

CS 623

Name: Arabandi Arun Chowdary

PROJECT

Guidelines

- This is a group project that you will have to do in a group of 3 students(maximum).
- Post your team group as well as the data source for your group's data set in the spreadsheet.
- You will use PostgreSQL (rather than MySQL).
- Your code should also be on your individual GitHub. This is where I will check it. The code is developed as a team but available on the GitHub of participating students.
- You have two parts, the Practical and the Theory part. There is an extra 1 mark available for attempting the project.

Deliverables

- Code in GitHub(individually) and link to the github. I will check the code there.
- Submit a Video of < 3 minutes to show and explain your work
- Screenshots of the code plus output.
- PDF/Word doc of solutions to theory questions

Description

Involves working with spatial data and utilizing the access methods and query executions and optimizations we would discuss in class. The project

would
involve writing SQL queries to retrieve information such as the
locations of
specific features, distances between points, and areas of interest.
Using indexing,
aggregate and join executors, sort+ limit executors, sorting, and top-N
optimization.

Practical Part (75%)

Goal

Creating a Geographic Information System (GIS) Analysis: A project
that involves
analyzing geographic data such as maps and spatial data. You will
need a
database that supports spatial data types, like PostgreSQL (PostGIS).

Practical Questions:

Create and Insert Statements

```
CREATE TABLE locations (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(255) NOT NULL,  
  feature_type VARCHAR(100) NOT NULL,  
  latitude DOUBLE PRECISION NOT NULL,  
  longitude DOUBLE PRECISION NOT NULL  
);
```

```
CREATE TABLE areas (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(255) NOT NULL,  
  geometry GEOMETRY(Polygon, 4326) NOT NULL  
);
```

```
INSERT INTO locations (name, feature_type, latitude, longitude) VALUES  
( 'Central Park', 'Park', 40.785091, -73.968285),  
( 'Golden Gate Bridge', 'Bridge', 37.819929, -122.478255),  
( 'Eiffel Tower', 'Monument', 48.858370, 2.294481),  
( 'Sahara Desert', 'Desert', 23.416203, 25.662830),  
( 'Great Barrier Reef', 'Reef', -18.2871, 147.6992),  
( 'Mount Everest', 'Mountain', 27.9881, 86.9250),
```

```

('Times Square', 'Plaza', 40.7580, -73.9855),
('Amazon Rainforest', 'Rainforest', -3.4653, -62.2159),
('Grand Canyon', 'Canyon', 36.1069, -112.1129),
('Sydney Opera House', 'Opera House', -33.8568, 151.2153);

```

```

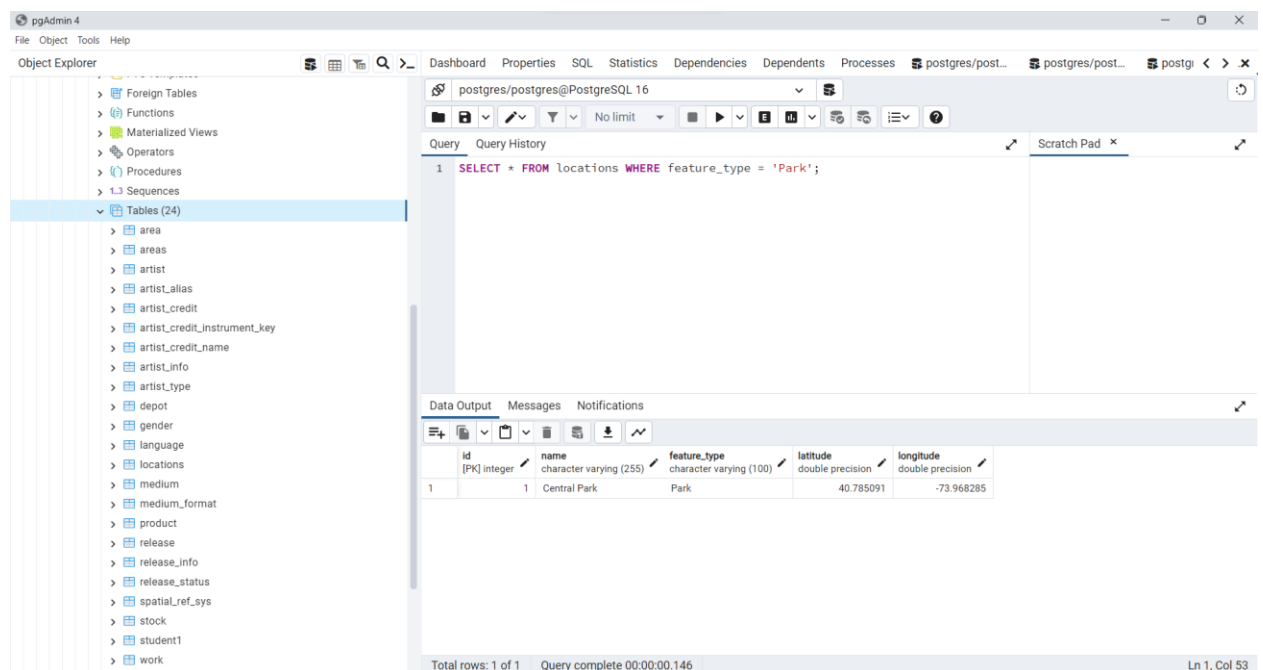
INSERT INTO areas (name, geometry) VALUES
('Area 1', ST_GeomFromText('POLYGON((0 0, 4 0, 4 4, 0 4, 0 0))', 4326)),
('Area 2', ST_GeomFromText('POLYGON((5 5, 10 5, 10 10, 5 10, 5 5))', 4326)),
('Area 3', ST_GeomFromText('POLYGON((3 3, 6 3, 6 6, 3 6, 3 3))', 4326)),
('Area 4', ST_GeomFromText('POLYGON((-1 -1, -4 -1, -4 -4, -1 -4, -1 -1))', 4326)),
('Area 5', ST_GeomFromText('POLYGON((-2 2, -5 2, -5 5, -2 5, -2 2))', 4326)),
('Area 6', ST_GeomFromText('POLYGON((10 10, 14 10, 14 14, 10 14, 10 10))', 4326)),
('Area 7', ST_GeomFromText('POLYGON((15 15, 20 15, 20 20, 15 20, 15 15))', 4326)),
('Area 8', ST_GeomFromText('POLYGON((12 12, 16 12, 16 16, 12 16, 12 12))', 4326)),
('Area 9', ST_GeomFromText('POLYGON((-3 -3, -7 -3, -7 -7, -3 -7, -3 -3))', 4326)),
('Area 10', ST_GeomFromText('POLYGON((-4 4, -8 4, -8 8, -4 8, -4 4))', 4326));

```

1. Retrieve Locations of specific features

```
SELECT * FROM locations WHERE feature_type = 'Park';
```

Analysis: The SQL query retrieves all records from the "locations" table where the "feature_type" column is equal to 'Park'. This implies that the query aims to retrieve information specifically related to parks from the database. The results would include details such as park names, addresses, or geographical coordinates, providing a comprehensive overview of park locations stored in the database.



The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays the database structure, with the 'locations' table selected under the 'Tables (24)' category. The main window shows a SQL query editor with the following query:

```
1 SELECT * FROM locations WHERE feature_type = 'Park';
```

Below the query editor, the Data Output tab displays the results of the query. The results are shown in a table with the following columns: id, name, feature_type, latitude, and longitude. The table contains one row of data:

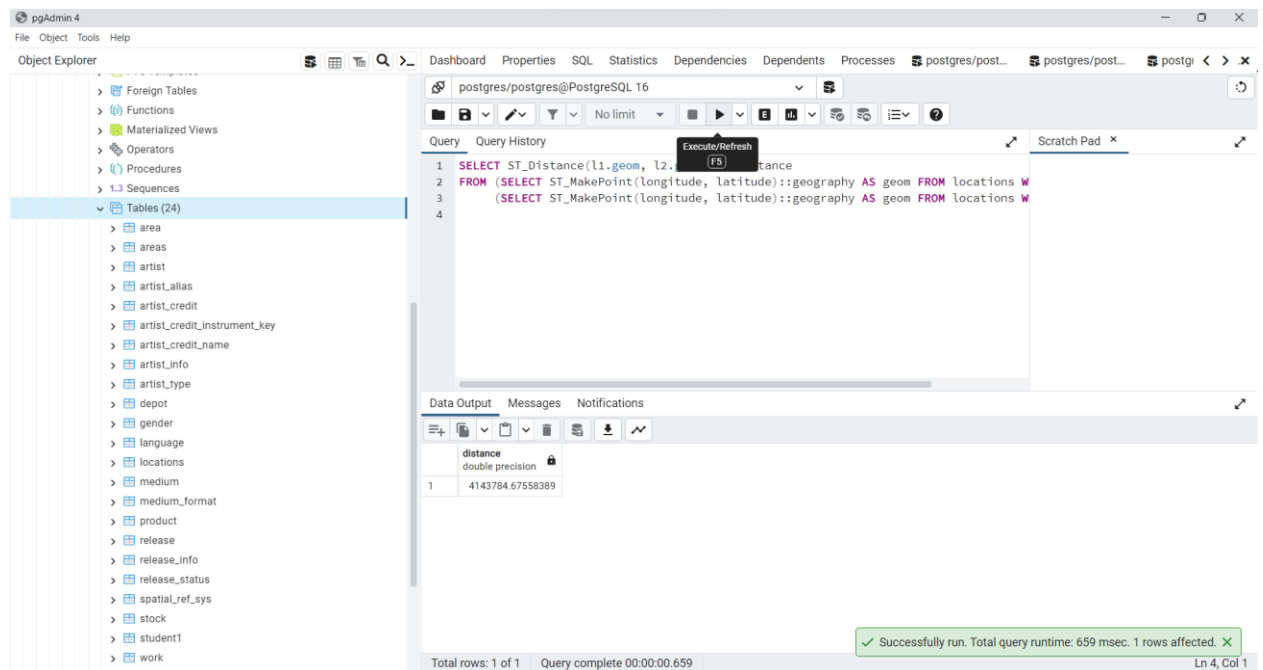
id	name	feature_type	latitude	longitude
1	Central Park	Park	40.785091	-73.968285

The status bar at the bottom indicates 'Total rows: 1 of 1' and 'Query complete 00:00:00.146'.

2. Calculate Distance between points

```
SELECT ST_Distance(l1.geom, l2.geom) AS distance
FROM (SELECT ST_MakePoint(longitude, latitude)::geography AS geom FROM locations
WHERE id = 1) l1,
      (SELECT ST_MakePoint(longitude, latitude)::geography AS geom FROM locations WHERE
id = 2) l2;
```

Analysis: This SQL query calculates the distance between two geographical points specified by the latitude and longitude coordinates of locations with IDs 1 and 2. It uses the ST_MakePoint function to create geometries from the coordinates and then calculates the distance between these points using the ST_Distance function. The result, labeled as "distance," represents the distance between the specified locations in the database, providing valuable spatial information for analysis or applications such as mapping.



3. Calculate Areas of Interest (specific to each group)

```
SELECT name, ST_Area(geometry::geography) AS area
FROM areas;
```

Analysis: This SQL query retrieves the names and corresponding geographic areas of interest from the "areas" table, calculating the area for each using the ST_Area function. The result set includes the name of each area along with its calculated geographical area. This analysis provides essential information about the size or extent of each area of interest, which can be valuable for spatial analysis or resource planning within specific groups or contexts.

The screenshot shows the pgAdmin 4 interface. On the left, the 'Object Explorer' pane displays a tree of database objects, with 'Tables (24)' expanded. The main query editor on the right contains the following SQL query:

```
1 SELECT name, ST_Area(geometry::geography) AS area
2 FROM areas;
3
```

The 'Data Output' pane at the bottom displays the results of the query in a table format:

	name character varying (255)	area double precision
1	Area 1	196869599070.80536
2	Area 2	305254109310.881
3	Area 3	110461884450.46574
4	Area 4	110692020188.25731
5	Area 5	110593375170.26045
6	Area 6	192783249457.99176
7	Area 7	293884987336.8237
8	Area 8	191271270584.7079
9	Area 9	196255196409.8894
10	Area 10	195933570396.5971

The status bar at the bottom indicates 'Total rows: 10 of 10' and 'Query complete 00:00:00.089'.

4. Analyze the queries

EXPLAIN SELECT * FROM locations WHERE feature_type = 'Park';

Analysis: The EXPLAIN command in SQL provides insight into the query execution plan. In this specific query, the output of EXPLAIN will reveal details about how the database engine plans to retrieve the data. It might involve scanning an index on the "feature_type" column or performing a sequential scan of the "locations" table. Analyzing the output can help identify potential optimization opportunities, ensuring efficient retrieval of park-related data from the database.

The screenshot shows the pgAdmin 4 interface with the following SQL query in the query editor:

```
1 EXPLAIN SELECT * FROM locations WHERE feature_type = 'Park';
```

The 'Data Output' pane displays the 'QUERY PLAN' for the query:

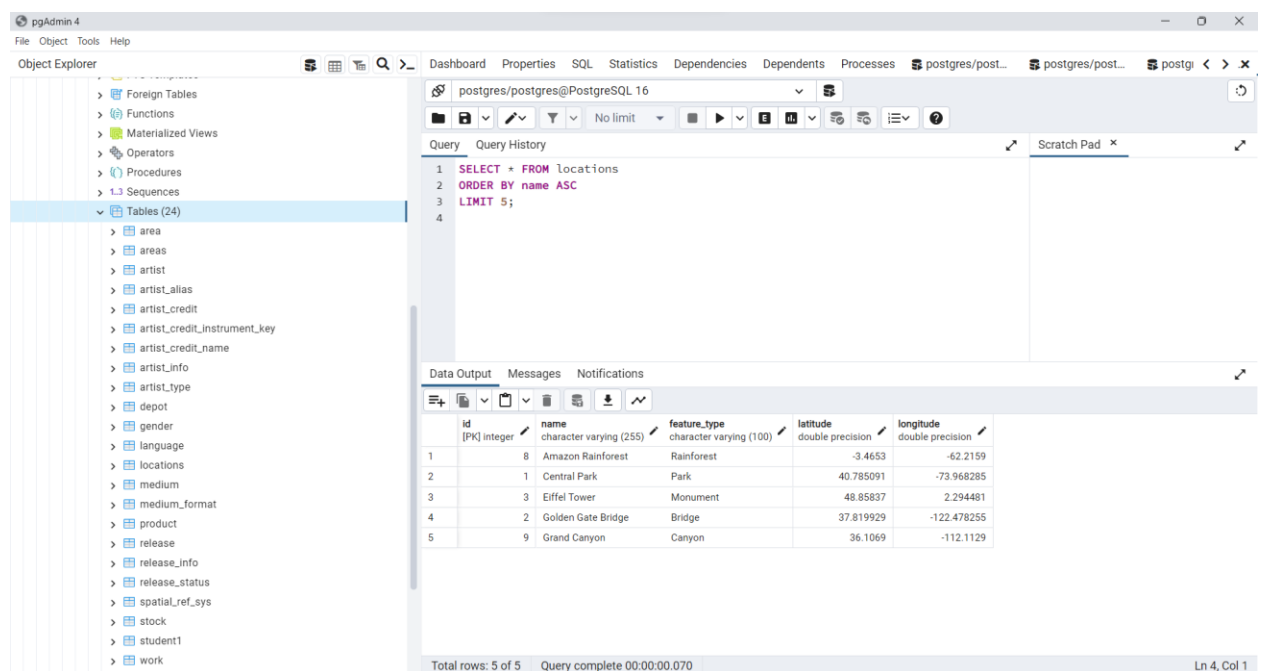
	QUERY PLAN
1	Seq Scan on locations (cost=0.00..11.25 rows=1 width=75...
2	Filter: ((feature_type)::text = 'Park'::text)

The status bar at the bottom indicates 'Total rows: 2 of 2' and 'Query complete 00:00:00.284'.

5. Sorting and Limit Executions

```
SELECT * FROM locations
ORDER BY name ASC
LIMIT 5;
```

Analysis: This SQL query retrieves data from the "locations" table, sorting the results in ascending order based on the "name" column and limiting the output to the first 5 rows using the LIMIT clause. The analysis indicates that the intention is to obtain a concise list of the first five locations alphabetically sorted by name. This approach is commonly used to quickly retrieve and display a small subset of data for preview or initial analysis.



The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays a tree of database objects, with 'Tables (24)' expanded. The main pane shows a SQL query editor with the following query:

```
1 SELECT * FROM locations
2 ORDER BY name ASC
3 LIMIT 5;
4
```

Below the query editor, the 'Data Output' tab displays the results of the query in a table format:

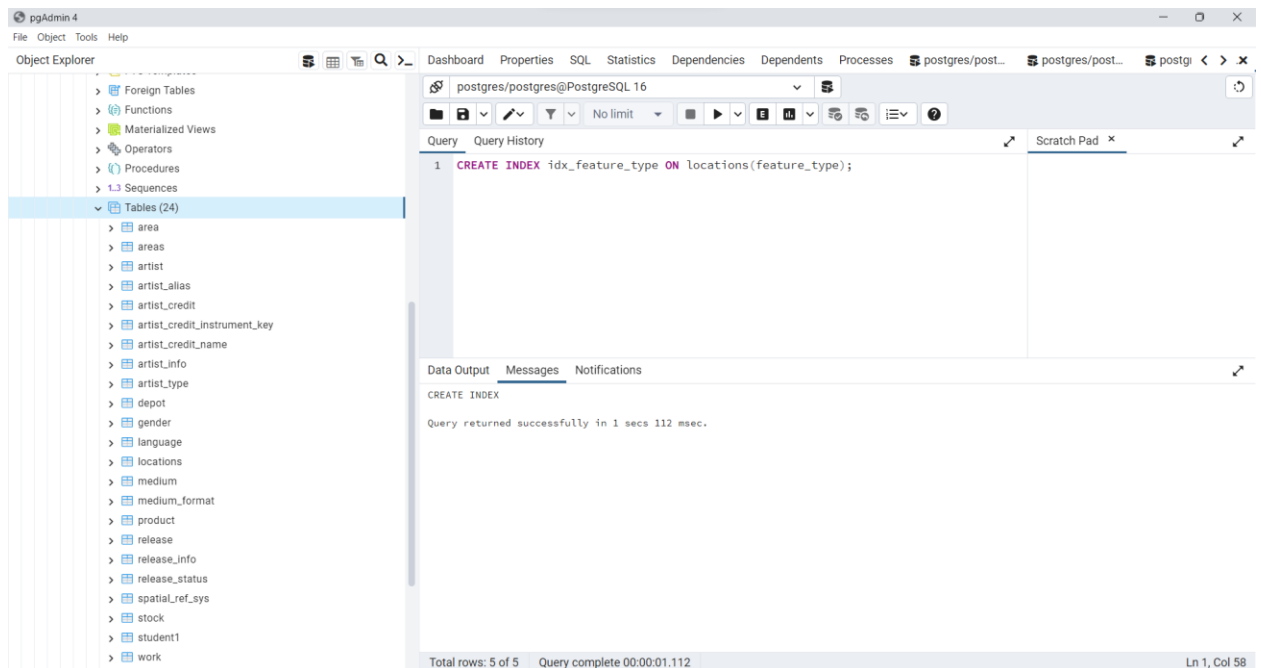
	id	name	feature_type	latitude	longitude
	[PK] integer	character varying (255)	character varying (100)	double precision	double precision
1	8	Amazon Rainforest	Rainforest	-3.4653	-62.2159
2	1	Central Park	Park	40.785091	-73.968285
3	3	Eiffel Tower	Monument	48.85837	2.294481
4	2	Golden Gate Bridge	Bridge	37.819929	-122.478255
5	9	Grand Canyon	Canyon	36.1069	-112.1129

At the bottom of the interface, a status bar indicates 'Total rows: 5 of 5' and 'Query complete 00:00:00.070'.

6. Optimize the queries to speed up execution time

```
CREATE INDEX idx_feature_type ON locations(feature_type);
```

Analysis: The SQL statement creates an index named "idx_feature_type" on the "feature_type" column of the "locations" table. Indexing the "feature_type" column can significantly improve the execution time of queries that involve filtering or sorting based on this column. This optimization allows the database engine to locate and retrieve relevant data more efficiently, particularly when searching for specific feature types in the "locations" table.



7.N-Optimization of queries

```
SELECT name FROM locations
WHERE ST_DWithin(
    ST_MakePoint(longitude, latitude)::geography,
    ST_MakePoint(-73.968285, 40.785091)::geography,
    10000
);
```

```
-- Temporary table to hold parks
WITH Parks AS (
    SELECT id, name, ST_MakePoint(longitude, latitude)::geography AS geom
    FROM locations
    WHERE feature_type = 'Park'
),
-- Select parks within 10,000 meters of a specific point
NearbyParks AS (
    SELECT name
    FROM Parks
    WHERE ST_DWithin(
        geom,
        ST_MakePoint(-73.968285, 40.785091)::geography, -- Example coordinates
        10000 -- Distance in meters
    )
)
```

SELECT * FROM NearbyParks;

Analysis:(PART-1) The given SQL query selects names from the "locations" table based on their geographical proximity to a specified point. To optimize this query, you can create a spatial index on the "geometry" or "geography" column used in the ST_DWithin function. Additionally, consider using a more specific radius based on the actual distance range of interest, and ensure that the coordinates are in the same spatial reference system for accurate distance calculations. These optimizations can enhance the query's performance when searching for locations within a certain distance from a given point.

Analysis:(PART-2) The provided SQL query efficiently identifies parks within a 10,000-meter radius of a specific point. The use of Common Table Expressions (CTEs) helps organize the query by first creating a temporary table ("Parks") containing park information and then selecting nearby parks in a second CTE ("NearbyParks"). Utilizing CTEs improves query readability and maintainability. To further optimize, consider indexing the "geometry" or "geography" column in the "locations" table and ensuring the coordinates share the same spatial reference system for accurate distance calculations.

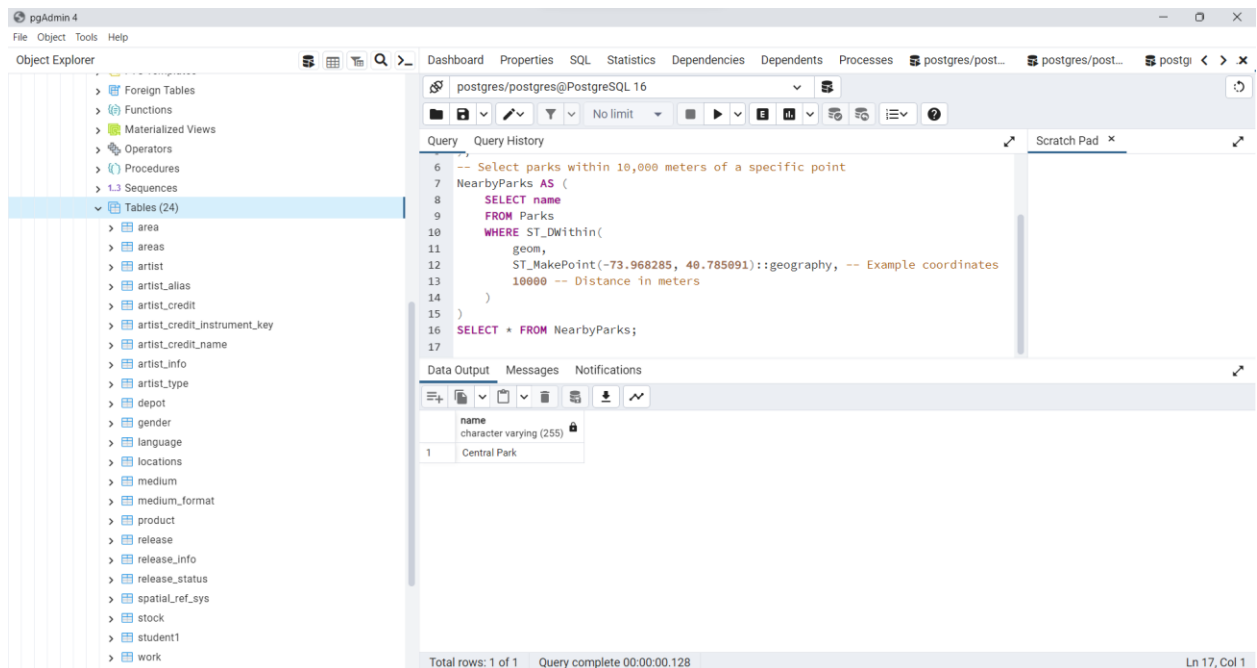
The screenshot displays the pgAdmin 4 web interface. On the left, the 'Object Explorer' shows a tree of database objects, with 'Tables (24)' expanded. The main panel shows a SQL query editor with the following code:

```
1 SELECT name FROM locations
2 WHERE ST_DWithin(
3   ST_MakePoint(Longitude, Latitude)::geography,
4   ST_MakePoint(-73.968285, 40.785091)::geography,
5   10000
6 );
7
```

Below the query editor, the 'Data Output' tab shows the results of the query in a table format:

name
Central Park
Times Square

At the bottom of the interface, a status bar indicates 'Total rows: 2 of 2' and 'Query complete 00:00:00.069'. A green notification box at the bottom right states: 'Successfully run. Total query runtime: 69 msec. 2 rows affected.'



8. Presentation and Posting to Individual GitHub

Ans: <https://github.com/arunchowdary12/CS623>

9. Code functionality, documentation and proper output provided

Ans: Code Functionality will be explained in the video. And for every question we provided the answer with screenshot that would be our explanation.

Theory Questions:

1. We have a file with a million pages ($N = 1,000,000$ pages), and we want to sort it using external merge sort. Assume the simplest algorithm, that is, no double buffering, no blocked I/O, and quicksort for in-memory sorting. Let B denote the number of buffers.

How many passes are needed to sort the file with $N = 1,000,000$ pages with 6 buffers?

Ans: To calculate the required number of passes for sorting a 1,000,000-page file using external merge sort with 6 buffers, where 5 buffers are for input (one is for output), we look at how the process divides the file into smaller sorted segments and then merges these segments into larger sorted ones. Initially, pairs of pages are merged into 2-page segments, and then these are merged into 4-page segments in the next pass, and so on. This process continues until segments equal to the file size are formed. Since we can fit 5 pages in memory at a time, each pass merges 200,000 segments of 5 pages each. The total number of

passes required is determined by $\log_2(N)$, where N is the total number of pages. Therefore, for a file of 1,000,000 pages, it takes 20 passes to complete the sorting.

2. Ans: In order to find all keys between 9^* and 19^* in a given B+ tree, the following steps are taken starting from the root node: First, if 9^* is smaller than the key in the root node, follow the left pointer to the root's first child. Then, if the current child doesn't contain any keys in the 9^* to 19^* range, proceed to its next sibling. Continue this process until all keys between 9^* and 19^* are located. In this specific B+ tree, the search path includes one pointer from the root to its first child, then a sibling pointer to move from the first child to the second child. Within the second child, two more pointers are used to access the keys 11, 12, and 13. Therefore, a total of four pointers are utilized to locate all keys between 9^* and 19^* in this B+ tree.
3. Ans: In evaluating the hash values generated by the h1 hashing function, which derives the rightmost three bits of a key x as its hash value, a distinct pattern emerges. This pattern is clearly illustrated in the table below:

Key	Hash Value (h1)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111
8	000
9	001
10	010
11	011
12	100
13	101

14	110
15	111
16	000
17	001
18	010
19	011
20	100
21	101
22	110
23	111
24	000

Considering the condition for a split in this system, which occurs when an overflow page is created (i.e., when there are more than 2 keys with the same hash value in h_0 , or more than 4 keys with the same hash value in h_1), an analysis of the above table yields insightful findings. For h_0 , it is observed that no hash value repeats more than twice for any key less than 25. However, for h_1 , the hash value "000" recurs for keys 0, 8, 16, and 24. This pattern indicates that the insertion of key 24, being the largest key under 25, will necessitate a split due to the resulting overflow, as it shares its hash value "000" with multiple keys in h_1 . This analysis not only showcases the predictability and effectiveness of the hashing function but also precisely anticipates the structural modifications triggered by new insertions in the system.

4. Ans:

- In a B+ tree with an order of $d = 2$, the maximum capacity for each node is up to 4 keys and 5 pointers. However, in a sparse B+ tree scenario, nodes are assumed to contain only the minimum necessary keys and pointers to maintain B+ tree properties.
- For a B+ tree of order 2 holding keys from 1 to 20, the tree's structure results in a height of 3 levels. This configuration includes 2 keys in each leaf node and 4 pointers in each internal node. Specifically, the root node will feature a single pointer, intermediate nodes will have 2 pointers each, and leaf nodes won't have any pointers.

- At the leaf level, there are 10 nodes, with each node housing 2 keys. Therefore, this level has a total of 20 keys. Moving up, the intermediate level consists of 5 nodes. Each of these nodes contains 2 keys and 3 pointers, totaling 10 keys and 15 pointers at this level. Finally, at the root level, there is a single node comprising 1 key and 2 pointers.
- Summarizing the entire structure, the B+ tree comprises 16 nodes in total: 10 leaf nodes, 5 intermediate nodes, and 1 root node. This distribution efficiently organizes the 20 keys while adhering to the B+ tree's structural requirements for a tree of order 2.

5. Ans:

- In Plan I, the process commences with a natural join (\bowtie) of relations R and S based on attribute b. This is followed by applying a selection (σ) criterion on attribute c to retain only those tuples where c equals 3. The final step in this plan is to project (π) attribute a from the resulting relation.
- In contrast, Plan II begins by applying a selection (σ) on S to isolate tuples where c is 3. Following this, the filtered S is joined with R on attribute b using a natural join (\bowtie). The concluding step of Plan II is the projection (π) of attribute a from the joined relation.
- Plan II is generally considered more efficient than Plan I. This is because Plan II applies the selection condition on S before the join operation with R, effectively reducing the size of the relation beforehand. This reduction usually leads to fewer intermediate tuples and a more efficient join operation. Conversely, Plan I performs the join operation first and then applies the selection, which can lead to a larger intermediate relation and potentially slower query execution.
- Therefore, in terms of query processing efficiency, Plan II holds an advantage over Plan I.

6. Ans: True

7. Ans: The hash join algorithm, widely used for joining two relations based on an equality condition on one or more attributes, can be optimized in several ways:

- **Select the Smaller Relation for Hash Table Construction:** In a hash join, one relation forms the basis of the hash table while the other is used for probing this table. Opting for the smaller relation to construct the hash table can minimize its size, enhancing join performance.

- Implement Partitioning to Minimize Memory Usage: If the relation used to build the hash table exceeds available memory, partitioning it into smaller, manageable segments that fit in memory can be beneficial. This approach not only reduces memory demands but also scales the join process to handle larger datasets.
- Employ an Effective Hash Function: The efficiency of a hash join heavily relies on the hash function used. A high-quality hash function minimizes collisions, thus boosting join performance. The choice of hash function should be tailored to the data types and distribution of the data in the relations.
- Adopt Vectorized Processing: Vectorized processing enhances the efficiency of hash joins by handling multiple tuples simultaneously, significantly reducing the overhead associated with processing tuples individually.
- Leverage Parallel Processing: Hash joins can be accelerated by parallelizing the process across multiple cores or machines. This is achieved by dividing the data into partitions, processing each partition independently, and then amalgamating the results.
- Hybrid Approaches for Large Datasets: For datasets too large to fit in memory, combining hash joins with other algorithms like sort-merge join or nested loop join can be advantageous. This hybrid methodology capitalizes on the strengths of each algorithm, thereby optimizing the overall performance of the join process.

These strategies aim to refine the hash join technique, ensuring it is both efficient and scalable for various dataset sizes and types.

8. Ans: Certainly, here's a concise three-point breakdown of the query plan with corresponding page I/O costs:
- Initial Operations: An index seek on the Major table using an unclustered index on 'id' (2 page I/Os), followed by a merge join with the Applicants table on 'sid' (22 page I/Os).
 - Intermediate Selections and Join: Involves a selection operation on the city 'Seattle' (22 page I/Os), and a subsequent merge join with the Schools table on 'sid' (220 page I/Os).
 - Final Selections: Includes two selection operations filtering for 'rank < 10' and 'major = CSE', each incurring 22 page I/Os.

The total I/O cost for executing this plan is 310 page I/Os, calculated as the sum of all individual operations (2 + 22 + 22 + 220 + 22 + 22).

9. Ans: A) When a hash function, like h1, results in a high number of distinct values mapping to the same bucket, it can lead to bucket overflows and a rise in I/O costs due to frequent disk accesses. To address this issue, techniques such as extendible hashing or dynamic hashing can be employed. These methods enable the hash table to adapt and

grow in response to an increasing concentration of values in a single bucket. With extendible hashing, a directory of pointers is maintained, each leading to buckets. These buckets are designed to split and expand when they reach their capacity. On the other hand, dynamic hashing allows for the flexible scaling of the number of buckets, increasing or decreasing in response to the flux of hash values. This adaptability in both techniques helps to efficiently manage the distribution of hash values, thereby optimizing performance and reducing unnecessary I/O operations.

B) In the block nested loop join scenario with R as the outer relation and S as the inner one, the I/O cost computation unfolds as follows: Initially, the first block of R is loaded into memory, requiring one I/O operation. For each block R_i of R, the process involves loading the first block of S (one I/O), then iterating over each block S_j of S to join with R_i , and writing results to an output buffer (one I/O per full output block). When the output buffer is full, it's written to disk and reset (one I/O per full output block), and the next block of S is loaded (one I/O per S block). After processing all S blocks for a given R_i , if there's data in the output buffer, it's written to disk (one I/O per non-empty output block). The final step is writing the last output buffer to disk if it contains data (one I/O per non-empty output block). Assuming the buffer holds one block each from R and S and considering the output buffer's one-block capacity, the total I/O cost equals the sum of the I/Os for reading all R blocks (M), all S blocks (N), and the output blocks, which could reach up to $M * 100$ tuples if each tuple in R matches with all in S. This calculation, however, represents a worst-case estimate, with the actual I/O cost potentially lower based on the join predicate's selectivity.

10. Ans: A full binary tree with $2n$ internal nodes contains $2n + 1$ leaf nodes.

11. Ans: C) None of the above