**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)),

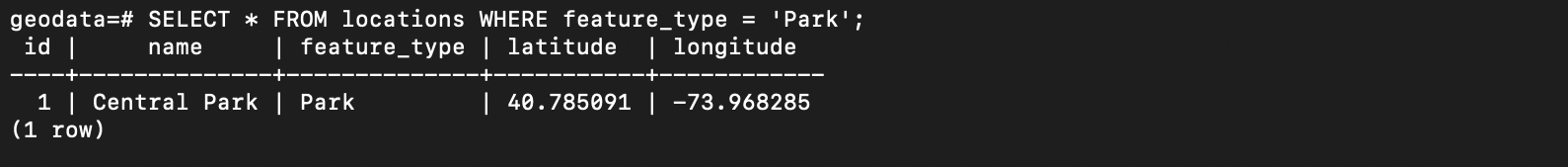
('Area 9', ST\_GeomFromText('POLYGON((-3 -3, -7 -3, -7 -7, -3 -7, -3 -3))', 4326)),

('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';

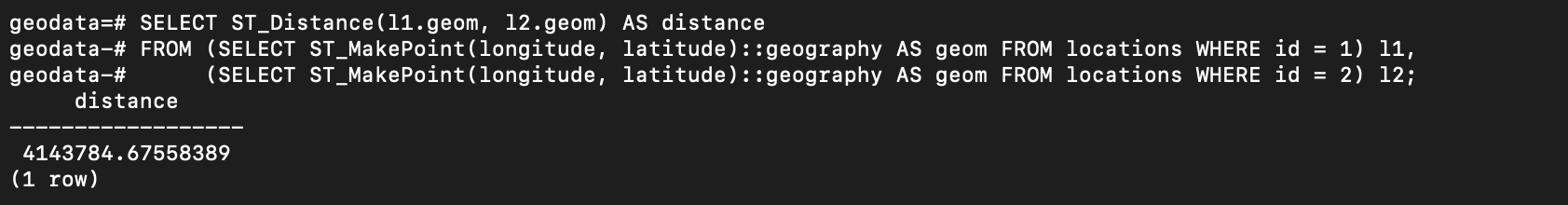


**2. Calculate Distance between points**

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;



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

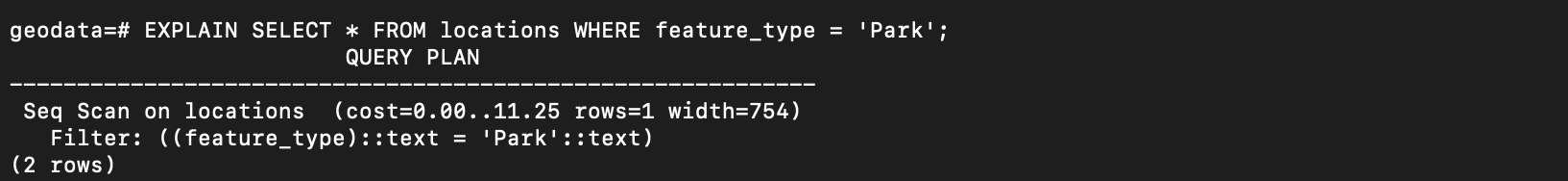
SELECT name, ST\_Area(geometry::geography) AS area

FROM areas;



**4. Analyze the queries**

EXPLAIN SELECT \* FROM locations WHERE feature\_type = 'Park';

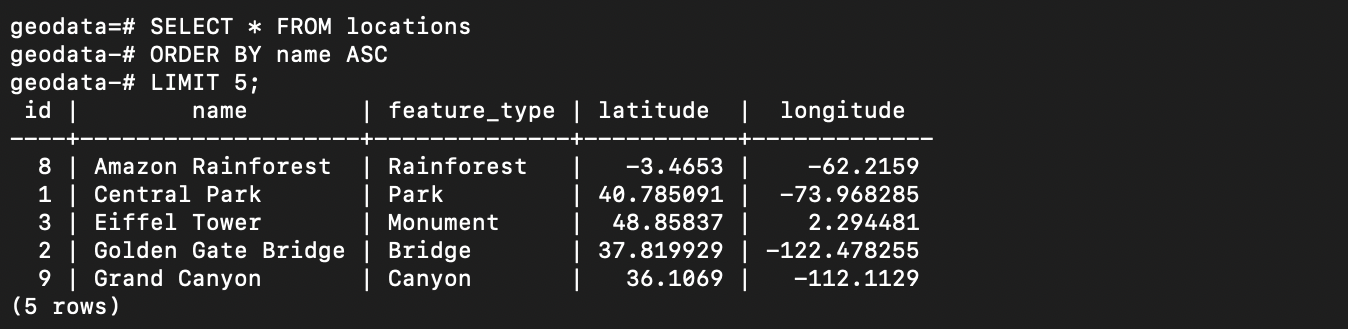


**5. Sorting and Limit Executions**

SELECT \* FROM locations

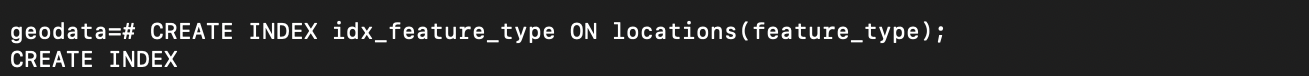
ORDER BY name ASC

LIMIT 5;



**6. Optimize the queries to speed up execution time**

CREATE INDEX idx\_feature\_type ON locations(feature\_type);



**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,

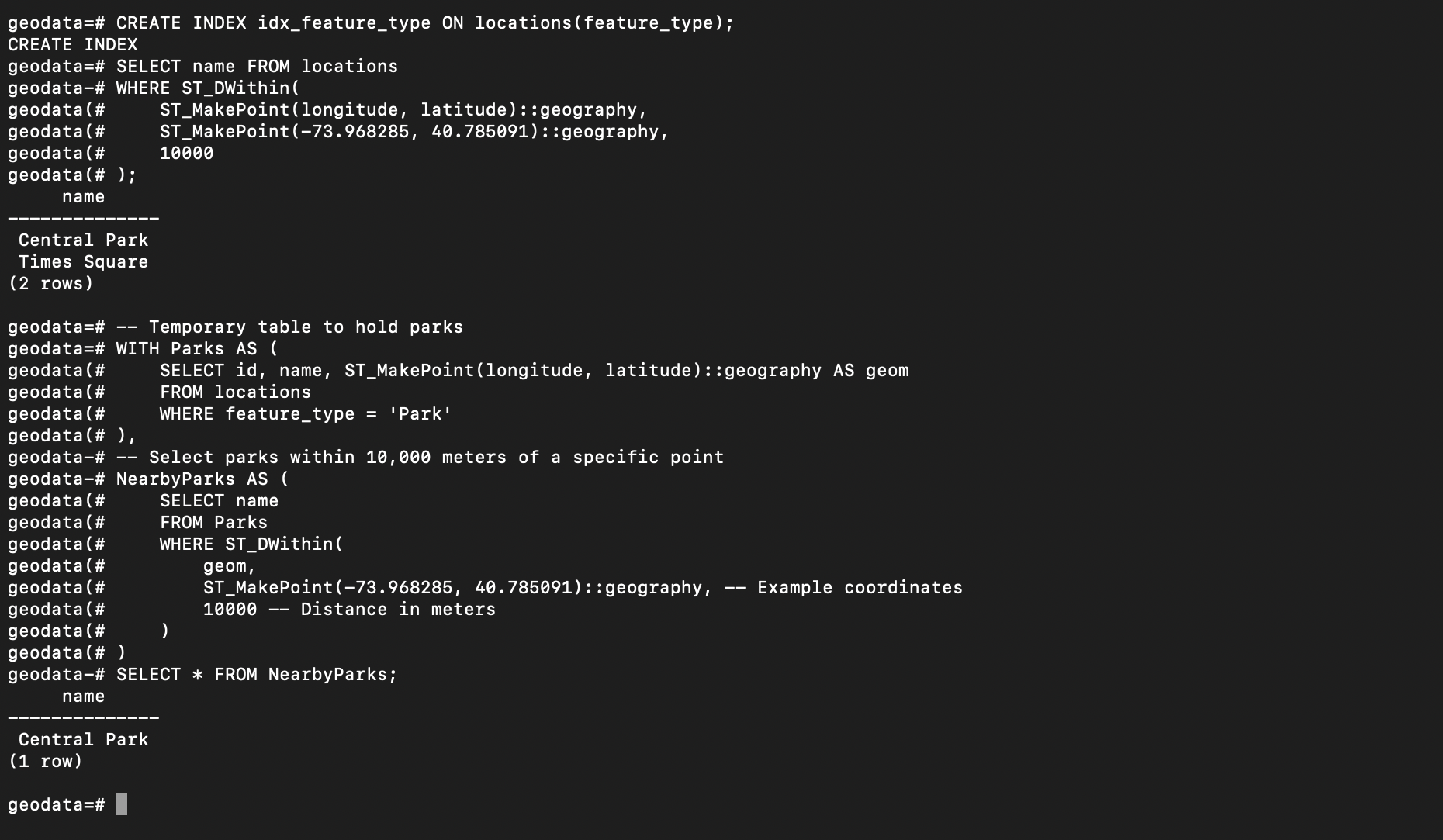
ST\_MakePoint(-73.968285, 40.785091)::geography, -- Example coordinates

10000 -- Distance in meters

)

)

SELECT \* FROM NearbyParks;



**Theory Questions:**

1. 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. 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. 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 h0, or more than 4 keys with the same hash value in h1), an analysis of the above table yields insightful findings. For h0, it is observed that no hash value repeats more than twice for any key less than 25. However, for h1, 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 h1. 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.

* 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.
* In Plan I, the process commences with a natural join (⋈) 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 (⋈). 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.

1. True
2. 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.

1. 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).

1. 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 Ri of R, the process involves loading the first block of S (one I/O), then iterating over each block Sj of S to join with Ri, 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 Ri, 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.

1. A full binary tree with 2n internal nodes contains 2n + 1 leaf nodes.
2. C) None of the above