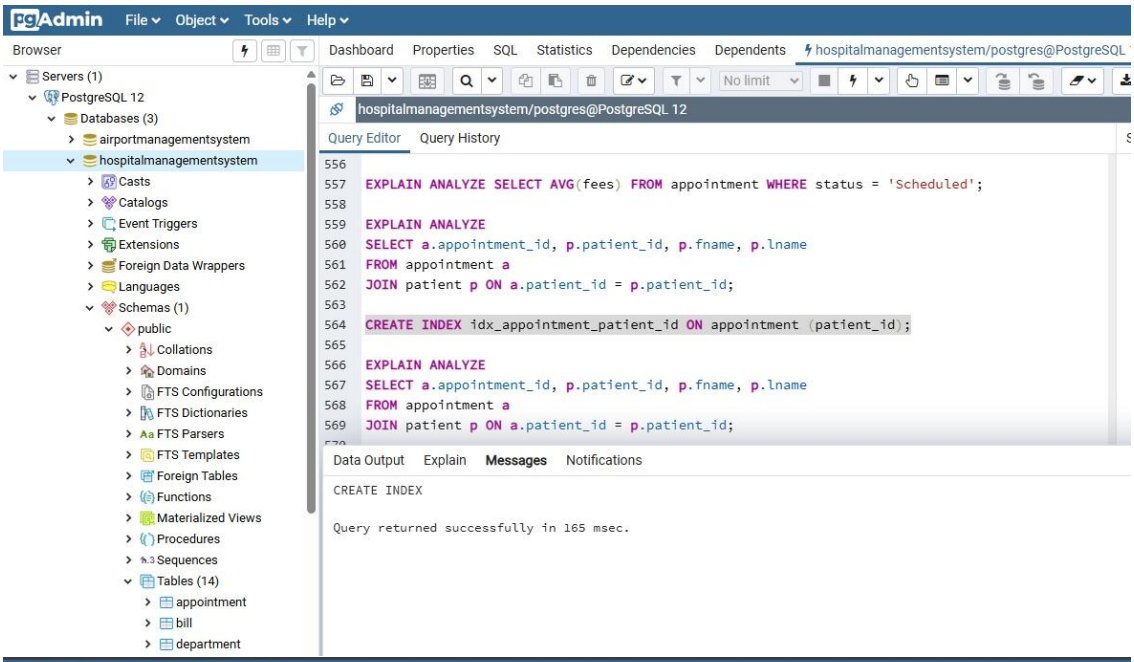


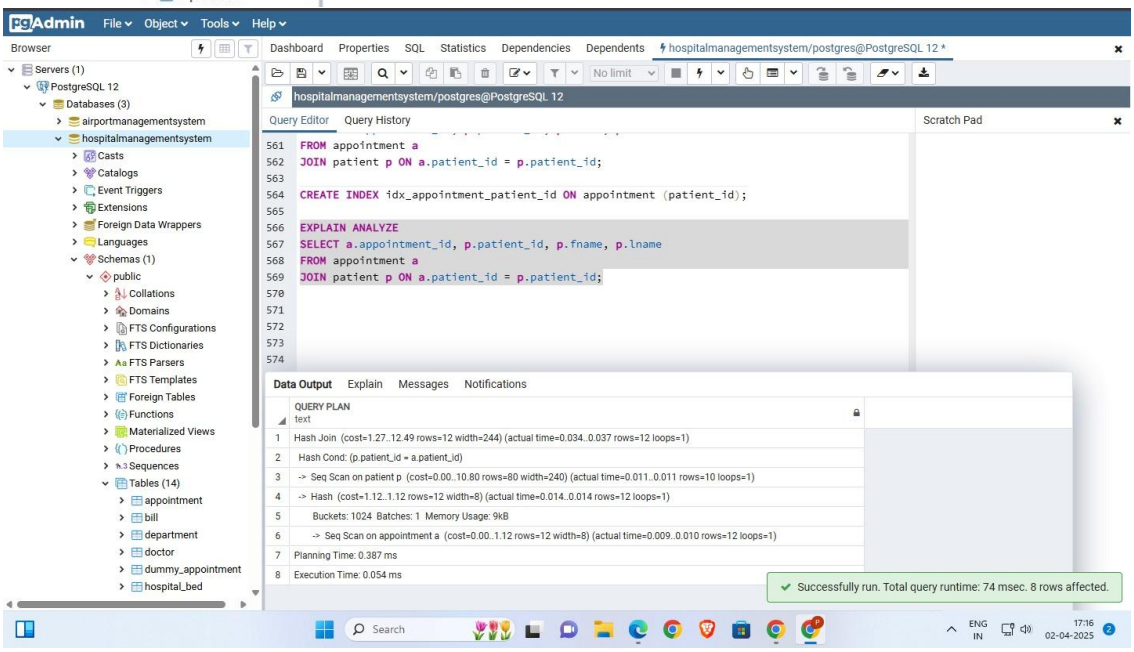
c. Select query with order by query



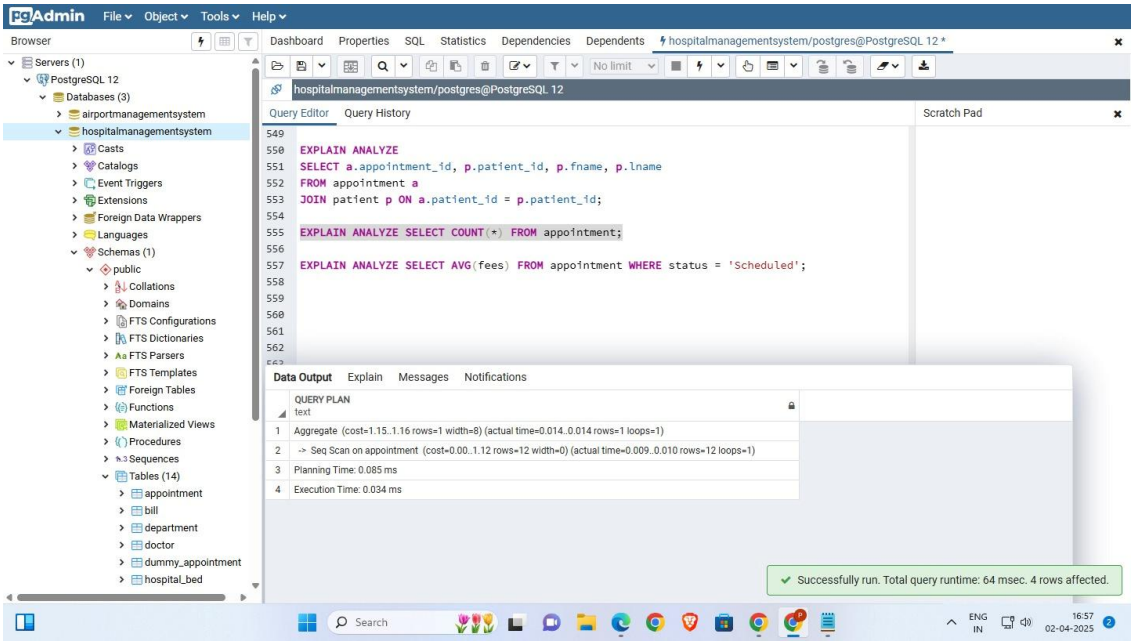
d. Select query with JOIN

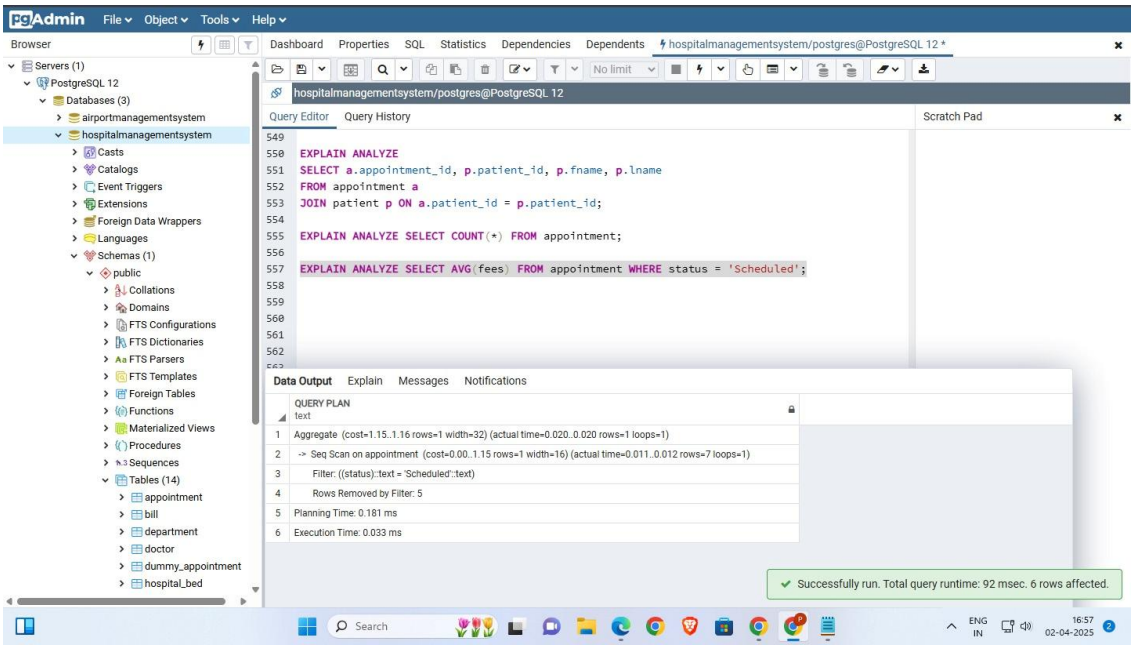



pgAdmin File Object Tools Help
Browser
Dashboard Properties SQL Statistics Dependencies Dependents hospitalmanagementsystem/postgres@PostgreSQL 12
Servers (1)
PostgreSQL 12
Databases (3)
airportmanagementsystem
hospitalmanagementsystem
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Schemas (1)
public
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Procedures
Sequences
Tables (14)
appointment
bill
department
hospitalmanagementsystem/postgres@PostgreSQL 12
Query Editor Query History
556
557 EXPLAIN ANALYZE SELECT AVG(fees) FROM appointment WHERE status = 'Scheduled';
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;
Data Output Explain Messages Notifications
CREATE INDEX
Query returned successfully in 165 msec.


pgAdmin File Object Tools Help
Browser
Dashboard Properties SQL Statistics Dependencies Dependents hospitalmanagementsystem/postgres@PostgreSQL 12 *
hospitalmanagementsystem/postgres@PostgreSQL 12
Query Editor Query History
Scratch Pad
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;
Data Output Explain Messages Notifications
QUERY PLAN
text
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
Successfully run. Total query runtime: 74 msec. 8 rows affected.

e. Select query with aggregation





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.