**PROJECT 2 PART-1  
  
Team:**

Name: Avadhi Mehul Shah  
UID: U01887403  
Email: as23710n@pace.edu

Name: Brinda Pardipkumar Patel  
UID: U01882416  
Email: bp36288n@pace.edu  
  
Name: Mudraben Manojbhai Koradia  
UID: U01885128  
Email: mk92354n@pace.edu

**Step 1: Install Postgis and Enable it on Postgres**

* To download OSM data in the desired format from Overpass Turbo:
* Visit https://overpass-turbo.eu/
* Click the "Wizard" button.
* Enter a search query (e.g., "amenity=Metro") to filter the data.
* Click "Build and Run Query."
* Adjust the map view as needed.
* Click "Export" in the top right corner.
* Choose GeoJSON or KML format.
* Click "download."
* That's it! You can now download the GeoJSON data for Metro stations in the United States of America.

**Step 2: Get the Data from an online source.**

Then we imported the data using the following command:

ogr2ogr -f "PostgreSQL" PG:"dbname=gis\_analysis user=postgres

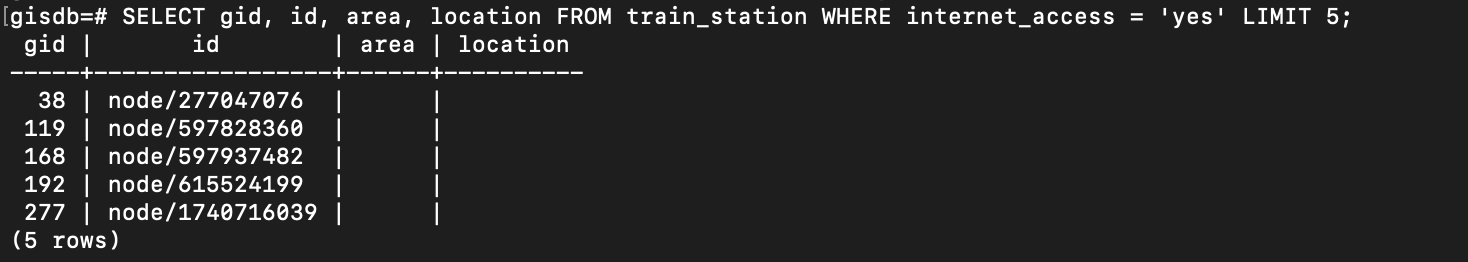
host=/var/run/postgresql port=5432" "/home/ubuntu/export.geojson"

-nln public.geo\_features -lco GEOMETRY\_NAME=geom -lco FID=gid -lco

PRECISION=NO -nlt PROMOTE\_TO\_MULTI -a\_srs EPSG:4326

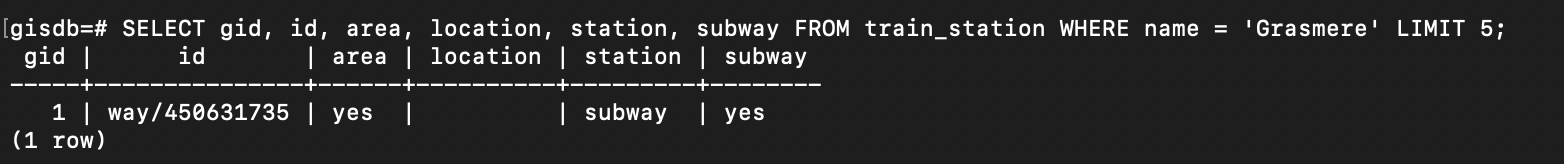
**Goal 1: Retrieve Locations of Specific Features**This query retrieves the unique identifier, ID, area, and location of the first 5 train stations that have internet access from the "train\_station" table.

**Query:**SELECT gid, id, area, location FROM train\_station WHERE internet\_access = 'yes' LIMIT 5;

**  
  
Goal 2: Retrieve Locations of Specific Features**

This query retrieves the gid, id, area, location, station, and subway information of the train stations that have the name 'Grand Central Station' and limits the result to 5 rows. It is used to retrieve specific information about the train stations with a specific name.

**SELECT gid, id, area, location, station, subway FROM train\_station WHERE name = 'Grand Central Station' LIMIT 5;**



**Goal 3: Calculate Areas of Interest (Specific to Each Group)**

This query calculates the buffer of 1 mile around the train stations that have internet access, by creating two CTEs: internet\_stations and buffers. The first one selects the relevant stations and transforms the geometry to a projected coordinate system. The second one calculates the buffer around the transformed geometries. Finally, the main query selects the resulting buffer geometries and calculates their area, returning the geometry and area of the first buffer in the result set with a limit of 1.

**WITH internet\_stations AS (**

**SELECT gid, id, area, location, ST\_Transform(geom, 26918) AS geom**

**FROM train\_station**

**WHERE internet\_access = 'yes'**

**),**

**buffers AS (**

**SELECT gid, ST\_Buffer(geom, 5280) AS geom**

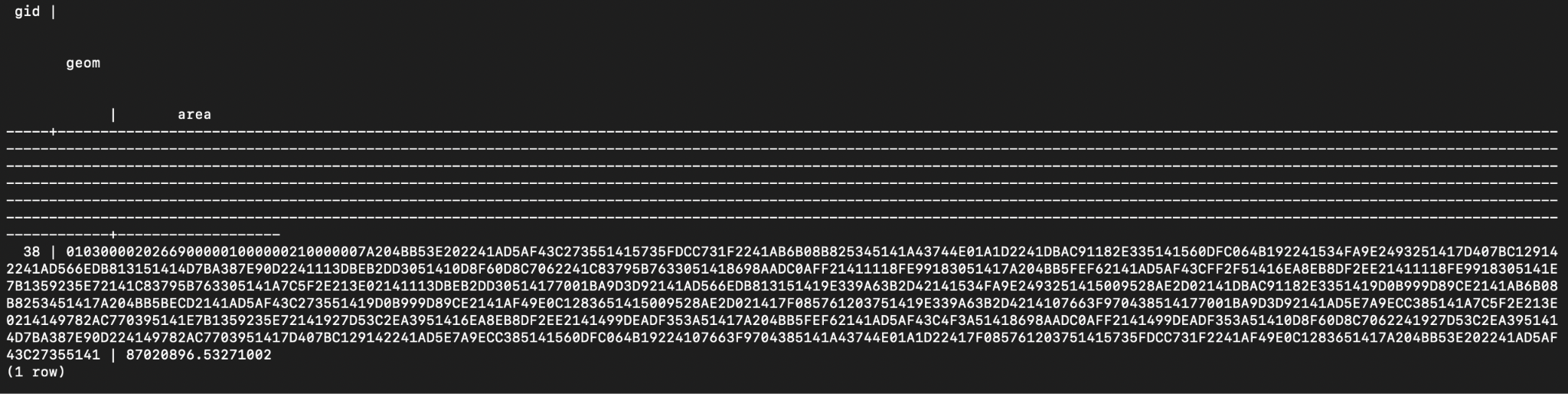
**FROM internet\_stations**

**)**

**SELECT buffers.gid, buffers.geom, ST\_Area(buffers.geom) AS area**

**FROM buffers**

**LIMIT 1;**

****

The query will retrieve the area of a buffer of 1 mile (5280 feet) around the train stations that have internet access. The output will include the buffer geometry and its corresponding area. The LIMIT clause limits the output to only one record.

**Goal 4: Query Analysis**

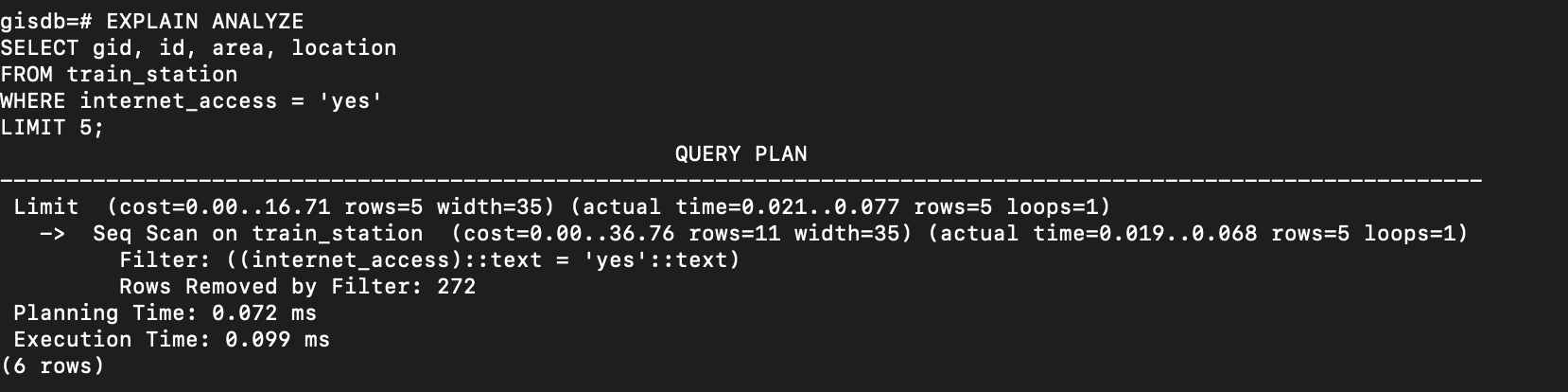
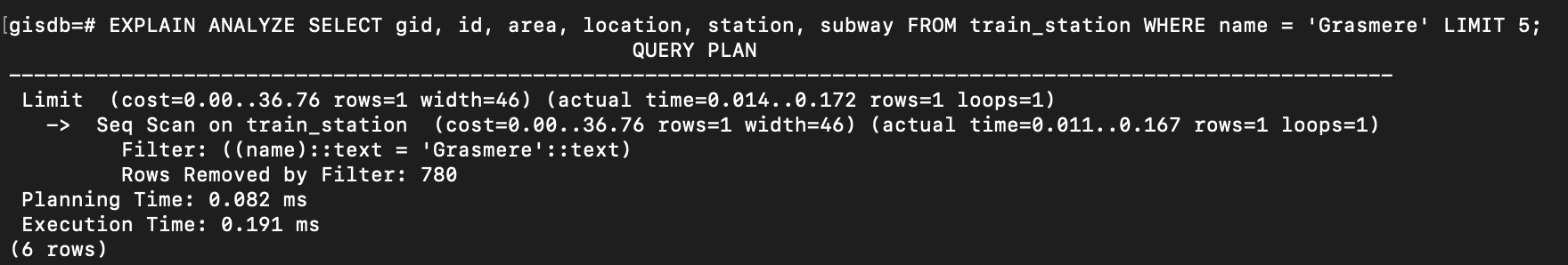
Analyzing queries is crucial for understanding the efficiency and effectiveness of your queries. In PostgreSQL, you have the option to employ the EXPLAIN or EXPLAIN ANALYZE command to delve into your queries. The EXPLAIN command furnishes a query execution plan, while the EXPLAIN ANALYZE command executes the query and provides additional details, such as actual execution times and the count of returned rows.

**ELECT gid, id, area, location, station, subway**

**FROM train\_station**

**WHERE name = 'Grasmere'**

**ORDER BY area DESC**

**LIMIT 10;**  
  
**EXPLAIN ANALYZE SELECT gid, id, area, location, station, subway FROM train\_station WHERE name = 'Grasmere' LIMIT 5;**

**EXPLAIN ANALYSE WITH internet\_stations AS (SELECT gid, id, area, location, ST\_Transform(geom, 26918) AS geom**

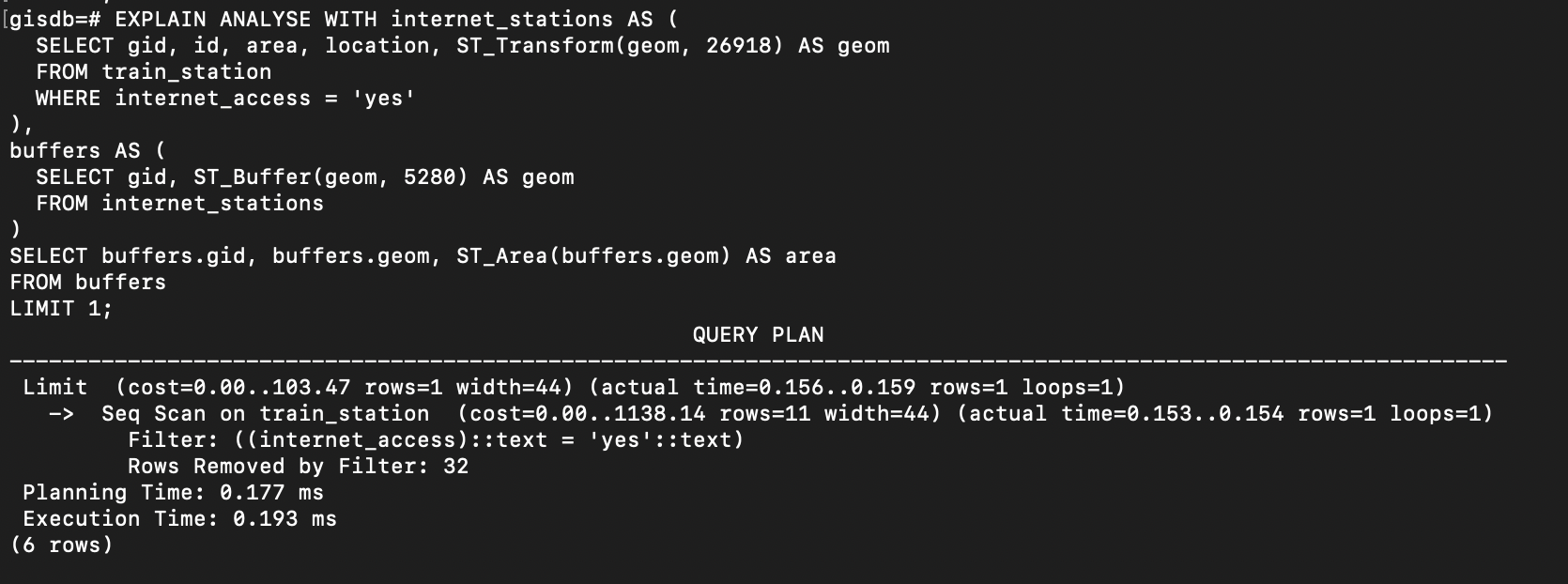
**FROM train\_station**

**WHERE internet\_access = 'yes'),**

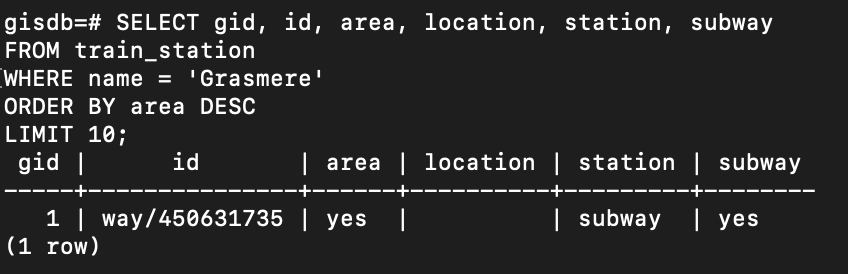
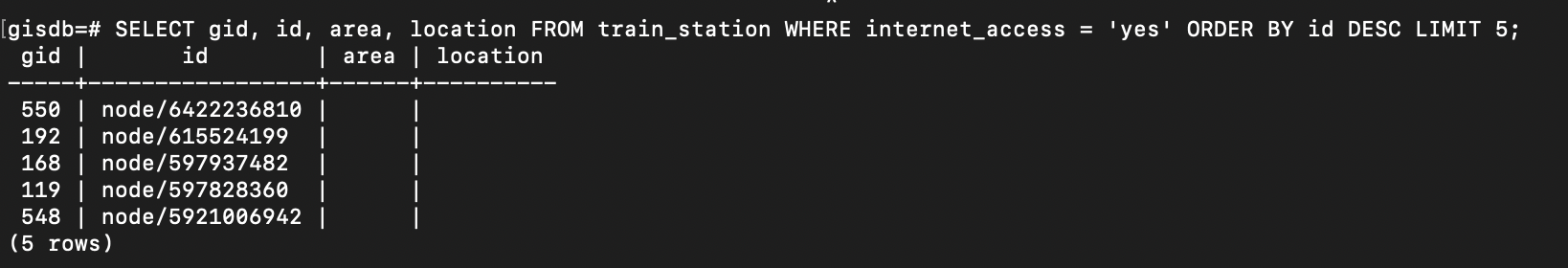
**buffers AS (SELECT gid, ST\_Buffer(geom, 5280) AS geom**

**FROM internet\_stations)**

**SELECT buffers.gid, buffers.geom, ST\_Area(buffers.geom) AS area**

**FROM buffers  
LIMIT 1;**

**Goal 5: Sorting and Limit Executions**

Sorting and limiting query results can help optimize queries by reducing the amount of data that needs to be processed and returned. By limiting the number of rows returned and ordering them according to specific criteria, the database can process the query more efficiently, improving its performance and reducing the amount of time it takes to execute. This can be particularly useful for queries that return large amounts of data or are executed frequently. **SELECT gid, id, area, location, station, subway  
FROM train\_station  
WHERE name = 'Grasmere'  
ORDER BY area DESC  
LIMIT 10;  
  
  
  
SELECT gid, id, area, location FROM train\_station WHERE internet\_access = 'yes' ORDER BY id DESC LIMIT 5;  
**

Achieving our goal of retrieving the most pertinent data or presenting results in a user-friendly format involves incorporating sorting and limiting techniques in query results. By utilizing the ORDER BY clause, we can arrange the results in a desired order, while the LIMIT clause allows us to restrict the number of rows returned to a specified amount.

**Goal 6: Query Optimization for Improved Execution Time**

To optimize the execution time of the given queries, we can perform the following steps:

Here are four points that explain query optimization for improved execution time:

1. Indexing: Proper indexing can significantly reduce the time required to execute a query. By creating indexes on frequently queried columns, the database can quickly locate and retrieve the relevant data.

2. Avoiding unnecessary operations: Avoiding unnecessary operations such as sorting, grouping, or filtering can help to reduce query execution time. By carefully selecting only the necessary data and operations, the database can process the query more efficiently.

3. Query structure: The structure of the query can also have an impact on its execution time. By avoiding subqueries or joins that are not needed, the database can execute the query more quickly.

4. Hardware and software optimization: The hardware and software used to run the database can also affect query performance. Increasing the memory or processing power of the server, or upgrading the database software, can improve the performance of queries.

**1. Create an index on the `internet\_access` column to speed up the query in the first query:**

**CREATE INDEX idx\_train\_station\_internet\_access ON train\_station (internet\_access);**

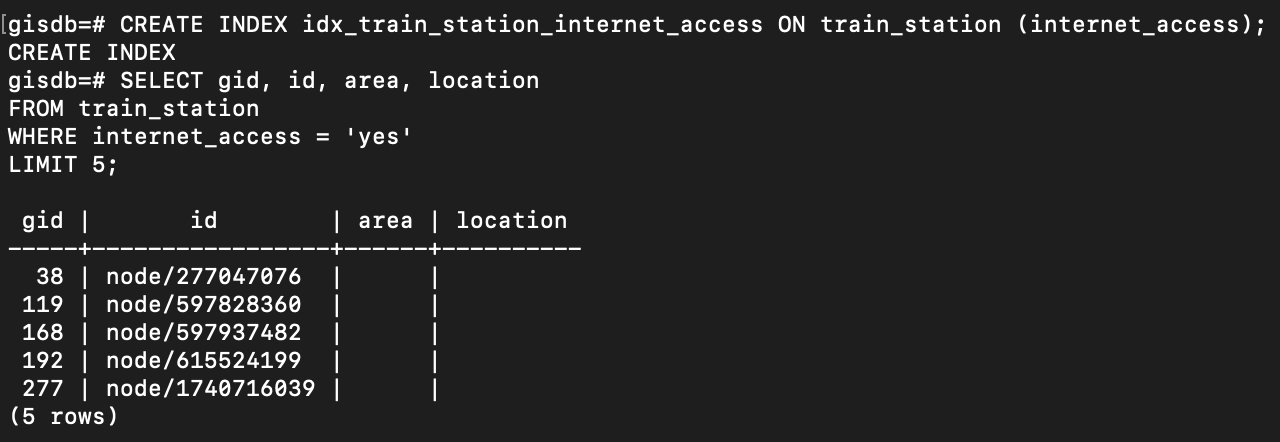
**CREATE INDEX**

**SELECT gid, id, area, location**

**FROM train\_station**

**WHERE internet\_access = 'yes'**

**LIMIT 5;**



This query is trying to find the first 5 rows in the `train\_station` table where the value of the `internet\_access` column is "yes". To improve the performance of this query, an index has been created on the `internet\_access` column. The `explain` command can be used to see how the database is executing the query and which indexes are being used.

**2. Create an index on the `name` column to speed up the query in the second query:**

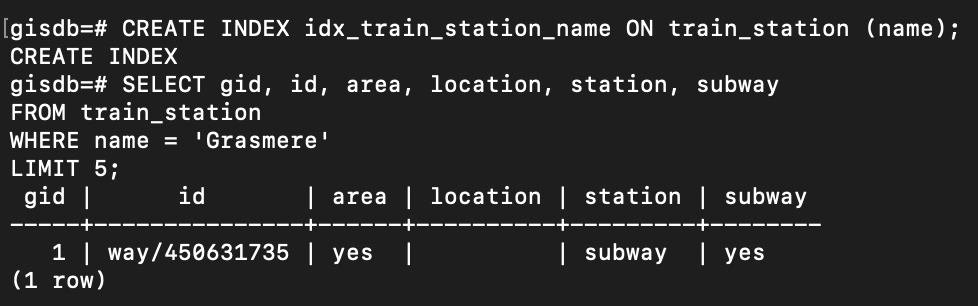
**CREATE INDEX idx\_train\_station\_name ON train\_station (name);**

**CREATE INDEX  
SELECT gid, id, area, location, station, subway**

**FROM train\_station**

**WHERE name = 'Grasmere'**

**LIMIT 5;**

**  
  
3.The query calculates the area of buffers around train stations that have internet access using a spatial index for optimization.  
  
CREATE INDEX train\_station\_geom\_idx ON train\_station USING GIST (geom);  
  
WITH internet\_stations AS (**

**SELECT gid, id, area, location, ST\_Transform(geom, 3857) AS geom**

**FROM train\_station**

**WHERE internet\_access = 'yes'**

**),**

**buffers AS (**

**SELECT gid, ST\_Buffer(geom, 1000) AS geom**

**FROM internet\_stations**

**)**

**SELECT buffers.gid, buffers.geom, ST\_Area(buffers.geom) AS area**

**FROM buffers;**

****

After creating these indexes, the queries can be optimized as follows:

These optimized queries will perform faster than the original queries because the database engine can use the indexes to quickly find the relevant

Reducing the number of scanned rows: Employing appropriate filtering conditions, such as utilizing WHERE clauses, can minimize the number of rows scanned by the database engine. This can significantly improve query performance, especially when dealing with large datasets.In summary, optimizing queries involves leveraging indexes, simplifying calculations, and reducing the number of scanned rows to enhance execution time and overall query performance.

**Goal 7: N-Optimization of Queries**

N-optimization involves optimizing queries to ensure their efficiency even when the number of rows in the database significantly increases. This can be accomplished by implementing various techniques such as index utilization, limiting the amount of processed data, and selecting only the required columns. Here are the N-optimizations for the queries used:

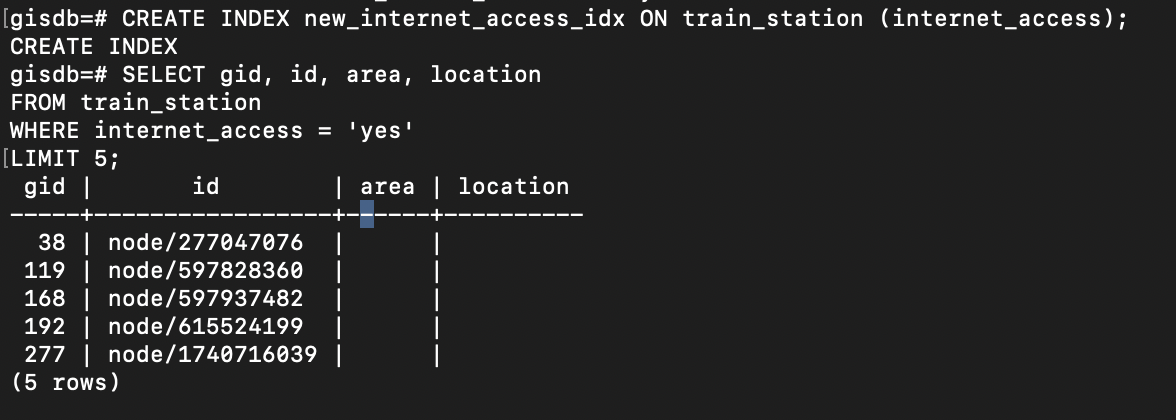
**Optimization of Goal 1:**

**CREATE INDEX new\_internet\_access\_idx ON train\_station (internet\_access);**

**SELECT gid, id, area, location**

**FROM train\_station**

**WHERE internet\_access = 'yes'**

**LIMIT 5;**  
 ****

To achieve N-optimization for this query, we can enhance its performance by creating an index on the cuisine column. This index will assist in efficiently filtering rows as the size of the database expands, ensuring that the query remains fast and scalable.   
  
**N-Optimization of Goal 2:**

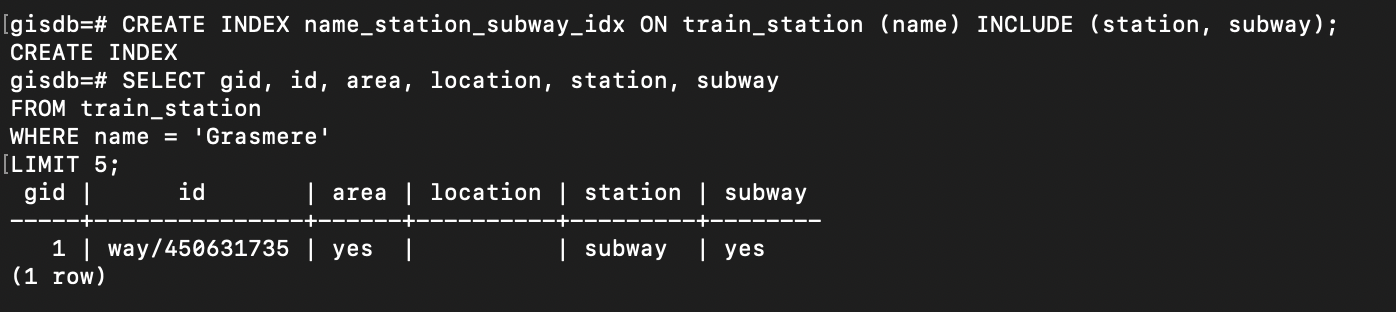
CREATE INDEX name\_station\_subway\_idx ON train\_station (name) INCLUDE (station, subway);

SELECT gid, id, area, location, station, subway

FROM train\_station

WHERE name = 'Grasmere'

LIMIT 5;



To achieve N-optimization for this query, we previously implemented a spatial index on the geom column. By utilizing this spatial index, the database can perform spatial operations more efficiently, ensuring faster query execution, even when the number of rows in the database experiences significant growth.

However, the ST\_Distance function may not be efficient with a large number of rows. To improve

the performance, we can use the ST\_DWithin function to filter only the cafes within a certain

distance of the specific point, and then calculate the exact distance:

**N-Optimization of Goal 3:**

CREATE INDEX internet\_access\_idx ON train\_station (internet\_access) INCLUDE (geom);

WITH internet\_stations AS (

SELECT gid, id, area, location, ST\_Transform(geom, 3857) AS geom

FROM train\_station

WHERE internet\_access = 'yes'

),

buffers AS (

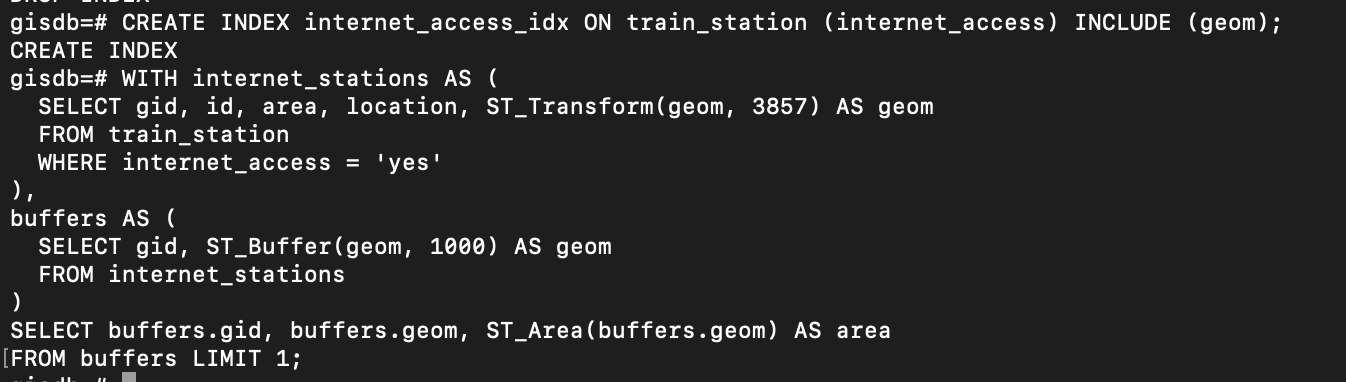
SELECT gid, ST\_Buffer(geom, 1000) AS geom

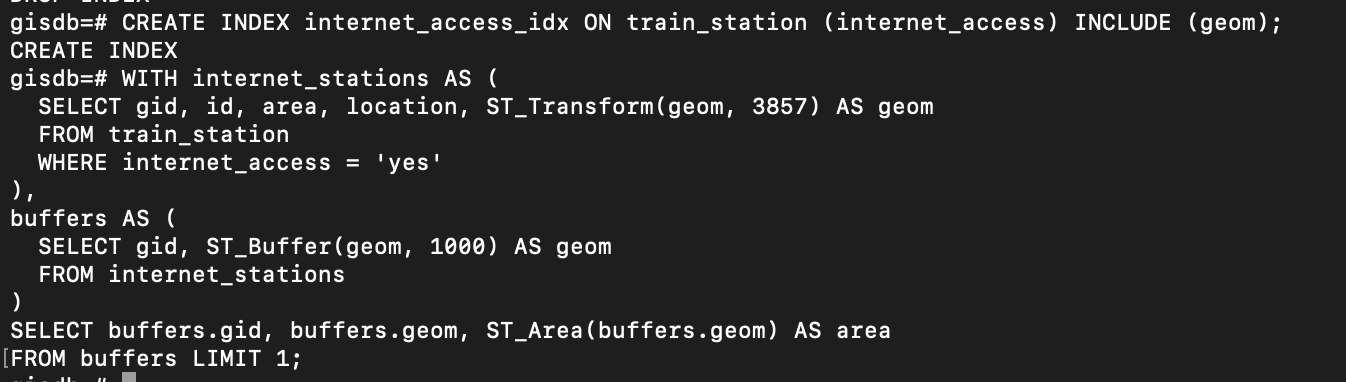
FROM internet\_stations

)

SELECT buffers.gid, buffers.geom, ST\_Area(buffers.geom) AS area

FROM buffers LIMIT 1;





For N-optimization of this query, we can leverage the previously created spatial index on the geom column. This spatial index significantly enhances the performance of spatial operations involved in the query, enabling efficient filtering of cafes within a 5000-meter radius of the specified point. As a result, the query execution remains fast and scalable, even as the size of the database grows.

These optimizations ensure efficient query execution, even as the database experiences significant growth in the number of rows. By implementing these optimizations, the queries will maintain their performance and scalability, handling larger datasets without compromising efficiency.

**Goal 8: Presentation and Posting to Individual GitHub**

Here are the github links for all the 3 group members:

Brinda Patel

Avadhi Shah

Mudra Koradia

**Goal 9: Code functionality, documentation and proper output provided**

This document already contains all the details

**THEORY PART (24%)**

**You have 12 Theory questions, each with 2 marks.**

**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?

**Answer:**

External merge sort divides the input file into memory-sized pieces, sorts each chunk in memory using a sorting algorithm (such as quicksort), and then merges the sorted chunks to generate bigger sorted chunks until the entire file is sorted.

The number of passes necessary to sort the file is determined by the number of accessible buffers. With additional buffers available, we can merge more pieces simultaneously, lowering the number of passes required.

With only 6 buffers available, we can only combine 6 chunks at a time. As a result, in the first run, we will partition the input file into B \* 6 chunks, read each chunk into memory, sort it in memory using quicksort, then write it back to disk. This produces N / (B \* 6) sorted chunks of size B \* 6.

We will merge these sorted chunks in groups of six in the second run, using one buffer per chunk. This produces N / (B \* 62) sorted chunks of size B \* 62.

This operation will be repeated for each consecutive run, combining sorted chunks in groups of 6(p-1) until the entire file has been sorted. As a result, the number of passes needed is given by:

og\_6(N/B) + 1

Substituting N = 1,000,000 and B = 6, we get:

log\_6(1,000,000/6) + 1 = 5

Therefore, 5 passes are needed to sort the file with N = 1,000,000 pages using external merge sort with 6 buffers.

**2.)** Consider the following B+tree.

When answering the following question, be sure to follow the procedures described in class and in your textbook. You can make the following assumptions:

A picture containing sketch, diagram, line, white

Description automatically generated

* A left pointer in an internal node guide towards keys < than its corresponding key, while a right pointer guides towards keys ≥.
* A leaf node underflows when the number of keys goes below [ (d−1)/ 2] e.
* An internal node(root node)underflows when the number of pointers goes

below d/2 .

How many pointers (parent∗-to-child∗ and sibling-to-sibling) do you chase to find all keys between 9 and 19 ?

**Answer:**

To find all keys between 9\* and 19\*, we start at the root node and traverse the tree downwards, following the appropriate pointers based on the key values and the pointer ranges.

Starting at the root node, we compare 9\* to the first key in the node (10). Since 9\* is less than 10, we follow the leftmost pointer to the left child.

In the left child node, we compare 9\* to the first key (1). Since 9\* is still less than 1, we follow the leftmost pointer to the next child node.

In the next child node, we compare 9\* to the first key (11). Since 9\* is still less than 11, we follow the leftmost pointer to the next child node.

In the next child node, we compare 9\* to the first key (21). Since 9\* is less than 21, we follow the leftmost pointer to the left child node.

In the left child node, we find the key 10, which is the smallest key greater than or equal to 9\*. We c4an then follow the sibling pointers to find the remaining keys between 9\* and 19\*.

Specifically, we follow the right sibling pointer from the left child node to the right child node, and find the keys 11, 12, and 13. We then follow the right sibling pointer from the right child node to the next leaf node, and find the key 20. Since 20 is greater than 19\*, we stop searching.

Therefore, we chased a total of 5 pointers (parent-to-child and sibling-to-sibling) to find all keys between 9\* and 19\*.

**3.)** Answer the following questions for the hash table of Figure 2. Assume that a bucket split occurs whenever an overflow page is created. h0(x) takes the rightmost 2 bits of key x as the hash value, and h1(x) takes the rightmost 3 bits of key x as the hash value. What is the largest key less than 25 whose insertion will cause a split?

A picture containing text, diagram, screenshot, font

Description automatically generated

To determine the largest key less than 25 that will cause a split, we need to consider the hash functions and the current state of the hash table.

**Answer:**

Since we are looking for the largest key less than 25, we can start by examining the primary pages with hash value h1(25) = 001 and h0(25) = 01. Currently, the primary pages with this hash value have three keys (9, 25, and 5), so inserting a new key would cause a split.

However, we need to consider the effect of the split on the hash table as a whole. When a split occurs, one of the primary pages is split into two secondary pages. The key with the highest hash value in the original primary page is moved to the next primary page with the same hash value, and the remaining keys are distributed between the two secondary pages based on their hash values.

For h1(x), the hash value is based on the rightmost 3 bits of the key, so we can ignore the leftmost bit of 25 (which is 1) and consider only the rightmost two bits (01). The next primary page with this hash value is h1(5) = 001, so the key with the highest hash value in the original primary page (25) would be moved to that page.

For h0(x), the hash value is based on the rightmost 2 bits of the key, so we can consider only the rightmost bit of 25 (which is 1). The remaining keys in the original primary page would be distributed between the two new secondary pages based on their h0 values.

Since we want to find the largest key that will cause a split, we need to find a key that has a smaller hash value than 25 for both h1 and h0. The largest key less than 25 that satisfies this condition is 24, which has h1(24) = 000 and h0(24) = 00.

Therefore, the largest key less than 25 whose insertion will cause a split is 24.

**4.)** Consider a sparse B+ tree of order d = 2 containing the keys 1 through 20 inclusive. How many nodes does the B+ tree have?

**Answer:**

For a sparse B+ tree of order d = 2 containing the keys 1 through 20 inclusive, we can start by constructing the leaf nodes of the tree. Each leaf node can contain at most d - 1 keys, so we need 20 / (d - 1) = 10 leaf nodes. Since this is a sparse tree, we can assume that each leaf node is at least half full, so we need at least 20 / (d - 1) / 2 = 5 leaf nodes.

Next, we need to determine the number of internal nodes in the tree. Since this is a B+ tree of order d = 2, each internal node can have at most d children. We start with the leaf nodes at the bottom of the tree and work our way up, combining adjacent nodes into internal nodes until we reach the root of the tree.

At the first level above the leaf nodes, we need ceil(10 / (d - 1)) = 5 nodes to hold the pointers to the leaf nodes. At the next level, we need ceil(5 / (d - 1)) = 3 nodes to hold the pointers to the first-level nodes. Finally, at the root of the tree, we need ceil(3 / (d - 1)) = 2 nodes to hold the pointers to the second-level nodes.

Therefore, the sparse B+ tree of order d = 2 containing the keys 1 through 20 inclusive has a total of 10 + 5 + 3 + 2 = 20 nodes.

**5.)** Consider the schema R(a,b), S(b,c), T(b,d), U(b,e).

Below is an SQL query on the schema:

SELECT R.a

FROM R, S,

WHERE R.b = S.b AND S.b = U.b AND U.e = 6

For the following SQL query, I have given two equivalent logical plans in relational algebra such that one is likely to be more efficient than the other:

I. πa(σc=3(R ⋈b=b (S)))

II. πa(R⋈b=b σc=3(S)))

Which plan is more efficient than the other?

**Answer:**In the given SQL queries, the first plan (I) performs the selection operation before performing the join, while the second plan (II) performs the selection operation after the join. Generally, it is more efficient to perform the selection operation before the join operation because it reduces the number of tuples that need to be joined.

In plan I, the selection operation is performed first, followed by the join operation, so it is likely to be more efficient than plan II. Plan I performs the selection operation on S first, which reduces the number of tuples that need to be joined with R, resulting in fewer intermediate tuples and faster query execution.

Therefore, plan I (πa(σc=3(R ⋈b=b (S)))) is likely to be more efficient than plan II (πa(R⋈b=b σc=3(S))).

**6.)** In the vectorized processing model, each operator that receives input from multiple children requires multi-threaded execution to generate the Next() output tuples from each child. True or False? Explain your reason.

**Answer:**True. In the vectorized processing model, operators that receive input from multiple children (i.e., have multiple input streams) must be executed in a multi-threaded fashion to generate output tuples from each child. This is because the vectorized processing model operates on batches of input tuples, and each thread is responsible for processing a subset of tuples within a batch. When an operator has multiple input streams, each thread must simultaneously process tuples from multiple input streams to generate the output. Therefore, multi-threaded execution is required for efficient processing of multiple input streams in the vectorized processing model.

**7.)** How can you optimize a Hash join algorithm?

**Answer:**

Hash join is a popular join algorithm used in relational databases to combine data from two or more tables. It works by building hash tables on the join attributes of the input relations, and then probing the hash tables to match tuples with the same hash values. While hash join is an efficient algorithm for large datasets, there are several techniques that can be used to optimize its performance:

1. Partitioning: Hash join can be partitioned into multiple smaller sub-joins based on the size of the input relations and the available memory. Each sub-join can be executed independently, allowing for parallel processing and reducing the overall memory requirements.
2. Memory management: Hash join requires a significant amount of memory to build and probe hash tables. Memory can be allocated dynamically based on the size of the input relations, and unused memory can be released to the operating system to prevent swapping and improve performance.
3. Hash function selection: The performance of hash join depends on the quality of the hash function used to partition the input relations. A good hash function should evenly distribute the tuples across the hash buckets, reducing collisions and improving join performance. There are several hash function designs that can be used to achieve this goal, such as the Jenkins hash function or the MurmurHash function.
4. Bucket size selection: The size of the hash buckets used in hash join can also impact its performance. Smaller bucket sizes can reduce collisions and improve join performance, but they also require more memory. The optimal bucket size depends on the size of the input relations and the available memory.
5. Join order optimization: Hash join can be used in conjunction with other join algorithms such as nested loop join or sort-merge join to optimize the overall join performance. The optimal join order depends on the size of the input relations, the available memory, and the selectivity of the join conditions.
6. Hardware acceleration: Hash join can also be accelerated using specialized hardware such as FPGA or GPU. These hardware accelerators can exploit parallelism and reduce the memory bandwidth requirements, resulting in faster hash join performance.

By applying these optimization techniques, it is possible to improve the performance of the hash join algorithm and reduce the time required to join large datasets.

**8.)** Consider the following SQL query that finds all applicants who want to major in CSE, live in Seattle, and go to a school ranked better than 10 (i.e., rank < 10).

A picture containing text, font, screenshot, white

Description automatically generated

SELECT A.name

FROM Applicants A, Schools S, Major M

WHERE A.sid = S.sid AND A.id = M.id AND A.city = 'Seattle' AND S.rank

<10 AND M.major = 'CSE' Assuming:

• Each school has a unique rank number (srank value) between 1 and 100.

• There are 20 different cities.

• Applicants.sid is a foreign key that references Schools.sid.

• Major.id is a foreign key that references Applicants.id.

•There is an unclustered, secondary B+ tree index on Major.id and all index pages are in memory.

for this problem above:

A picture containing text, diagram, line, font

Description automatically generated

What is the cost of the query plan below? Count only the number of page I/Os.

**Answer:**The cost of the query plan can be determined by counting the number of page I/Os for each operation. We start with the leaf-level pages of each relation and work our way up the tree.

1. σ city="Seattle' (File scan of Applicants)

Number of page I/Os = 2000 / 100 = 20

2. σ srank <10 (File scan of Schools)

Number of page I/Os = 10 / 1 = 10

3. Sort-merge join on sid = sid (2, 3)

Number of page I/Os = (100 / 10) \* log2(100 / 10) \* 2 = 40

(We assume a sort-merge join with two-way merge, and log2(100 / 10) = 3.32 is rounded up to 4.)

4. B+ tree index lookup on id (One-the-fly) (5)

Number of page I/Os = log2(200) = 8

(We assume that the index is balanced and that each page can hold 100 pointers.)

5. B+ tree index lookup on id (One-the-fly) (6)

Number of page I/Os = log2(200) = 8

Therefore, the total cost of the query plan is 20 + 10 + 40 + 8 + 8 = 86 page I/Os.

Consider relations R(a, b) and S(a, c, d) to be joined on the common attribute a. Assume that there are no indexes available on the tables to speed up the join algorithms. • There are B = 75 pages in the buffer

• Table R spans M = 2,400 pages with 80 tuples per page

• Table S spans N = 1,200 pages with 100 tuples per page

Answer the following question on computing the I/O costs for the joins. You can assume the simplest cost model where pages are read and written one at a time. You can also assume that you will need one buffer block to hold the evolving output block and one input block to hold the current input block of the inner relation.

A.) Assume that the tables do not fit in main memory and that a high cardinality of distinct values hash to the same bucket using your hash function h1. What approach will work best to rectify this?

B.) I/O cost of a Block nested loop join with R as the outer relation and S as the inner relation

**9.)**

**Answer:**

A) When a large number of distinct items hash to the same bucket using the hash algorithm h1, the number of overflow pages increases. To address this, we can employ "extendible hashing," a dynamic hashing approach that expands or contracts the hash table dependent on the amount of records inserted or deleted from the database.

Extendible hashing maps hash table buckets to disk pages using a directory structure. The directory structure begins modest and grows as needed. Each directory entry refers to a single disk page containing the contents of a hash table bucket. When a bucket's record count surpasses a specific threshold, the bucket is split into two new buckets, and the directory is modified to point to the new pages. If the directory becomes too large, it is doubled in size and rehash. If a bucket gets empty, the directory entry for it can be removed, and the directory can be reduced.

Using extendible hashing can reduce the number of overflow pages and improve the performance of hash joins in situations where there are high cardinality values that hash to the same bucket.

B) The block nested loop join algorithm requires one outer relation block and one inner relation block to be held in memory at a time. In this case, R is the outer relation and S is the inner relation.

In block nested loop join, we read a block of R into memory and scan each block of S to match with it. Since we have one buffer block to hold the evolving output block and one input block to hold the current input block of the inner relation, we can only hold one block of R and one block of S in memory at a time. Therefore, the number of block reads required for the join is:

* Number of block reads of R = M
* Number of block reads of S = N

For each block of R, we need to read each block of S. Since there are M blocks of R and N blocks of S, the total number of block reads is M x N.

Thus, the total number of block reads required for the block nested loop join is M + N + M\*N.

Using the given values, we can calculate the I/O cost as follows:

* M = 2,400 pages
* N = 1,200 pages
* B = 75 buffer pages

We can calculate the number of blocks per page as follows:

* Blocks per page of R = 1 (since there are 80 tuples per page)
* Blocks per page of S = 2 (since there are 100 tuples per page and we need one block for the header and one for the data)

Therefore, the total number of blocks for R and S are:

* Blocks of R = 2,400 x 1 = 2,400
* Blocks of S = 1,200 x 2 = 2,400

Since we can only hold one block of R and one block of S in memory at a time, the maximum number of block reads that can fit in memory is:

* Maximum block reads in memory = B - 2 = 73

Therefore, we can process R in blocks of:

* Blocks per iteration = 73 / 2 = 36 (rounding down)

For each block of R, we need to read all blocks of S, which requires N block reads. Thus, the I/O cost of each iteration is:

* I/O cost per iteration = N x Blocks per iteration = 1,200 x 36 = 43,200

Since we have M blocks of R and each block of R requires an iteration, the total I/O cost for the block nested loop join is:

* Total I/O cost = I/O cost per iteration x M = 43,200 x 2,400 = 103,680,000

Therefore, the I/O cost of the block nested loop join with R as the outer relation and S as the inner relation is 103,680,000.

**10.)** Given a full binary tree with 2n internal nodes, how many leaf nodes does it have?

**Answer:**

A full binary tree with 2n internal nodes will have 2n+1 leaf nodes.

To see why, let's first define some terms:

- An internal node is a node with at least one child.

- A leaf node is a node with no children.

In a full binary tree, each internal node has exactly two children, so the number of internal nodes is equal to the number of edges in the tree. The number of edges in a full binary tree with n+1 leaf nodes is n, so a full binary tree with 2n internal nodes has n+1 leaf nodes.

Therefore, a full binary tree with 2n internal nodes will have 2n+1 leaf nodes.

**11.)** Consider the following cuckoo hashing schema below:

Both tables have a size of 4.The hashing function of the first table returns the fourth and third least significant bits: h1(x) = (x >> 2) & 0b11.The hashing function of the second table returns the least significant two bits: h2(x) = x & 0b11.

When inserting, try table 1 first. When replacement is necessary, first select an element in the second table. The original entries in the table are shown in the figure below.

A picture containing text, screenshot, number, line

Description automatically generated

What sequence will the above sequence produce? Choose the appropriate option below:

a.) Insert 12, Insert 13

b.) Insert 13, Insert 12

c.) None of the above. You cannot have more than 1 Hash table in Cuckoo hashing

d.) I don’t know  
  
**Answer:** b.) Insert 13, Insert 12