SCHEMA 1

Query1:

First of all , we did some modification to insertion in Java code such that we have 15 departments (CS1 – CS15) , each department offering between 10 & 15 courses and would have between 7 & 12 instructors and would have at least 400 students , So our query result was 400 rows for the 400 students in department CS1 , semester 1 and year 2019

then we created the tables in pgadmin and run the java code so that the data would be inserted into the tables.

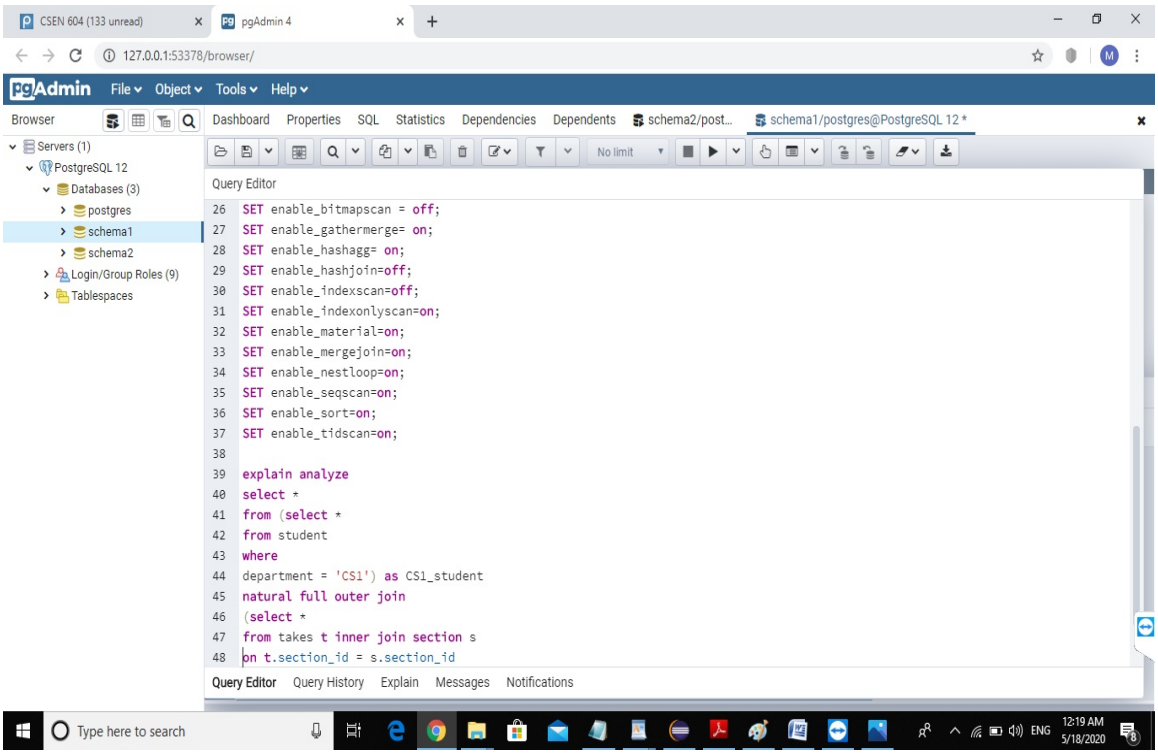
After that , when running the query in pgadmin , the output result was in 400 rows .

We created the statistics to create a new extended statistics object tracking data about the specified table. The statistics object will be created in the current database and will be owned by the user issuing the command.

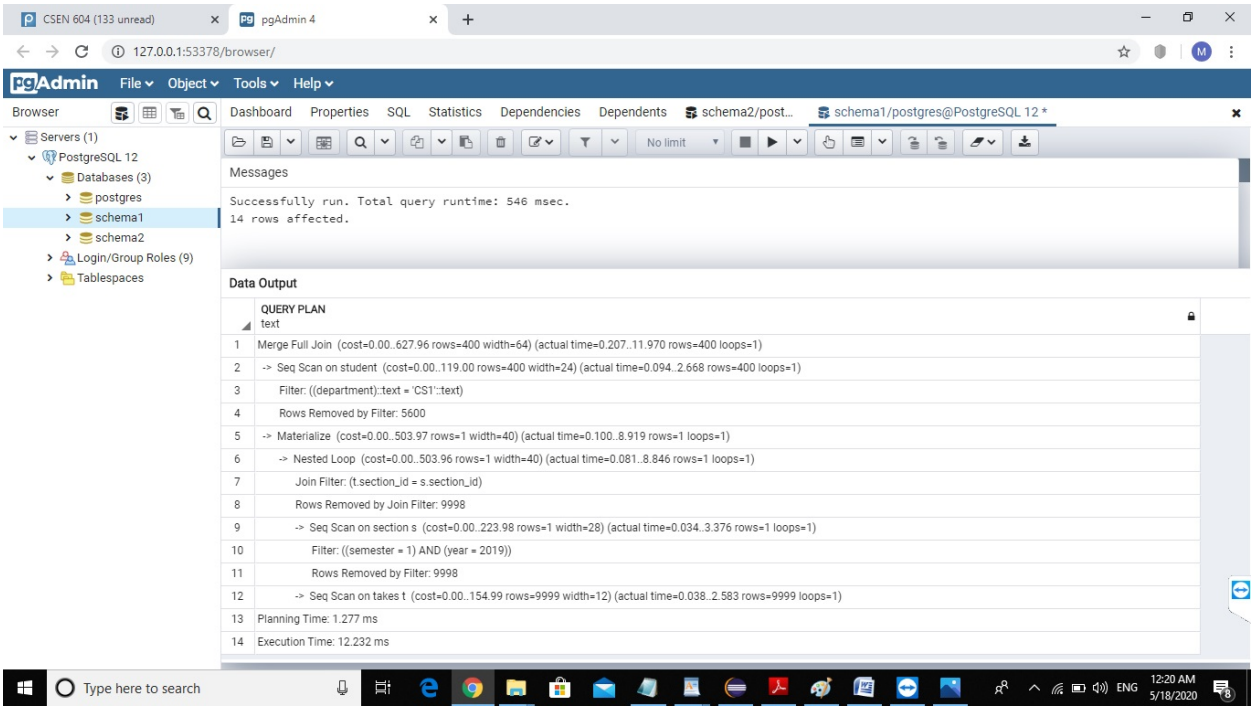
1-**without an index :**

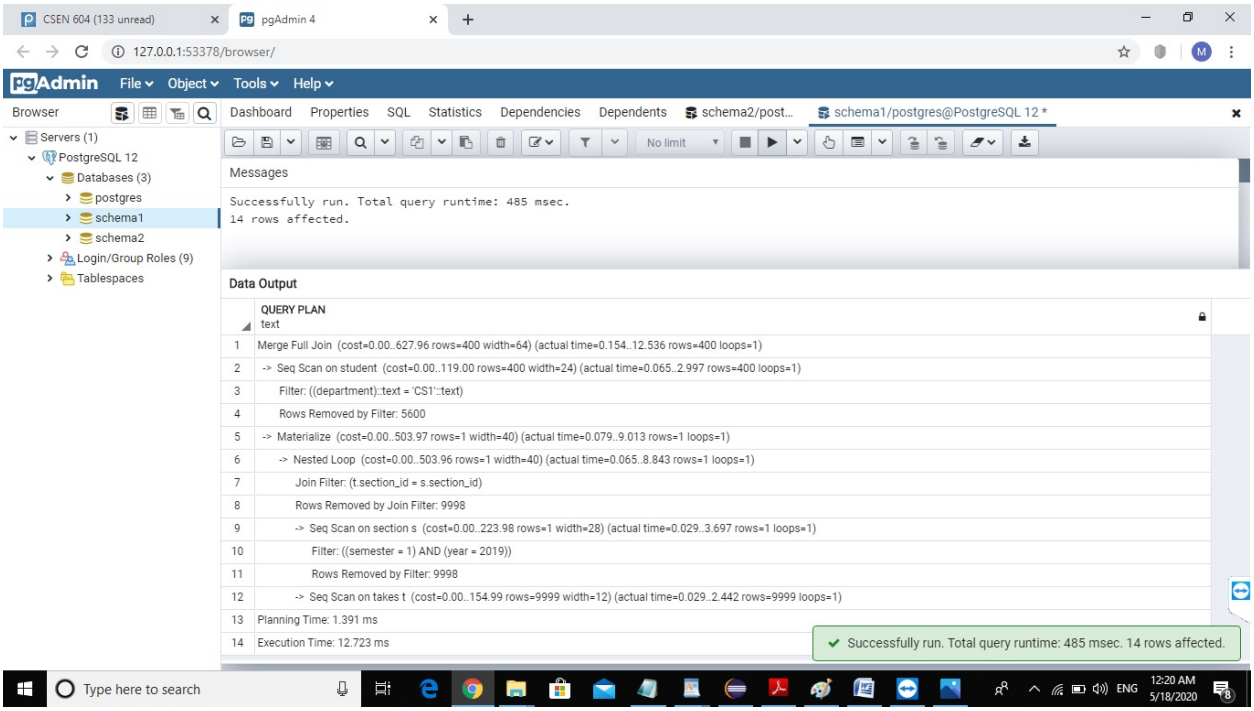
We changed some of the default planner method configuration because these configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a temporary solution is to use one of these configuration parameters to force the optimizer to choose a different plan. So we did so :

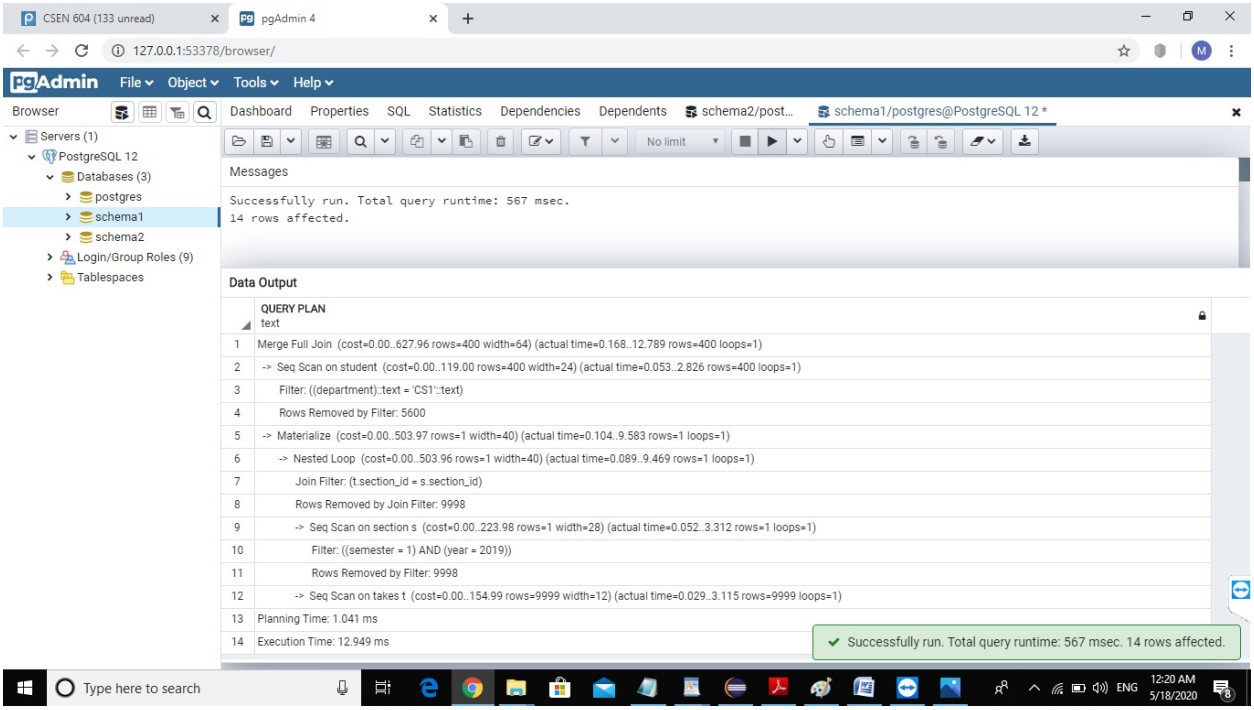
1. Configurations :



2- We run analyze manually : “Explain analyze (The query)” to see the query plan as follows:





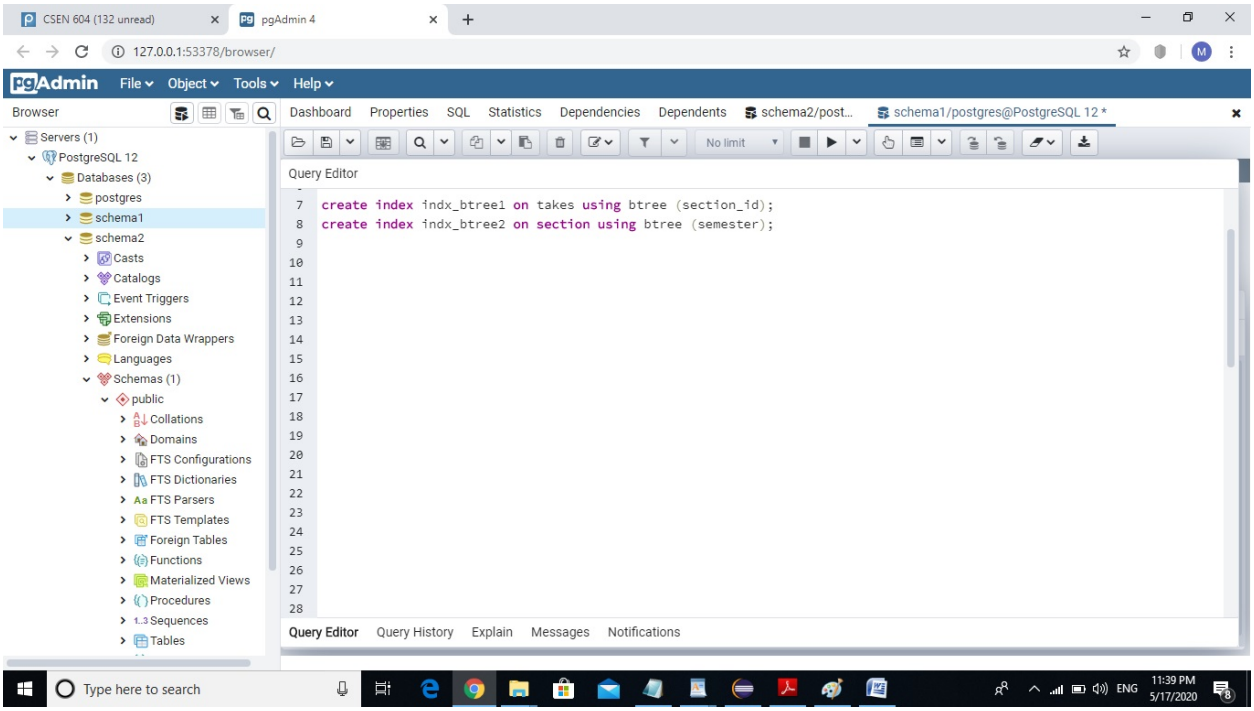


The average execution time : (12.232 + 12. 723 + 12.949) / 3 = 12.634 ms

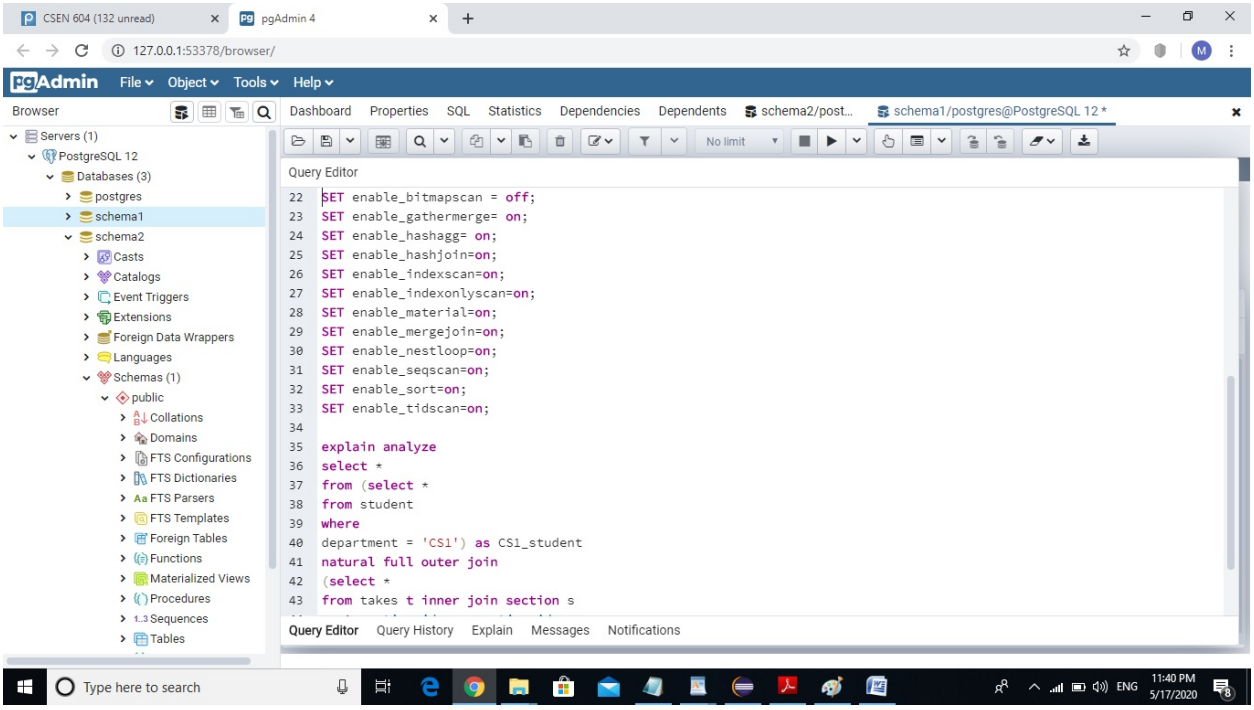
The estimated cost of the plan : 627.96

2-B+ Tree index :

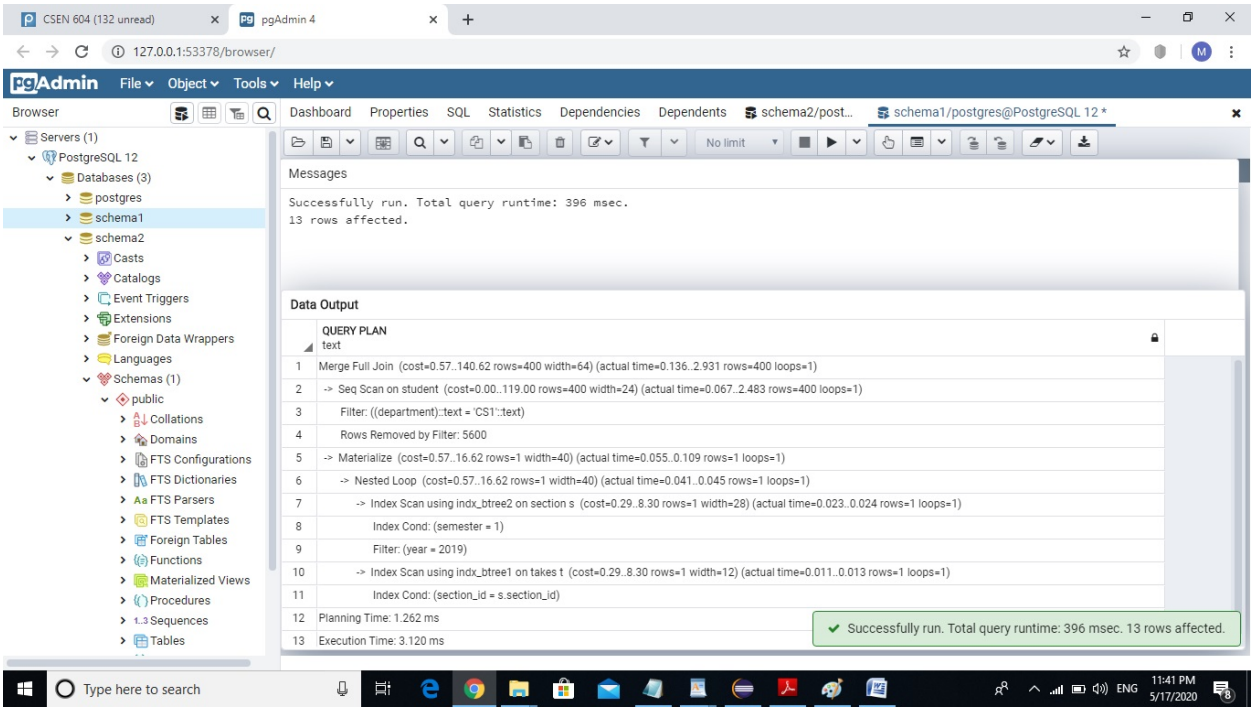
1-Creating the index:

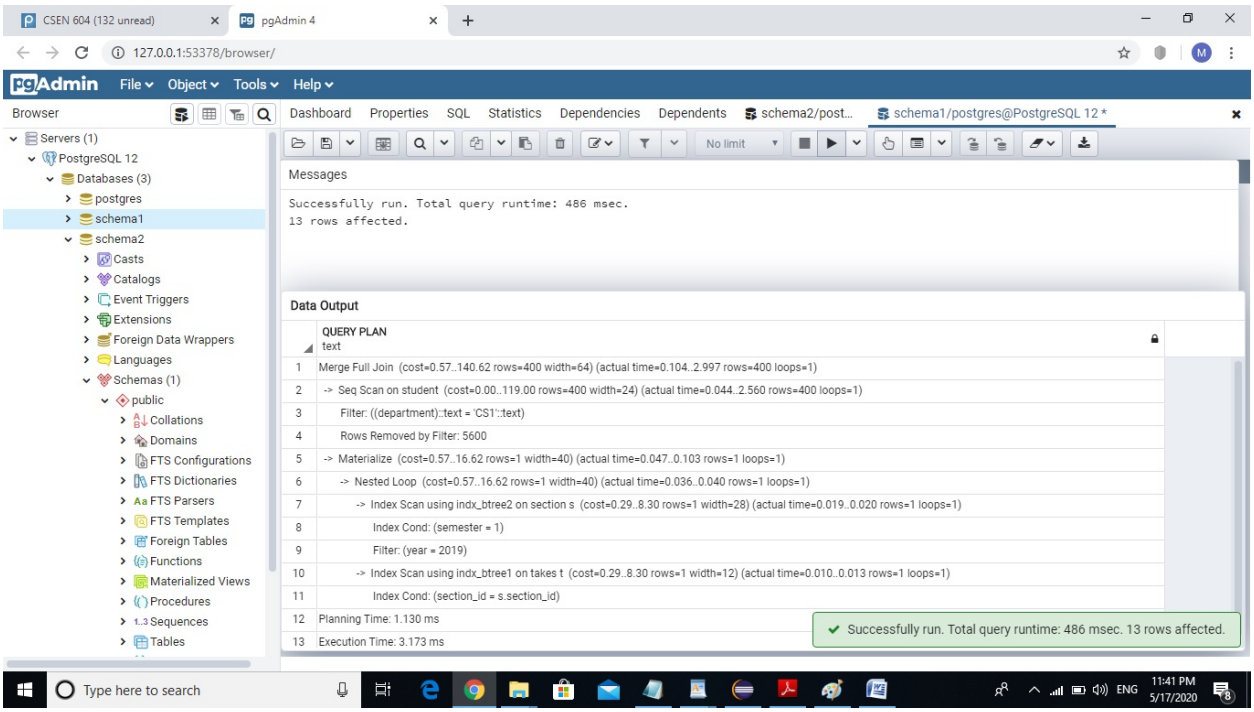


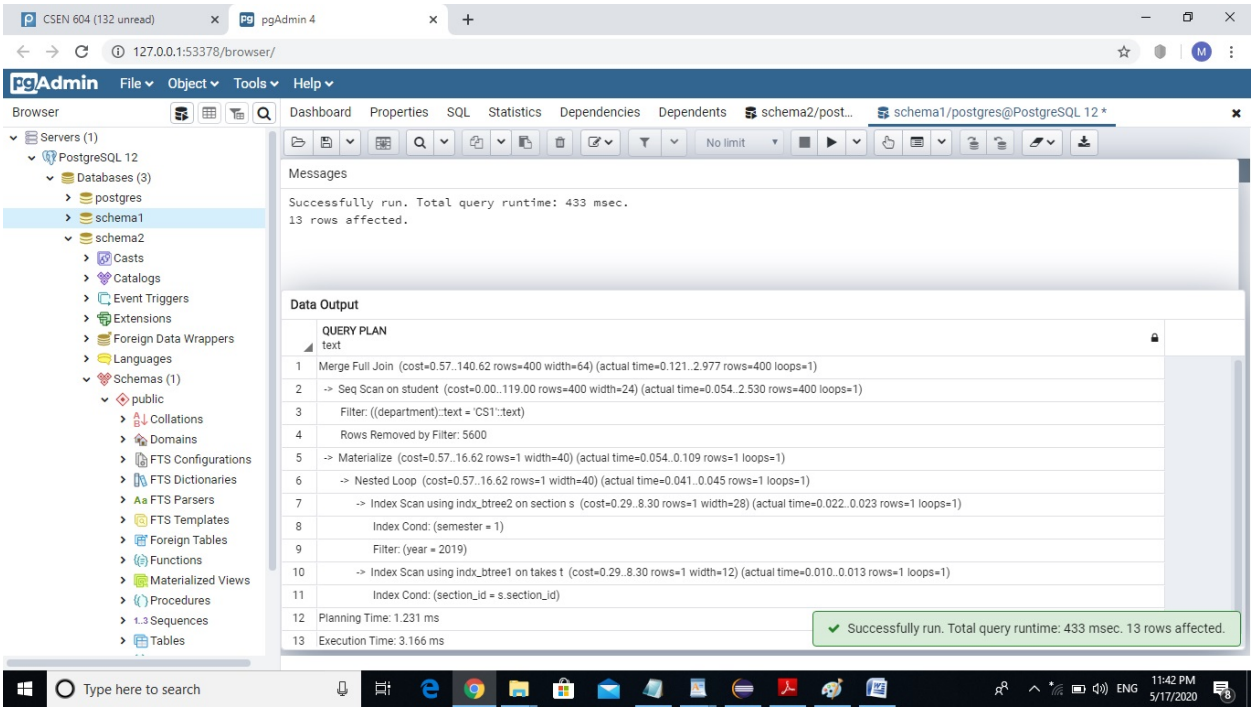
2-Configurations :



3-Query plan:


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Messages
Successfully run. Total query runtime: 396 msec.
13 rows affected.
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=0.57..140.62 rows=400 width=64) (actual time=0.136..2.931 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.067..2.483 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=0.57..16.62 rows=1 width=40) (actual time=0.055..0.109 rows=1 loops=1)
6 -> Nested Loop (cost=0.57..16.62 rows=1 width=40) (actual time=0.041..0.045 rows=1 loops=1)
7 -> Index Scan using indx_btree2 on section s (cost=0.29..8.30 rows=1 width=28) (actual time=0.023..0.024 rows=1 loops=1)
8 Index Cond: (semester = 1)
9 Filter: (year = 2019)
10 -> Index Scan using indx_btree1 on takes t (cost=0.29..8.30 rows=1 width=12) (actual time=0.011..0.013 rows=1 loops=1)
11 Index Cond: (section_id = s.section_id)
12 Planning Time: 1.262 ms
13 Execution Time: 3.120 ms
Successfully run. Total query runtime: 396 msec. 13 rows affected.
Type here to search
ENG
11:41 PM
5/17/2020


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Messages
Successfully run. Total query runtime: 486 msec.
13 rows affected.
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=0.57..140.62 rows=400 width=64) (actual time=0.104..2.997 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.044..2.560 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=0.57..16.62 rows=1 width=40) (actual time=0.047..0.103 rows=1 loops=1)
6 -> Nested Loop (cost=0.57..16.62 rows=1 width=40) (actual time=0.036..0.040 rows=1 loops=1)
7 -> Index Scan using indx_btree2 on section s (cost=0.29..8.30 rows=1 width=28) (actual time=0.019..0.020 rows=1 loops=1)
8 Index Cond: (semester = 1)
9 Filter: (year = 2019)
10 -> Index Scan using indx_btree1 on takes t (cost=0.29..8.30 rows=1 width=12) (actual time=0.010..0.013 rows=1 loops=1)
11 Index Cond: (section_id = s.section_id)
12 Planning Time: 1.130 ms
13 Execution Time: 3.173 ms
Successfully run. Total query runtime: 486 msec. 13 rows affected.
Type here to search
ENG
11:41 PM
5/17/2020



Average execution time : (3.120 + 3.173 + 3.166) / 3 = 3.153 ms

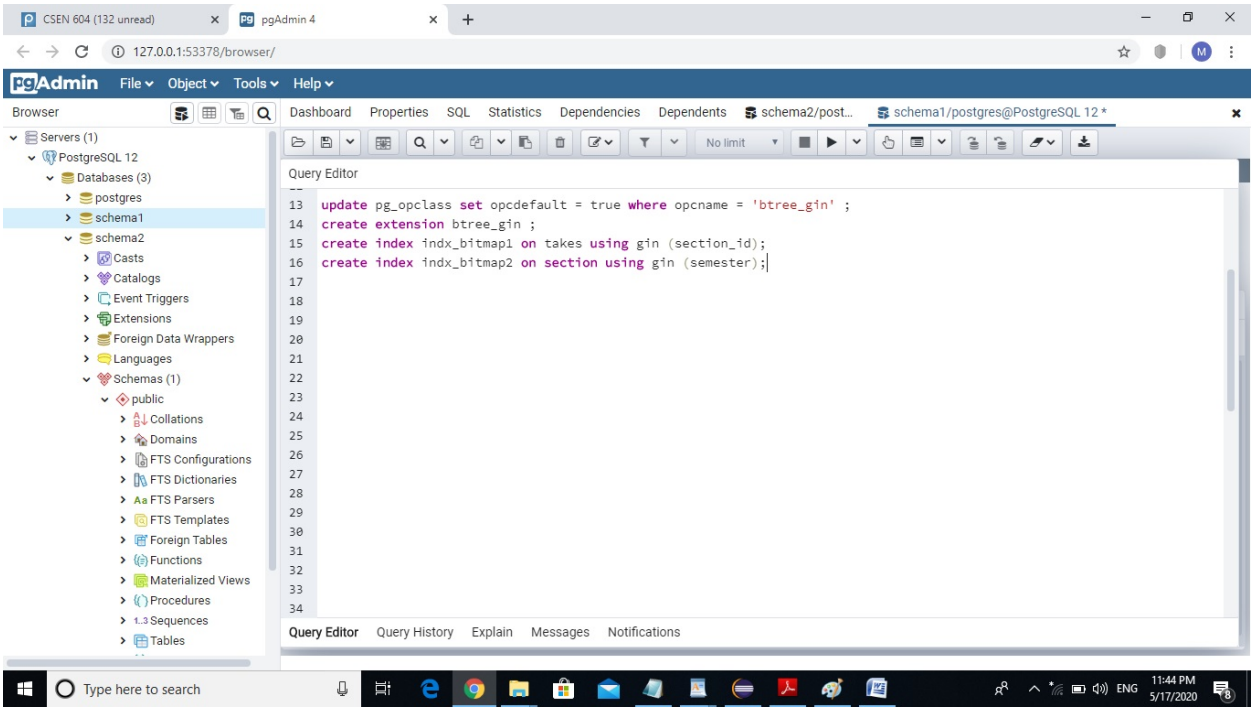
The estimated cost of the plan : 140.62

We can see the huge difference between the execution time without index(12.634 ms) and the execution time with index (3.153 ms) because B+ Tree index support queries on exact values , queries involving a range search and queries containing aggregate functions.

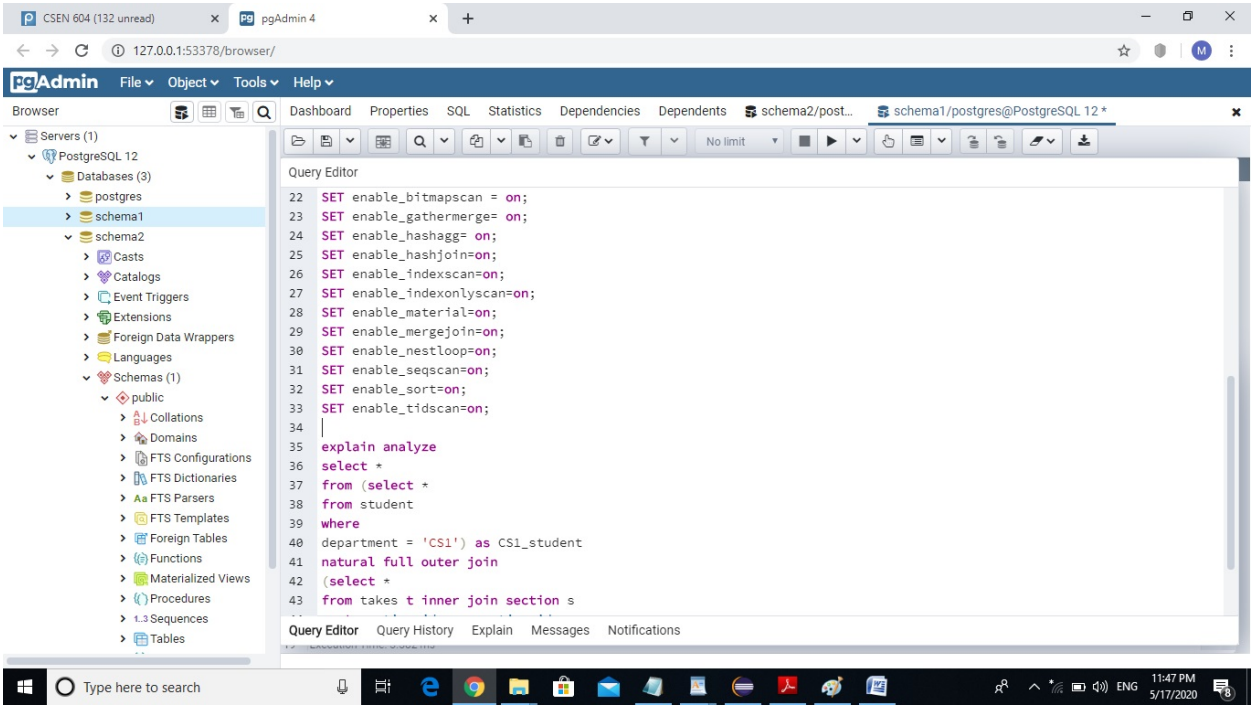
And we have in our query exact values terms such that (semester =1 , year = 2019…) that's why it will work efficiently and in less time when executing the query.

3-Bitmap index:

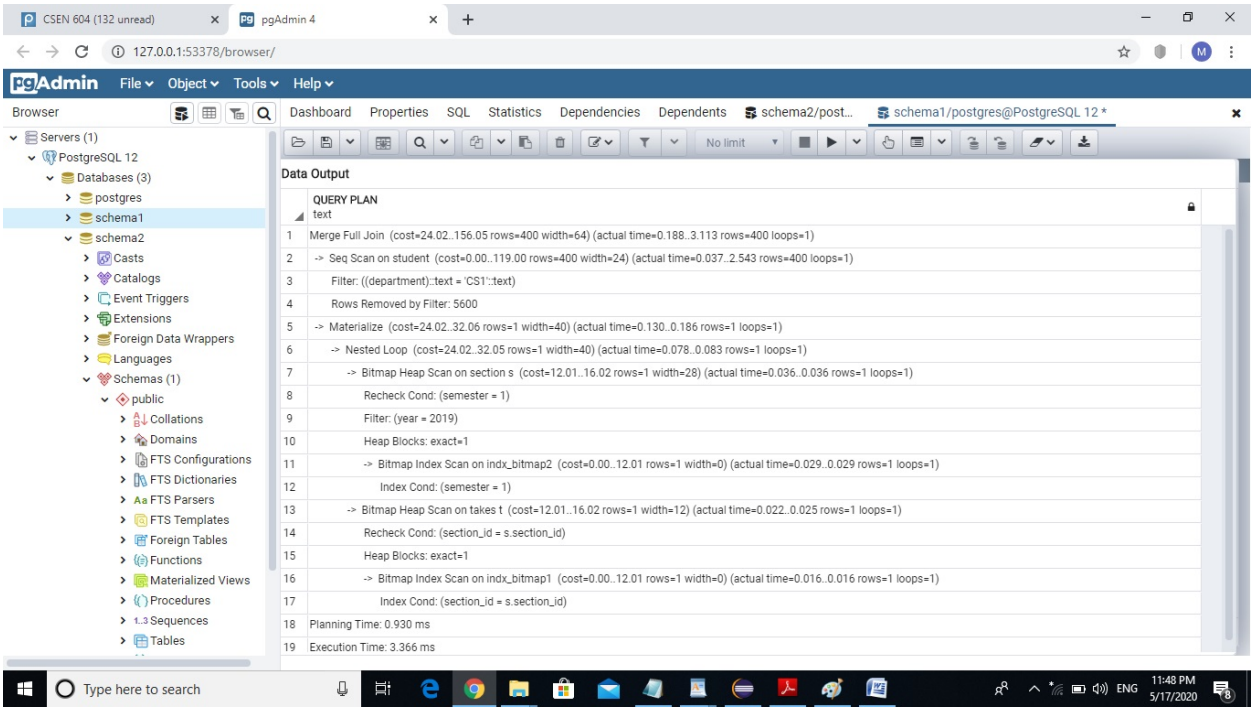
1-Creating the index:

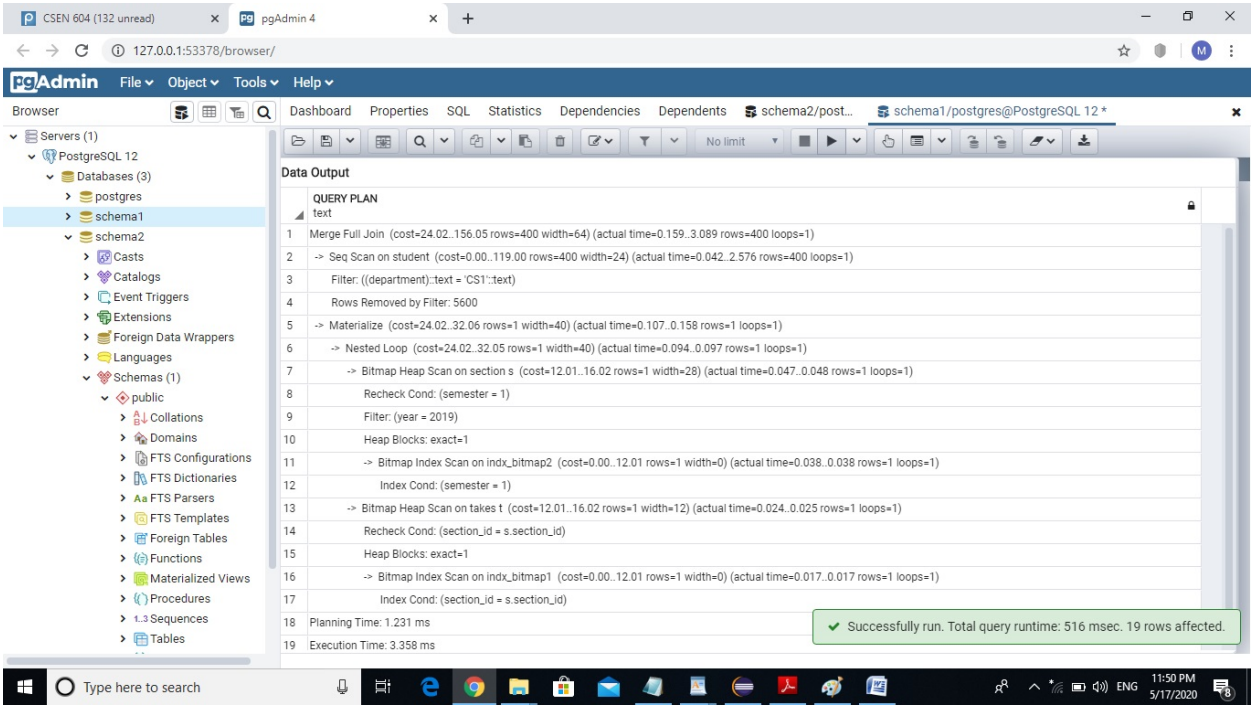


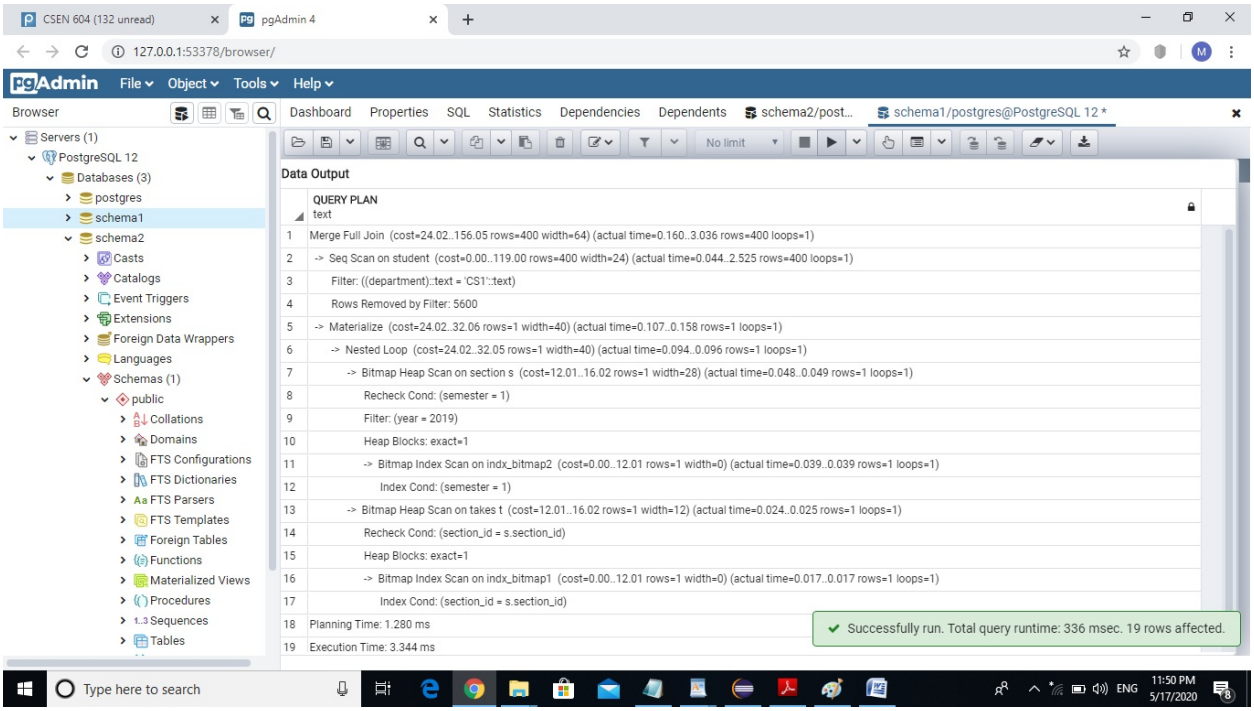
2- Configurations :



3-Query plan :


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=24.02..156.05 rows=400 width=64) (actual time=0.188..3.113 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.037..2.543 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=24.02..32.06 rows=1 width=40) (actual time=0.130..0.186 rows=1 loops=1)
6 -> Nested Loop (cost=24.02..32.05 rows=1 width=40) (actual time=0.078..0.083 rows=1 loops=1)
7 -> Bitmap Heap Scan on section s (cost=12.01..16.02 rows=1 width=28) (actual time=0.036..0.036 rows=1 loops=1)
8 Recheck Cond: (semester = 1)
9 Filter: (year = 2019)
10 Heap Blocks: exact=1
11 -> Bitmap Index Scan on indx_bitmap2 (cost=0.00..12.01 rows=1 width=0) (actual time=0.029..0.029 rows=1 loops=1)
12 Index Cond: (semester = 1)
13 -> Bitmap Heap Scan on takes t (cost=12.01..16.02 rows=1 width=12) (actual time=0.022..0.025 rows=1 loops=1)
14 Recheck Cond: (section_id = s.section_id)
15 Heap Blocks: exact=1
16 -> Bitmap Index Scan on indx_bitmap1 (cost=0.00..12.01 rows=1 width=0) (actual time=0.016..0.016 rows=1 loops=1)
17 Index Cond: (section_id = s.section_id)
18 Planning Time: 0.930 ms
19 Execution Time: 3.366 ms
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Type here to search
ENG
11:48 PM
5/17/2020


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=24.02..156.05 rows=400 width=64) (actual time=0.159..3.089 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.042..2.576 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=24.02..32.06 rows=1 width=40) (actual time=0.107..0.158 rows=1 loops=1)
6 -> Nested Loop (cost=24.02..32.05 rows=1 width=40) (actual time=0.094..0.097 rows=1 loops=1)
7 -> Bitmap Heap Scan on section s (cost=12.01..16.02 rows=1 width=28) (actual time=0.047..0.048 rows=1 loops=1)
8 Recheck Cond: (semester = 1)
9 Filter: (year = 2019)
10 Heap Blocks: exact=1
11 -> Bitmap Index Scan on indx_bitmap2 (cost=0.00..12.01 rows=1 width=0) (actual time=0.038..0.038 rows=1 loops=1)
12 Index Cond: (semester = 1)
13 -> Bitmap Heap Scan on takes t (cost=12.01..16.02 rows=1 width=12) (actual time=0.024..0.025 rows=1 loops=1)
14 Recheck Cond: (section_id = s.section_id)
15 Heap Blocks: exact=1
16 -> Bitmap Index Scan on indx_bitmap1 (cost=0.00..12.01 rows=1 width=0) (actual time=0.017..0.017 rows=1 loops=1)
17 Index Cond: (section_id = s.section_id)
18 Planning Time: 1.231 ms
19 Execution Time: 3.358 ms
Successfully run. Total query runtime: 516 msec. 19 rows affected.
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Type here to search
ENG
11:50 PM
5/17/2020



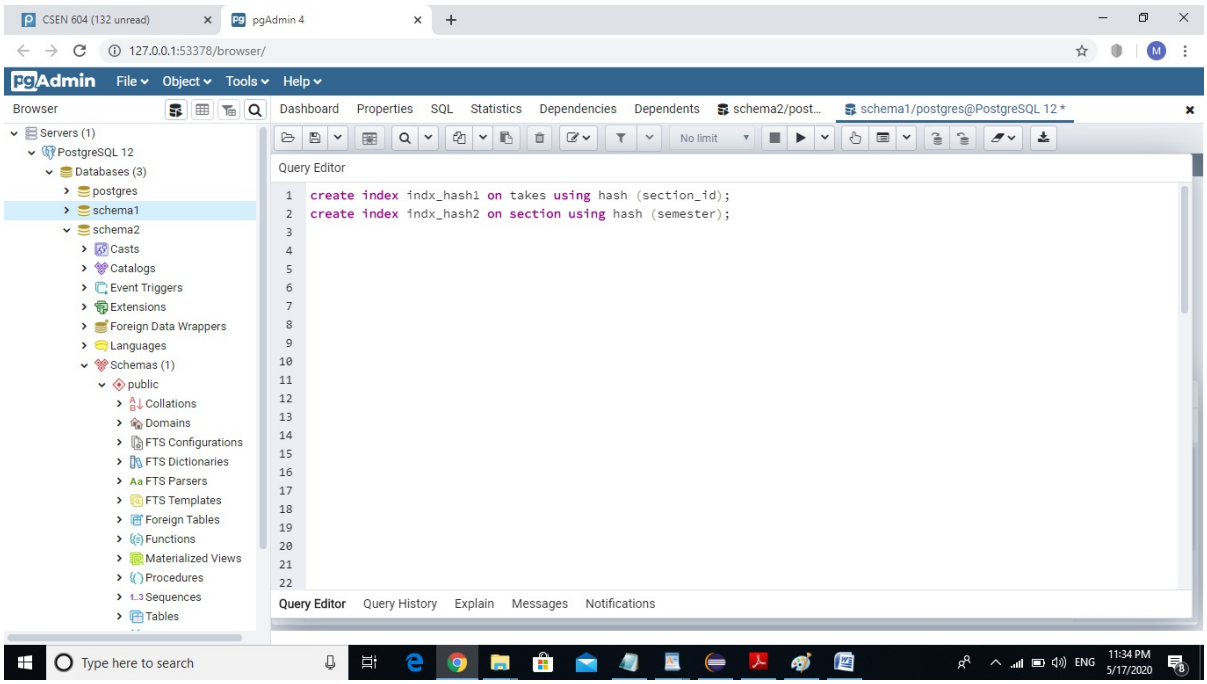
Average execution time : (3.366 + 3.358 + 3.344) / 3 = 3.356 ms.

The estimated cost of the plan : 156.05

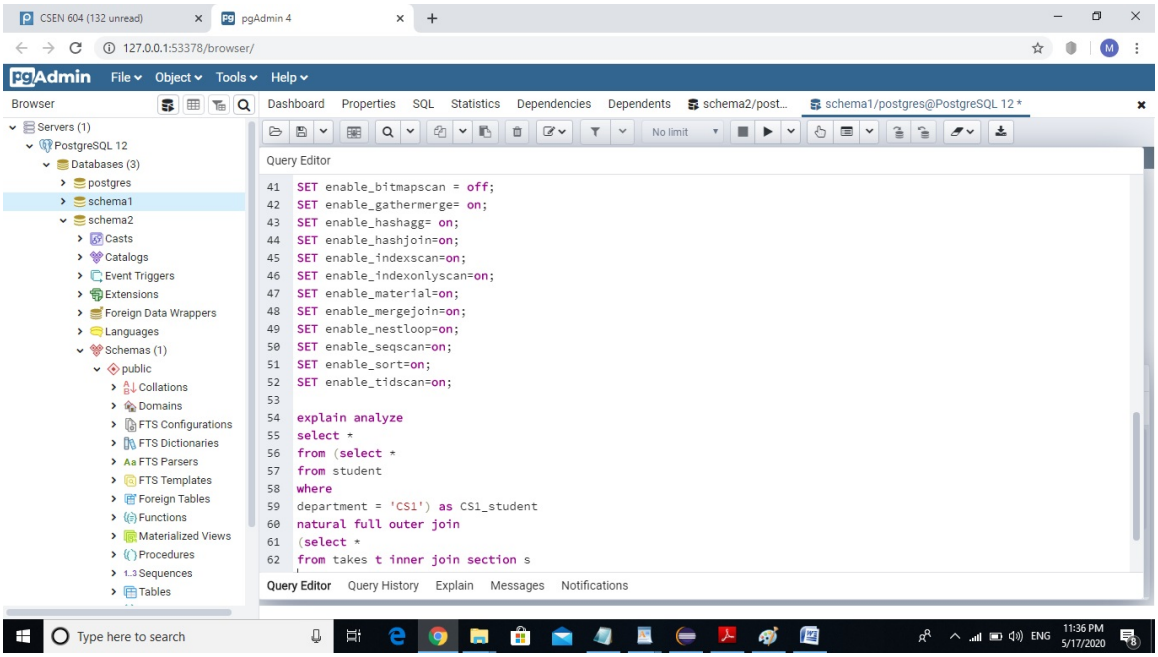
The huge difference between the execution time without index(12.634 ms) and the execution time with index (3.356 ms) because BitMap index answer many query types including on specific value, And, OR, XOR, and range .And we have in our query specific values terms such that (semester =1 , year = 2019…), we also have a bitwise operation ‘and’ that’s why it will work efficiently and in less time when executing the query.

4-HashBased index:

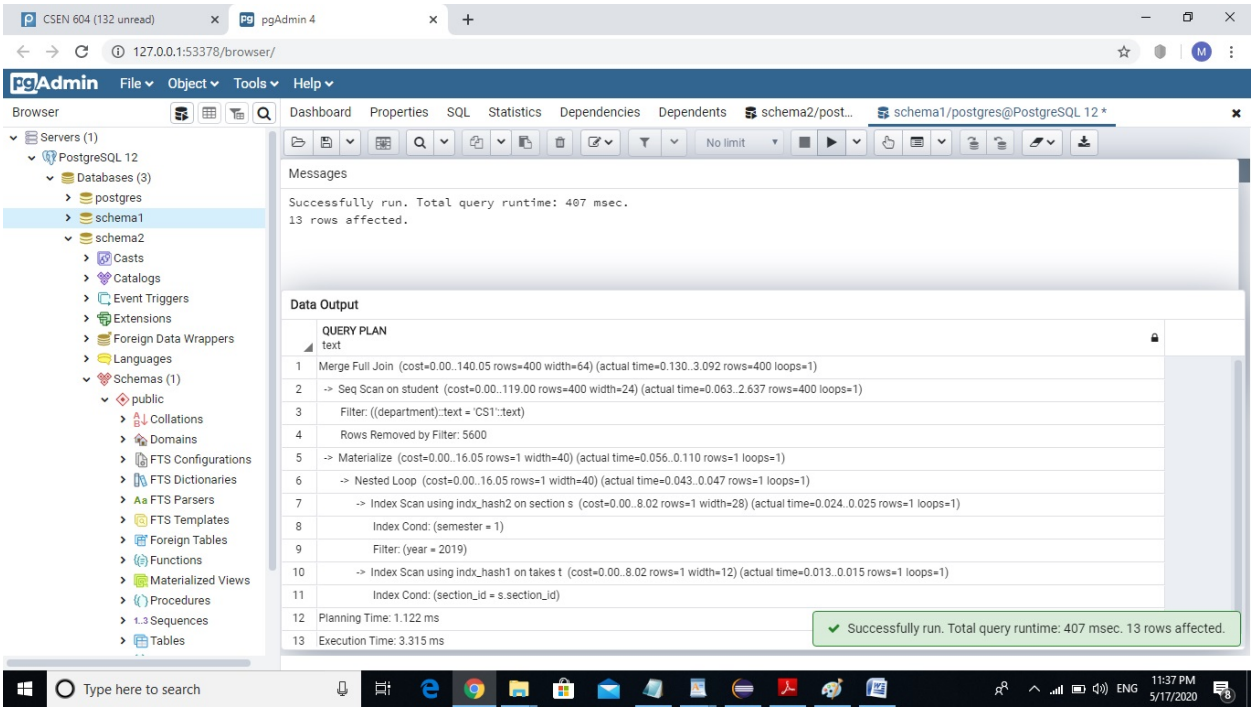
1. Creating the index:

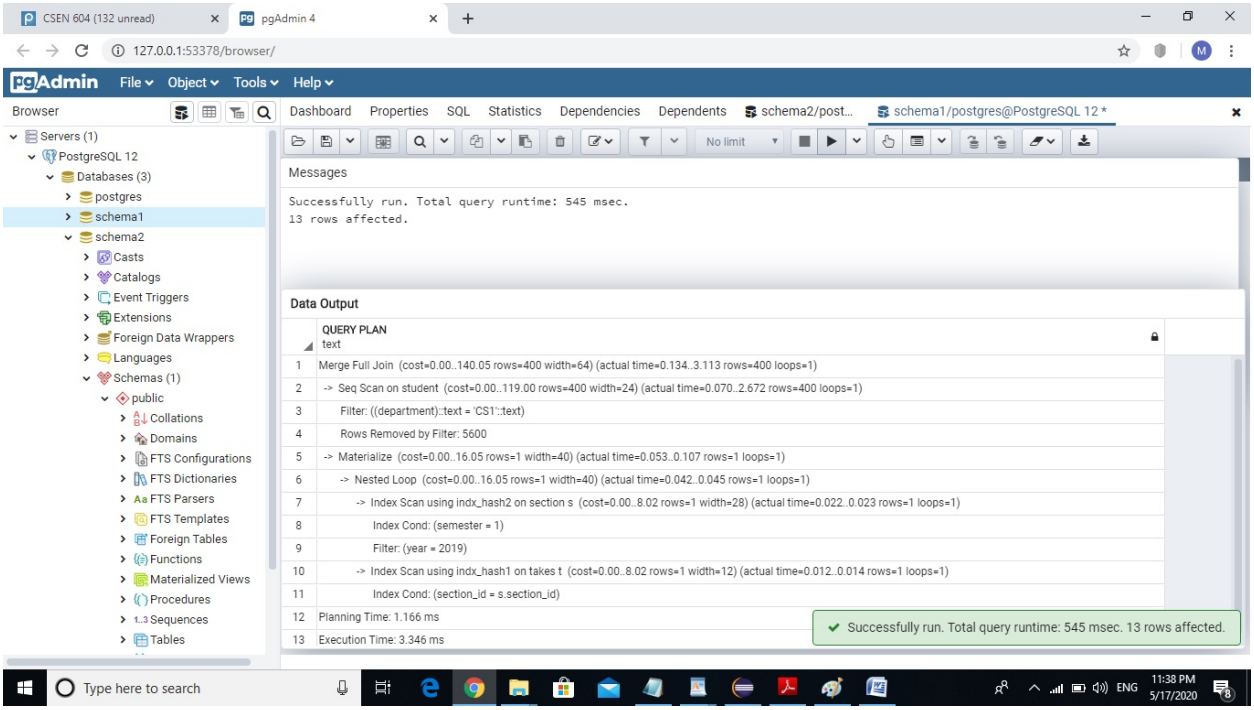


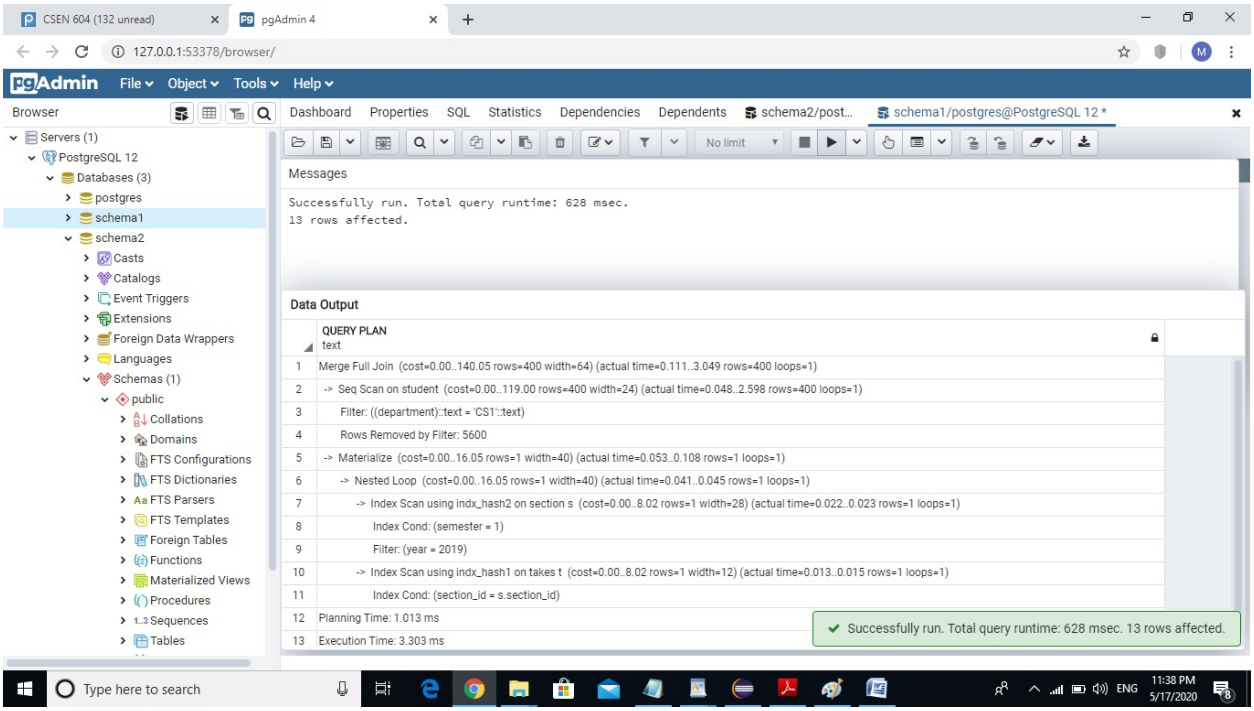
2. Configurations :



3-Query plan :


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Messages
Successfully run. Total query runtime: 407 msec.
13 rows affected.
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=0.00..140.05 rows=400 width=64) (actual time=0.130..3.092 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.063..2.637 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=0.00..16.05 rows=1 width=40) (actual time=0.056..0.110 rows=1 loops=1)
6 -> Nested Loop (cost=0.00..16.05 rows=1 width=40) (actual time=0.043..0.047 rows=1 loops=1)
7 -> Index Scan using indx_hash2 on section s (cost=0.00..8.02 rows=1 width=28) (actual time=0.024..0.025 rows=1 loops=1)
8 Index Cond: (semester = 1)
9 Filter: (year = 2019)
10 -> Index Scan using indx_hash1 on takes t (cost=0.00..8.02 rows=1 width=12) (actual time=0.013..0.015 rows=1 loops=1)
11 Index Cond: (section_id = s.section_id)
12 Planning Time: 1.122 ms
13 Execution Time: 3.315 ms
Successfully run. Total query runtime: 407 msec. 13 rows affected.
Type here to search
ENG
11:37 PM
5/17/2020


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/post...
schema1/postgres@PostgreSQL 12 *
No limit
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Messages
Successfully run. Total query runtime: 545 msec.
13 rows affected.
Data Output
QUERY PLAN
text
1 Merge Full Join (cost=0.00..140.05 rows=400 width=64) (actual time=0.134..3.113 rows=400 loops=1)
2 -> Seq Scan on student (cost=0.00..119.00 rows=400 width=24) (actual time=0.070..2.672 rows=400 loops=1)
3 Filter: ((department)::text = 'CS1'::text)
4 Rows Removed by Filter: 5600
5 -> Materialize (cost=0.00..16.05 rows=1 width=40) (actual time=0.053..0.107 rows=1 loops=1)
6 -> Nested Loop (cost=0.00..16.05 rows=1 width=40) (actual time=0.042..0.045 rows=1 loops=1)
7 -> Index Scan using indx_hash2 on section s (cost=0.00..8.02 rows=1 width=28) (actual time=0.022..0.023 rows=1 loops=1)
8 Index Cond: (semester = 1)
9 Filter: (year = 2019)
10 -> Index Scan using indx_hash1 on takes t (cost=0.00..8.02 rows=1 width=12) (actual time=0.012..0.014 rows=1 loops=1)
11 Index Cond: (section_id = s.section_id)
12 Planning Time: 1.166 ms
13 Execution Time: 3.346 ms
Successfully run. Total query runtime: 545 msec. 13 rows affected.
Type here to search
ENG
11:38 PM
5/17/2020



The average execution time : (3.315 + 3.346+ 3.303)/3 = 3.321 ms

The estimated cost of the plan : 140.05

The huge difference between the execution time without index(12.634 ms) and the execution time with index (3.321 ms) because HashBased index is used to enhance the performance of queries on exact values only.

And that's what we have in our query exact values terms such that (semester =1 , year = 2019…) that's why it will work efficiently and in less time when executing the query.

Explanation of the best performer :

First of all , for a fair comparison to all the scenarios : we put the indexes on takes (section_id) and on section (semester) , and we substituted one query planner choosen index by a seq scan and kept it always seq scan across all scenarios.

And now by comparing the cost and the average execution time of each scenario we can see:

Scenario1: Without an index

Average execution time: 12.634 ms

Estimated cost of the plan : 627.96

Scenario2: B+Tree index

Average execution time : 3.153 ms

Estimated cost of the plan: 140.62

Scenario3: Bitmap index

Average execution time : 3.356 ms.

Estimated cost of the plan : 156.05

Scenario4: HashBased index

Average execution time : 3.321 ms

Estimated cost of the plan : 140.05

-It is clear now that Scenario1 is the most costly and has the longest execution time due to a sequential scan on all tables (student , section and takes).

-using The HashBased index, was considered the best performer of average execution time of 3.321 ms and a cost of 140.05 because HashBased index work efficiently on exact values queries which is our case here in our query (semester = 1 , year = 2019) and also due to our insertion and the duplicate rows.

-The B+Tree index is also acceptable in averge execution time 3.153 ms and cost 140.62 because B+Tree index is efficient when we have queries on exact values also.

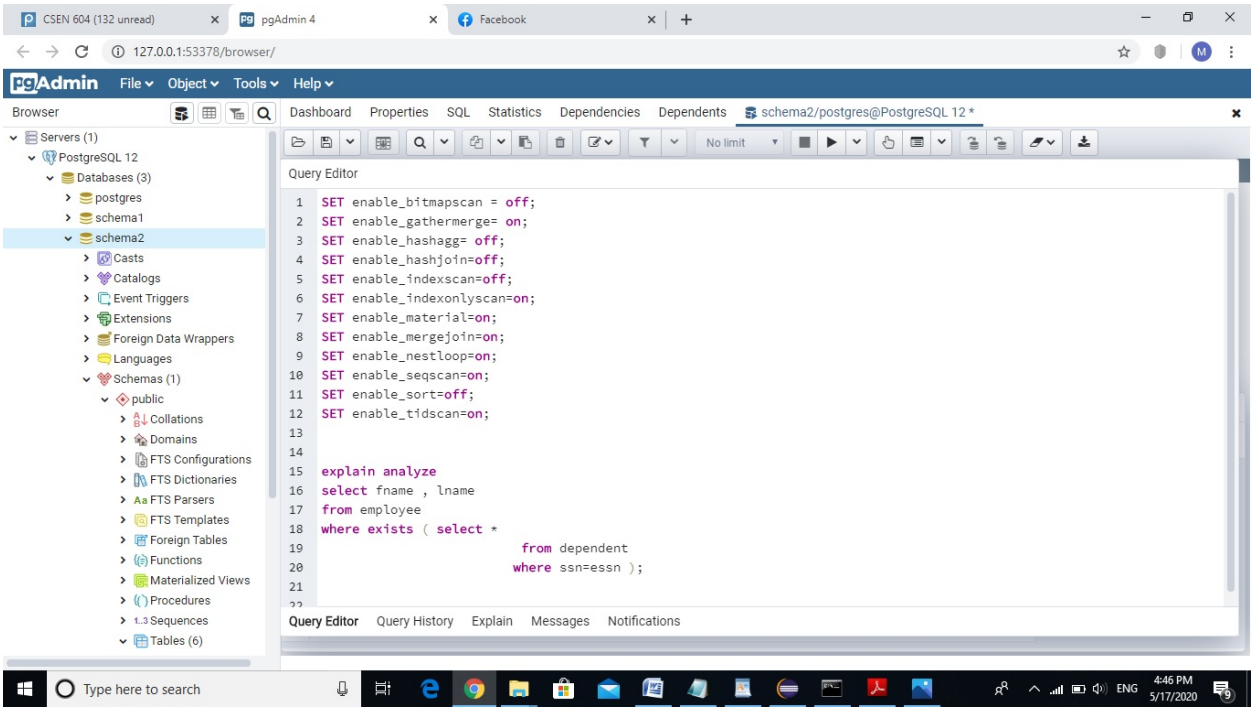
-Finally , The bitmap index has the longest execution time and the highest cost among all other scenarios.

SCHEMA 2

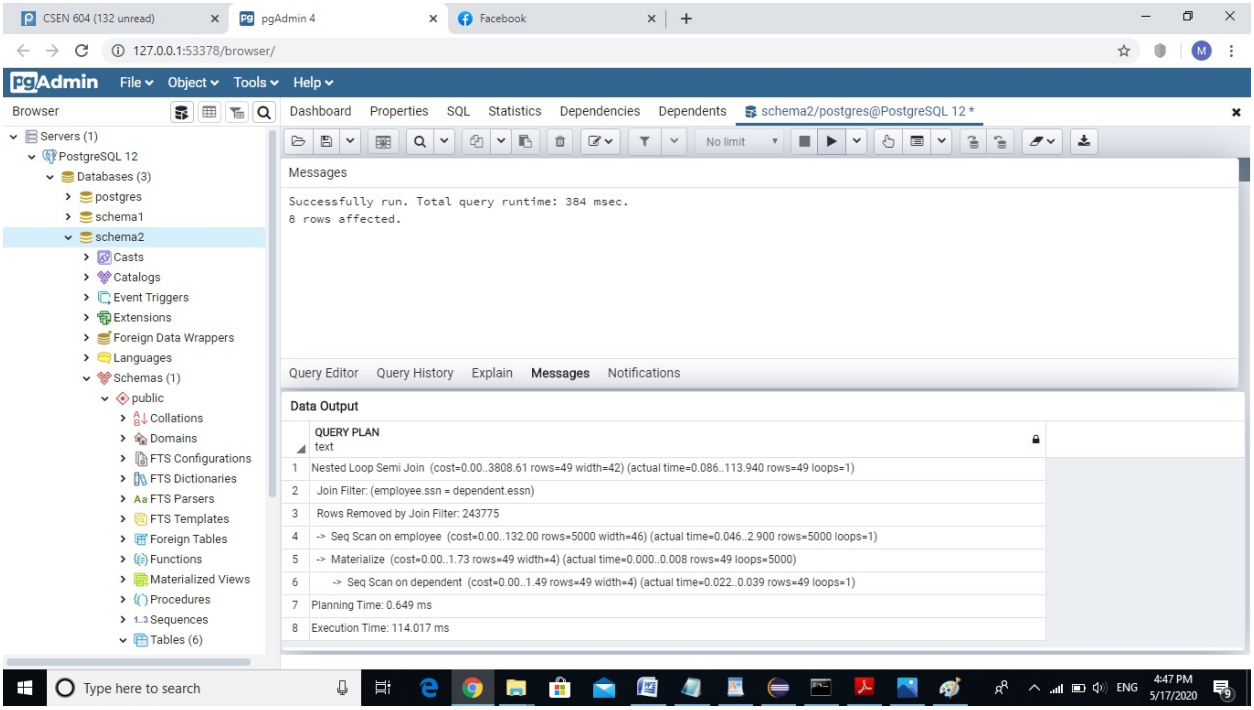
Query 5:

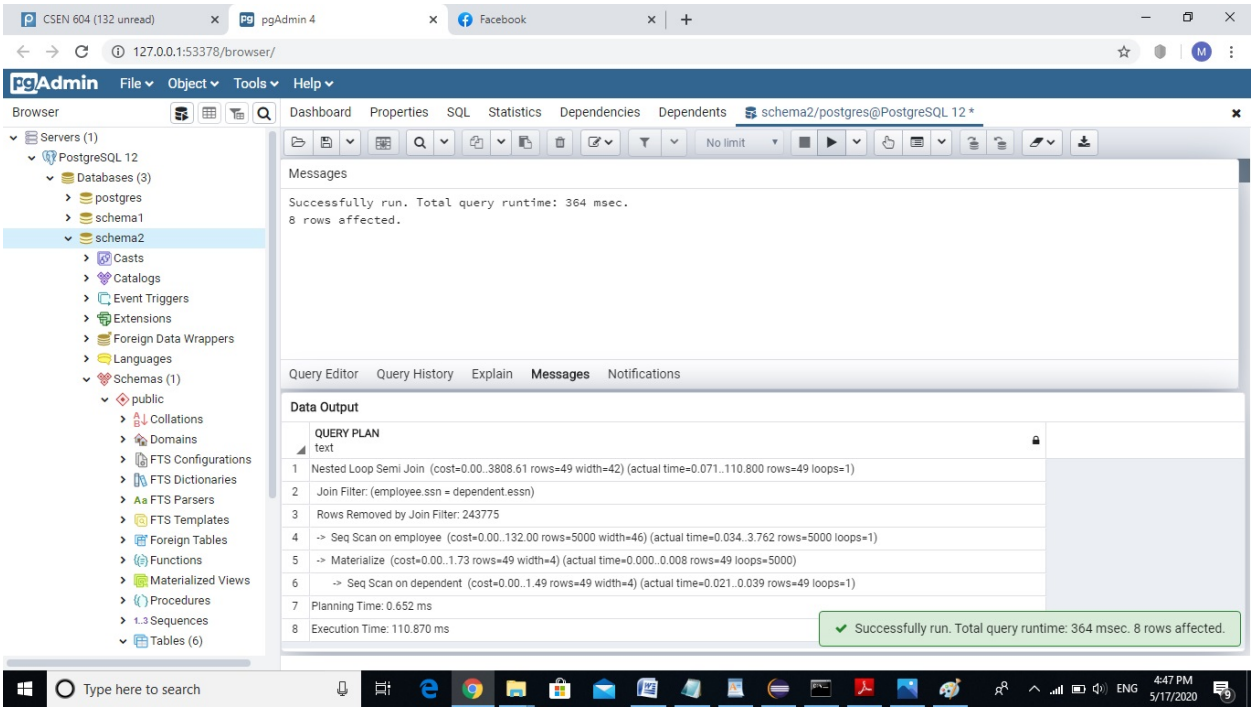
1-Without index :

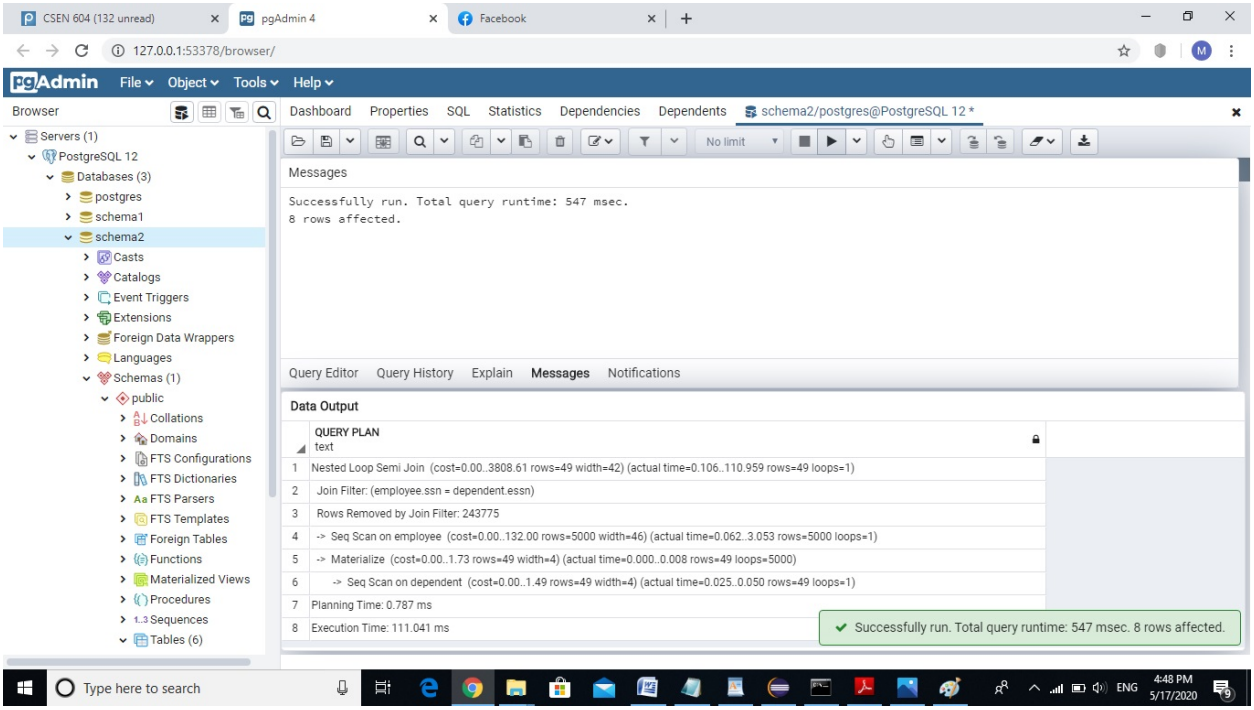
1-Configurations :


CSEN 604 (132 unread)
pgAdmin 4
Facebook
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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 (6)
No limit
Query Editor
1 SET enable_bitmapscan = off;
2 SET enable_gathermerge= on;
3 SET enable_hashagg= off;
4 SET enable_hashjoin=off;
5 SET enable_indexscan=off;
6 SET enable_indexonlyscan=on;
7 SET enable_material=on;
8 SET enable_mergejoin=on;
9 SET enable_nestloop=on;
10 SET enable_seqscan=on;
11 SET enable_sort=off;
12 SET enable_tidscan=on;
13
14
15 explain analyze
16 select fname , lname
17 from employee
18 where exists (select *
19 from dependent
20 where ssn=essn);
21
Query Editor
Query History
Explain
Messages
Notifications
Type here to search
ENG
4:46 PM
5/17/2020

2-Query plan :


CSEN 604 (132 unread)
pgAdmin 4
Facebook
127.0.0.1:53378/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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 (6)
No limit
Messages
Successfully run. Total query runtime: 384 msec.
8 rows affected.
Query Editor
Query History
Explain
Messages
Notifications
Data Output
QUERY PLAN
text
1 Nested Loop Semi Join (cost=0.00..3808.61 rows=49 width=42) (actual time=0.086..113.940 rows=49 loops=1)
2 Join Filter: (employee.ssn = dependent.essn)
3 Rows Removed by Join Filter: 243775
4 -> Seq Scan on employee (cost=0.00..132.00 rows=5000 width=46) (actual time=0.046..2.900 rows=5000 loops=1)
5 -> Materialize (cost=0.00..1.73 rows=49 width=4) (actual time=0.000..0.008 rows=49 loops=5000)
6 -> Seq Scan on dependent (cost=0.00..1.49 rows=49 width=4) (actual time=0.022..0.039 rows=49 loops=1)
7 Planning Time: 0.649 ms
8 Execution Time: 114.017 ms
Type here to search
ENG
4:47 PM
5/17/2020



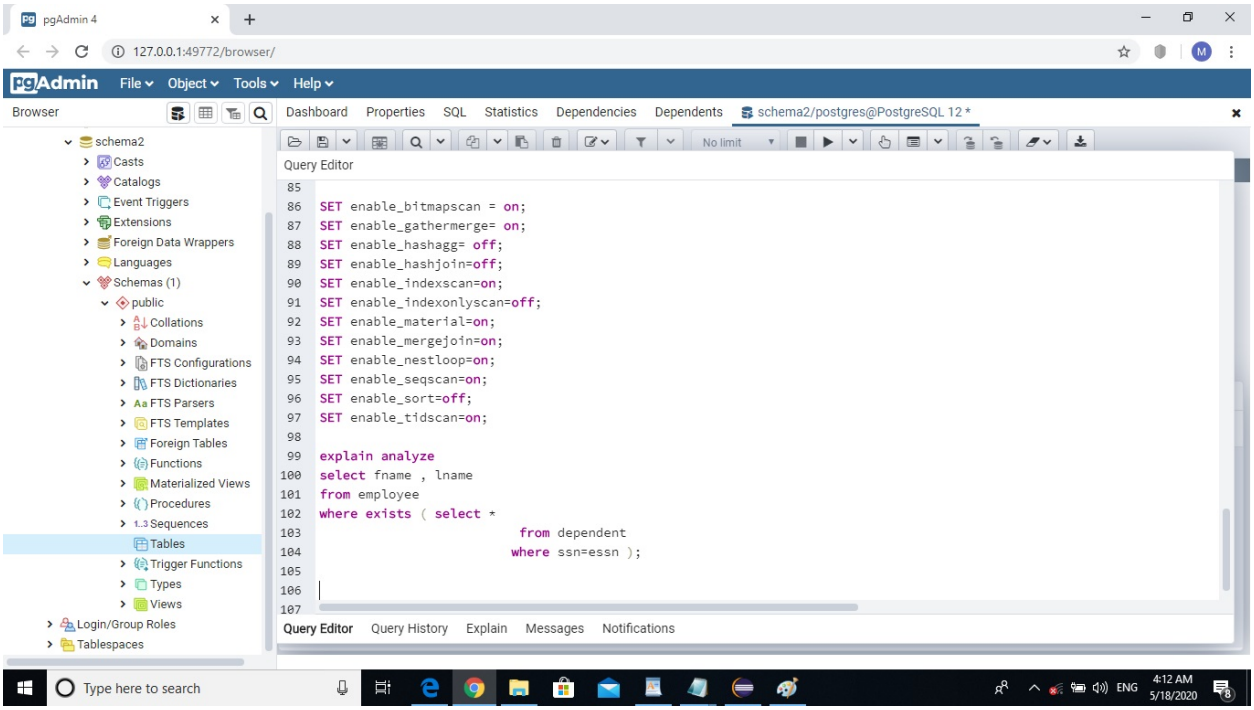


The Average execution time : (114.017 + 110.870 + 111.041) / 3 = 125.629 ms

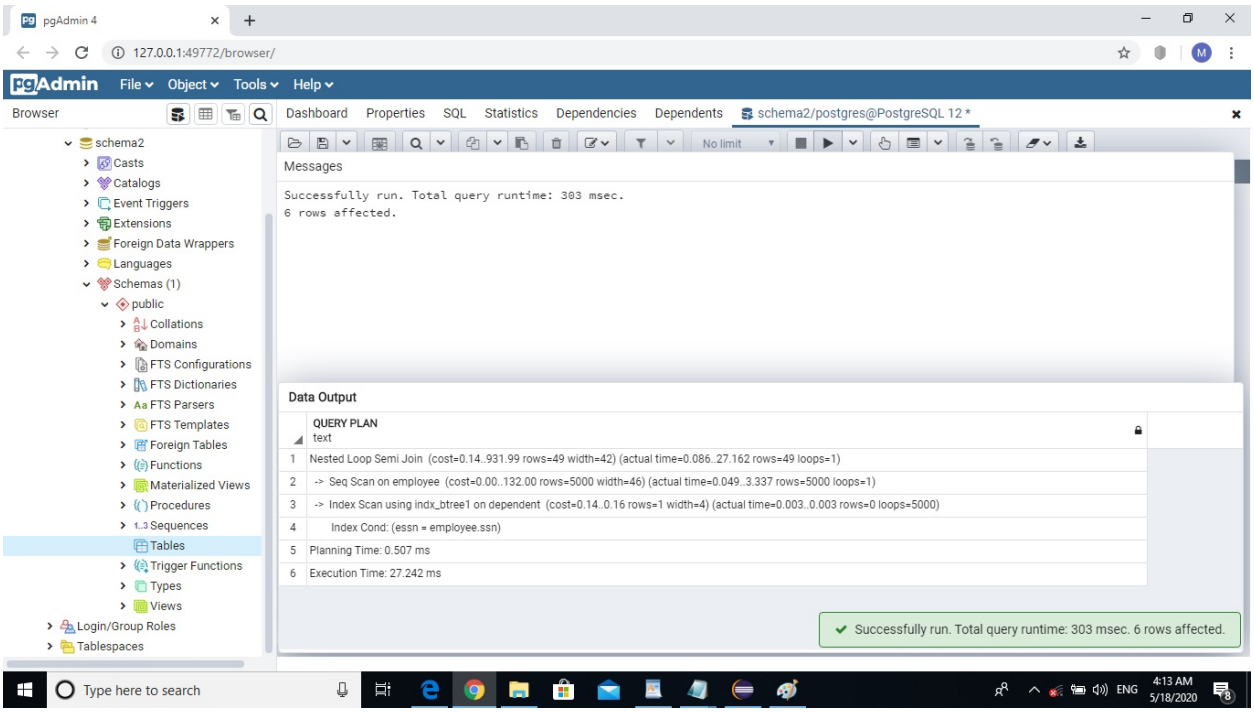
The Estimated cost of the plan : 3808.61

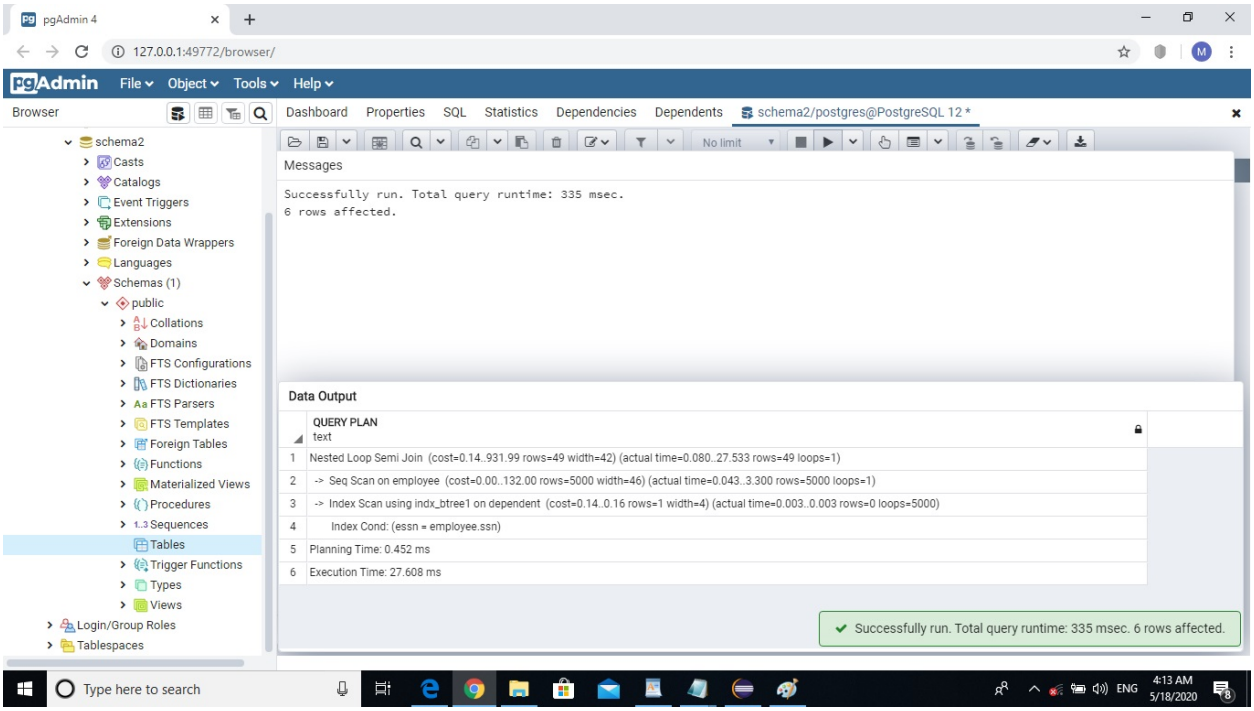
2-B+Tree index:

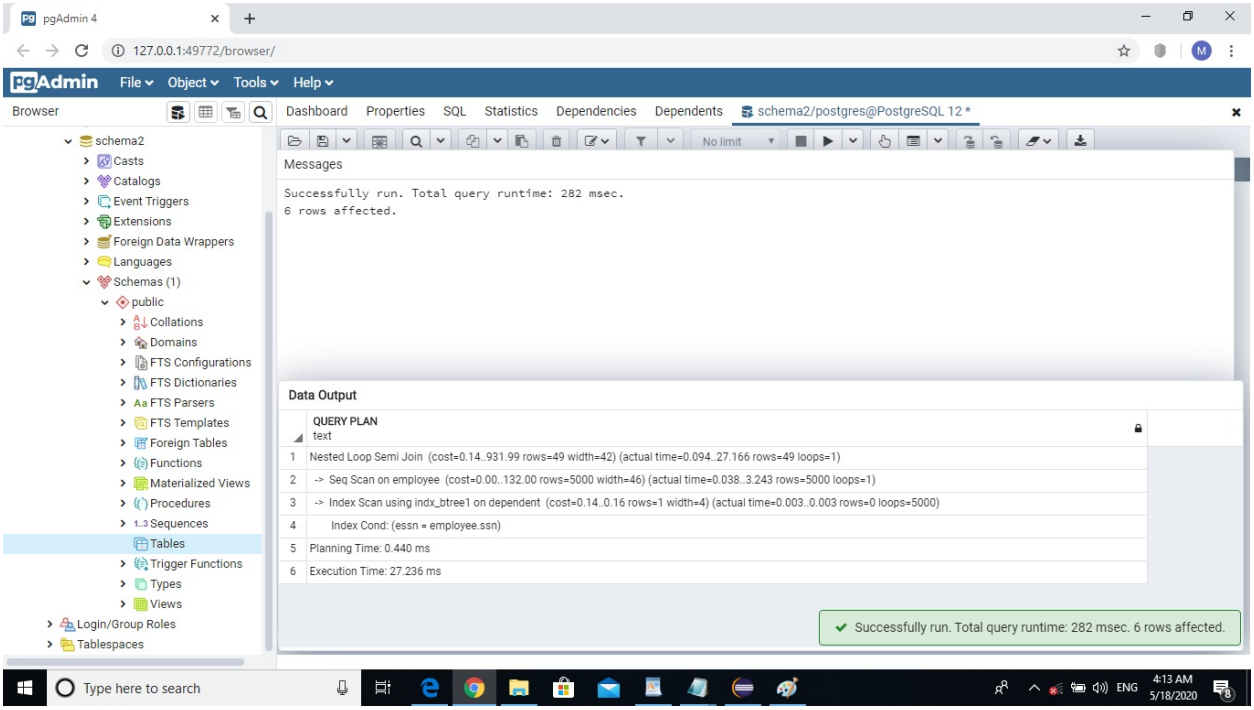
1-Configurations :



2-Query plan :





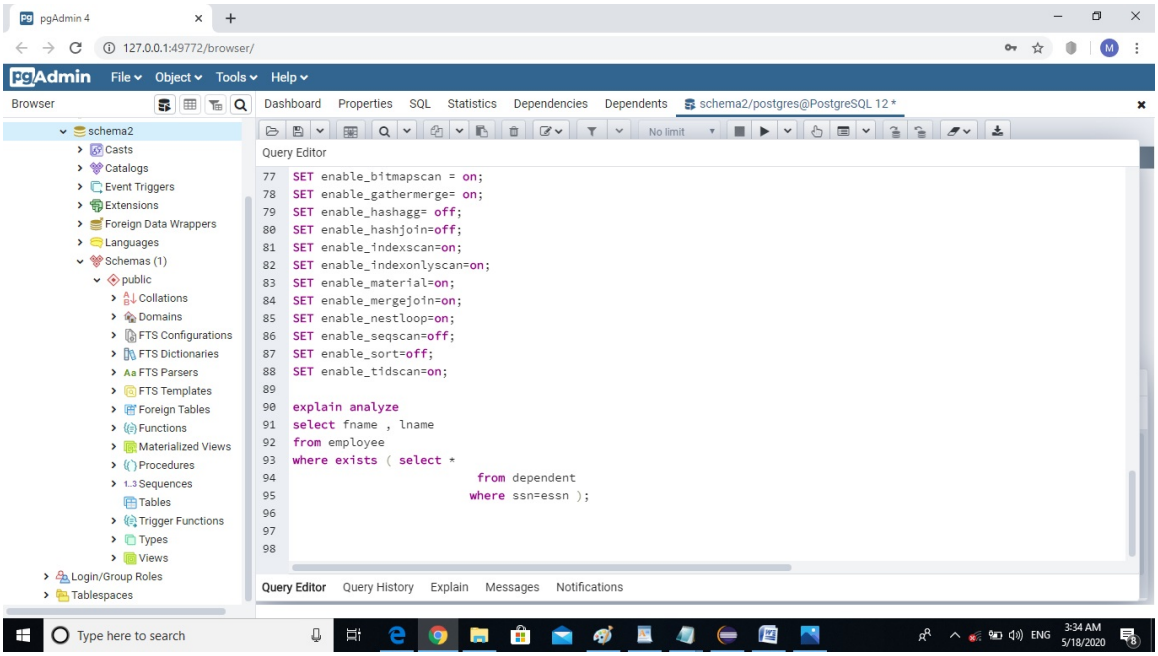


The Average execution time : (27.242 + 27.608 +27.236) /3 = 27.362 ms

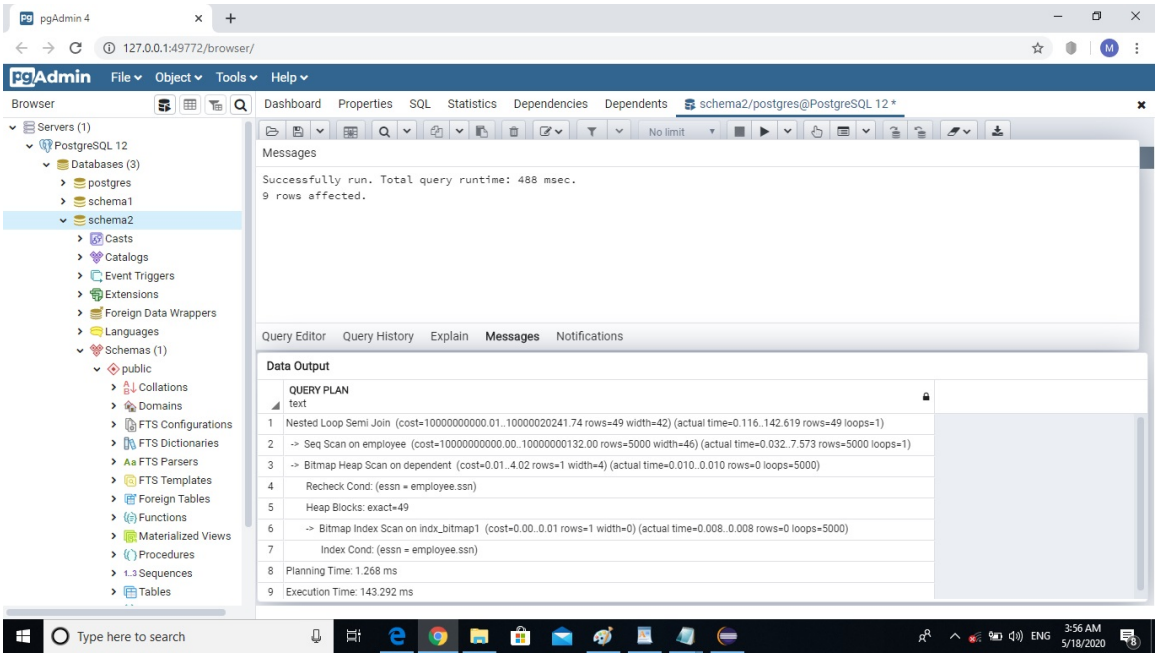
The Estimated cost of the plan : 931.99

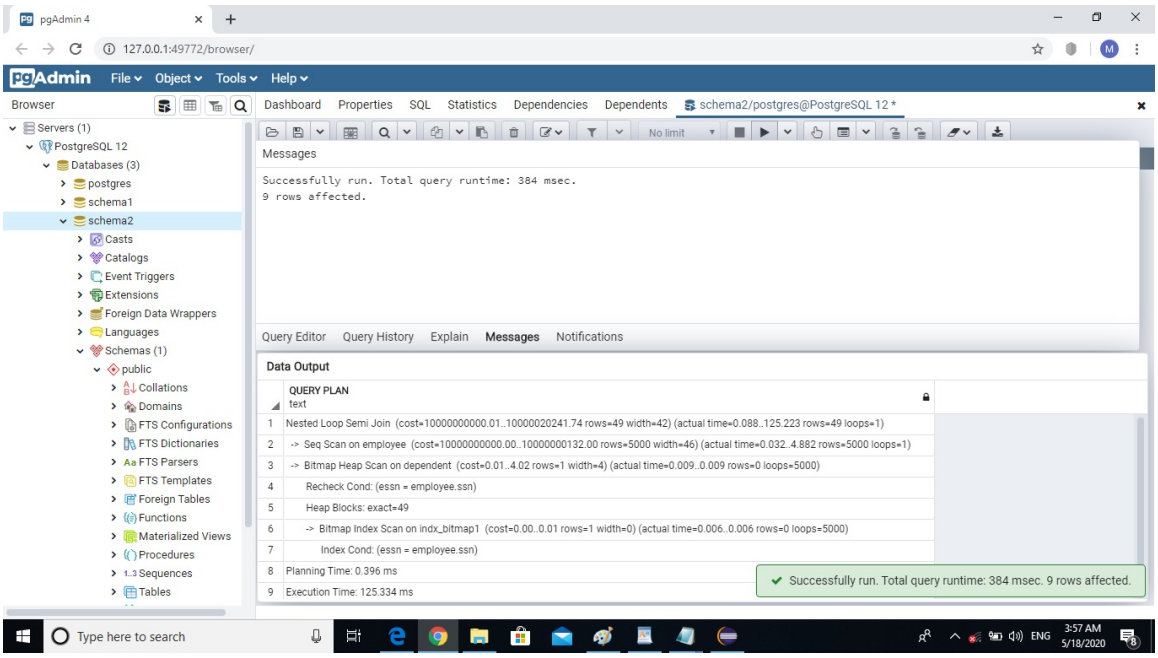
3-Bitmap index :

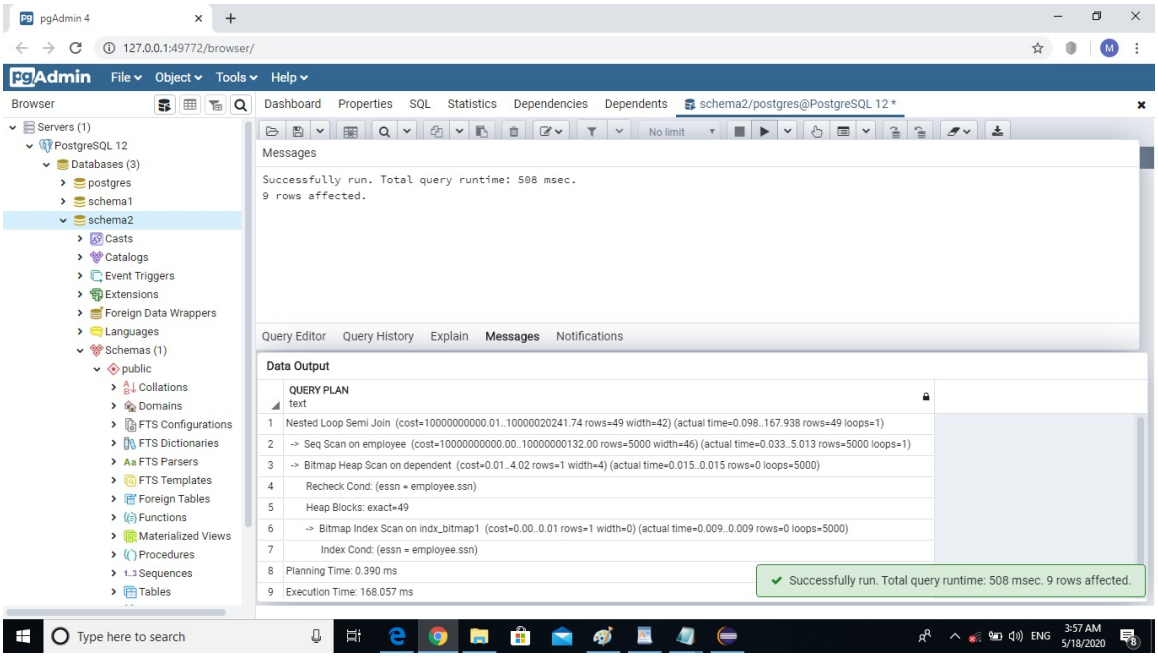
1. Configurations :



2. Query plan :





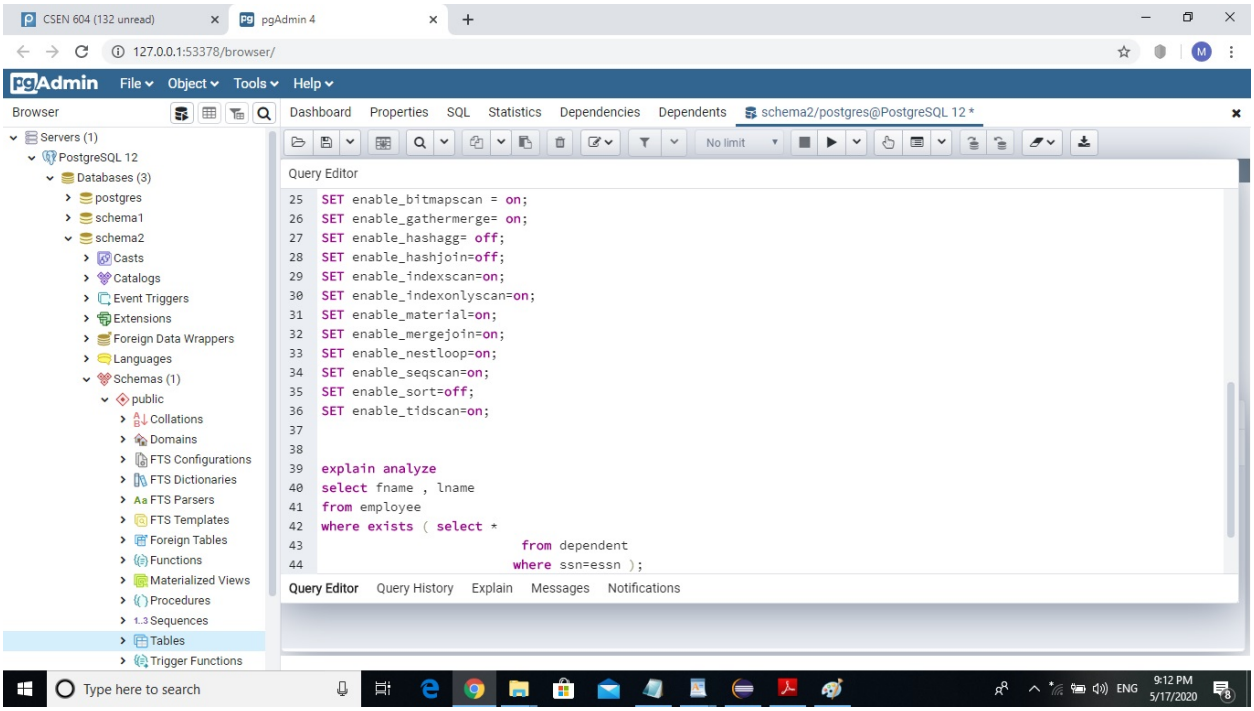


The average execution time: (143.292 + 125.334 + 168.057)/3 =145.561 ms

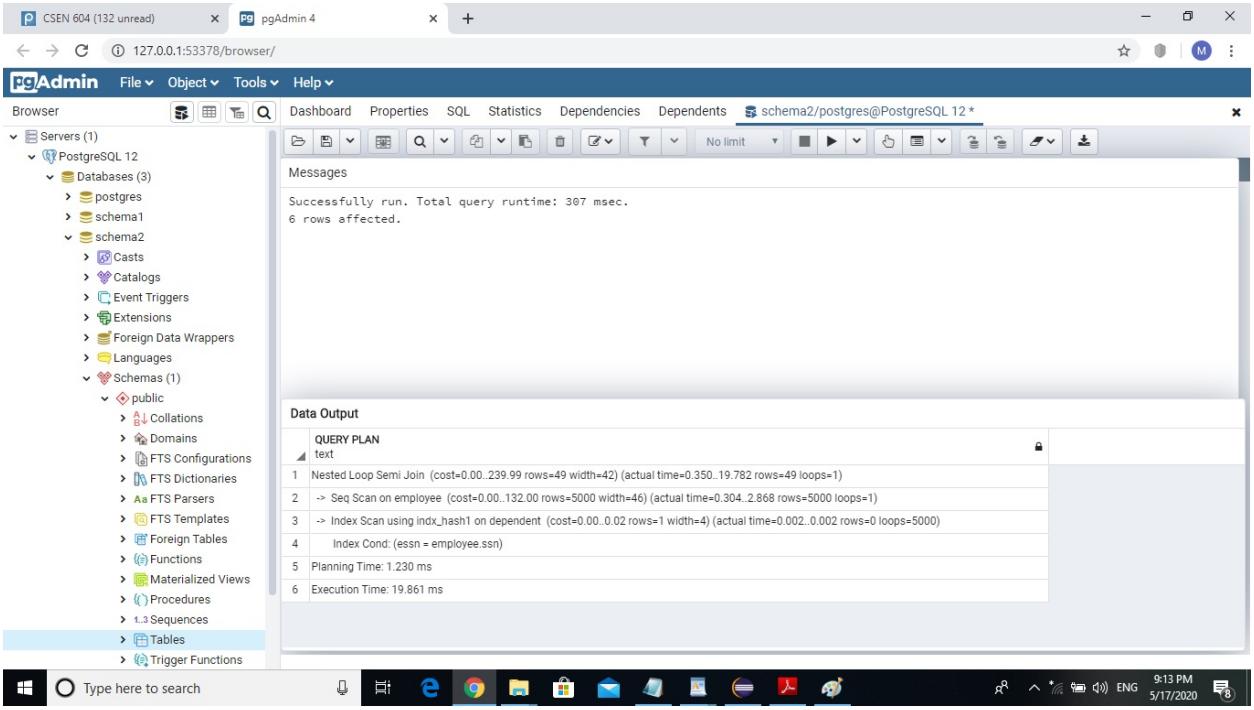
The estimated cost of the plan : 10000020241.74

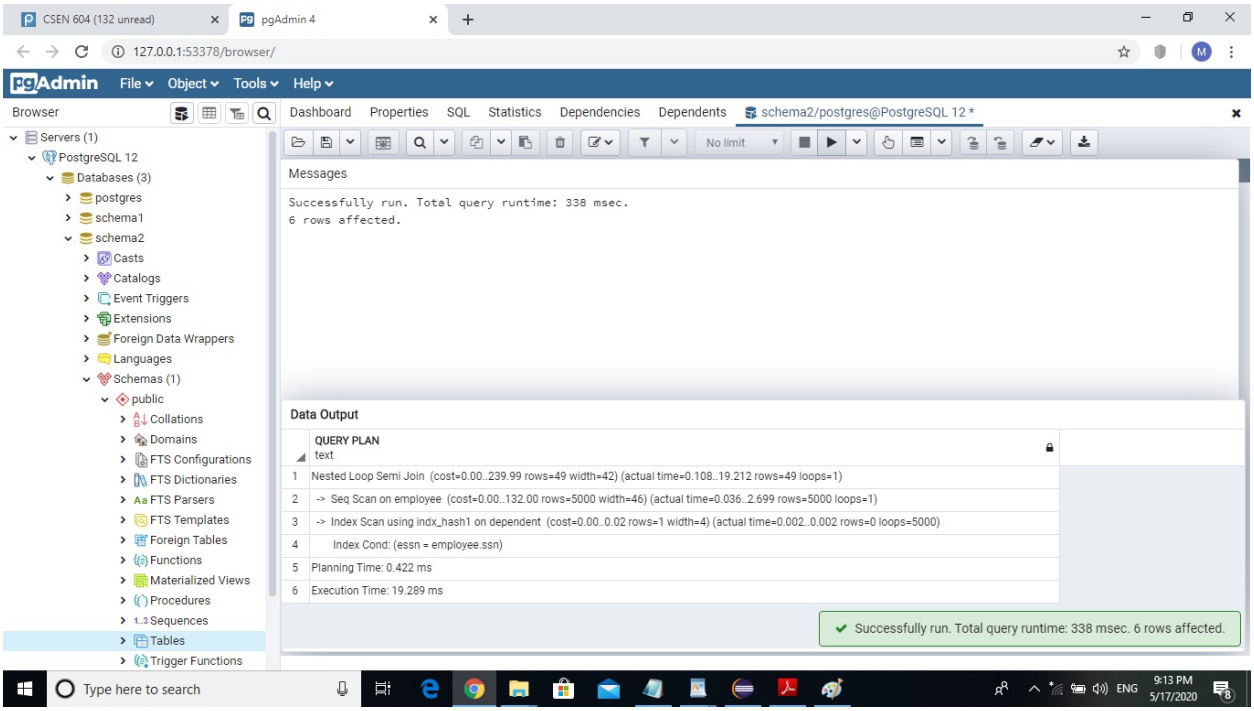
4-HashBased index :

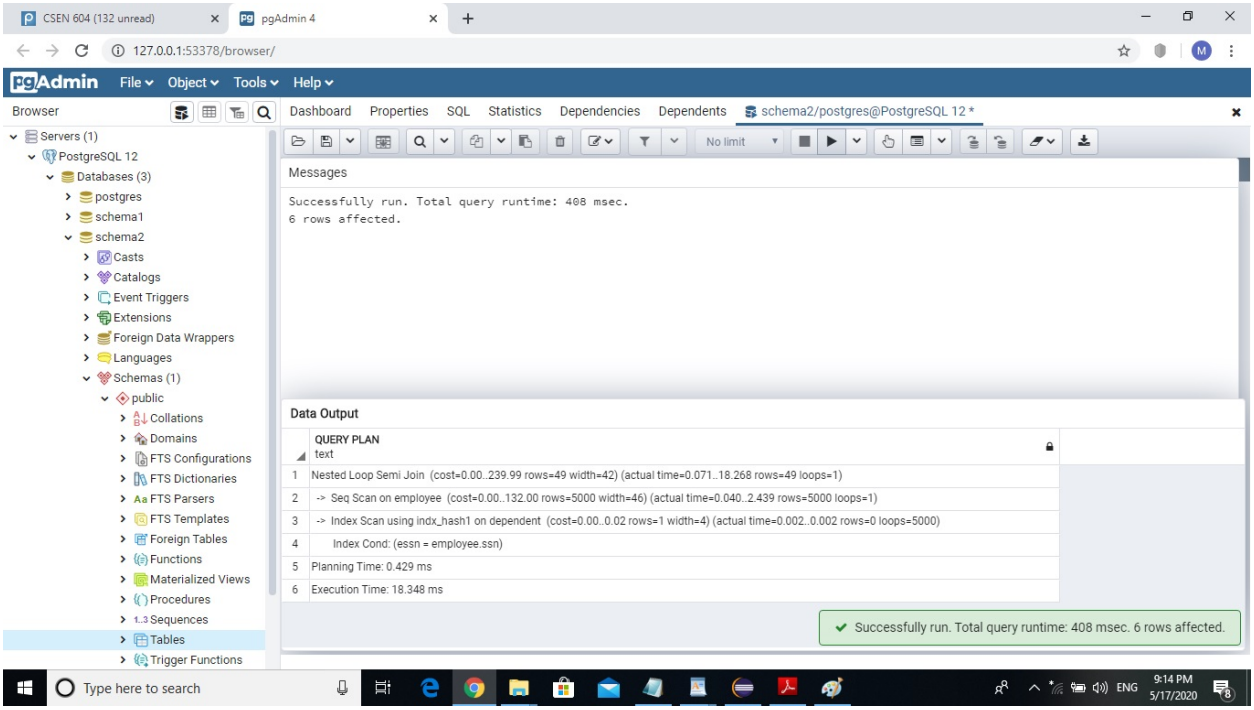
1-Configurations :


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin File Object Tools Help
Browser
Dashboard Properties SQL Statistics Dependencies Dependents schema2/postgres@PostgreSQL 12 *
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Trigger Functions
Query Editor
25 SET enable_bitmapscan = on;
26 SET enable_gathermerge= on;
27 SET enable_hashagg= off;
28 SET enable_hashjoin=off;
29 SET enable_indexscan=on;
30 SET enable_indexonlyscan=on;
31 SET enable_material=on;
32 SET enable_mergejoin=on;
33 SET enable_nestloop=on;
34 SET enable_seqscan=on;
35 SET enable_sort=off;
36 SET enable_tidscan=on;
37
38
39 explain analyze
40 select fname , lname
41 from employee
42 where exists (select *
43 from dependent
44 where ssn=essn);
Query Editor Query History Explain Messages Notifications
Type here to search
ENG
9:12 PM
5/17/2020

2-Query plan :


CSEN 604 (132 unread)
pgAdmin 4
127.0.0.1:53378/browser/
pgAdmin File Object Tools Help
Browser
Dashboard Properties SQL Statistics Dependencies Dependents schema2/postgres@PostgreSQL 12 *
Servers (1)
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Trigger Functions
Messages
Successfully run. Total query runtime: 307 msec.
6 rows affected.
Data Output
QUERY PLAN
text
1 Nested Loop Semi Join (cost=0.00..239.99 rows=49 width=42) (actual time=0.350..19.782 rows=49 loops=1)
2 -> Seq Scan on employee (cost=0.00..132.00 rows=5000 width=46) (actual time=0.304..2.868 rows=5000 loops=1)
3 -> Index Scan using indx_hash1 on dependent (cost=0.00..0.02 rows=1 width=4) (actual time=0.002..0.002 rows=0 loops=5000)
4 Index Cond: (essn = employee.ssn)
5 Planning Time: 1.230 ms
6 Execution Time: 19.861 ms
Type here to search
ENG
9:13 PM
5/17/2020





The average execution time : (19.861 + 19.289 + 18.348) /3 = 19.166 ms

The average cost of the plan : 239.99

Explanation :

We restricted our index to be only on table dependent (essn) , and do a SeqScan on employee table across all the scenarios.

By comparing the results of the query plan of all the scenarios , HashBased index was the fastest in execution time(19.166 ms) and the lowest costing(239.99) compared to all other scenarios , because HashBased index work efficiently on exact values only which is our case here in our query and our insertion.

However , Bitmap index was the most costly and took the most execution time to complete , because the sequential scan configuration was set to off , and it did a SeqScan on the employee table which increased the cost and the execution time .

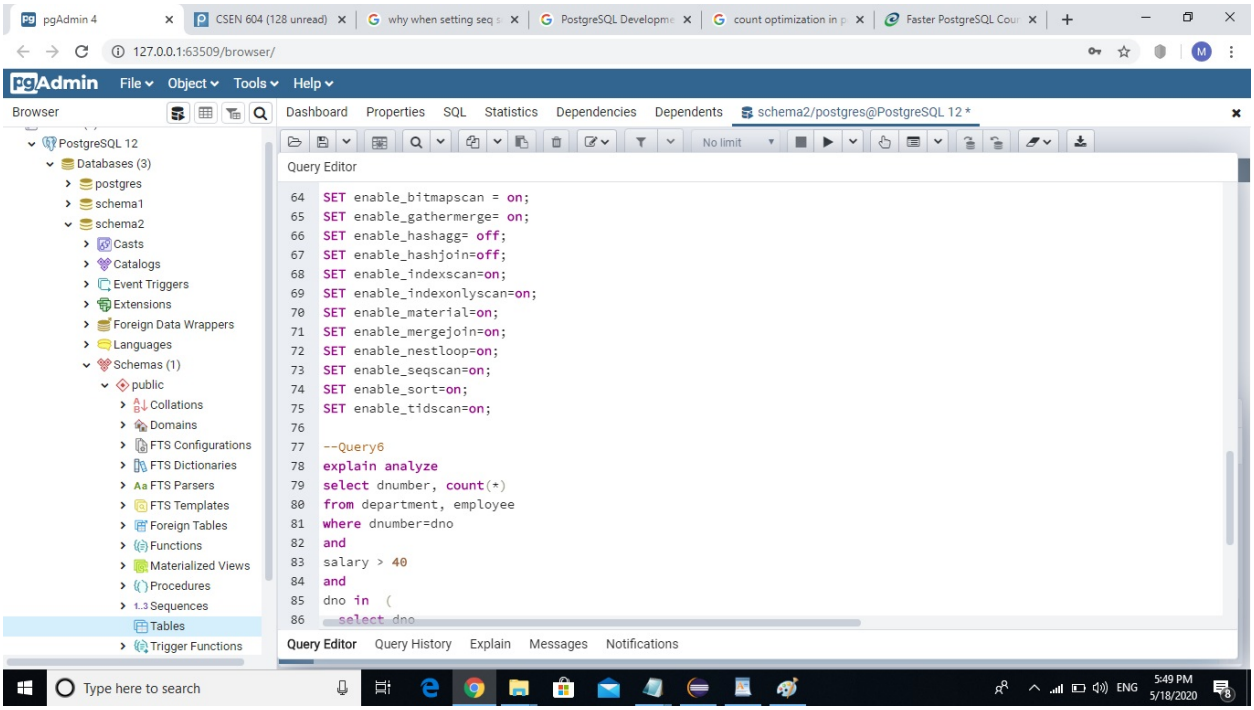
Regarding The B+Tree index , it is considered an average one in cost(931.99) and in execution time(27.362 ms) compared to others scenarios because it works on exact values terms queries as well.

And the without index scenario , it is costly(3808.61) and has an execution time of(125.629 ms) because it goes through dependent and employee tables sequentially without an index usage which makes the execution faster.

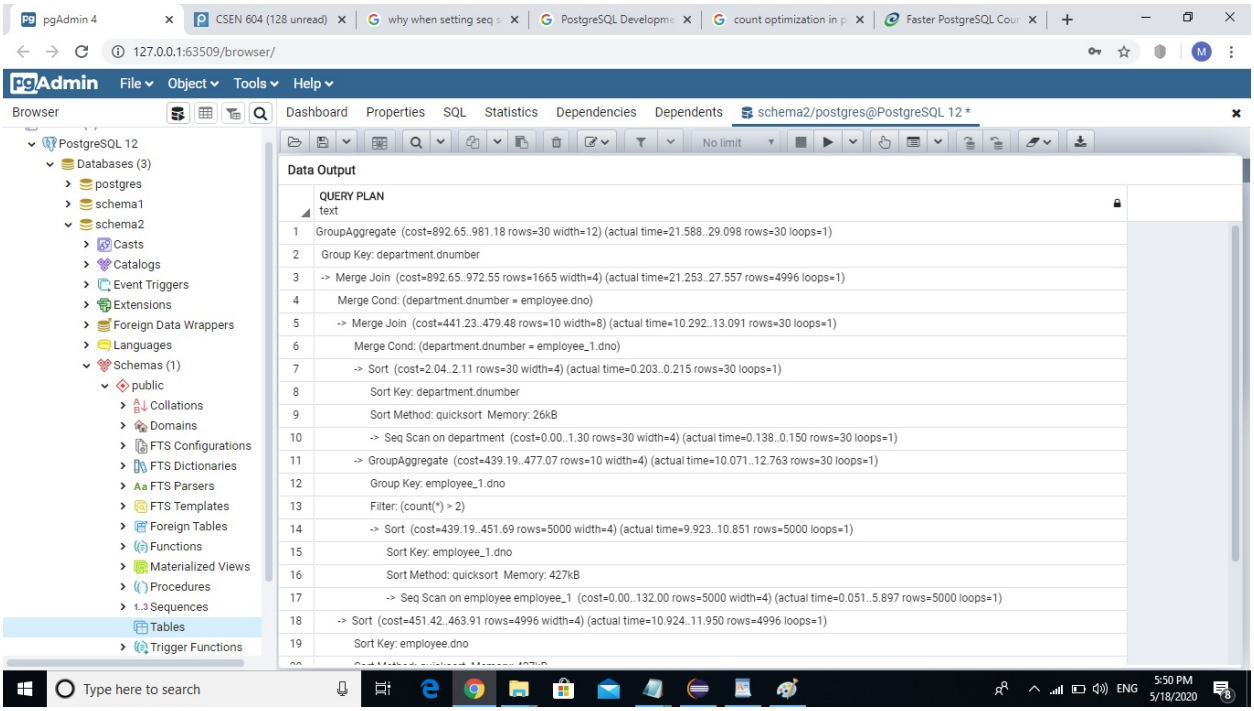
Query 6:

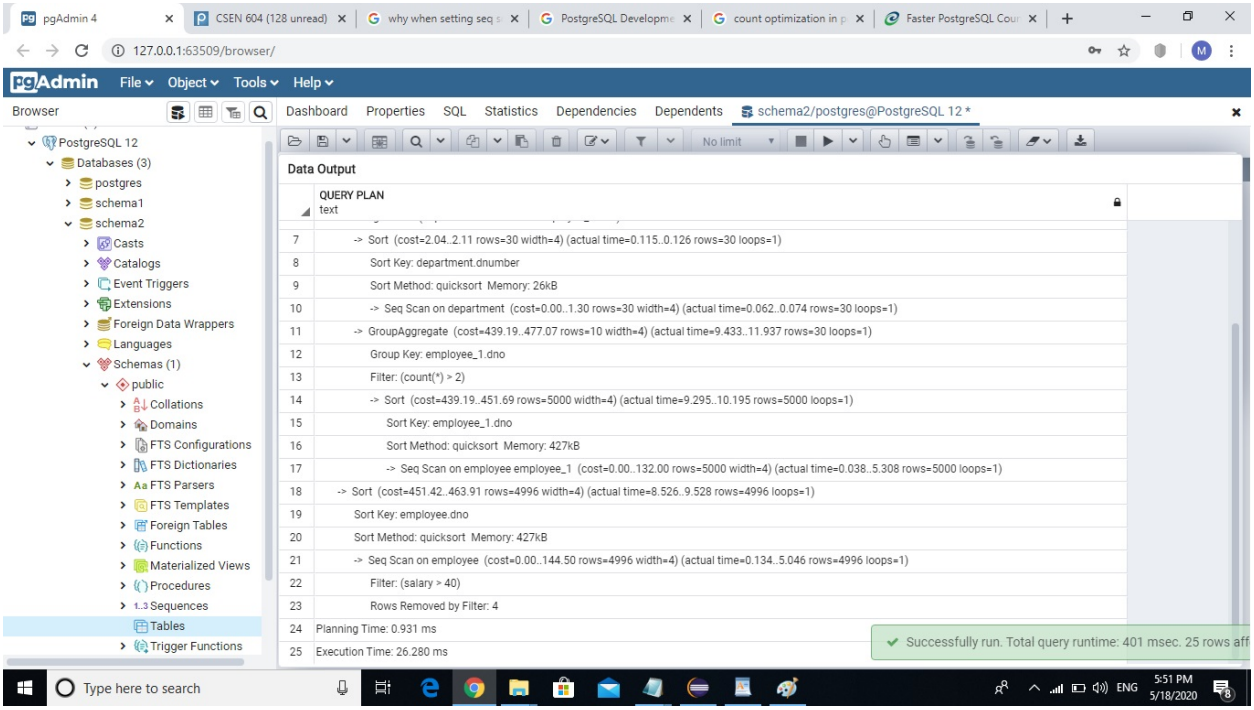
1-Without index :

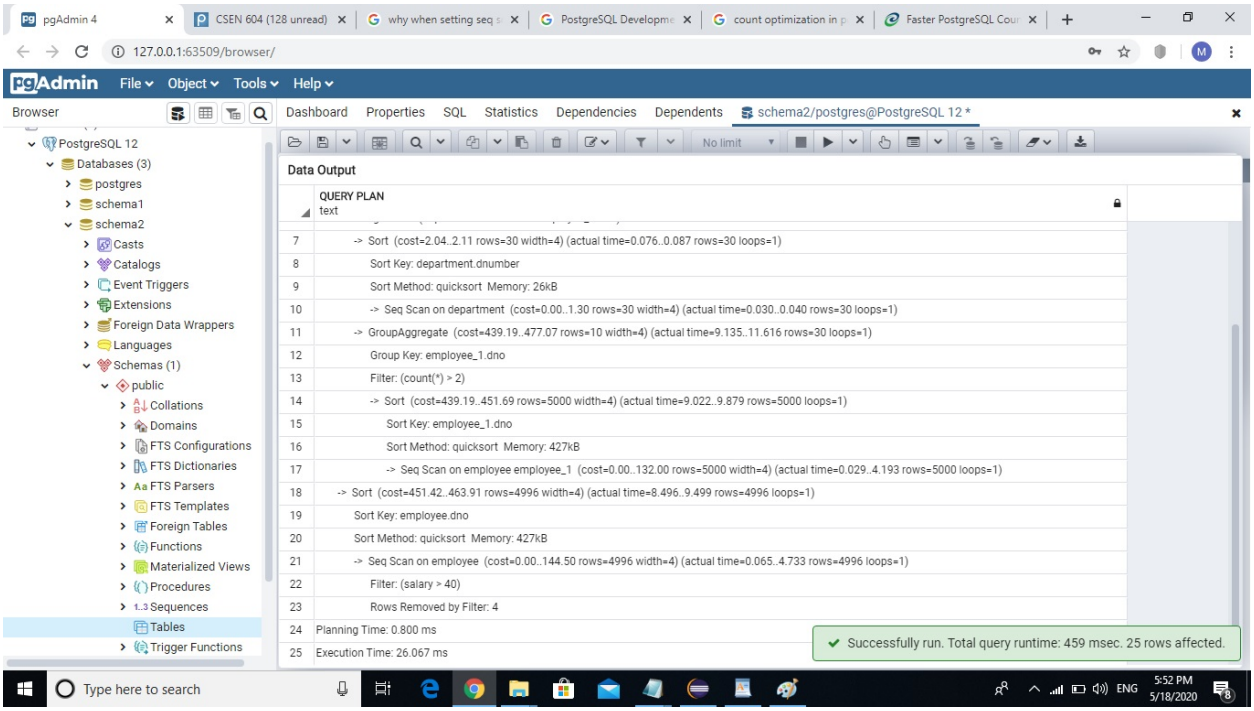
1-Configurations :

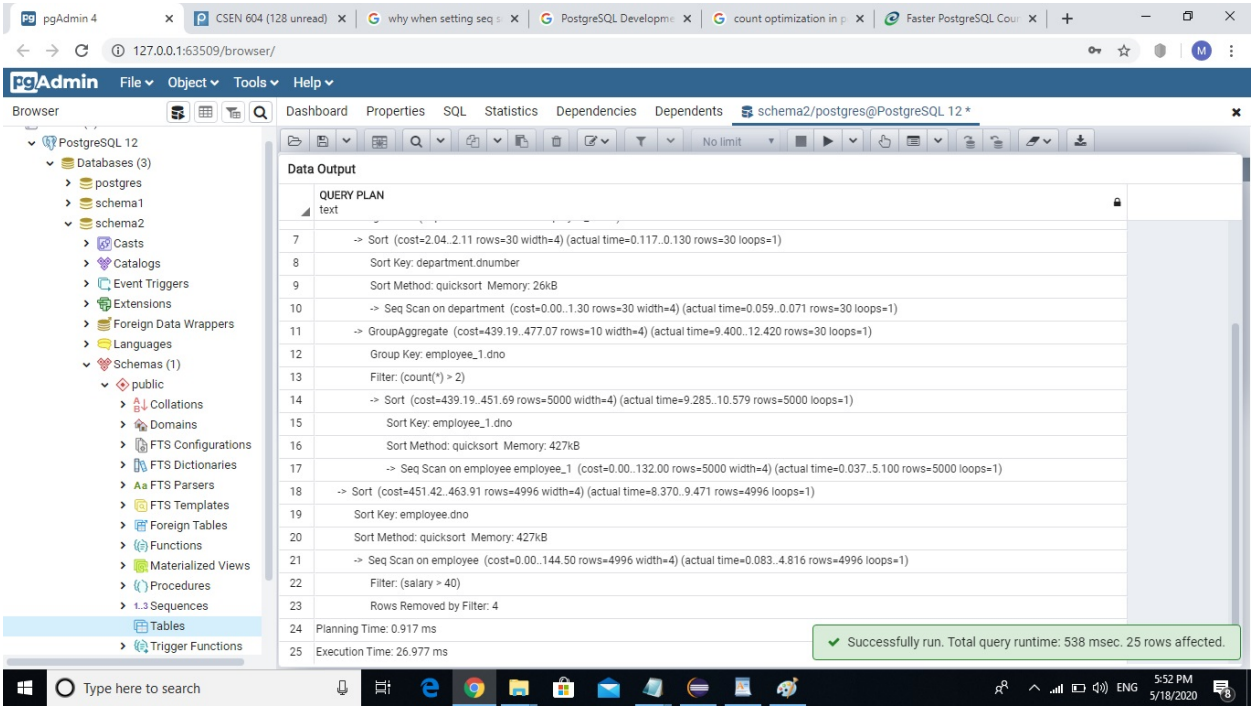


2-Query plan :


pgAdmin 4
CSEN 604 (128 unread)
why when setting seq
PostgreSQL Developme
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
1 GroupAggregate (cost=892.65..981.18 rows=30 width=12) (actual time=21.588..29.098 rows=30 loops=1)
2 Group Key: department.dnumber
3 -> Merge Join (cost=892.65..972.55 rows=1665 width=4) (actual time=21.253..27.557 rows=4996 loops=1)
4 Merge Cond: (department.dnumber = employee.dno)
5 -> Merge Join (cost=441.23..479.48 rows=10 width=8) (actual time=10.292..13.091 rows=30 loops=1)
6 Merge Cond: (department.dnumber = employee_1.dno)
7 -> Sort (cost=2.04..2.11 rows=30 width=4) (actual time=0.203..0.215 rows=30 loops=1)
8 Sort Key: department.dnumber
9 Sort Method: quicksort Memory: 26kB
10 -> Seq Scan on department (cost=0.00..1.30 rows=30 width=4) (actual time=0.138..0.150 rows=30 loops=1)
11 -> GroupAggregate (cost=439.19..477.07 rows=10 width=4) (actual time=10.071..12.763 rows=30 loops=1)
12 Group Key: employee_1.dno
13 Filter: (count(*) > 2)
14 -> Sort (cost=439.19..451.69 rows=5000 width=4) (actual time=9.923..10.851 rows=5000 loops=1)
15 Sort Key: employee_1.dno
16 Sort Method: quicksort Memory: 427kB
17 -> Seq Scan on employee employee_1 (cost=0.00..132.00 rows=5000 width=4) (actual time=0.051..5.897 rows=5000 loops=1)
18 -> Sort (cost=451.42..463.91 rows=4996 width=4) (actual time=10.924..11.950 rows=4996 loops=1)
19 Sort Key: employee.dno
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Trigger Functions
Type here to search
ENG
5:50 PM
5/18/2020


pgAdmin 4
CSEN 604 (128 unread)
why when setting seq
PostgreSQL Developme
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
7 -> Sort (cost=2.04..2.11 rows=30 width=4) (actual time=0.115..0.126 rows=30 loops=1)
8 Sort Key: department.dnumber
9 Sort Method: quicksort Memory: 26kB
10 -> Seq Scan on department (cost=0.00..1.30 rows=30 width=4) (actual time=0.062..0.074 rows=30 loops=1)
11 -> GroupAggregate (cost=439.19..477.07 rows=10 width=4) (actual time=9.433..11.937 rows=30 loops=1)
12 Group Key: employee_1.dno
13 Filter: (count(*) > 2)
14 -> Sort (cost=439.19..451.69 rows=5000 width=4) (actual time=9.295..10.195 rows=5000 loops=1)
15 Sort Key: employee_1.dno
16 Sort Method: quicksort Memory: 427kB
17 -> Seq Scan on employee employee_1 (cost=0.00..132.00 rows=5000 width=4) (actual time=0.038..5.308 rows=5000 loops=1)
18 -> Sort (cost=451.42..463.91 rows=4996 width=4) (actual time=8.526..9.528 rows=4996 loops=1)
19 Sort Key: employee.dno
20 Sort Method: quicksort Memory: 427kB
21 -> Seq Scan on employee (cost=0.00..144.50 rows=4996 width=4) (actual time=0.134..5.046 rows=4996 loops=1)
22 Filter: (salary > 40)
23 Rows Removed by Filter: 4
24 Planning Time: 0.931 ms
25 Execution Time: 26.280 ms
Successfully run. Total query runtime: 401 msec. 25 rows aff
PostgreSQL 12
Databases (3)
postgres
schema1
schema2
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
Trigger Functions
Type here to search
ENG
5:51 PM
5/18/2020



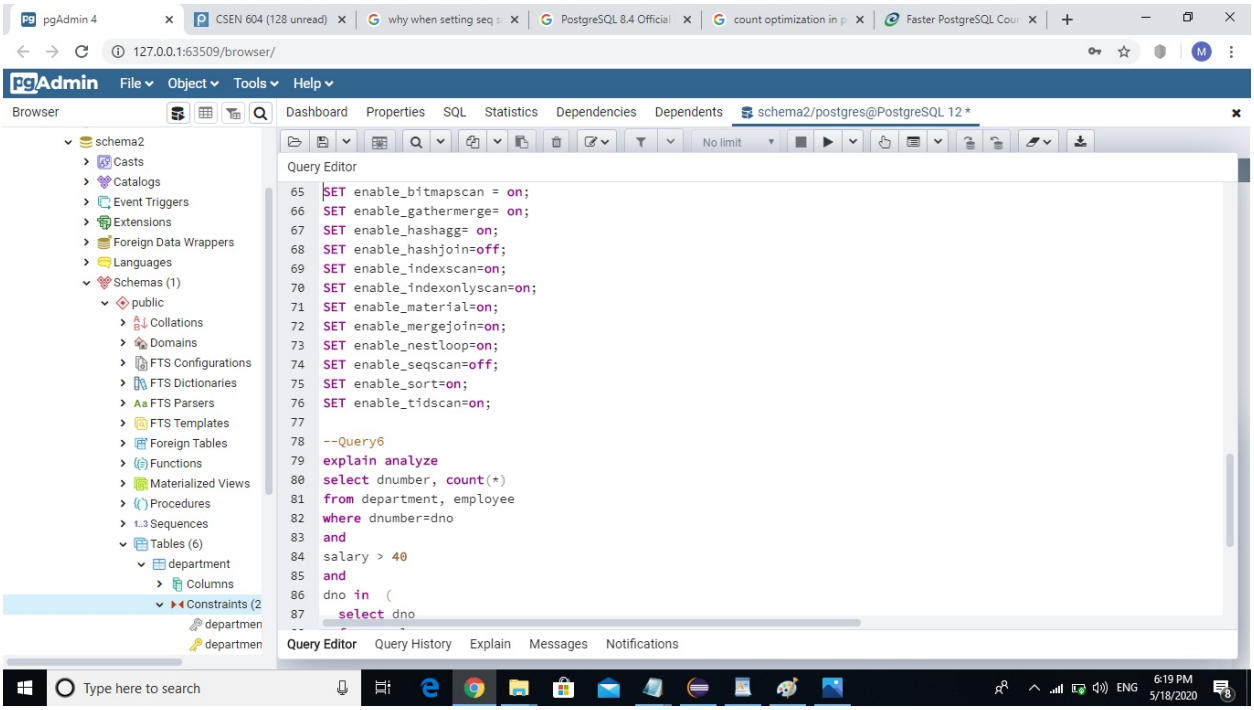


The Average execution time : (26.280 + 26.067 + 26.977) / 3 = 26.441 ms

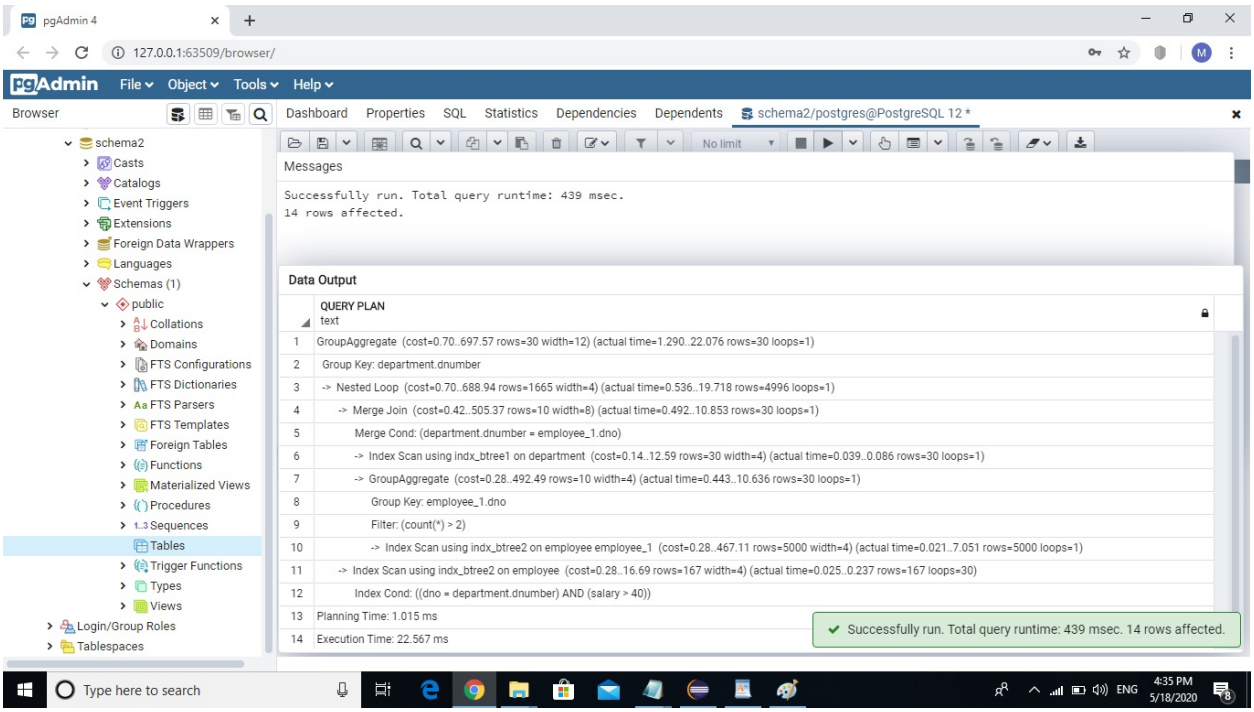
The Estimated cost of the plan : 981.18

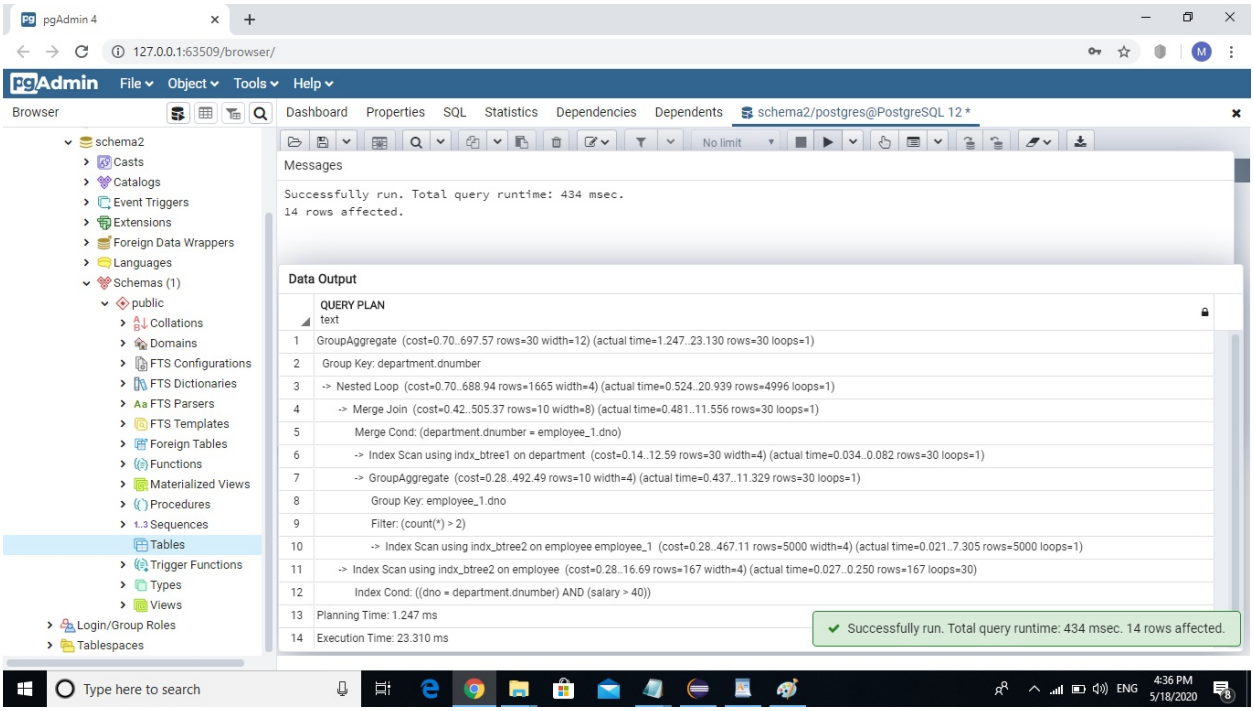
2-B+Tree index:

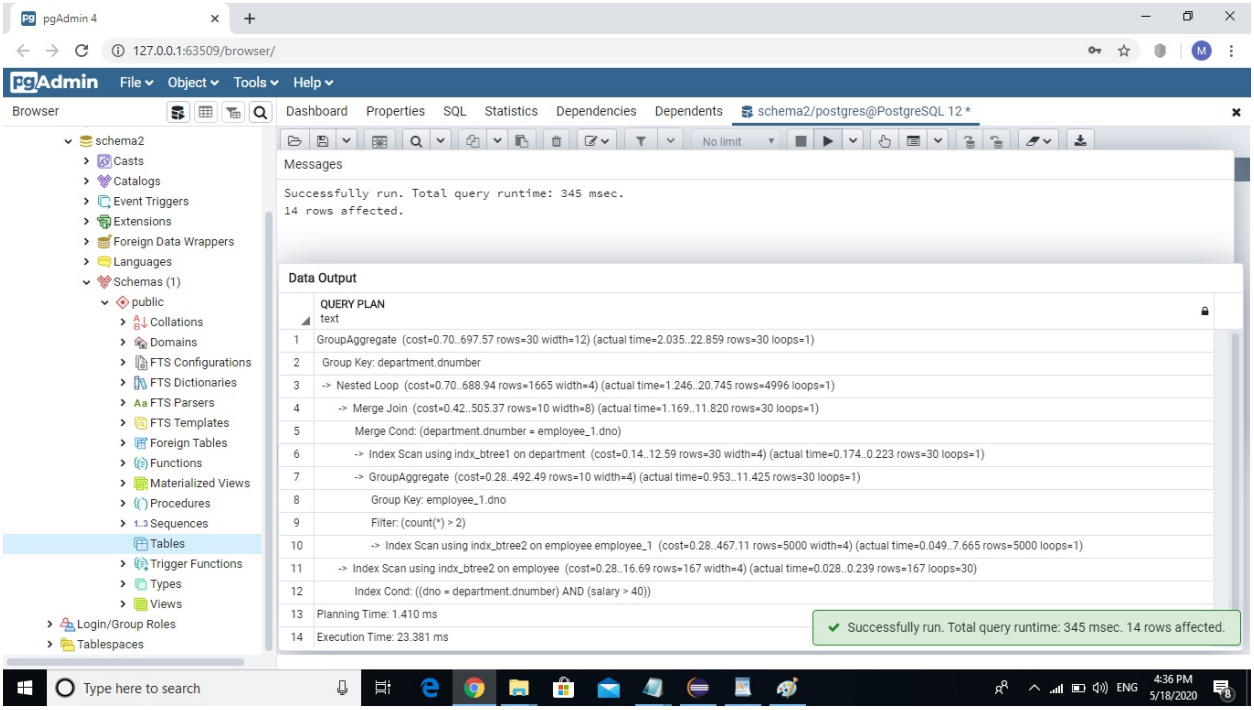
1-Configurations :



2-Query plan :





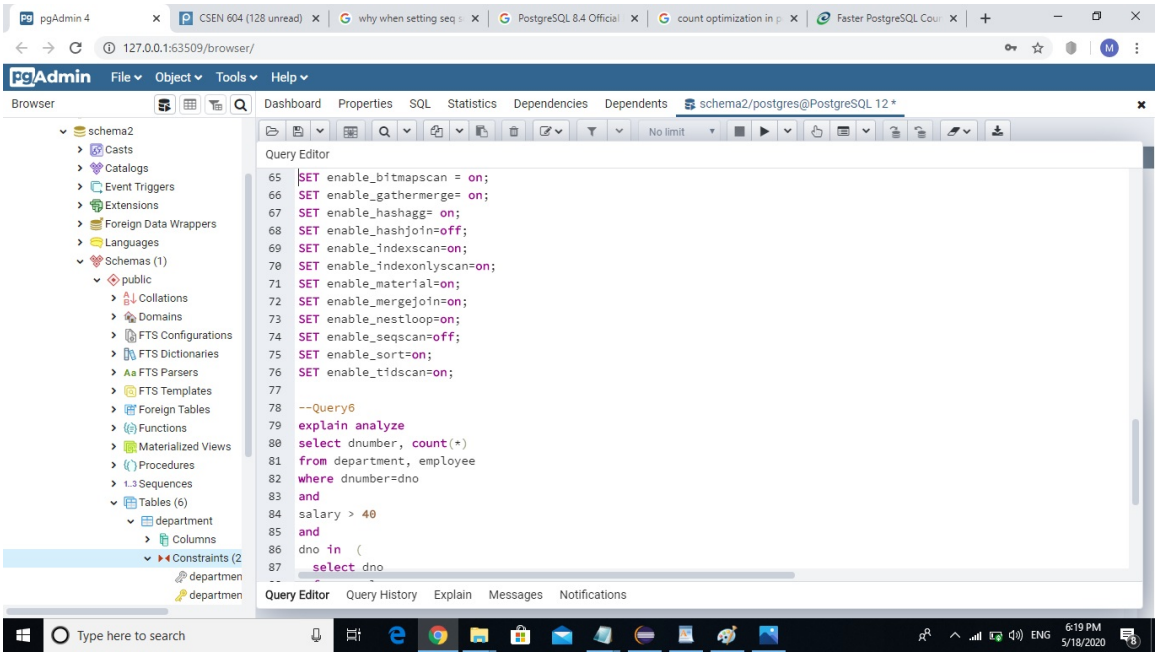


The Average execution time : (22.567 + 23.310 + 23.381) /3 = 23.086 ms

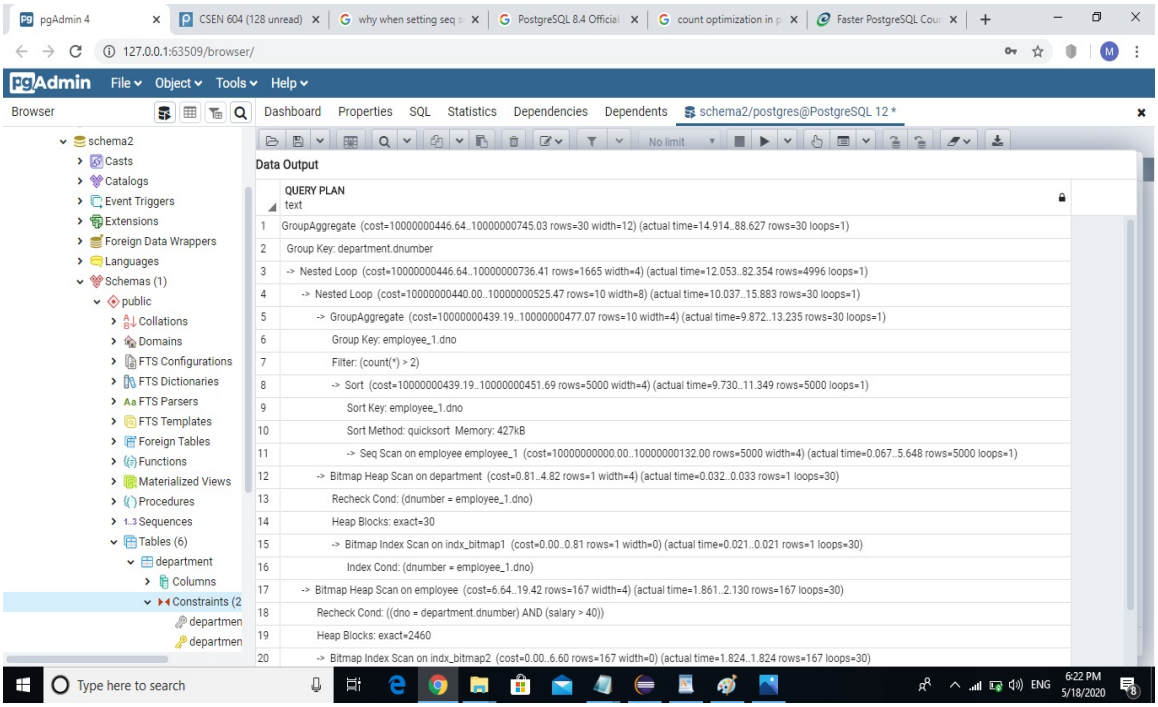
The Estimated cost of the plan : 697.57

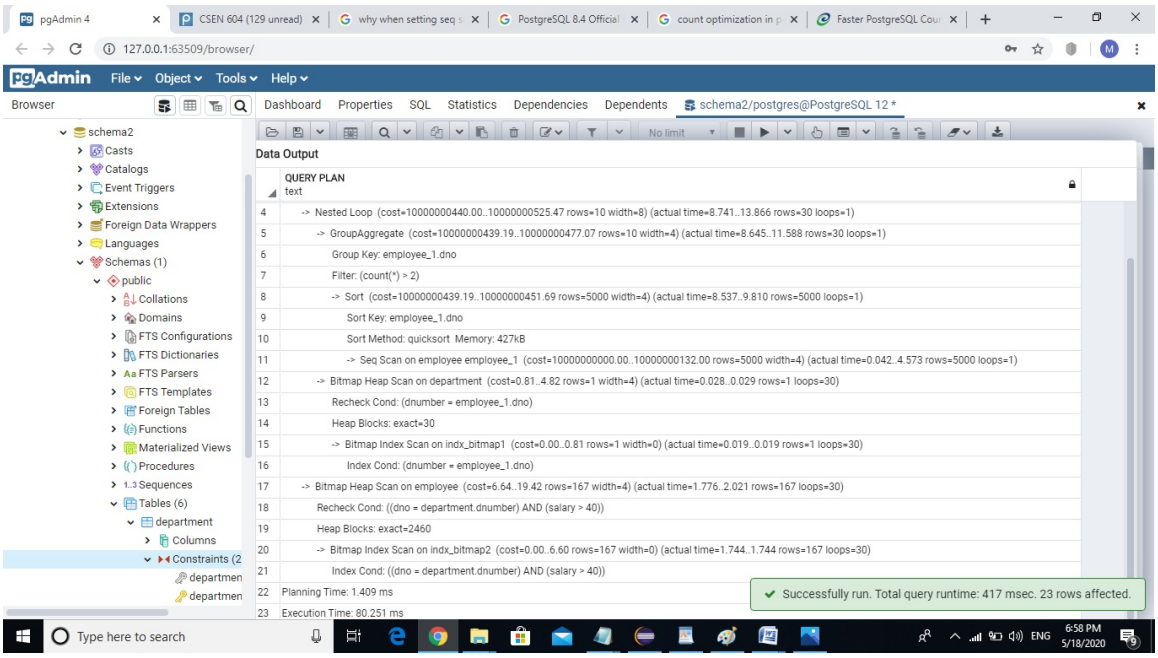
3-Bitmap index :

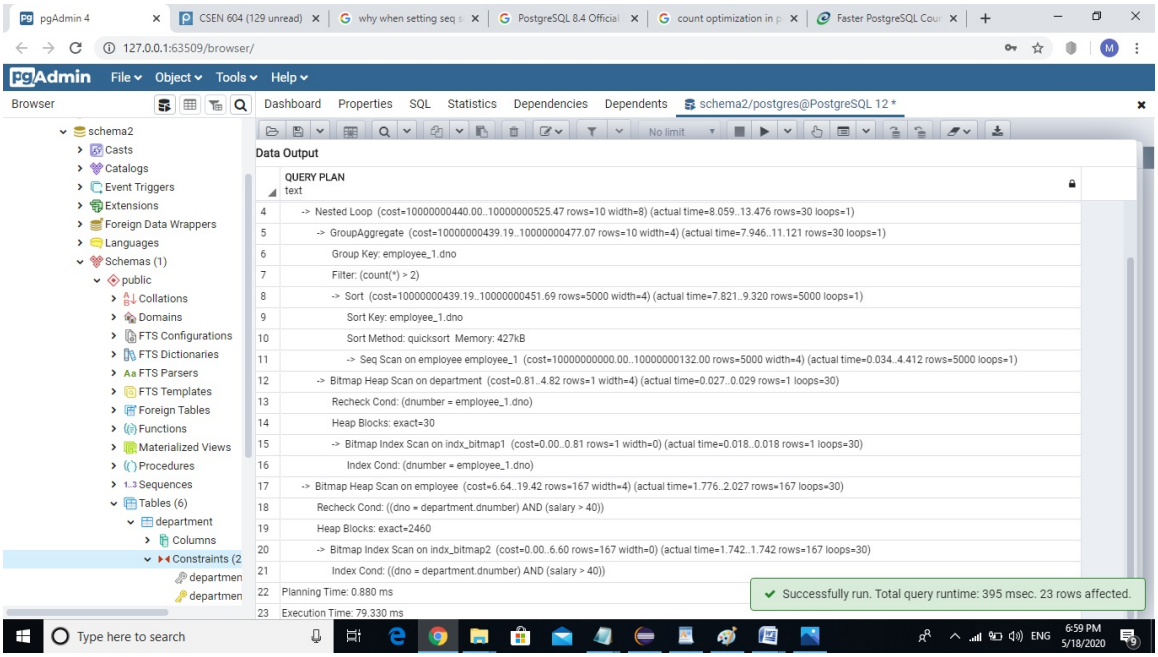
1. Configurations :

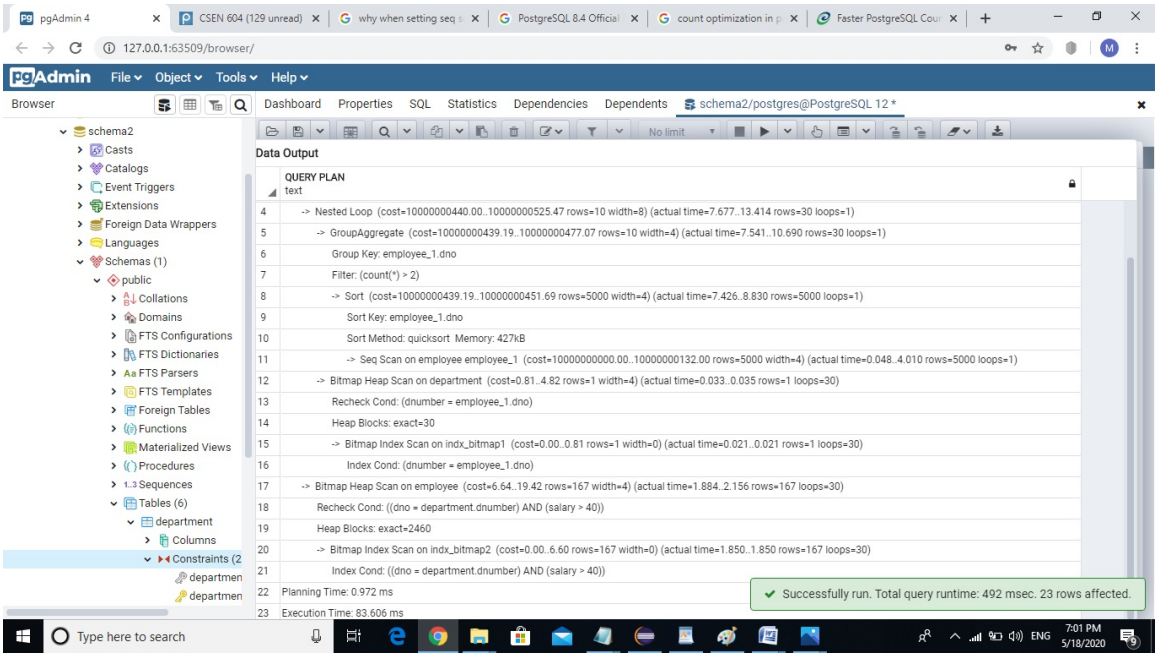


2. Query plan :




pgAdmin 4
CSEN 604 (129 unread)
why when setting seq
PostgreSQL 8.4 Official
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
4 -> Nested Loop (cost=10000000440.00..10000000525.47 rows=10 width=8) (actual time=8.741..13.866 rows=30 loops=1)
5 -> GroupAggregate (cost=10000000439.19..10000000477.07 rows=10 width=4) (actual time=8.645..11.588 rows=30 loops=1)
6 Group Key: employee_1.dno
7 Filter: (count(*) > 2)
8 -> Sort (cost=10000000439.19..10000000451.69 rows=5000 width=4) (actual time=8.537..9.810 rows=5000 loops=1)
9 Sort Key: employee_1.dno
10 Sort Method: quicksort Memory: 427kB
11 -> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.042..4.573 rows=5000 loops=1)
12 -> Bitmap Heap Scan on department (cost=0.81..4.82 rows=1 width=4) (actual time=0.028..0.029 rows=1 loops=30)
13 Recheck Cond: (dnumber = employee_1.dno)
14 Heap Blocks: exact=30
15 -> Bitmap Index Scan on indx_bitmap1 (cost=0.00..0.81 rows=1 width=0) (actual time=0.019..0.019 rows=1 loops=30)
16 Index Cond: (dnumber = employee_1.dno)
17 -> Bitmap Heap Scan on employee (cost=6.64..19.42 rows=167 width=4) (actual time=1.776..2.021 rows=167 loops=30)
18 Recheck Cond: ((dno = department.dnumber) AND (salary > 40))
19 Heap Blocks: exact=2460
20 -> Bitmap Index Scan on indx_bitmap2 (cost=0.00..6.60 rows=167 width=0) (actual time=1.744..1.744 rows=167 loops=30)
21 Index Cond: ((dno = department.dnumber) AND (salary > 40))
22 Planning Time: 1.409 ms
23 Execution Time: 80.251 ms
Successfully run. Total query runtime: 417 msec. 23 rows affected.
Type here to search
ENG
6:58 PM
5/18/2020


pgAdmin 4
CSEN 604 (129 unread)
why when setting seq
PostgreSQL 8.4 Official
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
Data Output
QUERY PLAN
text
4 -> Nested Loop (cost=10000000440.00..10000000525.47 rows=10 width=8) (actual time=8.059..13.476 rows=30 loops=1)
5 -> GroupAggregate (cost=10000000439.19..10000000477.07 rows=10 width=4) (actual time=7.946..11.121 rows=30 loops=1)
6 Group Key: employee_1.dno
7 Filter: (count(*) > 2)
8 -> Sort (cost=10000000439.19..10000000451.69 rows=5000 width=4) (actual time=7.821..9.320 rows=5000 loops=1)
9 Sort Key: employee_1.dno
10 Sort Method: quicksort Memory: 427kB
11 -> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.034..4.412 rows=5000 loops=1)
12 -> Bitmap Heap Scan on department (cost=0.81..4.82 rows=1 width=4) (actual time=0.027..0.029 rows=1 loops=30)
13 Recheck Cond: (dnumber = employee_1.dno)
14 Heap Blocks: exact=30
15 -> Bitmap Index Scan on indx_bitmap1 (cost=0.00..0.81 rows=1 width=0) (actual time=0.018..0.018 rows=1 loops=30)
16 Index Cond: (dnumber = employee_1.dno)
17 -> Bitmap Heap Scan on employee (cost=6.64..19.42 rows=167 width=4) (actual time=1.776..2.027 rows=167 loops=30)
18 Recheck Cond: ((dno = department.dnumber) AND (salary > 40))
19 Heap Blocks: exact=2460
20 -> Bitmap Index Scan on indx_bitmap2 (cost=0.00..6.60 rows=167 width=0) (actual time=1.742..1.742 rows=167 loops=30)
21 Index Cond: ((dno = department.dnumber) AND (salary > 40))
22 Planning Time: 0.880 ms
23 Execution Time: 79.330 ms
Successfully run. Total query runtime: 395 msec. 23 rows affected.
Type here to search
ENG
6:59 PM
5/18/2020

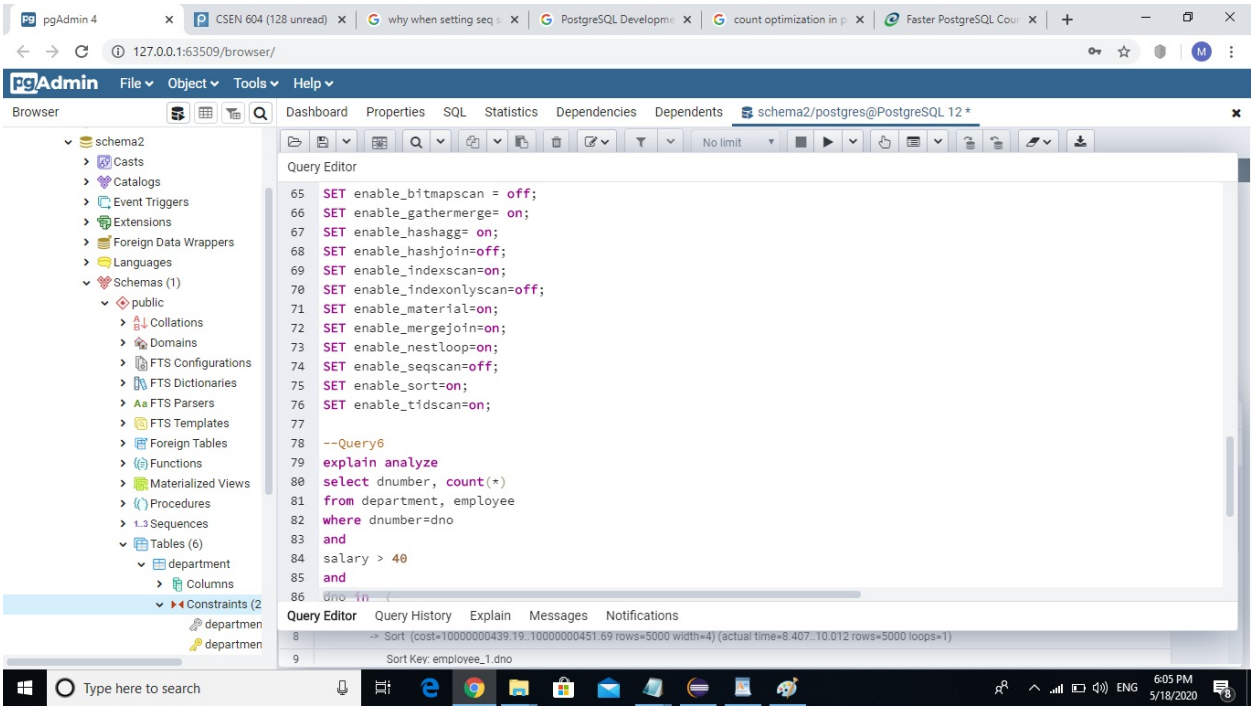


The average execution time: (80.251 + 79.330 + 83.606)/3 =164.645 ms

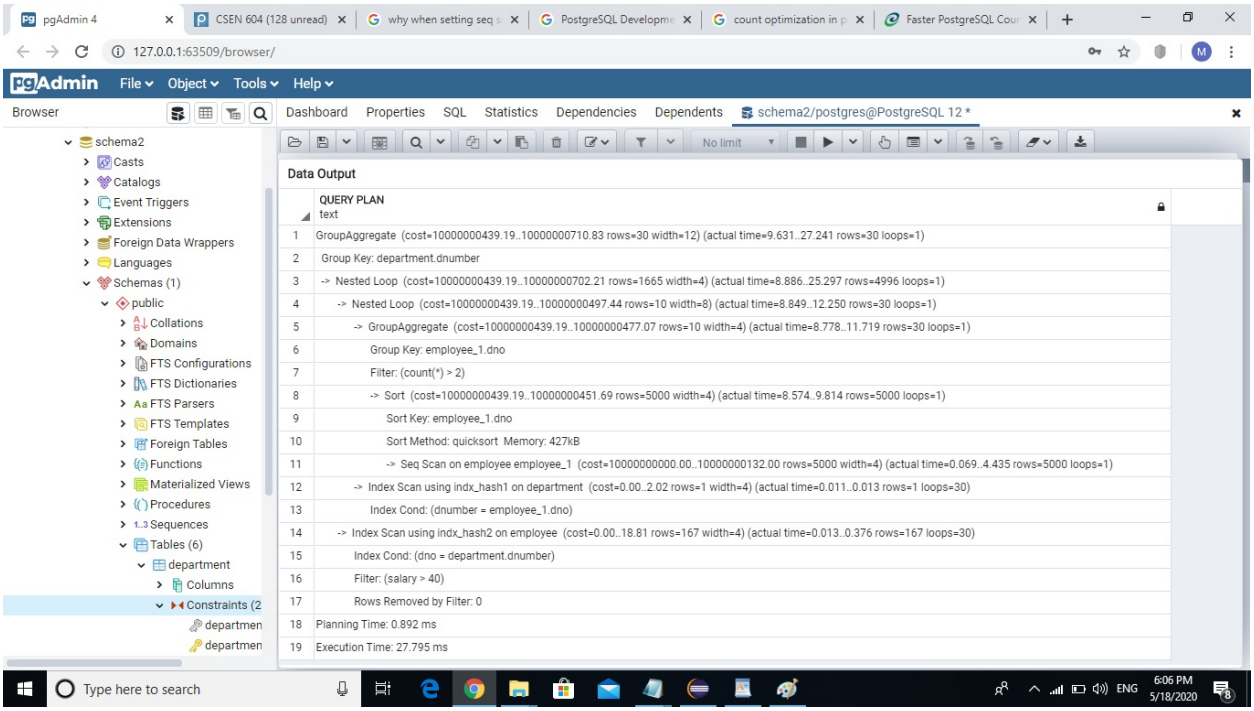
The estimated cost of the plan : 10000000745.03

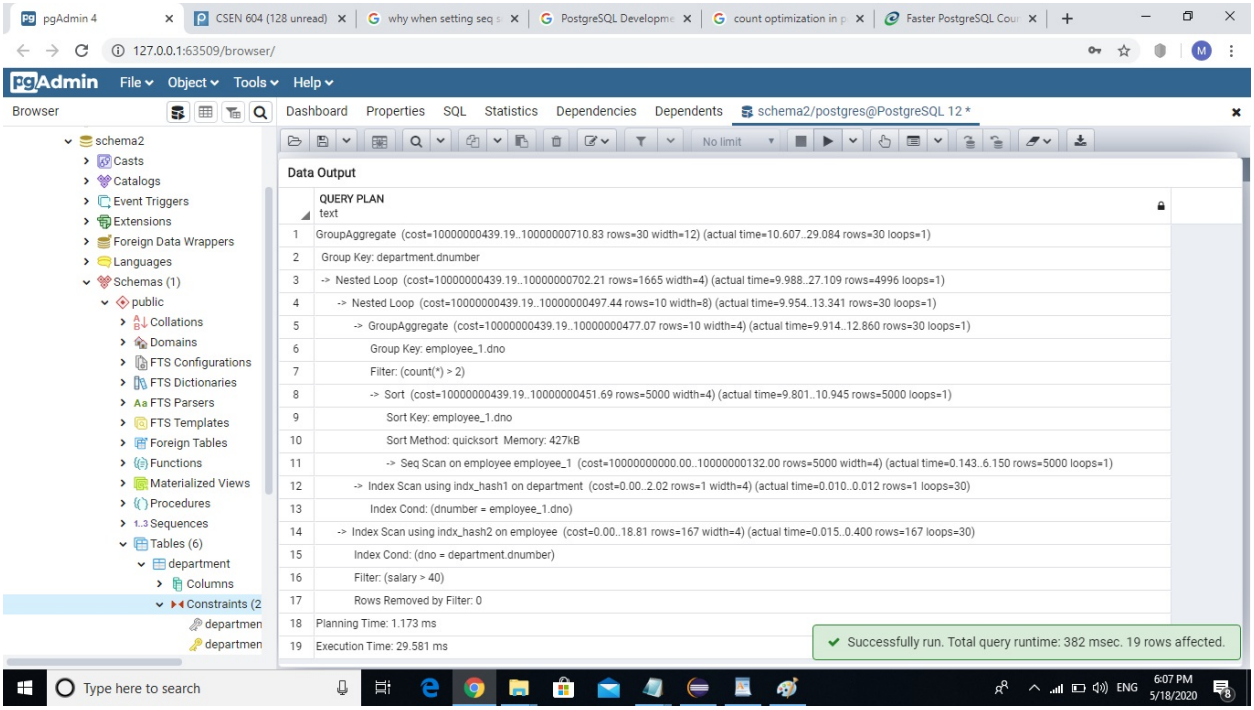
4-HashBased index :

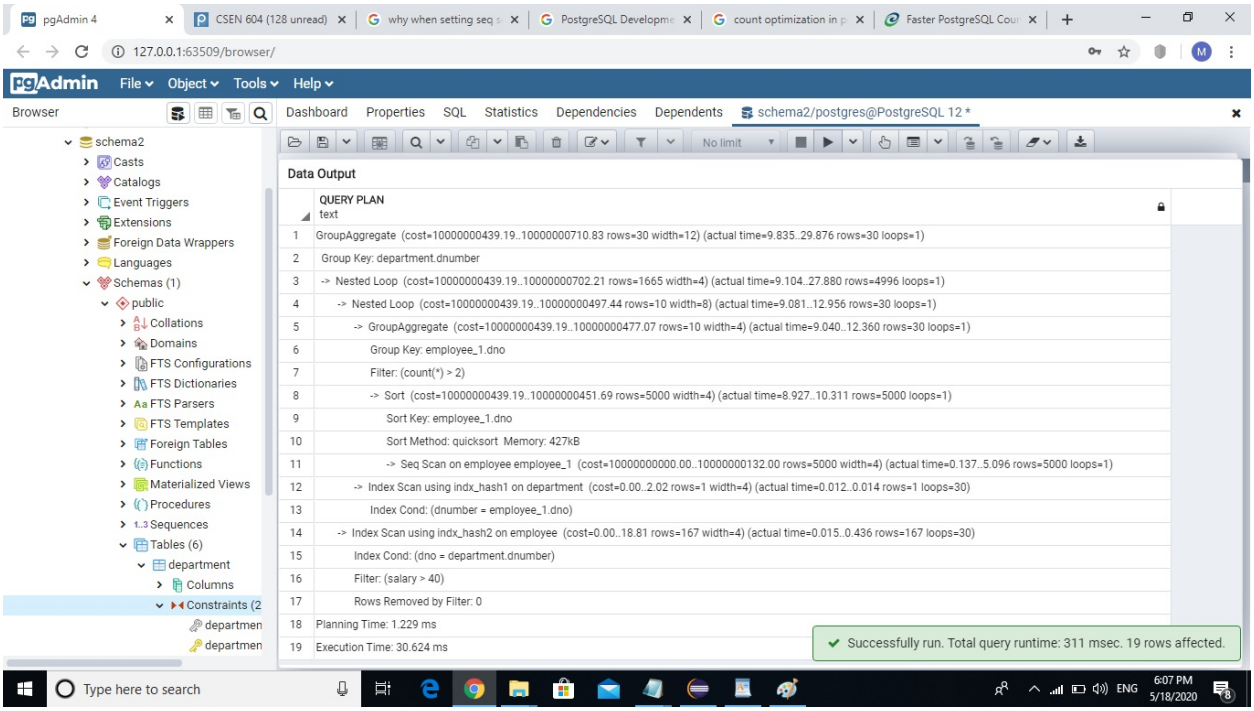
1-Configurations :



2-Query plan :


pgAdmin 4
CSEN 604 (128 unread)
why when setting seq
PostgreSQL Developme
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
schema2
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 (6)
department
Columns
Constraints (2
departmen
departmen
Data Output
QUERY PLAN
text
1 GroupAggregate (cost=10000000439.19..10000000710.83 rows=30 width=12) (actual time=9.631..27.241 rows=30 loops=1)
2 Group Key: department.dnumber
3 -> Nested Loop (cost=10000000439.19..10000000702.21 rows=1665 width=4) (actual time=8.886..25.297 rows=4996 loops=1)
4 -> Nested Loop (cost=10000000439.19..10000000497.44 rows=10 width=8) (actual time=8.849..12.250 rows=30 loops=1)
5 -> GroupAggregate (cost=10000000439.19..10000000477.07 rows=10 width=4) (actual time=8.778..11.719 rows=30 loops=1)
6 Group Key: employee_1.dno
7 Filter: (count(*) > 2)
8 -> Sort (cost=10000000439.19..10000000451.69 rows=5000 width=4) (actual time=8.574..9.814 rows=5000 loops=1)
9 Sort Key: employee_1.dno
10 Sort Method: quicksort Memory: 427kB
11 -> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.069..4.435 rows=5000 loops=1)
12 -> Index Scan using indx_hash1 on department (cost=0.00..2.02 rows=1 width=4) (actual time=0.011..0.013 rows=1 loops=30)
13 Index Cond: (dnumber = employee_1.dno)
14 -> Index Scan using indx_hash2 on employee (cost=0.00..18.81 rows=167 width=4) (actual time=0.013..0.376 rows=167 loops=30)
15 Index Cond: (dno = department.dnumber)
16 Filter: (salary > 40)
17 Rows Removed by Filter: 0
18 Planning Time: 0.892 ms
19 Execution Time: 27.795 ms
Type here to search
ENG
6:06 PM
5/18/2020


pgAdmin 4
CSEN 604 (128 unread)
why when setting seq
PostgreSQL Developme
count optimization in p
Faster PostgreSQL Cour
127.0.0.1:63509/browser/
pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
schema2/postgres@PostgreSQL 12 *
No limit
schema2
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 (6)
department
Columns
Constraints (2
departmen
departmen
Data Output
QUERY PLAN
text
1 GroupAggregate (cost=10000000439.19..10000000710.83 rows=30 width=12) (actual time=10.607..29.084 rows=30 loops=1)
2 Group Key: department.dnumber
3 -> Nested Loop (cost=10000000439.19..10000000702.21 rows=1665 width=4) (actual time=9.988..27.109 rows=4996 loops=1)
4 -> Nested Loop (cost=10000000439.19..10000000497.44 rows=10 width=8) (actual time=9.954..13.341 rows=30 loops=1)
5 -> GroupAggregate (cost=10000000439.19..10000000477.07 rows=10 width=4) (actual time=9.914..12.860 rows=30 loops=1)
6 Group Key: employee_1.dno
7 Filter: (count(*) > 2)
8 -> Sort (cost=10000000439.19..10000000451.69 rows=5000 width=4) (actual time=9.801..10.945 rows=5000 loops=1)
9 Sort Key: employee_1.dno
10 Sort Method: quicksort Memory: 427kB
11 -> Seq Scan on employee employee_1 (cost=10000000000.00..10000000132.00 rows=5000 width=4) (actual time=0.143..6.150 rows=5000 loops=1)
12 -> Index Scan using indx_hash1 on department (cost=0.00..2.02 rows=1 width=4) (actual time=0.010..0.012 rows=1 loops=30)
13 Index Cond: (dnumber = employee_1.dno)
14 -> Index Scan using indx_hash2 on employee (cost=0.00..18.81 rows=167 width=4) (actual time=0.015..0.400 rows=167 loops=30)
15 Index Cond: (dno = department.dnumber)
16 Filter: (salary > 40)
17 Rows Removed by Filter: 0
18 Planning Time: 1.173 ms
19 Execution Time: 29.581 ms
Successfully run. Total query runtime: 382 msec. 19 rows affected.
Type here to search
ENG
6:07 PM
5/18/2020



The average execution time : (27.795 + 29.581 + 30.624) /3 = 29.333 ms

The average cost of the plan : 10000000525.47

Explanation :

We restricted our index to beon table department (dnumber) and on employee (dno).

We did some changes to the SQL query as follows :

```
select dnumber, count(*)
from department, employee
where dnumber=dno
and
salary > 40
and
dno in  (
  select dno
  from employee
```

```
  group by dno

  having count (*) > 2)

group by dnumber;
```

The B+Tree index is considered the best performer compared to all others scenarios with an average execution time of 23.086 ms and a cost of 697.57 , it worked efficiently in our query because it has a range search (salary > 40) and in our insertion.

Regarding the Bitmap index and the HashBased index , they are the most costly and have the highest execution time , due to their usage of configurations that are set to off , we restricted that so they can use our index , and they are not considered an optimization scenario in our type of query here .

When looking at the without index scenario , we can see that the average execution time is 26.441 ms and the cost is 981.18 because it walked through dependent and employee tables sequentially.

SCHEMA 4

- **Description of data inserted:**

The Actor table was populated with 12000, Movie table with 1000 and Director with 2000 records. Any other table that has any of these attributes as foreign key also had to be changed e.g. the method populateMovieDirection loop was changed to loop till 2000 only with the dir_id foreign key starting from 1 till 2000 while mov_id foreign key starts from 1 till 1000 then repeats itself from the beginning.

QUERY 10

- **Description of any changes in sql code:**

In the last line of the query (where mov_title =‘movie1’), the movie title is changed from ‘movie1’ to ‘Movie1’ becoming (where mov_title =‘Movie1’).

- **Flags set:**

For Seq Scan:

```
set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
```

set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

For BTree/Hash:

set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

For Bitmap:

set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

- **Output of Explain Analyze:**

A. no indexes.

- Estimate Cost: 548.93
- *Average Execution* time: 4.291 ms

	QUERY PLAN text
1	Hash Semi Join (cost=284.30..548.93 rows=12 width=48) (actual time=2.303..4.216 rows=12 loops=1)
2	Hash Cond: (actor.act_id = movie_cast.act_id)
3	-> Seq Scan on actor (cost=0.00..233.00 rows=12000 width=48) (actual time=0.015..0.956 rows=12000 loops=1)
4	-> Hash (cost=284.15..284.15 rows=12 width=4) (actual time=2.280..2.280 rows=12 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 9kB
6	-> Hash Join (cost=32.51..284.15 rows=12 width=4) (actual time=0.278..2.268 rows=12 loops=1)
7	Hash Cond: (movie_cast.mov_id = movie.mov_id)
8	-> Seq Scan on movie_cast (cost=0.00..220.00 rows=12000 width=8) (actual time=0.005..0.978 rows=1200…
9	-> Hash (cost=32.50..32.50 rows=1 width=4) (actual time=0.265..0.265 rows=1 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 9kB
11	-> Seq Scan on movie (cost=0.00..32.50 rows=1 width=4) (actual time=0.013..0.258 rows=1 loops=1)
12	Filter: (mov_title = 'Movie1'::bpchar)
13	Rows Removed by Filter: 999
14	Planning time: 0.943 ms
15	Execution time: 4.291 ms

Seq scan finds relevant records by sequentially scanning the input record set.
It first scans the movie relation while filtering with the condition: mov_title = ‘Movie1’ removing 999 rows. Then, a hash table was generated for the rest of the rows as the database engine decided to build a hash index on the fly— Result set 2
At the same time, a seq scan was being performed on the relation movie_cast — Result set 1
The 2 sets are joined with a hash join under the condition: movie_cast.mov_id = movie.mov_id
Again, the database engine decided to build a hash index on the fly for the new result set. — Result set 3.
Finally, a seq scan is performed on the relation actor which is joined with Result set 3 with a semi-join under the condition actor.act_id = movie_cast.act_id.

B. Btree indexes:

	QUERY PLAN text
1	Nested Loop (cost=60.91..64.97 rows=12 width=48) (actual time=0.180..0.217 rows=12 loops=1)
2	-> HashAggregate (cost=60.62..60.74 rows=12 width=4) (actual time=0.160..0.162 rows=12 loops=1)
3	Group Key: movie_cast.act_id
4	-> Nested Loop (cost=0.56..60.59 rows=12 width=4) (actual time=0.135..0.149 rows=12 loops=1)
5	-> Index Scan using btree_movie_title on movie (cost=0.28..8.29 rows=1 width=4) (actual time=0....
6	Index Cond: (mov_title = 'Movie1'::bpchar)
7	-> Index Scan using btree_movie_id on movie_cast (cost=0.29..52.18 rows=12 width=8) (actual ti...
8	Index Cond: (mov_id = movie.mov_id)
9	-> Index Scan using btree_actor_id on actor (cost=0.29..0.35 rows=1 width=48) (actual time=0.004..0.0...
10	Index Cond: (act_id = movie_cast.act_id)
11	Planning time: 0.750 ms
12	Execution time: 0.286 ms

- Estimate Cost: 64.97
- *Average Execution* time: 0.133 ms

At first, a btree indexing is applied on the relation movie_cast under the condition: (mov_id = movie.mov_id) while, another btree indexing is applied on the relation movie under the condition: (mov_title = 'Movie1'::bpchar). Then, with both result sets we perform a nested loop, which merges the 2 record sets by looping through every record in the first set and trying to find a match in the second set. — Result set 1
HashAggregate then grouped the result set 1 records based on the grouping key: movie_cast.act_id, while another tree index was applied to relation actor under condition: act_id = movie_cast.act_id. The results of those 2 are then iterated on using a nested loop, merging both of them.

There is a big difference between execution time where there was no index (4.291 ms) and with the BTree index(0.133 ms) because B+ Tree index support queries on exact values , queries involving a range search and queries containing aggregate functions.
And we have in our query exact values terms such that (Mov_id = 1) that's why it will work efficiently and in less time when executing the query.

C. Hash indexes:

	QUERY PLAN text
1	Nested Loop (cost=60.38..61.42 rows=12 width=48) (actual time=0.035..0.057 rows=12 loops=1)
2	-> HashAggregate (cost=60.38..60.50 rows=12 width=4) (actual time=0.032..0.032 rows=12 loops=1)
3	Group Key: movie_cast.act_id
4	-> Nested Loop (cost=0.00..60.35 rows=12 width=4) (actual time=0.015..0.025 rows=12 loops=1)
5	-> Index Scan using hash_mov_title on movie (cost=0.00..8.02 rows=1 width=4) (actual time=0.011..0.011 rows=1 loops=1)
6	Index Cond: (mov_title = 'Movie1'::bpchar)
7	-> Index Scan using hash_mov_id on movie_cast (cost=0.00..52.21 rows=12 width=8) (actual time=0.003..0.013 rows=12 loops=1
8	Index Cond: (mov_id = movie.mov_id)
9	-> Index Scan using hash_act_id on actor (cost=0.00..0.08 rows=1 width=48) (actual time=0.002..0.002 rows=1 loops=12)
10	Index Cond: (act_id = movie_cast.act_id)
11	Planning time: 1.310 ms
12	Execution time: 0.136 ms

- Estimate Cost 61.42
- *Average Execution* time: 0.136 ms

At first, hash indexing is applied on the relation movie under condition: mov_title = 'Movie1' and another hash indexing is applied on the relation movie_cast under condition: mov_id = movie.mov_id. the results of those 2 are merged together when nested loop is performed then, HashAggregate groups the nested loop results over the grouping key: movie_cast.act_id. — result set 1.
On the other hand, another hash indexing is performed over relation actor under the condition: act_id = movie_cast.act_id — result set 2
Those 2 result sets (1 and 2) are merged together when the final nested loop is applied forming the final result set.
The huge difference between the execution time without index(4.291 ms) and the execution time with index (0.136 ms) because HashBased index is used to enhance the performance of queries on exact values only.
And that's what we have in our query exact values terms such that (mov_id = 1, act_id = 1 ..) that's why it will work efficiently and in less time when executing the query.

D. Bitmap indexes:

	QUERY PLAN text
1	Nested Loop (cost=10000000080.47..10000000129.25 rows=12 width=48) (actual time=0.469..0.714 rows=12 loops=1)
2	-> HashAggregate (cost=10000000080.43..10000000080.55 rows=12 width=4) (actual time=0.439..0.452 rows=12 loops=1)
3	Group Key: movie_cast.act_id
4	-> Nested Loop (cost=10000000044.60..10000000080.40 rows=12 width=4) (actual time=0.394..0.424 rows=12 loops=1)
5	-> HashAggregate (cost=10000000032.50..10000000032.51 rows=1 width=4) (actual time=0.308..0.309 rows=1 loops=1)
6	Group Key: movie.mov_id
7	-> Seq Scan on movie (cost=10000000000.00..10000000032.50 rows=1 width=4) (actual time=0.028..0.291 rows=1 loops=1)
8	Filter: (mov_title = 'Movie1'::bpchar)
9	Rows Removed by Filter: 999
10	-> Bitmap Heap Scan on movie_cast (cost=12.09..47.77 rows=12 width=8) (actual time=0.061..0.089 rows=12 loops=1)
11	Recheck Cond: (mov_id = movie.mov_id)
12	Heap Blocks: exact=12
13	-> Bitmap Index Scan on bitmap_mov_id (cost=0.00..12.09 rows=12 width=0) (actual time=0.043..0.043 rows=12 loops=1)
14	Index Cond: (mov_id = movie.mov_id)
15	-> Bitmap Heap Scan on actor (cost=0.04..4.05 rows=1 width=48) (actual time=0.015..0.015 rows=1 loops=12)
16	Recheck Cond: (act_id = movie_cast.act_id)
17	Heap Blocks: exact=12
18	-> Bitmap Index Scan on bitmap_act_id (cost=0.00..0.04 rows=1 width=0) (actual time=0.010..0.010 rows=1 loops=12)
19	Index Cond: (act_id = movie_cast.act_id)
20	Planning time: 0.400 ms
21	Execution time: 0.970 ms

- Estimate Cost: 10000000129.25
- *Average Execution* time: 1.970 ms

At first, a seq scan is performed on relation movie (even though bitmap should have been applied here, it was inefficient that the database engine decided to use seq scan instead) with filtering: mov_title = 'Movie1' , removing 999 rows then, grouped by mov_id when HashAggregate is applied to it while, a bitmap heap is performed on movie_cast, applying a bitmap indexing on mov_id with condition: mov_id = movie.mov_id. The results of those 2 are merged together using nested loop which merges the 2 record sets by looping through every record in the first set and trying to find a match in the second set then, the result of the loop is then grouped by movie_cast.act_id when HashAggregate is applied. Meanwhile, another bitmap heap is performed on relation actor applying bitmap indexing on act_id where the condition is act_id = movie_cast.act_id. The results of the HashAggregate and the last bitmap heap are also merged together using nested loop producing the final result set.
The difference between the execution time without index(4.291 ms) and the execution time with index (1.970 ms) because BitMap index answer many query types including on specific value, And, OR, XOR, and range .And we have in our query specific values terms such that (semester =1 , year = 2019…), we also have a 'in' operation that's why it will work efficiently and in less time when executing the query, even though the difference is not that huge

- **Best Query Performance:**

using The HashBased index, was considered the best performer of average execution time of 0.136 ms and a cost of 61.42 because HashBased index work efficiently on exact values queries which is our case here in our query and also due to our insertion.

QUERY 11

- **Description of any changes in sql code:**

In the last line of the query (where mov_title='movie2'), the movie title is changed from ‘movie2’ to ‘Movie2’ becoming (where mov_title =‘Movie2’).

- **Flags set:**

For Seq Scan:

```
set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For BTree/Hash:

```
set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For Bitmap:

```
set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

- **Output of Explain Analyze:**

A. no indexes.

	QUERY PLAN text
1	Nested Loop (cost=6140.79..9152.03 rows=48 width=42) (actual time=50.011..50.419 rows=6 loops=1)
2	Join Filter: (director.dir_id = movie_direction.dir_id)
3	Rows Removed by Join Filter: 3006
4	-> Unique (cost=6140.79..6141.03 rows=48 width=4) (actual time=50.002..50.005 rows=6 loops=1)
5	-> Sort (cost=6140.79..6140.91 rows=48 width=4) (actual time=50.001..50.002 rows=6 loops=1)
6	Sort Key: movie_direction.dir_id
7	Sort Method: quicksort Memory: 25kB
8	-> Nested Loop (cost=4843.33..6139.45 rows=48 width=4) (actual time=48.884..49.978 rows=6 loops=1)
9	Join Filter: (movie_direction.mov_id = movie_cast.mov_id)
10	Rows Removed by Join Filter: 5994
11	-> Unique (cost=4843.33..4843.45 rows=24 width=4) (actual time=48.871..48.875 rows=3 loops=1)
12	-> Sort (cost=4843.33..4843.39 rows=24 width=4) (actual time=48.871..48.871 rows=16 loops=1)
13	Sort Key: movie_cast.mov_id
14	Sort Method: quicksort Memory: 25kB
15	-> Nested Loop (cost=402.72..4842.78 rows=24 width=4) (actual time=4.961..48.803 rows=16 loops=1)
16	Join Filter: (movie_cast.role = movie_cast_1.role)
17	Rows Removed by Join Filter: 143984
18	-> Unique (cost=402.72..402.78 rows=12 width=31) (actual time=2.594..2.622 rows=12 loops=1)
19	-> Sort (cost=402.72..402.75 rows=12 width=31) (actual time=2.591..2.600 rows=12 loops=1)
20	Sort Key: movie_cast_1.role
21	Sort Method: quicksort Memory: 25kB
22	-> Nested Loop (cost=0.00..402.50 rows=12 width=31) (actual time=0.022..2.535 rows=12 loops=1)
23	Join Filter: (movie_cast_1.mov_id = movie.mov_id)
24	Rows Removed by Join Filter: 11988
25	-> Seq Scan on movie (cost=0.00..32.50 rows=1 width=4) (actual time=0.013..0.279 rows=1 loops=1)
26	Filter: (mov_title = 'Movie2'::bpchar)
27	Rows Removed by Filter: 999
28	-> Seq Scan on movie_cast movie_cast_1 (cost=0.00..220.00 rows=12000 width=35) (actual time=0.007..1.052 rows=12000 loops=1)
29	-> Seq Scan on movie_cast (cost=0.00..220.00 rows=12000 width=35) (actual time=0.009..1.771 rows=12000 loops=12)
30	-> Seq Scan on movie_direction (cost=0.00..29.00 rows=2000 width=8) (actual time=0.005..0.141 rows=2000 loops=3)
31	-> Seq Scan on director (cost=0.00..39.00 rows=2000 width=46) (actual time=0.002..0.037 rows=502 loops=6)
32	Planning time: 0.845 ms
33	Execution time: 50.479 ms

- *Estimate Cost: 9152.03*
- *Average Execution time: 50.479 ms*

At first, a sequential scan is perform on the relation movie jilting: mov_title = ‘Movie2’, removing 999 rows from the initial table; while another sequential scan was performed on movie_cast _1 (the second movie_cast query). Those 2 sets are joined with a nested loop filtering only: movie_cast_1.mov_id = movie.mov_id then, the result of the nested loop set is then sorted on sorting key: movie_cast_1.role and then the Unique constraint is applied to it removing duplicates; The result set of the unique constraint is merged using nested loop to a sequential scan performed on movie_cast (the first movie_cast query) filtering only: movie_cast.role = movie_cast_1.role. This filter resulted in removing 143984 rows. The filtered set is then sorted on sorting key: movie_cast.mov_id and the duplicates again are removed then, its result set is

merged using a nested loop with a sequential scan on relation movie_direction with join filter: movie_direction.mov_id = movie_cast.mov_id. The result of the iteration is sorted on sorting key: movie_direction.dir_id and the duplicates are removed. The result of the last unique constraint is then merged to a seq scan on relation director using nested loop, filtering: director.dir_id = movie_direction.dir_id, producing the final set of director names.

B. Btree indexes:

	QUERY PLAN text
1	Nested Loop (cost=77.36..94.01 rows=48 width=42) (actual time=0.243..0.253 rows=6 loops=1)
2	-> Unique (cost=77.08..77.32 rows=48 width=4) (actual time=0.239..0.243 rows=6 loops=1)
3	-> Sort (cost=77.08..77.20 rows=48 width=4) (actual time=0.238..0.241 rows=6 loops=1)
4	Sort Key: movie_direction.dir_id
5	Sort Method: quicksort Memory: 25kB
6	-> Nested Loop (cost=66.29..75.74 rows=48 width=4) (actual time=0.216..0.223 rows=6 loops=1)
7	-> Unique (cost=66.01..66.13 rows=24 width=4) (actual time=0.211..0.213 rows=3 loops=1)
8	-> Sort (cost=66.01..66.07 rows=24 width=4) (actual time=0.211..0.212 rows=16 loops=1)
9	Sort Key: movie_cast.mov_id
10	Sort Method: quicksort Memory: 25kB
11	-> Nested Loop (cost=61.09..65.46 rows=24 width=4) (actual time=0.144..0.187 rows=16 loops=1)
12	-> Unique (cost=60.81..60.87 rows=12 width=31) (actual time=0.135..0.139 rows=12 loops=1)
13	-> Sort (cost=60.81..60.84 rows=12 width=31) (actual time=0.135..0.136 rows=12 loops=1)
14	Sort Key: movie_cast_1.role
15	Sort Method: quicksort Memory: 25kB
16	-> Nested Loop (cost=0.56..60.59 rows=12 width=31) (actual time=0.093..0.105 rows=12 loops=1)
17	-> Index Scan using btree_mov_title on movie (cost=0.28..8.29 rows=1 width=4) (actual time=0.020..0.020 rows=1 loops=1)
18	Index Cond: (mov_title = 'Movie2'::bpchar)
19	-> Index Scan using btree_mov_id2 on movie_cast movie_cast_1 (cost=0.29..52.18 rows=12 width=35) (actual time=0.071..0.082 rows=12 loops=1)
20	Index Cond: (mov_id = movie.mov_id)
21	-> Index Scan using btree_role on movie_cast (cost=0.29..0.37 rows=1 width=35) (actual time=0.003..0.004 rows=1 loops=12)
22	Index Cond: (role = movie_cast_1.role)
23	-> Index Scan using btree_mov_id on movie_direction (cost=0.28..0.38 rows=2 width=8) (actual time=0.002..0.002 rows=2 loops=3)
24	Index Cond: (mov_id = movie_cast.mov_id)
25	-> Index Scan using btree_dir_id on director (cost=0.28..0.35 rows=1 width=46) (actual time=0.002..0.002 rows=1 loops=6)
26	Index Cond: (dir_id = movie_direction.dir_id)
27	Planning time: 2.516 ms
28	Execution time: 0.288 ms

- *Estimate Cost: 94.01*
- *Average Execution time: 0.288 ms*

At first, a btree index is applied to the relation movie where mov_title =‘Movie2’ and another is applied to the relation movie_cast_1 (the second movie_cast query) where mov_id = movie.mov_id. They are both merged together using nested loop then sorted and duplicates are removed with sort key: movie_cast_1.role. The result set produced after the unique constraint is merged using another nested loop with a btree scan on relation movie_cast (the first movie_cast query) where role = movie_cast_1.role. The result set of the iteration is then sorted on movie_cast.mov_id and the duplicates are removed; then the new result set is merged by another nested loop with a btree scan on movie _direction with mov_id = movie_cast.mov_id. The set produced from the iteration is sorted on movie_direction.dir_id and the duplicates are removed; then this new set is merged using a final nested loop with a btree

scan on relation director with dir_id = movie_direction.dir_id producing the final result set of the directors fname and lname.
There is a huge difference between execution time where there was no index (*50.479* ms) and with the BTree index(*0.288* ms) because B+ Tree index support queries on exact values , queries involving a range search.
And we have in our query exact values terms such that (Mov_id = 1) that's why it will work efficiently and in less time when executing the query.

C. Hash indexes:

	QUERY PLAN text
1	Nested Loop (cost=66.90..70.70 rows=48 width=42) (actual time=0.185..0.193 rows=6 loops=1)
2	-> Unique (cost=66.90..67.14 rows=48 width=4) (actual time=0.171..0.171 rows=6 loops=1)
3	-> Sort (cost=66.90..67.02 rows=48 width=4) (actual time=0.171..0.171 rows=6 loops=1)
4	Sort Key: movie_direction.dir_id
5	Sort Method: quicksort Memory: 25kB
6	-> Nested Loop (cost=62.30..65.56 rows=48 width=4) (actual time=0.151..0.159 rows=6 loops=1)
7	-> Unique (cost=62.30..62.42 rows=24 width=4) (actual time=0.143..0.146 rows=3 loops=1)
8	-> Sort (cost=62.30..62.36 rows=24 width=4) (actual time=0.143..0.144 rows=16 loops=1)
9	Sort Key: movie_cast.mov_id
10	Sort Method: quicksort Memory: 25kB
11	-> Nested Loop (cost=60.56..61.75 rows=24 width=4) (actual time=0.096..0.120 rows=16 loops=1)
12	-> Unique (cost=60.56..60.62 rows=12 width=31) (actual time=0.082..0.083 rows=12 loops=1)
13	-> Sort (cost=60.56..60.59 rows=12 width=31) (actual time=0.082..0.082 rows=12 loops=1)
14	Sort Key: movie_cast_1.role
15	Sort Method: quicksort Memory: 25kB
16	-> Nested Loop (cost=0.00..60.35 rows=12 width=31) (actual time=0.040..0.051 rows=12 loops=1)
17	-> Index Scan using hash_mov_title on movie (cost=0.00..8.02 rows=1 width=4) (actual time=0.031..0.033 rows=1 loops=1)
18	Index Cond: (mov_title = 'Movie2'::bpchar)
19	-> Index Scan using hash_mov_id2 on movie_cast movie_cast_1 (cost=0.00..52.21 rows=12 width=35) (actual time=0.008..0.014 rows=12 loops=1)
20	Index Cond: (mov_id = movie.mov_id)
21	-> Index Scan using hash_role on movie_cast (cost=0.00..0.08 rows=1 width=35) (actual time=0.002..0.002 rows=1 loops=12)
22	Index Cond: (role = movie_cast_1.role)
23	-> Index Scan using hash_mov_id on movie_direction (cost=0.00..0.11 rows=2 width=8) (actual time=0.003..0.004 rows=2 loops=3)
24	Index Cond: (mov_id = movie_cast.mov_id)
25	-> Index Scan using hash_dir_id on director (cost=0.00..0.08 rows=1 width=46) (actual time=0.003..0.003 rows=1 loops=6)
26	Index Cond: (dir_id = movie_direction.dir_id)
27	Planning time: 0.801 ms
28	Execution time: 0.245 ms

- *Estimate Cost: 70.70*
- *Average Execution time: 0.245 ms*

At first, an index scan is applied on relation movie using hash indexing with mov_title = 'Movie2' and another one is applied to relation movie_cast_1 where mov_id of this relation = movie.mov_id; they are both merged using a nested loop then sorted on movie_cast_1.role then the duplicates were removed then, merged using another nested loop with another hash indexing on relation movie_cast where the role = movie_cast_1.role. The set produced after the iteration is sorted on movie_cast.mov_id with no duplicates then merged with a hash index on movie_direction relation where mov_id = movie_cast.mov_id, using nested loop that is then sorted on movie_direction.dir_id then the duplicates are removed and merged with the last hash indexing over relation director where dir_id = movie_direction.dir_id.

The huge difference between the execution time without index(*50.479* ms) and the execution time with index (*0.245* ms) because HashBased index is used to enhance the performance of queries on exact values only.
And that's what we have in our query exact values terms such that (mov_id = 1, act_id = 1 ..) that's why it will work efficiently and in less time when executing the query.

D. Bitmap indexes:

	QUERY PLAN text
1	Nested Loop (cost=130000004621.15..130000004816.24 rows=48 width=42) (actual time=38.025..38.043 rows=6 loops=1)
2	-> Unique (cost=130000004621.11..130000004621.35 rows=48 width=4) (actual time=38.003..38.004 rows=6 loops=1)
3	-> Sort (cost=130000004621.11..130000004621.23 rows=48 width=4) (actual time=38.002..38.003 rows=6 loops=1)
4	Sort Key: movie_direction.dir_id
5	Sort Method: quicksort Memory: 25kB
6	-> Nested Loop (cost=130000004521.29..130000004619.77 rows=48 width=4) (actual time=37.959..37.976 rows=6 loops=1)
7	-> Unique (cost=130000004521.24..130000004521.36 rows=24 width=4) (actual time=37.912..37.915 rows=3 loops=1)
8	-> Sort (cost=130000004521.24..130000004521.30 rows=24 width=4) (actual time=37.911..37.912 rows=16 loops=1)
9	Sort Key: movie_cast.mov_id
10	Sort Method: quicksort Memory: 25kB
11	-> Nested Loop (cost=20000000080.62..130000004520.68 rows=24 width=4) (actual time=0.808..37.823 rows=16 loops=1)
12	Join Filter: (movie_cast.role = movie_cast_1.role)
13	Rows Removed by Join Filter: 143984
14	-> Unique (cost=10000000080.62..10000000080.68 rows=12 width=31) (actual time=0.466..0.506 rows=12 loops=1)
15	-> Sort (cost=10000000080.62..10000000080.65 rows=12 width=31) (actual time=0.465..0.484 rows=12 loops=1)
16	Sort Key: movie_cast_1.role
17	Sort Method: quicksort Memory: 25kB
18	-> Nested Loop (cost=10000000044.60..10000000080.41 rows=12 width=31) (actual time=0.387..0.420 rows=12 loops=1)
19	-> Unique (cost=10000000032.51..10000000032.52 rows=1 width=4) (actual time=0.209..0.211 rows=1 loops=1)
20	-> Sort (cost=10000000032.51..10000000032.51 rows=1 width=4) (actual time=0.208..0.208 rows=1 loops=1)
21	Sort Key: movie.mov_id
22	Sort Method: quicksort Memory: 25kB
23	-> Seq Scan on movie (cost=10000000000.00..10000000032.50 rows=1 width=4) (actual time=0.034..0.174 rows=1 loops=1)

23	-> Seq Scan on movie (cost=10000000000.00..10000000032.50 rows=1 width=4) (actual time=0.034..0.174 rows=1 loops=1)
24	Filter: (mov_title = 'Movie2'::bpchar)
25	Rows Removed by Filter: 999
26	-> Bitmap Heap Scan on movie_cast movie_cast_1 (cost=12.09..47.77 rows=12 width=35) (actual time=0.162..0.188 rows=12 loops=1)
27	Recheck Cond: (mov_id = movie.mov_id)
28	Heap Blocks: exact=12
29	-> Bitmap Index Scan on bitmap_mov_id2 (cost=0.00..12.09 rows=12 width=0) (actual time=0.148..0.148 rows=12 loops=1)
30	Index Cond: (mov_id = movie.mov_id)
31	-> Seq Scan on movie_cast (cost=10000000000.00..10000000220.00 rows=12000 width=35) (actual time=0.007..1.433 rows=12000 loops=12)
32	-> Bitmap Heap Scan on movie_direction (cost=0.06..4.08 rows=2 width=8) (actual time=0.016..0.017 rows=2 loops=3)
33	Recheck Cond: (mov_id = movie_cast.mov_id)
34	Heap Blocks: exact=6
35	-> Bitmap Index Scan on bitmap_mov_id (cost=0.00..0.05 rows=2 width=0) (actual time=0.014..0.014 rows=2 loops=3)
36	Index Cond: (mov_id = movie_cast.mov_id)
37	-> Bitmap Heap Scan on director (cost=0.04..4.05 rows=1 width=46) (actual time=0.005..0.005 rows=1 loops=6)
38	Recheck Cond: (dir_id = movie_direction.dir_id)
39	Heap Blocks: exact=6
40	-> Bitmap Index Scan on bitmap_dir_id (cost=0.00..0.04 rows=1 width=0) (actual time=0.004..0.004 rows=1 loops=6)
41	Index Cond: (dir_id = movie_direction.dir_id)
42	Planning time: 0.667 ms
43	Execution time: 38.309 ms

- *Estimate Cost: 130000004816.24*
- *Average Execution time: 38.309 ms*

At first, a seq scan is applied to the relation movie where mov_title ='Movie2', removing 999 rows. The new set is sorted on key movie.mov_id and duplicates are removed then, merged with a bitmap heap applied on relation movie_cast_1 where bitmap index is on mov_id and mov_id = movie_mov_id, using nested loops that are then sorted as well on movie_cast_1.role and duplicates are removed, then it is joined with a seq scan on relation movie_cast with a filter: movie_cast.role = movie_cast_1.role causing the removal of 143984 more rows. The set produced after the nested loop is sorted on movie_cast.mov_id and duplicates again are removed, then the whole set is joined with a bitmap heap on relation movie_direction with bitmap index on mov_id where mov_id = movie_cast.mov_id. The new set produced from this nested loop is again sorted on movie_direction.dir_id and duplicates are removed then joined with bitmap heap on director with bitmap index on dir_id where dir_id = movie_direction.dir_id producing the final result set.

The difference between the execution time without index(*50.479* ms) and the execution time with index (*38.309* ms) because BitMap index answer many query types including on specific value, And, OR, XOR, and range .And we have in our query specific values terms such that (semester =1 , year = 2019…), we also have a 'in' and 'any' operations that's why it will work efficiently and in less time when executing the query, even though the difference is not that huge

- **Best Query Performance:**

using The HashBased index, was considered the best performer of average execution time of *0.245* ms and a cost of *70.70* because HashBased index work efficiently on exact values queries which is our case here in our query and also due to our insertion.

Query 12

- **Description of any changes in sql code:**

In the last 3 lines of the query (where dir_fname=‘actor1' and dir_lname='actor1'), the direction first name and last name are changed from ‘actor1’ to ‘Actor1’ for both, becoming (where dir_fname=‘Actor1' and dir_lname='Actor1').

- **Flags set:**

For Seq Scan:

```
set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For BTree/Hash:

```
set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For Bitmap:

```
set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

- **Output of Explain Analyze:**

A. no indexes.

	QUERY PLAN text
1	Seq Scan on movie (cost=83.00..115.50 rows=1 width=51) (actual time=0.395..0.469 rows=1 loops=1)
2	Filter: (mov_id = $1)
3	Rows Removed by Filter: 999
4	InitPlan 2 (returns $1)
5	-> Seq Scan on movie_direction (cost=49.00..83.00 rows=1 width=4) (actual time=0.259..0.383 rows=1 loops=1)
6	Filter: (dir_id = $0)
7	Rows Removed by Filter: 1999
8	InitPlan 1 (returns $0)
9	-> Seq Scan on director (cost=0.00..49.00 rows=1 width=4) (actual time=0.029..0.241 rows=1 loops=1)
10	Filter: ((dir_fname = 'Actor1'::bpchar) AND (dir_lname = 'Actor1'::bpchar))
11	Rows Removed by Filter: 1999
12	Planning time: 0.206 ms
13	Execution time: 0.514 ms

- *Estimate Cost: 115.50*
- *Average Execution time: 0.514 ms*

At first, a seq scan is applied to the relation director, with a filter on dir_fname and dir_lname both = ‘Actor1’. That resulted in the removal of 1999 rows. Another seq scan is then applied on movie_direction with another filter removing another 1999 rows. Then a last seq scan is applied on movie relation filtering mov_id removing 999 rows leaving only the remaining rows to be the final result set.

B. Btree indexes:

	QUERY PLAN text
1	Index Scan using btree_mov_id on movie (cost=16.87..24.89 rows=1 width=51) (actual time=0.060..0.060 rows=1 loops=1)
2	Index Cond: (mov_id = $1)
3	InitPlan 2 (returns $1)
4	-> Index Scan using btree_dir_id on movie_direction (cost=8.57..16.59 rows=1 width=4) (actual time=0.026..0.026 rows=1 loops=1)
5	Index Cond: (dir_id = $0)
6	InitPlan 1 (returns $0)
7	-> Index Scan using btree_dir_fname on director (cost=0.28..8.30 rows=1 width=4) (actual time=0.015..0.015 rows=1 loops=1)
8	Index Cond: (dir_fname = 'Actor1'::bpchar)
9	Filter: (dir_lname = 'Actor1'::bpchar)
10	Planning time: 0.341 ms
11	Execution time: 0.081 ms

- *Estimate Cost : 24.89*
- *Average Execution time: 0.054 ms*

There is a big difference between execution time where there was no index (0.514 ms) and with the BTree index(*0.054* ms) because B+ Tree index support queries on exact values , queries involving a range search..
And we have in our query exact values terms such that (Mov_id = 1) that's why it will work efficiently and in less time when executing the query.

C. Hash indexes:

	QUERY PLAN text
1	Index Scan using hash_mov_id on movie (cost=16.04..24.05 rows=1 width=51) (actual time=0.016..0.017 rows=1 loops=1)
2	Index Cond: (mov_id = $1)
3	InitPlan 2 (returns $1)
4	-> Index Scan using hash_dir_id on movie_direction (cost=8.02..16.04 rows=1 width=4) (actual time=0.013..0.013 rows=1 loops=1)
5	Index Cond: (dir_id = $0)
6	InitPlan 1 (returns $0)
7	-> Index Scan using hash_dir_fname on director (cost=0.00..8.02 rows=1 width=4) (actual time=0.006..0.006 rows=1 loops=1)
8	Index Cond: (dir_fname = 'Actor1'::bpchar)
9	Filter: (dir_lname = 'Actor1'::bpchar)
10	Planning time: 0.315 ms
11	Execution time: 0.055 ms

- *Estimate Cost: 24.05*
- *Average Execution time: 0.055 ms*

The huge difference between the execution time without index(0.514 ms) and the execution time with index (*0.055* ms) because HashBased index is used to enhance the performance of queries on exact values only.
And that's what we have in our query exact values terms such that (mov_id = 1, act_id = 1 ..) that's why it will work efficiently and in less time when executing the query.

D. Bitmap indexes:

	QUERY PLAN text
1	Bitmap Heap Scan on movie (cost=10000000069.03..10000000073.04 rows=1 width=51) (actual time=0.397..0.397 rows=1 loops=1)
2	Recheck Cond: (mov_id = $1)
3	Heap Blocks: exact=1
4	InitPlan 2 (returns $1)
5	-> Bitmap Heap Scan on movie_direction (cost=10000000057.01..10000000061.02 rows=1 width=4) (actual time=0.370..0.371 r...
6	Recheck Cond: (dir_id = $0)
7	Heap Blocks: exact=1
8	InitPlan 1 (returns $0)
9	-> Seq Scan on director (cost=10000000000.00..10000000049.00 rows=1 width=4) (actual time=0.011..0.327 rows=1 loops...
10	Filter: ((dir_fname = 'Actor1'::bpchar) AND (dir_lname = 'Actor1'::bpchar))
11	Rows Removed by Filter: 1999
12	-> Bitmap Index Scan on bitmap_dir_id (cost=0.00..8.01 rows=1 width=0) (actual time=0.362..0.362 rows=1 loops=1)
13	Index Cond: (dir_id = $0)
14	-> Bitmap Index Scan on bitmap_mov_id (cost=0.00..8.01 rows=1 width=0) (actual time=0.390..0.390 rows=1 loops=1)
15	Index Cond: (mov_id = $1)
16	Planning time: 0.819 ms
17	Execution time: 0.498 ms

- *Estimate Cost: 10000000073.04*
- *Average Execution time: 0.316 ms*

The huge difference between the execution time without index(0.514 ms) and the execution time with index (*0.316* ms) because HashBased index is used to enhance the performance of queries on exact values only.
And that's what we have in our query exact values terms such that (mov_id = 1, act_id = 1 ..) that's why it will work efficiently and in less time when executing the query.

- **Best Query Performance:**

The HashBased index, was considered the best performer of average execution time of *0.055* ms and a cost of *24.05* because HashBased index work efficiently on exact values queries which is our case here in our query and also due to our insertion. The btree is the second best with values very close to that of the hash based index.

SCHEMA 5

1. Description of data inserted:

I populated the soccer_country table with 195 records, player_must with 10000 records and the match_must with 5000 records. Any other table that has any of these attributes as foreign key was changed as well e.g. The match_details table has 10000 records, the match_no foreign key starts from 1 till 5000 then it repeats itself from the beginning; the country_id foreign key starts from 1 till 195 then repeats itself every 195 records till the end of the 10000 record loop; everything else took the values that came with the project.

QUERY 13

1) Description of any changes in sql code:

No changes were made to the sql code.

2) Flags set:

Case a: SEQSCAN

```
set enable_nestloop to on;
set enable_seqscan to on;
set enable_bitmapscan to off;
set enable_hashagg to on;
set enable_hashjoin to on;
set enable_material to on;
set enable_mergejoin to on;
set enable_indexscan to off;
set enable_indexonlyscan to off;
```

case b: BTREE/HASH

```
set enable_indexscan to on;
set enable_indexonlyscan to on;
set enable_bitmapscan to off;
set enable_hashagg to on;
set enable_hashjoin to on;
set enable_material to on;
set enable_mergejoin to on;
set enable_nestloop to on;
set enable_seqscan to off;
```

case c: BITMAP/GIN

```
set enable_indexscan to on;
set enable_indexonlyscan to on;
set enable_bitmapscan to on;
set enable_hashagg to on;
```

```
set enable_hashjoin to on;
set enable_material to on;
set enable_mergejoin to on;
set enable_nestloop to on;
set enable_seqscan to off;
```

3) Output of Explain Analyze:

Without index (Seq Scan):

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=258.35..264.98 rows=195 width=10) (actual time=5.058..5.122 rows=195 loops=1)
  Hash Cond: (soccer_country.country_id = match_details.team_id)
  ->  Seq Scan on soccer_country  (cost=0.00..3.95 rows=195 width=15) (actual time=0.019..0.030 rows=195 loops=1)
  ->  Hash  (cost=255.91..255.91 rows=195 width=5) (actual time=5.022..5.022 rows=195 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 16kB
        ->  HashAggregate  (cost=253.96..255.91 rows=195 width=5) (actual time=4.931..4.972 rows=195 loops=1)
              Group Key: match_details.team_id
              ->  Seq Scan on match_details  (cost=0.00..233.37 rows=8237 width=5) (actual time=0.017..2.794 rows=5001 loops=1)
                    Filter: ((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar))
                    Rows Removed by Filter: 4999
Planning Time: 0.905 ms
Execution Time: 5.193 ms
```

Seq scan finds relevant records by sequentially scanning the input record set.
It first scans the match_details relation while filtering with the condition:
((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar)) cost: 0.00..233.37 (1)
The rows removed by the filter were 4999 then, HashAggregate grouped the remaining 5001 records to become 195 based on match_details.team_id cost:253.96..255.91 (2)
then, a hash table was generated for the 195 records as the database engine decided to build a hash index on the fly. —> ResultSet1 cost: 255.91 .. 255.91 (3)
At the same time, the soccer_country relation was also sequentially scanned to produce 195 records (the whole soccer_country table) —> ResultSet2 cost:0.00..3.95 (3)
Then hash join was applied on the ResultSet1 and ResultSet2 based on the hash condition (soccer_country.country_id = match_details.team_id) cost: 258.35 .. 264.98 (4)

(1) took the biggest cost, because he had to sequentially iterate the table, even though we only needed about half.

BTree:

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.43..1227.60 rows=195 width=10) (actual time=0.076..12.495 rows=195 loops=1)
2	-> Index Scan using soccer_country_pkey on soccer_country (cost=0.14..16.07 rows=195 width=15) (actual time=0.012..0.106 rows=195 loops=1)
3	-> Index Scan using match_details_pkey on match_details (cost=0.29..79.00 rows=26 width=5) (actual time=0.061..0.061 rows=1 loops=195)
4	Index Cond: (team_id = soccer_country.country_id)
5	Filter: ((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar))
6	Rows Removed by Filter: 0
7	Planning Time: 0.486 ms
8	Execution Time: 12.588 ms

We perform an index scan on the relation match_details using the match_details_pk to find the relevant records based on the index condition:
(team_id = soccer_country.country_id)
And the filter:
((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar)) cost: 0.29..79
At the same time, we create an index scan on relation soccer_country using the soccer_country_pk (cost: 0.14..16.07)
Then, with both result sets we perform a nested loop, which merges the 2 record sets by looping through every record in the first set and trying to find a match in the second set (cost: 0.43..1227.60)

We traverse the table multiple times and hence that produces a bigger cost
There is a big difference between execution time where there was no index (4.291 ms) and with the BTree index(0.133 ms) because B+ Tree index support queries on exact values , queries involving a range search and queries containing aggregate functions.
And we have in our query exact values terms such that (Mov_id = 1) that's why it will work efficiently and in less time when executing the query.

Hash:

	QUERY PLAN text
1	Nested Loop (cost=632.52..663.84 rows=195 width=10) (actual time=3.400..4.042 rows=195 loops=1)
2	-> HashAggregate (cost=632.52..634.47 rows=195 width=5) (actual time=3.386..3.507 rows=195 loops=1)
3	Group Key: match_details.team_id
4	-> Index Scan using hash_winlose on match_details (cost=0.00..620.02 rows=5001 width=5) (actual time=0.016..1.834 rows=5001 loops=1)
5	Index Cond: (win_lose = 'W'::bpchar)
6	Filter: (play_stage = 'a'::bpchar)
7	-> Index Scan using hash_countryid_pk on soccer_country (cost=0.00..0.14 rows=1 width=15) (actual time=0.002..0.002 rows=1 loops=195)
8	Index Cond: (country_id = match_details.team_id)
9	Planning Time: 0.353 ms
10	Execution Time: 4.086 ms

First, we perform an index scan on the relation match_details using a hash index on the column match_details.win_lose . (cost: 0.00..620.02)
The index condition: (win_lose = 'W'::bpchar))
Then, HashAggregate grouped the remaining 5001 records to become 195 based on match_details.team_id (cost: 632.52..634.37)
At the same time, an index scan was performed on relation soccer_country using soccer_country.country_id as a hash index with the index condition :
(country_id = match_details.team_id) (cost: 0.00..0.14)
Then, with both result sets we perform a nested loop, which merges the 2 record sets by looping through every record in the first set and trying to find a match in the second set (cost: 632.52..663.84)

Bitmap:

Scratch Pad | Data Output

	QUERY PLAN text
1	Nested Loop Semi Join (cost=10000000001.46..10000000363.63 rows=195 width=10) (actual time=0.065..3.257 rows=195 loops=1)
2	-> Seq Scan on soccer_country (cost=10000000000.00..10000000003.95 rows=195 width=15) (actual time=0.012..0.035 rows=195 loops=1)
3	-> Bitmap Heap Scan on match_details (cost=1.46..6.35 rows=26 width=5) (actual time=0.012..0.012 rows=1 loops=195)
4	Recheck Cond: (team_id = soccer_country.country_id)
5	Filter: ((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar))
6	Heap Blocks: exact=195
7	-> Bitmap Index Scan on bitmap_matchdetails_pk (cost=0.00..1.45 rows=51 width=0) (actual time=0.008..0.008 rows=51 loops=195)
8	Index Cond: (team_id = soccer_country.country_id)
9	Planning Time: 0.843 ms
10	Execution Time: 3.317 ms

First, a bitmap index scan was performed using index bitmap on match_details.team_country with the index condition: (team_id = soccer_country.country_id) (cost: 0.00..1.45)
Results of this are fed to bitmap heap scan then, the bitmap heap scan searches through the pages returned by the bitmap index scan for relevant rows. (cost: 1.46..6.35)
Meanwhile, a sequential scan in performed on soccer_country (cost: 10000000000.00..10000000003.95)
Then, a nested loop merges the 2 record sets and returns the matching record. (cost: 10000000001.46..10000000363.63)

Mixed indexes:

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.14..84.41 rows=195 width=10) (actual time=0.124..8.494 rows=195 loops=1)
2	-> Index Scan using soccer_country_pkey on soccer_country (cost=0.14..16.07 rows=195 width=15) (actual time=0.010..0.117 rows=195 loops=1)
3	-> Index Scan using hash_matchdetails_pk on match_details (cost=0.00..4.41 rows=26 width=5) (actual time=0.042..0.042 rows=1 loops=195)
4	Index Cond: (team_id = soccer_country.country_id)
5	Filter: ((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar))
6	Rows Removed by Filter: 22
7	Planning Time: 1.909 ms
8	Execution Time: 8.567 ms

We perform an index scan using hash index on match_details.team_id (cost: 0.00..4.41)
Index condition: (team_id = soccer_country.country_id)
Filter condition: ((play_stage = 'a'::bpchar) AND (win_lose = 'W'::bpchar))

At the same time, we perform an index scan using BTree index on soccer_country.country_id.
(cost: 0.14..16.07)
Then, a nested loop merges the 2 record sets and returns the matching record.
(cost: 0.14..84.41)

- **Best Query Performance:**

The mixed indices scenario was the best. It performed a btree index on country_id and a hash index on matchdetails primary keys with a cost of 84.41 and execution time 8.5 ms

QUERY 14

- **Description of any changes in sql code:**

No changes were made in the sql queries.

- **Flags set:**

For Seq Scan:

```
set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For BTree/Hash:

```
set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For Bitmap:

```
set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

- **Output of Explain Analyze:**

A. *no indexes.*

	QUERY PLAN text
1	GroupAggregate (cost=246.28..248.31 rows=1 width=5) (actual time=7.380..7.746 rows=104 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	InitPlan 1 (returns $0)
5	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.012..0.054 rows=1 loops=1)
6	Filter: ((country_name)::text = 'Germany1'::text)
7	Rows Removed by Filter: 194
8	InitPlan 2 (returns $1)
9	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.011..0.054 rows=1 loo...
10	Filter: ((country_name)::text = 'Germany2'::text)
11	Rows Removed by Filter: 194
12	-> Sort (cost=237.40..237.66 rows=102 width=10) (actual time=7.344..7.358 rows=104 loops=1)
13	Sort Key: match_details.match_no
14	Sort Method: quicksort Memory: 29kB
15	-> Seq Scan on match_details (cost=0.00..234.00 rows=102 width=10) (actual time=0.076..7.217 rows=104 loops=1)
16	Filter: ((team_id = $0) OR (team_id = $1))
17	Rows Removed by Filter: 9896
18	Planning Time: 0.231 ms
19	Execution Time: 7.809 ms

- *Estimate Cost: 248.31*
- *Average Execution time: 7.809 ms*

B. Btree indexes:

	QUERY PLAN text
1	GroupAggregate (cost=16.61..612.10 rows=1 width=5) (actual time=0.121..12.231 rows=104 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	InitPlan 1 (returns $0)
5	-> Index Scan using btree_countryname on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.034..0.035 rows=1 loops=1)
6	Index Cond: ((country_name)::text = 'Germany1'::text)
7	InitPlan 2 (returns $1)
8	-> Index Scan using btree_countryname on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.012..0.013 rows=1 loops=1)
9	Index Cond: ((country_name)::text = 'Germany2'::text)
10	-> Index Scan using btree_matchno on match_details (cost=0.29..594.00 rows=102 width=10) (actual time=0.086..11.821 rows=104 loops=1)
11	Filter: ((team_id = $0) OR (team_id = $1))
12	Rows Removed by Filter: 9896
13	Planning Time: 0.748 ms
14	Execution Time: 12.303 ms

- *Average Execution Cost : 612.10*
- *Average Execution time: 12.303 ms*

C. Hash indexes:
(Bitmap scan enable is off)

QUERY PLAN text
GroupAggregate (cost=10000000118.21..10000000120.24 rows=1 width=5) (actual time=0.215..0.324 rows=104 loops=1)
Group Key: match_details.match_no
Filter: (count(DISTINCT match_details.team_id) = 1)
InitPlan 1 (returns $0)
-> Index Scan using hash_countryname on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.014..0.015 rows=1 loops=1)
Index Cond: ((country_name)::text = 'Germany1'::text)
InitPlan 2 (returns $1)
-> Index Scan using hash_countryname on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.004..0.005 rows=1 loo...
Index Cond: ((country_name)::text = 'Germany2'::text)
-> Sort (cost=10000000102.17..10000000102.43 rows=102 width=10) (actual time=0.195..0.199 rows=104 loops=1)
Sort Key: match_details.match_no
Sort Method: quicksort Memory: 29kB
-> Bitmap Heap Scan on match_details (cost=10000000008.82..10000000098.77 rows=102 width=10) (actual time=0.074..0.157 rows=104 loops=1)
Recheck Cond: ((team_id = $0) OR (team_id = $1))
Heap Blocks: exact=52
-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.065..0.065 rows=0 loops=1)
-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.035..0.035 rows=52 loops=1)
Index Cond: (team_id = $0)
-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.012..0.012 rows=52 loops=1)
Index Cond: (team_id = $1)
Planning Time: 0.683 ms
Execution Time: 0.501 ms

- *Estimate Cost: 10000000120.24*
- *Average Execution time: 0.501 ms*

(bitmap on)

	QUERY PLAN text
1	GroupAggregate (cost=118.21..120.24 rows=1 width=5) (actual time=0.163..0.297 rows=104 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	InitPlan 1 (returns $0)
5	-> Index Scan using hash_countryname on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.012..0.013 rows=1 loops=1)
6	Index Cond: ((country_name)::text = 'Germany1'::text)
7	InitPlan 2 (returns $1)
8	-> Index Scan using hash_countryname on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.002..0.002 rows=1 loops=1)
9	Index Cond: ((country_name)::text = 'Germany2'::text)
10	-> Sort (cost=102.17..102.43 rows=102 width=10) (actual time=0.153..0.157 rows=104 loops=1)
11	Sort Key: match_details.match_no
12	Sort Method: quicksort Memory: 29kB
13	-> Bitmap Heap Scan on match_details (cost=8.82..98.77 rows=102 width=10) (actual time=0.042..0.118 rows=104 loops=1)
14	Recheck Cond: ((team_id = $0) OR (team_id = $1))
15	Heap Blocks: exact=52
16	-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.032..0.032 rows=0 loops=1)
17	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.024..0.025 rows=52 loops=1)
18	Index Cond: (team_id = $0)
19	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.007..0.007 rows=52 loops=1)
20	Index Cond: (team_id = $1)
21	Planning Time: 0.265 ms
22	Execution Time: 0.352 ms

- *Estimate Cost: 120.24*
- *Average Execution time: 0.352 ms*

D. Bitmap indexes:

	QUERY PLAN text
1	GroupAggregate (cost=134.22..136.24 rows=1 width=5) (actual time=0.161..0.283 rows=104 loops=1)
2	Group Key: match_details.match_no
3	Filter: (count(DISTINCT match_details.team_id) = 1)
4	InitPlan 1 (returns $0)
5	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=5) (actual time=0.035..0.035 rows=1 loops=1)
6	Recheck Cond: ((country_name)::text = 'Germany1'::text)
7	Heap Blocks: exact=1
8	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.032..0.032 rows=1 loops=1)
9	Index Cond: ((country_name)::text = 'Germany1'::text)
10	InitPlan 2 (returns $1)
11	-> Bitmap Heap Scan on soccer_country soccer_country_1 (cost=8.01..12.02 rows=1 width=5) (actual time=0.005..0.006 rows=1 loops=1)
12	Recheck Cond: ((country_name)::text = 'Germany2'::text)
13	Heap Blocks: exact=1
14	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.005..0.005 rows=1 loops=1)
15	Index Cond: ((country_name)::text = 'Germany2'::text)
16	-> Sort (cost=110.18..110.43 rows=102 width=10) (actual time=0.152..0.155 rows=104 loops=1)
17	Sort Key: match_details.match_no
18	Sort Method: quicksort Memory: 29kB
19	-> Bitmap Heap Scan on match_details (cost=16.82..106.77 rows=102 width=10) (actual time=0.074..0.120 rows=104 loops=1)
20	Recheck Cond: ((team_id = $0) OR (team_id = $1))
21	Heap Blocks: exact=52
22	-> BitmapOr (cost=16.82..16.82 rows=103 width=0) (actual time=0.065..0.066 rows=0 loops=1)
23	-> Bitmap Index Scan on bitmap_teamid (cost=0.00..8.38 rows=51 width=0) (actual time=0.053..0.053 rows=52 loops=1)
24	Index Cond: (team_id = $0)
25	-> Bitmap Index Scan on bitmap_teamid (cost=0.00..8.38 rows=51 width=0) (actual time=0.012..0.012 rows=52 loops=1)
26	Index Cond: (team_id = $1)
27	Planning Time: 0.507 ms
28	Execution Time: 0.359 ms

- *Estimate Cost: 136.24*
- *Average Execution time: 0.359 ms*

- **The Best Performance:**

The best performance was that of the hash based scan and bitmap combined. Bitmap is best for Or operations and hash is best for exact values queries.

QUERY 15

- **Description of any changes in sql code:**

No changes were made in the sql queries.

- **Flags set:**

For Seq Scan:

```
set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For BTree/Hash:

```
set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

For Bitmap:

```
set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;
```

- **Output of Explain Analyze:**

A) *no indexes.*

	QUERY PLAN text
1	Hash Join (cost=248.33..363.46 rows=1 width=23) (actual time=3.142..44.886 rows=104 loops=1)
2	Hash Cond: (match_mast.match_no = match_details.match_no)
3	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=23) (actual time=0.019..40.881 rows=5000 loops=1)
4	-> Hash (cost=248.32..248.32 rows=1 width=5) (actual time=3.111..3.111 rows=104 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 12kB
6	-> GroupAggregate (cost=246.28..248.31 rows=1 width=5) (actual time=2.959..3.086 rows=104 loops=1)
7	Group Key: match_details.match_no
8	Filter: (count(DISTINCT match_details.team_id) = 1)
9	InitPlan 1 (returns $0)
10	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.029 rows=1 loops=1)
11	Filter: ((country_name)::text = 'Germany1'::text)
12	Rows Removed by Filter: 194
13	InitPlan 2 (returns $1)
14	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.006..0.023 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany2'::text)
16	Rows Removed by Filter: 194
17	-> Sort (cost=237.40..237.66 rows=102 width=10) (actual time=2.935..2.940 rows=104 loops=1)
18	Sort Key: match_details.match_no
19	Sort Method: quicksort Memory: 29kB
20	-> Seq Scan on match_details (cost=0.00..234.00 rows=102 width=10) (actual time=0.043..2.893 rows=104 loops=1)
21	Filter: ((team_id = $0) OR (team_id = $1))
22	Rows Removed by Filter: 9896
23	Planning Time: 7.689 ms
24	Execution Time: 44.953 ms

- *Estimate Cost: 363.46*
- *Average Execution time: 44.953 ms*

B. Btree indexes:

Scratch Pad | Explain | Messages | Notifications | Data Output

	QUERY PLAN text
1	Nested Loop (cost=16.89..620.41 rows=1 width=23) (actual time=0.091..7.632 rows=104 loops=1)
2	-> GroupAggregate (cost=16.61..612.10 rows=1 width=5) (actual time=0.070..7.143 rows=104 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	InitPlan 1 (returns $0)
6	-> Index Scan using btree_countryname on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.022..0.022 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1'::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using btree_countryname on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.007..0.007 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2'::text)
11	-> Index Scan using btree_matchno on match_details (cost=0.29..594.00 rows=102 width=10) (actual time=0.049..6.862 rows=104 loops=1)
12	Filter: ((team_id = $0) OR (team_id = $1))
13	Rows Removed by Filter: 9896
14	-> Index Scan using btree_matchno_matchmast on match_mast (cost=0.28..8.30 rows=1 width=23) (actual time=0.004..0.004 rows=1 loops=104)
15	Index Cond: (match_no = match_details.match_no)
16	Planning Time: 0.715 ms
17	Execution Time: 7.698 ms

- *Average Execution Cost : 620.41*
- *Average Execution time: 7.698 ms*

C. Hash indexes:
(Bitmap scan enable is off)

Data Output

	QUERY PLAN text
1	Nested Loop (cost=10000000118.21..10000000128.27 rows=1 width=23) (actual time=0.193..0.524 rows=104 loops=1)
2	-> GroupAggregate (cost=10000000118.21..10000000120.24 rows=1 width=5) (actual time=0.186..0.318 rows=104 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	InitPlan 1 (returns $0)
6	-> Index Scan using hash_countryname on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.008..0.008 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1'::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using hash_countryname on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.002..0.002 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2'::text)
11	-> Sort (cost=10000000102.17..10000000102.43 rows=102 width=10) (actual time=0.174..0.178 rows=104 loops=1)
12	Sort Key: match_details.match_no
13	Sort Method: quicksort Memory: 29kB
14	-> Bitmap Heap Scan on match_details (cost=10000000008.82..10000000098.77 rows=102 width=10) (actual time=0.036..0.140 rows=104 loops=1)
15	Recheck Cond: ((team_id = $0) OR (team_id = $1))
16	Heap Blocks: exact=52
17	-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.028..0.028 rows=0 loops=1)
18	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.021..0.021 rows=52 loops=1)
19	Index Cond: (team_id = $0)
20	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.006..0.006 rows=52 loops=1)
21	Index Cond: (team_id = $1)
22	-> Index Scan using hash_matchno_matchmast on match_mast (cost=0.00..8.02 rows=1 width=23) (actual time=0.002..0.002 rows=1 loops=104)
23	Index Cond: (match_no = match_details.match_no)
24	Planning Time: 0.428 ms
25	Execution Time: 0.618 ms

- *Estimate Cost: 10000000128.27*
- *Average Execution time: 0.618 ms*

(bitmap on)

- *Estimate Cost: 128.27*
- *Average Execution time: 0.588 ms*

	QUERY PLAN text
1	Nested Loop (cost=118.21..128.27 rows=1 width=23) (actual time=0.170..0.501 rows=104 loops=1)
2	-> GroupAggregate (cost=118.21..120.24 rows=1 width=5) (actual time=0.159..0.301 rows=104 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	InitPlan 1 (returns $0)
6	-> Index Scan using hash_countryname on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.009..0.010 rows=1 loops=1)
7	Index Cond: ((country_name)::text = 'Germany1''::text)
8	InitPlan 2 (returns $1)
9	-> Index Scan using hash_countryname on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.004 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany2''::text)
11	-> Sort (cost=102.17..102.43 rows=102 width=10) (actual time=0.148..0.153 rows=104 loops=1)
12	Sort Key: match_details.match_no
13	Sort Method: quicksort Memory: 29kB
14	-> Bitmap Heap Scan on match_details (cost=8.82..98.77 rows=102 width=10) (actual time=0.042..0.107 rows=104 loops=1)
15	Recheck Cond: ((team_id = $0) OR (team_id = $1))
16	Heap Blocks: exact=52
17	-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.033..0.033 rows=0 loops=1)
18	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.025..0.025 rows=52 loops=1)
19	Index Cond: (team_id = $0)
20	-> Bitmap Index Scan on hash_teamid (cost=0.00..4.38 rows=51 width=0) (actual time=0.008..0.008 rows=52 loops=1)
21	Index Cond: (team_id = $1)
22	-> Index Scan using hash_matchno_matchmast on match_mast (cost=0.00..8.02 rows=1 width=23) (actual time=0.001..0.001 rows=1 loops=104)
23	Index Cond: (match_no = match_details.match_no)
24	Planning Time: 0.428 ms
25	Execution Time: 0.588 ms

D. Bitmap indexes:

	QUERY PLAN text
1	Nested Loop (cost=146.23..152.29 rows=1 width=23) (actual time=0.388..1.347 rows=104 loops=1)
2	-> GroupAggregate (cost=134.22..136.24 rows=1 width=5) (actual time=0.350..0.542 rows=104 loops=1)
3	Group Key: match_details.match_no
4	Filter: (count(DISTINCT match_details.team_id) = 1)
5	InitPlan 1 (returns $0)
6	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=5) (actual time=0.055..0.055 rows=1 loops=1)
7	Recheck Cond: ((country_name)::text = 'Germany1'::text)
8	Heap Blocks: exact=1
9	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.043..0.043 rows=1 loops=1)
10	Index Cond: ((country_name)::text = 'Germany1'::text)
11	InitPlan 2 (returns $1)
12	-> Bitmap Heap Scan on soccer_country soccer_country_1 (cost=8.01..12.02 rows=1 width=5) (actual time=0.013..0.014 rows=1 loops=1)
13	Recheck Cond: ((country_name)::text = 'Germany2'::text)
14	Heap Blocks: exact=1
15	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.011..0.011 rows=1 loops=1)
16	Index Cond: ((country_name)::text = 'Germany2'::text)
17	-> Sort (cost=110.18..110.43 rows=102 width=10) (actual time=0.317..0.325 rows=104 loops=1)
18	Sort Key: match_details.match_no
19	Sort Method: quicksort Memory: 29kB
20	-> Bitmap Heap Scan on match_details (cost=16.82..106.77 rows=102 width=10) (actual time=0.143..0.250 rows=104 loops=1)
21	Recheck Cond: ((team_id = $0) OR (team_id = $1))
22	Heap Blocks: exact=52
23	-> BitmapOr (cost=16.82..16.82 rows=103 width=0) (actual time=0.130..0.130 rows=0 loops=1)
24	-> Bitmap Index Scan on bitmap_teamid (cost=0.00..8.38 rows=51 width=0) (actual time=0.100..0.100 rows=52 loops=1)
25	Index Cond: (team_id = $0)
26	-> Bitmap Index Scan on bitmap_teamid (cost=0.00..8.38 rows=51 width=0) (actual time=0.027..0.027 rows=52 loops=1)
27	Index Cond: (team_id = $1)
28	-> Bitmap Heap Scan on match_mast (cost=12.01..16.02 rows=1 width=23) (actual time=0.004..0.004 rows=1 loops=104)
29	Recheck Cond: (match_no = match_details.match_no)
30	Heap Blocks: exact=104
31	-> Bitmap Index Scan on bitmap_matchno_matchmast (cost=0.00..12.01 rows=1 width=0) (actual time=0.003..0.003 rows=1 loops=104)
32	Index Cond: (match_no = match_details.match_no)
33	Planning Time: 0.989 ms
34	Execution Time: 1.801 ms

- *Estimate Cost: 152.29*
- *Average Execution time: 1.801 ms*

- **The Best Performance:**

The best performance was that of the hash based scan and bitmap combined. Bitmap is best for Or operations and hash is best for exact values queries.

QUERY 17

- **Description of any changes in sql code:**

No changes were made in the sql queries.

- **Flags set:**

For Seq Scan:

set enable_seqscan = on;
set enable_bitmapscan = off;
set enable_indexonlyscan = off;
set enable_indexscan = off;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

For BTree/Hash:

set enable_seqscan = off;
set enable_bitmapscan = off;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

For Bitmap:

set enable_seqscan = off;
set enable_bitmapscan = on;
set enable_indexonlyscan = on;
set enable_indexscan = on;

set enable_hashjoin = on;
set enable_hashagg = on;
set enable_nestloop = on;
set enable_mergejoin = on;
set enable_gathermerge = on;
set enable_material = on;

- **Output of Explain Analyze:**

A) *no indexes.*

	QUERY PLAN text
1	Seq Scan on player_mast (cost=1009.47..1238.47 rows=1 width=9) (actual time=12.886..15.498 rows=1 loops=1)
2	Filter: (player_id = $9)
3	Rows Removed by Filter: 9999
4	InitPlan 10 (returns $9)
5	-> Seq Scan on goal_details goal_details_1 (cost=750.47..1009.47 rows=1 width=5) (actual time=8.975..12.864 rows=1 loops=1)
6	Filter: ((match_no = $2) AND (team_id = $3) AND (goal_time = $8))
7	Rows Removed by Filter: 9999
8	InitPlan 3 (returns $2)
9	-> GroupAggregate (cost=246.28..248.31 rows=1 width=5) (actual time=3.815..3.950 rows=1 loops=1)
10	Group Key: match_details.match_no
11	Filter: (count(DISTINCT match_details.team_id) = 2)
12	Rows Removed by Filter: 103
13	InitPlan 1 (returns $0)
14	-> Seq Scan on soccer_country (cost=0.00..4.44 rows=1 width=5) (actual time=0.014..0.058 rows=1 loops=1)
15	Filter: ((country_name)::text = 'Germany1'::text)
16	Rows Removed by Filter: 194
17	InitPlan 2 (returns $1)
18	-> Seq Scan on soccer_country soccer_country_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.013..0.055 rows=1 loops=1)
19	Filter: ((country_name)::text = 'Germany2'::text)
20	Rows Removed by Filter: 194
21	-> Sort (cost=237.40..237.66 rows=102 width=10) (actual time=3.788..3.794 rows=105 loops=1)
22	Sort Key: match_details.match_no
23	Sort Method: quicksort Memory: 29kB
24	-> Seq Scan on match_details (cost=0.00..234.00 rows=102 width=10) (actual time=0.073..3.739 rows=105 loops=1)
25	Filter: ((team_id = $0) OR (team_id = $1))
26	Rows Removed by Filter: 9895
27	InitPlan 4 (returns $3)
28	-> Hash Join (cost=4.45..9.92 rows=1 width=5) (actual time=0.060..0.092 rows=1 loops=1)
29	Hash Cond: (b.team_id = a.country_id)
30	-> Seq Scan on soccer_team b (cost=0.00..4.95 rows=195 width=5) (actual time=0.018..0.029 rows=195 loops=1)
31	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.029..0.030 rows=1 loops=1)
32	Buckets: 1024 Batches: 1 Memory Usage: 9kB
33	-> Seq Scan on soccer_country a (cost=0.00..4.44 rows=1 width=5) (actual time=0.010..0.026 rows=1 loops=1)
34	Filter: ((country_name)::text = 'Germany2'::text)
35	Rows Removed by Filter: 194
36	InitPlan 9 (returns $8)
37	-> Aggregate (cost=492.23..492.24 rows=1 width=32) (actual time=4.915..4.915 rows=1 loops=1)
38	InitPlan 7 (returns $6)
39	-> GroupAggregate (cost=246.28..248.31 rows=1 width=5) (actual time=2.781..2.887 rows=1 loops=1)
40	Group Key: match_details_1.match_no
41	Filter: (count(DISTINCT match_details_1.team_id) = 2)
42	Rows Removed by Filter: 103
43	InitPlan 5 (returns $4)
44	-> Seq Scan on soccer_country soccer_country_2 (cost=0.00..4.44 rows=1 width=5) (actual time=0.005..0.020 rows=1 loops=1)
45	Filter: ((country_name)::text = 'Germany1'::text)
46	Rows Removed by Filter: 194

Activate Windows
Go to Settings to activate Windows.

47	InitPlan 6 (returns $5)
48	-> Seq Scan on soccer_country soccer_country_3 (cost=0.00..4.44 rows=1 width=5) (actual time=0.005..0.020 rows=1 loops=1)
49	Filter: ((country_name)::text = 'Germany2'::text)
50	Rows Removed by Filter: 194
51	-> Sort (cost=237.40..237.66 rows=102 width=10) (actual time=2.767..2.772 rows=105 loops=1)
52	Sort Key: match_details_1.match_no
53	Sort Method: quicksort Memory: 29kB
54	-> Seq Scan on match_details match_details_1 (cost=0.00..234.00 rows=102 width=10) (actual time=0.029..2.735 rows=105 loops=1)
55	Filter: ((team_id = $4) OR (team_id = $5))
56	Rows Removed by Filter: 9895
57	InitPlan 8 (returns $7)
58	-> Hash Join (cost=4.45..9.92 rows=1 width=5) (actual time=0.045..0.076 rows=1 loops=1)
59	Hash Cond: (b_1.team_id = a_1.country_id)
60	-> Seq Scan on soccer_team b_1 (cost=0.00..4.95 rows=195 width=5) (actual time=0.011..0.023 rows=195 loops=1)
61	-> Hash (cost=4.44..4.44 rows=1 width=5) (actual time=0.024..0.024 rows=1 loops=1)
62	Buckets: 1024 Batches: 1 Memory Usage: 9kB
63	-> Seq Scan on soccer_country a_1 (cost=0.00..4.44 rows=1 width=5) (actual time=0.007..0.022 rows=1 loops=1)
64	Filter: ((country_name)::text = 'Germany2'::text)
65	Rows Removed by Filter: 194
66	-> Seq Scan on goal_details (cost=0.00..234.00 rows=1 width=5) (actual time=2.975..4.912 rows=1 loops=1)
67	Filter: ((match_no = $6) AND (team_id = $7))
68	Rows Removed by Filter: 9999
69	Planning Time: 1.161 ms
70	Execution Time: 15.721 ms

- *Estimate Cost: 1238.47*
- *Average Execution time: 15.721 ms*

B. Btree indexes:

	QUERY PLAN text
1	Index Scan using btree_playerid on player_mast (cost=1276.93..1284.95 rows=1 width=9) (actual time=24.666..24.666 rows=1 loops=1)
2	Index Cond: (player_id = $12)
3	InitPlan 10 (returns $12)
4	-> Index Scan using btree_goaltime on goal_details goal_details_1 (cost=1268.63..1276.65 rows=1 width=5) (actual time=24.601..24.602 rows=1 loops=1)
5	Index Cond: (goal_time = $11)
6	Filter: ((match_no = $2) AND (team_id = $4))
7	InitPlan 3 (returns $2)
8	-> GroupAggregate (cost=16.61..612.10 rows=1 width=5) (actual time=0.192..11.930 rows=1 loops=1)
9	Group Key: match_details.match_no
10	Filter: (count(DISTINCT match_details.team_id) = 2)
11	Rows Removed by Filter: 103
12	InitPlan 1 (returns $0)
13	-> Index Scan using btree_countryname on soccer_country (cost=0.14..8.16 rows=1 width=5) (actual time=0.009..0.010 rows=1 loops=1)
14	Index Cond: ((country_name)::text = 'Germany1'::text)
15	InitPlan 2 (returns $1)
16	-> Index Scan using btree_countryname on soccer_country soccer_country_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.007..0.008 rows=1 loops=1)
17	Index Cond: ((country_name)::text = 'Germany2'::text)
18	-> Index Scan using btree_matchno_matchdetails on match_details (cost=0.29..594.00 rows=102 width=10) (actual time=0.020..11.498 rows=105 loops=1)
19	Filter: ((team_id = $0) OR (team_id = $1))
20	Rows Removed by Filter: 9895
21	InitPlan 4 (returns $4)
22	-> Nested Loop (cost=0.29..16.41 rows=1 width=5) (actual time=0.053..0.054 rows=1 loops=1)
23	-> Index Scan using btree_countryname on soccer_country a (cost=0.14..8.16 rows=1 width=5) (actual time=0.023..0.023 rows=1 loops=1)
24	Index Cond: ((country_name)::text = 'Germany2'::text)
25	-> Index Only Scan using btree_teamid_soccerteam on soccer_team b (cost=0.14..8.16 rows=1 width=5) (actual time=0.026..0.026 rows=1 loops=1)
26	Index Cond: (team_id = a.country_id)
27	Heap Fetches: 1
28	InitPlan 9 (returns $11)
29	-> Aggregate (cost=639.83..639.84 rows=1 width=32) (actual time=12.576..12.576 rows=1 loops=1)
30	InitPlan 7 (returns $7)
31	-> GroupAggregate (cost=16.61..612.10 rows=1 width=5) (actual time=0.271..12.490 rows=1 loops=1)
32	Group Key: match_details_1.match_no
33	Filter: (count(DISTINCT match_details_1.team_id) = 2)
34	Rows Removed by Filter: 103
35	InitPlan 5 (returns $5)
36	-> Index Scan using btree_countryname on soccer_country soccer_country_2 (cost=0.14..8.16 rows=1 width=5) (actual time=0.035..0.036 rows=1 lo...
37	Index Cond: ((country_name)::text = 'Germany1'::text)
38	InitPlan 6 (returns $6)
39	-> Index Scan using btree_countryname on soccer_country soccer_country_3 (cost=0.14..8.16 rows=1 width=5) (actual time=0.008..0.009 rows=1 lo...
40	Index Cond: ((country_name)::text = 'Germany2'::text)
41	-> Index Scan using btree_matchno_matchdetails on match_details match_details_1 (cost=0.29..594.00 rows=102 width=10) (actual time=0.085..12.0...
42	Filter: ((team_id = $5) OR (team_id = $6))
43	Rows Removed by Filter: 9895
44	InitPlan 8 (returns $9)
45	-> Nested Loop (cost=0.29..16.41 rows=1 width=5) (actual time=0.032..0.033 rows=1 loops=1)
46	-> Index Scan using btree_countryname on soccer_country a_1 (cost=0.14..8.16 rows=1 width=5) (actual time=0.017..0.018 rows=1 loops=1)

```
47                    Index Cond: ((country_name)::text = 'Germany2'::text)
48                 -> Index Only Scan using btree_teamid_soccerteam on soccer_team b_1  (cost=0.14..8.16 rows=1 width=5) (actual time=0.012..0.012 rows=1 loops=1)
49                    Index Cond: (team_id = a_1.country_id)
50                    Heap Fetches: 1
51              -> Index Scan using btree_matchno_goaldetails on goal_details  (cost=0.29..11.32 rows=1 width=5) (actual time=12.568..12.570 rows=1 loops=1)
52                 Index Cond: (match_no = $7)
53                 Filter: (team_id = $9)
54                 Rows Removed by Filter: 1
55  Planning Time: 2.718 ms
56  Execution Time: 24.842 ms
```


- *Average Execution Cost : 1284.95*
- *Average Execution time: 24.842 ms*

C. Hash indexes:

(bitmap on)

	QUERY PLAN text
1	Index Scan using hash_playerid on player_mast (cost=291.86..299.88 rows=1 width=9) (actual time=0.604..0.604 rows=1 loops=1)
2	Index Cond: (player_id = $11)
3	InitPlan 10 (returns $11)
4	-> Index Scan using hash_goaltime on goal_details goal_details_1 (cost=283.84..291.86 rows=1 width=5) (actual time=0.593..0.594 rows=1 loops=1)
5	Index Cond: (goal_time = $10)
6	Filter: ((match_no = $2) AND (team_id = $4))
7	InitPlan 3 (returns $2)
8	-> GroupAggregate (cost=118.21..120.24 rows=1 width=5) (actual time=0.124..0.240 rows=1 loops=1)
9	Group Key: match_details.match_no
10	Filter: (count(DISTINCT match_details.team_id) = 2)
11	Rows Removed by Filter: 103
12	InitPlan 1 (returns $0)
13	-> Index Scan using hash_countryname on soccer_country (cost=0.00..8.02 rows=1 width=5) (actual time=0.004..0.004 rows=1 loops=1)
14	Index Cond: ((country_name)::text = 'Germany1'::text)
15	InitPlan 2 (returns $1)
16	-> Index Scan using hash_countryname on soccer_country soccer_country_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
17	Index Cond: ((country_name)::text = 'Germany2'::text)
18	-> Sort (cost=102.17..102.43 rows=102 width=10) (actual time=0.116..0.120 rows=105 loops=1)
19	Sort Key: match_details.match_no
20	Sort Method: quicksort Memory: 29kB
21	-> Bitmap Heap Scan on match_details (cost=8.82..98.77 rows=102 width=10) (actual time=0.026..0.084 rows=105 loops=1)
22	Recheck Cond: ((team_id = $0) OR (team_id = $1))
23	Heap Blocks: exact=53
24	-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.020..0.020 rows=0 loops=1)
25	-> Bitmap Index Scan on hash_teamid_matchdetails (cost=0.00..4.38 rows=51 width=0) (actual time=0.013..0.013 rows=53 loops=1)
26	Index Cond: (team_id = $0)
27	-> Bitmap Index Scan on hash_teamid_matchdetails (cost=0.00..4.38 rows=51 width=0) (actual time=0.007..0.007 rows=52 loops=1)
28	Index Cond: (team_id = $1)
29	InitPlan 4 (returns $4)
30	-> Nested Loop (cost=0.00..16.12 rows=1 width=5) (actual time=0.008..0.008 rows=1 loops=1)
31	-> Index Scan using hash_countryname on soccer_country a (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.004 rows=1 loops=1)
32	Index Cond: ((country_name)::text = 'Germany2'::text)
33	-> Index Scan using hash_teamid_soccerteam on soccer_team b (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
34	Index Cond: (team_id = a.country_id)
35	InitPlan 9 (returns $10)
36	-> Aggregate (cost=147.47..147.48 rows=1 width=32) (actual time=0.338..0.338 rows=1 loops=1)
37	InitPlan 7 (returns $7)
38	-> GroupAggregate (cost=118.21..120.24 rows=1 width=5) (actual time=0.187..0.309 rows=1 loops=1)
39	Group Key: match_details_1.match_no
40	Filter: (count(DISTINCT match_details_1.team_id) = 2)
41	Rows Removed by Filter: 103
42	InitPlan 5 (returns $5)
43	-> Index Scan using hash_countryname on soccer_country soccer_country_2 (cost=0.00..8.02 rows=1 width=5) (actual time=0.011..0.012 rows=1 loops=1)
44	Index Cond: ((country_name)::text = 'Germany1'::text)
45	InitPlan 6 (returns $6)
46	-> Index Scan using hash_countryname on soccer_country soccer_country_3 (cost=0.00..8.02 rows=1 width=5) (actual time=0.004..0.004 rows=1 loops=1)

Activate Windows
Go to Settings to activate Windows.

	QUERY PLAN text
47	Index Cond: ((country_name)::text = 'Germany2'::text)
48	-> Sort (cost=102.17..102.43 rows=102 width=10) (actual time=0.174..0.178 rows=105 loops=1)
49	Sort Key: match_details_1.match_no
50	Sort Method: quicksort Memory: 29kB
51	-> Bitmap Heap Scan on match_details match_details_1 (cost=8.82..98.77 rows=102 width=10) (actual time=0.046..0.141 rows=105 loops=1)
52	Recheck Cond: ((team_id = $5) OR (team_id = $6))
53	Heap Blocks: exact=53
54	-> BitmapOr (cost=8.82..8.82 rows=103 width=0) (actual time=0.036..0.036 rows=0 loops=1)
55	-> Bitmap Index Scan on hash_teamid_matchdetails (cost=0.00..4.38 rows=51 width=0) (actual time=0.026..0.026 rows=53 loops=1)
56	Index Cond: (team_id = $5)
57	-> Bitmap Index Scan on hash_teamid_matchdetails (cost=0.00..4.38 rows=51 width=0) (actual time=0.009..0.009 rows=52 loops=1)
58	Index Cond: (team_id = $6)
59	InitPlan 8 (returns $9)
60	-> Nested Loop (cost=0.00..16.12 rows=1 width=5) (actual time=0.009..0.009 rows=1 loops=1)
61	-> Index Scan using hash_countryname on soccer_country a_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.004..0.005 rows=1 loops=1)
62	Index Cond: ((country_name)::text = 'Germany2'::text)
63	-> Index Scan using hash_teamid_soccerteam on soccer_team b_1 (cost=0.00..8.02 rows=1 width=5) (actual time=0.003..0.003 rows=1 loops=1)
64	Index Cond: (team_id = a_1.country_id)
65	-> Bitmap Heap Scan on goal_details (cost=4.02..11.12 rows=1 width=5) (actual time=0.333..0.336 rows=1 loops=1)
66	Recheck Cond: (match_no = $7)
67	Filter: (team_id = $9)
68	Rows Removed by Filter: 1
69	Heap Blocks: exact=2
70	-> Bitmap Index Scan on hash_matchno_goaldetails (cost=0.00..4.01 rows=2 width=0) (actual time=0.319..0.319 rows=2 loops=1)
71	Index Cond: (match_no = $7)
72	Planning Time: 1.400 ms
73	Execution Time: 0.824 ms

Activate Windows
Go to Settings to activate Windows.

- *Estimate Cost: 299.88*
- *Average Execution time: 0.824 ms*

D. Bitmap indexes:

	QUERY PLAN text
1	Bitmap Heap Scan on player_mast (cost=367.76..371.77 rows=1 width=9) (actual time=1.082..1.083 rows=1 loops=1)
2	Recheck Cond: (player_id = $11)
3	Heap Blocks: exact=1
4	InitPlan 10 (returns $11)
5	-> Bitmap Heap Scan on goal_details goal_details_1 (cost=351.73..355.75 rows=1 width=5) (actual time=1.067..1.067 rows=1 loops=1)
6	Recheck Cond: (goal_time = $10)
7	Filter: ((match_no = $2) AND (team_id = $4))
8	Heap Blocks: exact=1
9	InitPlan 3 (returns $2)
10	-> GroupAggregate (cost=134.22..136.24 rows=1 width=5) (actual time=0.157..0.441 rows=1 loops=1)
11	Group Key: match_details.match_no
12	Filter: (count(DISTINCT match_details.team_id) = 2)
13	Rows Removed by Filter: 103
14	InitPlan 1 (returns $0)
15	-> Bitmap Heap Scan on soccer_country (cost=8.01..12.02 rows=1 width=5) (actual time=0.008..0.008 rows=1 loops=1)
16	Recheck Cond: ((country_name)::text = 'Germany1'::text)
17	Heap Blocks: exact=1
18	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.007..0.007 rows=1 loops=1)
19	Index Cond: ((country_name)::text = 'Germany1'::text)
20	InitPlan 2 (returns $1)
21	-> Bitmap Heap Scan on soccer_country soccer_country_1 (cost=8.01..12.02 rows=1 width=5) (actual time=0.006..0.006 rows=1 loops=1)
22	Recheck Cond: ((country_name)::text = 'Germany2'::text)
23	Heap Blocks: exact=1
24	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.005..0.005 rows=1 loops=1)
25	Index Cond: ((country_name)::text = 'Germany2'::text)
26	-> Sort (cost=110.18..110.43 rows=102 width=10) (actual time=0.143..0.151 rows=105 loops=1)
27	Sort Key: match_details.match_no
28	Sort Method: quicksort Memory: 29kB
29	-> Bitmap Heap Scan on match_details (cost=16.82..106.77 rows=102 width=10) (actual time=0.053..0.100 rows=105 loops=1)
30	Recheck Cond: ((team_id = $0) OR (team_id = $1))
31	Heap Blocks: exact=53
32	-> BitmapOr (cost=16.82..16.82 rows=103 width=0) (actual time=0.045..0.045 rows=0 loops=1)
33	-> Bitmap Index Scan on bitmap_teamid_matchdetails (cost=0.00..8.38 rows=51 width=0) (actual time=0.027..0.027 rows=53 loops=1)
34	Index Cond: (team_id = $0)
35	-> Bitmap Index Scan on bitmap_teamid_matchdetails (cost=0.00..8.38 rows=51 width=0) (actual time=0.017..0.017 rows=52 loops=1)
36	Index Cond: (team_id = $1)
37	InitPlan 4 (returns $4)
38	-> Nested Loop (cost=16.02..24.05 rows=1 width=5) (actual time=0.039..0.040 rows=1 loops=1)
39	-> Bitmap Heap Scan on soccer_country a (cost=8.01..12.02 rows=1 width=5) (actual time=0.014..0.015 rows=1 loops=1)
40	Recheck Cond: ((country_name)::text = 'Germany2'::text)
41	Heap Blocks: exact=1
42	-> Bitmap Index Scan on bitmap_countryname (cost=0.00..8.01 rows=1 width=0) (actual time=0.012..0.012 rows=1 loops=1)
43	Index Cond: ((country_name)::text = 'Germany2'::text)
44	-> Bitmap Heap Scan on soccer_team b (cost=8.01..12.02 rows=1 width=5) (actual time=0.011..0.011 rows=1 loops=1)
45	Recheck Cond: (team_id = a.country_id)
46	Heap Blocks: exact=1

```
47              ->  Bitmap Index Scan on bitmap_teamid_soccerteam  (cost=0.00..8.01 rows=1 width=0) (actual time=0.008..0.008 rows=1 loops=1)
48                    Index Cond: (team_id = a.country_id)
49        InitPlan 9 (returns $10)
50          ->  Aggregate  (cost=179.42..179.43 rows=1 width=32) (actual time=0.574..0.574 rows=1 loops=1)
51              InitPlan 7 (returns $7)
52                ->  GroupAggregate  (cost=134.22..136.24 rows=1 width=5) (actual time=0.260..0.520 rows=1 loops=1)
53                    Group Key: match_details_1.match_no
54                    Filter: (count(DISTINCT match_details_1.team_id) = 2)
55                    Rows Removed by Filter: 103
56                    InitPlan 5 (returns $5)
57                      ->  Bitmap Heap Scan on soccer_country soccer_country_2  (cost=8.01..12.02 rows=1 width=5) (actual time=0.026..0.026 rows=1 loops=1)
58                            Recheck Cond: ((country_name)::text = 'Germany1'::text)
59                            Heap Blocks: exact=1
60                            ->  Bitmap Index Scan on bitmap_countryname  (cost=0.00..8.01 rows=1 width=0) (actual time=0.022..0.022 rows=1 loops=1)
61                                  Index Cond: ((country_name)::text = 'Germany1'::text)
62                    InitPlan 6 (returns $6)
63                      ->  Bitmap Heap Scan on soccer_country soccer_country_3  (cost=8.01..12.02 rows=1 width=5) (actual time=0.010..0.010 rows=1 loops=1)
64                            Recheck Cond: ((country_name)::text = 'Germany2'::text)
65                            Heap Blocks: exact=1
66                            ->  Bitmap Index Scan on bitmap_countryname  (cost=0.00..8.01 rows=1 width=0) (actual time=0.008..0.008 rows=1 loops=1)
67                                  Index Cond: ((country_name)::text = 'Germany2'::text)
68                    ->  Sort  (cost=110.18..110.43 rows=102 width=10) (actual time=0.239..0.247 rows=105 loops=1)
69                          Sort Key: match_details_1.match_no
70                          Sort Method: quicksort  Memory: 29kB
71                          ->  Bitmap Heap Scan on match_details match_details_1  (cost=16.82..106.77 rows=102 width=10) (actual time=0.093..0.178 rows=105 loops=1)
72                                Recheck Cond: ((team_id = $5) OR (team_id = $6))
73                                Heap Blocks: exact=53
74                                ->  BitmapOr  (cost=16.82..16.82 rows=103 width=0) (actual time=0.082..0.082 rows=0 loops=1)
75                                      ->  Bitmap Index Scan on bitmap_teamid_matchdetails  (cost=0.00..8.38 rows=51 width=0) (actual time=0.055..0.055 rows=53 loops=1)
76                                            Index Cond: (team_id = $5)
77                                      ->  Bitmap Index Scan on bitmap_teamid_matchdetails  (cost=0.00..8.38 rows=51 width=0) (actual time=0.026..0.026 rows=52 loops=1)
78                                            Index Cond: (team_id = $6)
79              InitPlan 8 (returns $9)
80                ->  Nested Loop  (cost=16.02..24.05 rows=1 width=5) (actual time=0.026..0.027 rows=1 loops=1)
81                    ->  Bitmap Heap Scan on soccer_country a_1  (cost=8.01..12.02 rows=1 width=5) (actual time=0.009..0.009 rows=1 loops=1)
82                          Recheck Cond: ((country_name)::text = 'Germany2'::text)
83                          Heap Blocks: exact=1
84                          ->  Bitmap Index Scan on bitmap_countryname  (cost=0.00..8.01 rows=1 width=0) (actual time=0.008..0.008 rows=1 loops=1)
85                                Index Cond: ((country_name)::text = 'Germany2'::text)
86                    ->  Bitmap Heap Scan on soccer_team b_1  (cost=8.01..12.02 rows=1 width=5) (actual time=0.007..0.008 rows=1 loops=1)
87                          Recheck Cond: (team_id = a_1.country_id)
88                          Heap Blocks: exact=1
89                          ->  Bitmap Index Scan on bitmap_teamid_soccerteam  (cost=0.00..8.01 rows=1 width=0) (actual time=0.005..0.005 rows=1 loops=1)
90                                Index Cond: (team_id = a_1.country_id)
91              ->  Bitmap Heap Scan on goal_details  (cost=12.02..19.12 rows=1 width=5) (actual time=0.570..0.572 rows=1 loops=1)
92                    Recheck Cond: (match_no = $7)
93                    Filter: (team_id = $9)
```


Activate Windows
Go to Settings to activate Windows.

- *Estimate Cost: 371.77*
- *Average Execution time: 1.590 ms*

- **The Best Performance:**

The best performance was that of the hash based scan and bitmap combined. Bitmap is best for Or operations and hash is best for exact values queries.

Schema 2:

Data description:

employee id’s goes from 1->5000

here are the employees that have lname of employee1

	fname character (20)	minit character (10)	lname character (20)	ssn [PK] integer	bdate date	address character (20)	sex character (1)	salary integer	super_snn integer	dno integer
1	employee2	M1	employee1	2	1927-07-...	address1	M	20	2	2
2	employee16	M201	employee1	202	1927-07-...	address201	M	2020	202	16
3	employee30	M401	employee1	402	1927-07-...	address401	M	4020	402	30
4	employee13	M601	employee1	602	1927-07-...	address601	M	6020	602	13
5	employee27	M801	employee1	802	1927-07-...	address801	M	8020	802	27
6	employee10	M1001	employee1	1002	1927-07-...	address1001	M	10020	1002	10
7	employee24	M1201	employee1	1202	1927-07-...	address1201	M	12020	1202	24
8	employee7	M1401	employee1	1402	1927-07-...	address1401	M	14020	1402	7
9	employee21	M1601	employee1	1602	1927-07-...	address1601	M	16020	1602	21
10	employee4	M1801	employee1	1802	1927-07-...	address1801	M	18020	1802	4
11	employee18	M2001	employee1	2002	1927-07-...	address2001	M	20020	2002	18
12	employee1	M2201	employee1	2202	1927-07-...	address2201	M	22020	2202	1
13	employee15	M2401	employee1	2402	1927-07-...	address2401	M	24020	2402	15
14	employee29	M2601	employee1	2602	1927-07-...	address2601	F	26020	2602	29
15	employee12	M2801	employee1	2802	1927-07-...	address2801	F	28020	2802	12
16	employee26	M3001	employee1	3002	1927-07-...	address3001	F	30020	3002	26
17	employee9	M3201	employee1	3202	1927-07-...	address3201	F	32020	3202	9
18	employee23	M3401	employee1	3402	1927-07-...	address3401	F	34020	3402	23
19	employee6	M3601	employee1	3602	1927-07-...	address3601	F	36020	3602	6
20	employee20	M3801	employee1	3802	1927-07-...	address3801	F	38020	3802	20
21	employee3	M4001	employee1	4002	1927-07-...	address4001	F	40020	4002	3
22	employee17	M4201	employee1	4202	1927-07-...	address4201	F	42020	4202	17
23	employee31	M4401	employee1	4402	1927-07-...	address4401	F	44020	4402	31
24	employee14	M4601	employee1	4602	1927-07-...	address4601	F	46020	4602	14
25	employee28	M4801	employee1	4802	1927-07-...	address4801	F	48020	4802	28

Please notice there are a lot of employees that work for department 5 and even more that earn more than 40000 so I will not be able to attach screen shots for those data, but everything in employee table can be reproduced using the following code:

```
@SuppressWarnings("deprecation")
public static void populateEmployee(Connection conn) {
    for (int i = 0; i < 5000; i++) {
            String result = "M";
            if (i > 2500)
                result = "F";
            if (insertEmployee("employee" + i, "M" + i,"employee" + ((i%30)+1), i+1, new Date(22,1,1999), "address" + i ,result,((i+1)*10),i+1,(i%30)+1, conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    int i = 10000;
        insertEmployee("employee" + i, "M" + i,"employee" + i, i, new Date(22,1,1999), "address" + i ,"M",i,i,1, conn);
        i++;
        insertEmployee("employee" + i, "M" + i,"employee" + i, i, new Date(22,1,1999), "address" + i ,"M",i,i,1, conn);

}
```

Department numbers are from 1-30:

	dname character (20)	dnumber [PK] integer	mgr_snn integer	mgr_start_date date
1	Department1	1	1	1906-07-14
2	Department2	2	2	1906-07-14
3	Department3	3	3	1906-07-14
4	Department4	4	4	1906-07-14
5	Department0	5	5	1906-07-14
6	Department1	6	6	1906-07-14
7	Department2	7	7	1906-07-14
8	Department3	8	8	1906-07-14
9	Department4	9	9	1906-07-14
10	Department0	10	10	1906-07-14
11	Department1	11	11	1906-07-14
12	Department2	12	12	1906-07-14
13	Department3	13	13	1906-07-14
14	Department4	14	14	1906-07-14
15	Department0	15	15	1906-07-14
16	Department1	16	16	1906-07-14
17	Department2	17	17	1906-07-14
18	Department3	18	18	1906-07-14
19	Department4	19	19	1906-07-14
20	Department0	20	20	1906-07-14
21	Department1	21	21	1906-07-14
22	Department2	22	22	1906-07-14
23	Department3	23	23	1906-07-14
24	Department4	24	24	1906-07-14
25	Department0	25	25	1906-07-14
26	Department1	26	26	1906-07-14
27	Department2	27	27	1906-07-14
28	Department3	28	28	1906-07-14
29	Department4	29	29	1906-07-14
30	Department0	30	30	1906-07-14

```java
@SuppressWarnings("deprecation")
public static void populateDepartment(Connection conn) {
    //31
    for (int i = 1; i < 31; i++) {
                                                    //i
        if (insertDepartment("Department" + i, i,i,new Date(1,1,1990), conn) == 0) {
            System.err.println("insertion of record " + i + " failed");
            break;
        } else
            System.out.println("insertion was successful");
    }
}
    public static void populateDeptLocations(Connection conn) {
        for (int i = 1; i < 21; i++) {
            if (insertDeptLocations(i, "Location" + i, conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }

    public static void populateProject(Connection conn) {
                            //601
        for (int i = 0; i < 600; i++) {
            if (insertProject("Project" + i, i+1,"Location1" + i,(i%30)+1, conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }
    public static void populateWorksOn(Connection conn) {
        //didn't increase it
        for (int i = 0; i < 10000; i++) {
            if (insertWorksOn((i%5000)+1, (i%600)+1, i, conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }
    @SuppressWarnings("deprecation")
    public static void populateDependent(Connection conn) {
        for (int i = 1; i < 50; i++) {
            String result = "F";
            if (i > 25)
                result = "M";
            if (insertDependent(i, "Name" + i%10, result,new Date(1,1,1999),"child", conn) == 0) {
                System.err.println("insertion of record " + i + " failed");
                break;
            } else
                System.out.println("insertion was successful");
        }
    }
```

You can generate the rest of data using the code above:
1.As you can see that department locations are from 1-20
2.we have 600 projects running from 1-600

query 2

No index case:

Flags used:

set enable_bitmapscan=off;
set enable_hashjoin=off;
set enable_mergejoin=off;

Data Output

	QUERY PLAN text
1	Unique (cost=25643.18..25645.43 rows=450 width=4) (actual time=1.791..1.915 rows=600 loops=1)
2	-> Sort (cost=25643.18..25644.31 rows=450 width=4) (actual time=1.791..1.828 rows=600 loops=1)
3	Sort Key: project.pnumber
4	Sort Method: quicksort Memory: 53kB
5	-> Seq Scan on project (cost=25605.35..25623.35 rows=450 width=4) (actual time=1.562..1.688 rows=600 loops=1)
6	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
7	SubPlan 1
8	-> Nested Loop (cost=0.00..254.55 rows=20 width=4) (actual time=0.026..1.361 rows=600 loops=1)
9	-> Nested Loop (cost=0.00..233.55 rows=1 width=0) (actual time=0.023..1.228 rows=1 loops=1)
10	Join Filter: ((d.dnumber = e.dno) AND (d.mgr_snn = e.ssn))
11	Rows Removed by Join Filter: 5009
12	-> Seq Scan on employee e (cost=0.00..144.50 rows=167 width=8) (actual time=0.005..0.612 rows=167 loops=1)
13	Filter: (lname = 'employee1'::bpchar)
14	Rows Removed by Filter: 4833
15	-> Materialize (cost=0.00..1.45 rows=30 width=8) (actual time=0.000..0.002 rows=30 loops=167)
16	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.003..0.008 rows=30 loops=1)
17	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (actual time=0.003..0.071 rows=600 loops=1)
18	SubPlan 2
19	-> Nested Loop (cost=0.00..25349.92 rows=334 width=4) (never executed)
20	Join Filter: (works_on.essn = employee.ssn)
21	-> Seq Scan on works_on (cost=0.00..155.00 rows=10000 width=8) (never executed)
22	-> Materialize (cost=0.00..145.34 rows=167 width=4) (never executed)
23	-> Seq Scan on employee (cost=0.00..144.50 rows=167 width=4) (never executed)
24	Filter: (lname = 'employee1'::bpchar)
25	Planning Time: 0.218 ms
26	Execution Time: 1.970 ms

average execution is 2.25

BPlus case:

I created a Bplus indexes on table employee on column ssn
set enable_bitmapscan=off;
set enable_hashjoin=off;
set enable_mergejoin=off;

Data Output

	QUERY PLAN text
1	HashAggregate (cost=377.81..382.31 rows=450 width=4) (actual time=0.778..0.848 rows=600 loops=1)
2	Group Key: project.pnumber
3	-> Seq Scan on project (cost=358.68..376.68 rows=450 width=4) (actual time=0.487..0.623 rows=600 loops=1)
4	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
5	SubPlan 1
6	-> Nested Loop (cost=0.28..73.75 rows=20 width=4) (actual time=0.043..0.315 rows=600 loops=1)
7	-> Nested Loop (cost=0.28..52.75 rows=1 width=0) (actual time=0.035..0.115 rows=1 loops=1)
8	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.007..0.009 rows=30 loops=1)
9	-> Index Scan using btree_employee_ssn on employee e (cost=0.28..1.70 rows=1 width=8) (actual time=0.003..0.003 rows=0 loops=30)
10	Index Cond: (ssn = d.mgr_snn)
11	Filter: ((lname = 'employee1'::bpchar) AND (d.dnumber = dno))
12	Rows Removed by Filter: 1
13	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (actual time=0.006..0.090 rows=600 loops=1)
14	SubPlan 2
15	-> Hash Join (cost=88.21..284.05 rows=334 width=4) (never executed)
16	Hash Cond: (works_on.essn = employee.ssn)
17	-> Seq Scan on works_on (cost=0.00..155.00 rows=10000 width=8) (never executed)
18	-> Hash (cost=86.12..86.12 rows=167 width=4) (never executed)
19	-> Index Scan using btree_employee_lname on employee (cost=0.28..86.12 rows=167 width=4) (never executed)
20	Index Cond: (lname = 'employee1'::bpchar)
21	Planning Time: 0.668 ms
22	Execution Time: 0.936 ms

average 0.937ms

Gin case:

I created a gin index on table employee on column snn

Flags:

set enable_bitmapscan=off;

set enable_hashjoin=off;

set enable_mergejoin=off;

Data Output

	QUERY PLAN text
1	HashAggregate (cost=392.68..397.18 rows=450 width=4) (actual time=0.841..0.956 rows=600 loops=1)
2	Group Key: project.pnumber
3	-> Seq Scan on project (cost=373.55..391.55 rows=450 width=4) (actual time=0.476..0.687 rows=600 loops=1)
4	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
5	SubPlan 1
6	-> Nested Loop (cost=1.14..87.36 rows=20 width=4) (actual time=0.029..0.312 rows=600 loops=1)
7	-> Nested Loop (cost=1.14..66.36 rows=1 width=0) (actual time=0.024..0.152 rows=1 loops=1)
8	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.004..0.007 rows=30 loops=1)
9	-> Bitmap Heap Scan on employee e (cost=1.14..2.16 rows=1 width=8) (actual time=0.003..0.003 rows=0 loops=30)
10	Recheck Cond: (ssn = d.mgr_snn)
11	Filter: ((lname = 'employee1'::bpchar) AND (d.dnumber = dno))
12	Rows Removed by Filter: 1
13	Heap Blocks: exact=30
14	-> Bitmap Index Scan on gin_employee_ssn (cost=0.00..1.14 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=30)
15	Index Cond: (ssn = d.mgr_snn)
16	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (actual time=0.005..0.085 rows=600 loops=1)
17	SubPlan 2
18	-> Hash Join (cost=89.47..285.31 rows=334 width=4) (never executed)
19	Hash Cond: (works_on.essn = employee.ssn)
20	-> Seq Scan on works_on (cost=0.00..155.00 rows=10000 width=8) (never executed)
21	-> Hash (cost=87.38..87.38 rows=167 width=4) (never executed)
22	-> Bitmap Heap Scan on employee (cost=3.29..87.38 rows=167 width=4) (never executed)
23	Recheck Cond: (lname = 'employee1'::bpchar)
24	-> Bitmap Index Scan on gin_employee (cost=0.00..3.25 rows=167 width=0) (never executed)
25	Index Cond: (lname = 'employee1'::bpchar)
26	Planning Time: 0.839 ms
27	Execution Time: 1.073 ms

average 1.076ms

Hash case :

I have created a hash index on table employee on column ssn

Flags used:
set enable_bitmapscan=off;
set enable_hashjoin=off;
set enable_mergejoin=off;

Data Output

	QUERY PLAN text
1	HashAggregate (cost=375.13..379.63 rows=450 width=4) (actual time=0.717..0.786 rows=600 loops=1)
2	Group Key: project.pnumber
3	-> Seq Scan on project (cost=356.01..374.01 rows=450 width=4) (actual time=0.385..0.521 rows=600 loops=1)
4	Filter: ((hashed SubPlan 1) OR (hashed SubPlan 2))
5	SubPlan 1
6	-> Nested Loop (cost=0.00..70.28 rows=20 width=4) (actual time=0.019..0.201 rows=600 loops=1)
7	-> Nested Loop (cost=0.00..49.27 rows=1 width=0) (actual time=0.015..0.071 rows=1 loops=1)
8	-> Seq Scan on department d (cost=0.00..1.30 rows=30 width=8) (actual time=0.004..0.007 rows=30 loops=1)
9	-> Index Scan using hash_ssn on employee e (cost=0.00..1.59 rows=1 width=8) (actual time=0.002..0.002 rows=0 loops=30)
10	Index Cond: (ssn = d.mgr_snn)
11	Filter: ((lname = 'employee1'::bpchar) AND (d.dnumber = dno))
12	Rows Removed by Filter: 1
13	-> Seq Scan on project project_1 (cost=0.00..15.00 rows=600 width=4) (actual time=0.002..0.065 rows=600 loops=1)
14	SubPlan 2
15	-> Hash Join (cost=89.01..284.85 rows=334 width=4) (never executed)
16	Hash Cond: (works_on.essn = employee.ssn)
17	-> Seq Scan on works_on (cost=0.00..155.00 rows=10000 width=8) (never executed)
18	-> Hash (cost=86.92..86.92 rows=167 width=4) (never executed)
19	-> Index Scan using hash_lname on employee (cost=0.00..86.92 rows=167 width=4) (never executed)
20	Index Cond: (lname = 'employee1'::bpchar)
21	Planning Time: 0.575 ms
22	Execution Time: 0.869 ms

average 0.870ms

Best performance:

The top performance is from the hash case. Hash Index beat btree index(which comes second place) which has O(log(n)) access time, while the hash one has O(1) access time, so definitely hash is faster in that type of queries and also it is absolutely logical that the no index case is the slowest because it has access time of O(n), also from the screen shots we find out that the actual time difference between btree and hash is ~0.001 , but notice that it went on for 30 loops ,so the difference is 0.001*30=0.03 .

Query 3

No index case:

Flags used:

set enable_bitmapscan=off;

Data Output

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..364525.75 rows=2500 width=42) (actual time=67.874..68.169 rows=15 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4985
4	SubPlan 1
5	-> Materialize (cost=0.00..145.34 rows=167 width=4) (actual time=0.000..0.005 rows=85 loops=5000)
6	-> Seq Scan on employee employee_1 (cost=0.00..144.50 rows=167 width=4) (actual time=0.004..1.189 rows=167 loops=1)
7	Filter: (dno = 5)
8	Rows Removed by Filter: 4833
9	Planning Time: 0.646 ms
10	Execution Time: 68.201 ms

average execution=61.651ms

Btree:

I have created two btree indexes one on employee dno column and the other on employee salary column, the plan did not use the salary one because it is more efficient to do seq scan on employee table to answer salary> all(……., because now we don't have the over head of searching the salary index which is needed to be searched for all of dno=5 employees

Flags used:
set enable_bitmapscan=off;

Data Output

	QUERY PLAN text
1	Seq Scan on employee (cost=0.28..217835.74 rows=2500 width=42) (actual time=55.888..56.220 rows=15 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4985
4	SubPlan 1
5	-> Materialize (cost=0.28..86.94 rows=167 width=4) (actual time=0.000..0.004 rows=85 loops=5000)
6	-> Index Scan using btree_employee_dno on employee employee_1 (cost=0.28..86.11 rows=167 width=4) (actual time=0.006..0.287 rows=167 loops=1)
7	Index Cond: (dno = 5)
8	Planning Time: 0.151 ms
9	Execution Time: 56.245 ms

average 56.535ms

Gin Case:

I have created two gin indexes one on employee dno column and the other on employee salary column, the plan did not use the salary one because it is more efficient to do seq scan on employee table to answer salary> all(……., because now we don't have the overhead of searching the salary index which is needed to be searched for all of dno=5 employees

Flags:

set enable_bitmapscan=on;

Data Output

	QUERY PLAN text
1	Seq Scan on employee (cost=5.29..213499.79 rows=2500 width=42) (actual time=60.440..60.697 rows=15 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4985
4	SubPlan 1
5	-> Materialize (cost=5.29..90.22 rows=167 width=4) (actual time=0.000..0.005 rows=85 loops=5000)
6	-> Bitmap Heap Scan on employee employee_1 (cost=5.29..89.38 rows=167 width=4) (actual time=0.036..0.347 rows=167 loops=1)
7	Recheck Cond: (dno = 5)
8	Heap Blocks: exact=82
9	-> Bitmap Index Scan on gin_employee_dno (cost=0.00..5.25 rows=167 width=0) (actual time=0.025..0.025 rows=167 loops=1)
10	Index Cond: (dno = 5)
11	Planning Time: 0.155 ms
12	Execution Time: 60.737 ms

Average execution time: 74.107ms

Hash case:

I have created two hash indexes one on employee dno column and the other on employee salary column, the plan did not use the salary one because it is more efficient to do seq scan on employee table to answer salary> all(……., because now we don't have the overhead of searching the salary index which is needed to be searched for all of dno=5 employees

Flags:

set enable_bitmapscan=off;

Data Output

	QUERY PLAN text
1	Seq Scan on employee (cost=0.00..115582.00 rows=2500 width=42) (actual time=1.481..1.728 rows=15 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4985
4	SubPlan 1
5	-> Materialize (cost=0.00..45.76 rows=167 width=4) (actual time=0.000..0.000 rows=1 loops=5000)
6	-> Index Scan using hash_employee_dno on employee employee_1 (cost=0.00..44.92 rows=167 width=4) (actual time=0.011..0.066 rows=167 loops=1)
7	Index Cond: (dno = 5)
8	Planning Time: 0.127 ms
9	Execution Time: 1.756 ms

average 1.86ms

Best performance:

Best performance is from hash index, which makes sense because the access time to get all employees in dno=5 is O(1), which beats the O(log(n)) of btrees, and also it is logical that bitmaps did not come first, because of the overhead needed to make bitwise operations, and to know which page and which row has the record

Query number 4:

No index scenario:

Data Output

	QUERY PLAN text
1	Seq Scan on employee e (cost=0.00..41744.50 rows=2500 width=42) (actual time=0.017..68.333 rows=25 loops=1)
2	Filter: (SubPlan 1)
3	Rows Removed by Filter: 4975
4	SubPlan 1
5	-> Seq Scan on dependent d (cost=0.00..15.70 rows=376 width=4) (actual time=0.004..0.010 rows=24 loops=5000)
6	Filter: ((e.fname <> dependent_name) AND (e.sex <> sex))
7	Rows Removed by Filter: 24
8	Planning Time: 0.100 ms
9	Execution Time: 68.355 ms

Best performance:

The best performance in queries having != , will surely be from seq scan, the planner will not even bother to use any index, even when seq scan is off, and this is expected, because when != is used the index must be fully searched which causes low performance (ex. To search btree fully we need O(nlog(n)) which is a lower performance than O(n)), also the in database world we need to access the index pages then access table pages, which causes a huge I/O bottle-neck, so the planner sticks with seq scan .

Query 16:

No index:

Flags:

Bitmapscan= off

Lets try and understand this query plan first the planner made a seq scan on match_mast to get the max audience, to be used later, then the planner makes a hash semi join between soccer_country and goal_details, why semi hash join ? semi hash join is used because the condition is soccer_country.country_id **in**(……..), so regardless of the size returned from the query inside the **in** clause, the engine would still take it in the output.

Data Output

	QUERY PLAN text
1	Hash Semi Join (cost=438.03..442.52 rows=2 width=10) (actual time=6.208..6.239 rows=2 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=114.51..229.01 rows=1 width=10) (actual time=3.713..3.713 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=2.552..2.552 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.003..0.737 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.009..0.022 rows=195 loops=1)
11	-> Hash (cost=209.00..209.00 rows=2 width=5) (actual time=6.168..6.168 rows=2 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Seq Scan on goal_details (cost=0.00..209.00 rows=2 width=5) (actual time=4.514..6.129 rows=2 loops=1)
14	Filter: (match_no = $1)
15	Rows Removed by Filter: 9998
16	Planning Time: 0.623 ms
17	Execution Time: 6.330 ms

average execution time :8.06ms

Btree case:

Flags used

set enable_bitmapscan=off;

the index created is on table goal_details on column matchno

Data Output

	QUERY PLAN text
1	Hash Semi Join (cost=240.35..244.84 rows=2 width=10) (actual time=5.520..5.551 rows=2 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=114.51..229.01 rows=1 width=10) (actual time=5.435..5.436 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=2.999..2.999 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.005..1.183 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.017..0.031 rows=195 loops=1)
11	-> Hash (cost=11.32..11.32 rows=2 width=5) (actual time=5.463..5.463 rows=2 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Index Scan using btree_goal_details_matchno on goal_details (cost=0.29..11.32 rows=2 width=5) (actual time=5.456..5.460 rows=2 loops=1)
14	Index Cond: (match_no = $1)
15	Planning Time: 0.582 ms
16	Execution Time: 5.636 ms

average time execution 4.12ms

Gin case:

Flags used:

set enable_bitmapscan=on;

the index created is on table goal_details on column matchno

Data Output

	QUERY PLAN text
1	Hash Semi Join (cost=248.15..252.64 rows=2 width=10) (actual time=2.986..3.017 rows=2 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=114.51..229.01 rows=1 width=10) (actual time=2.914..2.915 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=1.649..1.650 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.002..0.417 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.006..0.023 rows=195 loops=1)
11	-> Hash (cost=19.11..19.11 rows=2 width=5) (actual time=2.959..2.959 rows=2 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Bitmap Heap Scan on goal_details (cost=12.02..19.11 rows=2 width=5) (actual time=2.953..2.955 rows=2 loops=1)
14	Recheck Cond: (match_no = $1)
15	Heap Blocks: exact=2
16	-> Bitmap Index Scan on gin_goal_details_matchno (cost=0.00..12.01 rows=2 width=0) (actual time=2.946..2.946 rows=2 loops=1)
17	Index Cond: (match_no = $1)
18	Planning Time: 0.287 ms
19	Execution Time: 3.090 ms

average time 3ms

Hash case:

Flags used:

set enable_bitmapscan=off;

the index created is on table goal_details on column matchno

Data Output

	QUERY PLAN text
1	Hash Semi Join (cost=241.07..245.56 rows=2 width=10) (actual time=2.790..2.818 rows=2 loops=1)
2	Hash Cond: (soccer_country.country_id = goal_details.team_id)
3	InitPlan 2 (returns $1)
4	-> Seq Scan on match_mast match_mast_1 (cost=114.51..229.01 rows=1 width=10) (actual time=2.708..2.708 rows=1 loops=1)
5	Filter: (audonce = $0)
6	Rows Removed by Filter: 4999
7	InitPlan 1 (returns $0)
8	-> Aggregate (cost=114.50..114.51 rows=1 width=32) (actual time=1.611..1.611 rows=1 loops=1)
9	-> Seq Scan on match_mast (cost=0.00..102.00 rows=5000 width=5) (actual time=0.003..0.530 rows=5000 loops=1)
10	-> Seq Scan on soccer_country (cost=0.00..3.95 rows=195 width=15) (actual time=0.015..0.026 rows=195 loops=1)
11	-> Hash (cost=12.04..12.04 rows=2 width=5) (actual time=2.745..2.745 rows=2 loops=1)
12	Buckets: 1024 Batches: 1 Memory Usage: 9kB
13	-> Index Scan using hash_goal_details_matchno on goal_details (cost=0.00..12.04 rows=2 width=5) (actual time=2.739..2.742 rows=2 loops=1)
14	Index Cond: (match_no = $1)
15	Planning Time: 0.306 ms
16	Execution Time: 2.861 ms

average execution time : 2.7 ms

Best performance:

Top performance is from btree index, but the difference between it and the hash one is So small, this can be due to differences in indexes size.

Data Output | Explain | Messages | Notifications

	schemaname name	tablename name	indexname name	num_rows real	table_size text	index_size text
1	public	asst_referee_m...	[null]	10000	512 kB	[null]
2	public	coach_mast	[null]	9999	440 kB	[null]
3	public	goal_details	hash_goal_details_matchno	10000	672 kB	528 kB
4	public	goal_details	goal_details_pkey	10000	672 kB	240 kB
5	public	goal_details	btree_goal_details_matchno	10000	672 kB	240 kB

as you can see the hash index is more than double the size of the btree one.

Why hash and btree are on the top ?

Because they are used to answer the query match_no=$1, where $1 is the match number having max audience number found in the first seq scan in the plan, and they are good on those types of queries.

Schema 3:

Query 7:

1. Without indexes:

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.00..1109.54 rows=5 width=21) (actual time=2.160..12.997 rows=5 loops=1)
2	Join Filter: (s.sid = r.sid)
3	Rows Removed by Join Filter: 44985
4	-> Seq Scan on sailors s (cost=0.00..165.00 rows=9000 width=25) (actual time=0.029..1.206 rows=9000 loops=1)
5	-> Materialize (cost=0.00..269.55 rows=5 width=4) (actual time=0.000..0.001 rows=5 loops=9000)
6	-> Seq Scan on reserves r (cost=0.00..269.52 rows=5 width=4) (actual time=0.032..1.989 rows=5 loops=1)
7	Filter: (bid = 103)
8	Rows Removed by Filter: 14997
9	Planning time: 0.163 ms
10	Execution time: 13.042 ms

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=off;
set enable_indexscan=off;
set enable_bitmapscan=off;
```

2. Btree index on sailors s.sid, reserves r.bid:

	QUERY PLAN text
1	Nested Loop (cost=24.03..65.36 rows=5 width=21) (actual time=0.096..0.138 rows=5 loops=1)
2	-> HashAggregate (cost=23.75..23.80 rows=5 width=4) (actual time=0.059..0.061 rows=5 loops=1)
3	Group Key: r.sid
4	-> Index Scan using btreer on reserves r (cost=0.29..23.73 rows=5 width=4) (actual time=0.046.....
5	Index Cond: (bid = 103)
6	-> Index Scan using btrees on sailors s (cost=0.29..8.30 rows=1 width=25) (actual time=0.014..0.01...
7	Index Cond: (sid = r.sid)
8	Planning time: 3.263 ms
9	Execution time: 0.194 ms

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

3. Hash index on sailors s.sid and reserves r.bid:

1	Nested Loop (cost=24.10..64.29 rows=5 width=21) (actual time=0.030..0.038 rows=5 loops=1)
2	-> HashAggregate (cost=24.10..24.15 rows=5 width=4) (actual time=0.016..0.017 rows=5 loops=1)
3	Group Key: r.sid
4	-> Index Scan using hashr on reserves r (cost=0.00..24.09 rows=5 width=4) (actual time=0.009..0.0...
5	Index Cond: (bid = 103)
6	-> Index Scan using hashs on sailors s (cost=0.00..8.02 rows=1 width=25) (actual time=0.004..0.004 r...
7	Index Cond: (sid = r.sid)
8	Planning time: 2.029 ms
9	Execution time: 0.066 ms

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

4. Gin index on sailors s.sid and reserves r.bid:

1	Nested Loop (cost=37.22..92.61 rows=5 width=21) (actual time=0.055..0.100 rows=5 loops=1)
2	-> HashAggregate (cost=28.41..28.46 rows=5 width=4) (actual time=0.037..0.039 rows=5 loops=1)
3	Group Key: r.sid
4	-> Bitmap Heap Scan on reserves r (cost=12.04..28.40 rows=5 width=4) (actual time=0.026..0.031 ...
5	Recheck Cond: (bid = 103)
6	Heap Blocks: exact=5
7	-> Bitmap Index Scan on ginr (cost=0.00..12.04 rows=5 width=0) (actual time=0.019..0.019 row...
8	Index Cond: (bid = 103)
9	-> Bitmap Heap Scan on sailors s (cost=8.81..12.82 rows=1 width=25) (actual time=0.005..0.005 rows...
10	Recheck Cond: (sid = r.sid)
11	Heap Blocks: exact=5
12	-> Bitmap Index Scan on gins (cost=0.00..8.81 rows=1 width=0) (actual time=0.003..0.003 rows=1 l...
13	Index Cond: (sid = r.sid)
14	Planning time: 0.820 ms
15	Execution time: 0.179 ms

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=on;
```

5. All indexes:

1	Nested Loop (cost=20.41..60.60 rows=5 width=21) (actual time=0.032..0.043 rows=5 loops=1)
2	-> HashAggregate (cost=20.41..20.46 rows=5 width=4) (actual time=0.023..0.023 rows=5 loops=1)
3	Group Key: r.sid
4	-> Bitmap Heap Scan on reserves r (cost=4.04..20.40 rows=5 width=4) (actual time=0.014..0.020 r...
5	Recheck Cond: (bid = 103)
6	Heap Blocks: exact=5
7	-> Bitmap Index Scan on hashr (cost=0.00..4.04 rows=5 width=0) (actual time=0.009..0.009 row...
8	Index Cond: (bid = 103)
9	-> Index Scan using hashs on sailors s (cost=0.00..8.02 rows=1 width=25) (actual time=0.003..0.003 r...
10	Index Cond: (sid = r.sid)
11	Planning time: 3.206 ms
12	Execution time: 0.087 ms

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=on;
```

Explanation and justification: From the above screenshots we can infer the that the least efficient thing is to use no indexes (just the sequential scan) which very costly. The most efficient index for query 7 was the Hash based index. It resulted in the least Cost and Execution time 0.066ms and 64.29 respectively. The reason behind such results is that the Hash based index in databases works best whenever there is an equality in the query being executed. Here in query 7 there was the following equality:

Also, when running all the Hash based index which

```
where r.bid = 103 );
```

indexes together, the Database engine chose the supports the efficiency of this index for the query executed.

Query 8:

1. Without indexes:

1	Nested Loop Semi Join (cost=0.00..675491.27 rows=3750 width=21) (actual time=0.110..6708.726 rows=3750 loops=1)
2	Join Filter: (s.sid = r.sid)
3	Rows Removed by Join Filter: 26716875
4	-> Seq Scan on sailors s (cost=0.00..165.00 rows=9000 width=25) (actual time=0.033..2.450 rows=9000 loops=1)
5	-> Materialize (cost=0.00..169085.64 rows=3750 width=4) (actual time=0.000..0.426 rows=2969 loops=9000)
6	-> Nested Loop Semi Join (cost=0.00..169066.89 rows=3750 width=4) (actual time=0.061..2123.966 rows=3750 loops...
7	Join Filter: (r.bid = b.bid)
8	Rows Removed by Join Filter: 9843375
9	-> Seq Scan on reserves r (cost=0.00..232.02 rows=15002 width=8) (actual time=0.025..3.898 rows=15002 loops=1)
10	-> Materialize (cost=0.00..64.25 rows=750 width=4) (actual time=0.000..0.048 rows=656 loops=15002)
11	-> Seq Scan on boat b (cost=0.00..60.50 rows=750 width=4) (actual time=0.023..1.262 rows=750 loops=1)
12	Filter: ((color)::text = 'red'::text)
13	Rows Removed by Filter: 2250
14	Planning time: 0.298 ms
15	Execution time: 6709.263 ms

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=off;
set enable_indexscan=off;
set enable_bitmapscan=off;
```

2. Btree index on sailors s.sid, reserves r.bid, & boat b.color:

1	Nested Loop (cost=948.17..2327.26 rows=3750 width=21) (actual time=7.144..17.346 rows=3750 loops=1)
2	-> HashAggregate (cost=947.88..985.38 rows=3750 width=4) (actual time=7.064..8.028 rows=3750 loops=1)
3	Group Key: r.sid
4	-> Nested Loop (cost=114.42..938.51 rows=3750 width=4) (actual time=0.609..5.778 rows=3750 loops=1)
5	-> HashAggregate (cost=114.13..121.63 rows=750 width=4) (actual time=0.566..0.774 rows=750 loops=1)
6	Group Key: b.bid
7	-> Index Scan using btreeb on boat b (cost=0.28..112.26 rows=750 width=4) (actual time=0.079..0.326 rows=750 loops=1)
8	Index Cond: ((color)::text = 'red'::text)
9	-> Index Scan using btreer on reserves r (cost=0.29..1.04 rows=5 width=8) (actual time=0.003..0.006 rows=5 loops=750)
10	Index Cond: (bid = b.bid)
11	-> Index Scan using btrees on sailors s (cost=0.29..0.35 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=3750)
12	Index Cond: (sid = r.sid)
13	Planning time: 5.849 ms
14	Execution time: 17.901 ms

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

3. Hash index on sailors s.sid, reserves r.bid, & boat b.color:

	text
1	Nested Loop (cost=843.00..1165.29 rows=3750 width=21) (actual time=6.850..15.607 rows=3750 loops=1)
2	-> HashAggregate (cost=843.00..880.50 rows=3750 width=4) (actual time=6.836..7.770 rows=3750 loops=1)
3	Group Key: r.sid
4	-> Nested Loop (cost=131.00..833.63 rows=3750 width=4) (actual time=0.628..5.503 rows=3750 loops=1)
5	-> HashAggregate (cost=131.00..138.50 rows=750 width=4) (actual time=0.616..0.800 rows=750 loops=1)
6	Group Key: b.bid
7	-> Index Scan using hashb on boat b (cost=0.00..129.13 rows=750 width=4) (actual time=0.017..0.393 rows=750 loops=1)
8	Index Cond: ((color)::text = 'red'::text)
9	-> Index Scan using hashr on reserves r (cost=0.00..0.88 rows=5 width=8) (actual time=0.002..0.005 rows=5 loops=750)
10	Index Cond: (bid = b.bid)
11	-> Index Scan using hashs on sailors s (cost=0.00..0.07 rows=1 width=25) (actual time=0.001..0.002 rows=1 loops=3750)
12	Index Cond: (sid = r.sid)
13	Planning time: 6.156 ms
14	Execution time: 16.226 ms

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

4. Gin index sailors s.sid, reserves r.bid, & boat b.color:

	QUERY PLAN text
1	Nested Loop (cost=3310.41..18566.32 rows=3750 width=21) (actual time=20.288..74.363 rows=3750 loops=1)
2	-> HashAggregate (cost=3310.38..3347.88 rows=3750 width=4) (actual time=20.262..21.679 rows=3750 loops=1)
3	Group Key: r.sid
4	-> Nested Loop (cost=48.28..3301.00 rows=3750 width=4) (actual time=0.622..18.430 rows=3750 loops=1)
5	-> HashAggregate (cost=48.06..55.56 rows=750 width=4) (actual time=0.587..0.921 rows=750 loops=1)
6	Group Key: b.bid
7	-> Bitmap Heap Scan on boat b (cost=13.81..46.19 rows=750 width=4) (actual time=0.140..0.307 rows=750 loops...
8	Recheck Cond: ((color)::text = 'red'::text)
9	Heap Blocks: exact=23
10	-> Bitmap Index Scan on ginb (cost=0.00..13.63 rows=750 width=0) (actual time=0.124..0.124 rows=750 loops...
11	Index Cond: ((color)::text = 'red'::text)
12	-> Bitmap Heap Scan on reserves r (cost=0.21..4.28 rows=5 width=8) (actual time=0.008..0.012 rows=5 loops=750)
13	Recheck Cond: (bid = b.bid)
14	Heap Blocks: exact=3750
15	-> Bitmap Index Scan on ginr (cost=0.00..0.21 rows=5 width=0) (actual time=0.005..0.005 rows=5 loops=750)
16	Index Cond: (bid = b.bid)
17	-> Bitmap Heap Scan on sailors s (cost=0.04..4.05 rows=1 width=25) (actual time=0.004..0.004 rows=1 loops=3750)
18	Recheck Cond: (sid = r.sid)
19	Heap Blocks: exact=3750
20	-> Bitmap Index Scan on gins (cost=0.00..0.04 rows=1 width=0) (actual time=0.003..0.003 rows=1 loops=3750)
21	Index Cond: (sid = r.sid)
22	Planning time: 5.769 ms
23	Execution time: 75.373 ms

5. All indexes:

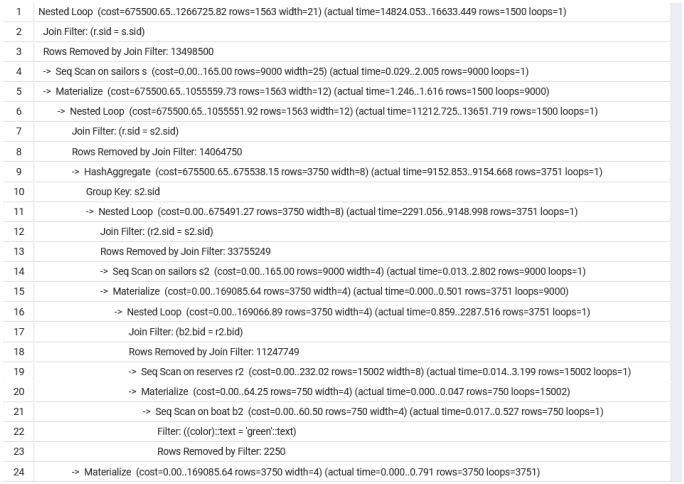
	QUERY PLAN text
1	Nested Loop (cost=760.06..1082.35 rows=3750 width=21) (actual time=6.628..14.823 rows=3750 loops=1)
2	-> HashAggregate (cost=760.06..797.56 rows=3750 width=4) (actual time=6.616..7.506 rows=3750 loops=1)
3	Group Key: r.sid
4	-> Nested Loop (cost=48.06..750.69 rows=3750 width=4) (actual time=0.523..5.113 rows=3750 loops=1)
5	-> HashAggregate (cost=48.06..55.56 rows=750 width=4) (actual time=0.508..0.696 rows=750 loops=1)
6	Group Key: b.bid
7	-> Bitmap Heap Scan on boat b (cost=13.81..46.19 rows=750 width=4) (actual time=0.118..0.273 rows=750 loops=1)
8	Recheck Cond: ((color)::text = 'red'::text)
9	Heap Blocks: exact=23
10	-> Bitmap Index Scan on ginb (cost=0.00..13.63 rows=750 width=0) (actual time=0.101..0.102 rows=750 loops=1)
11	Index Cond: ((color)::text = 'red'::text)
12	-> Index Scan using hashr on reserves r (cost=0.00..0.88 rows=5 width=8) (actual time=0.001..0.005 rows=5 loops=750)
13	Index Cond: (bid = b.bid)
14	-> Index Scan using hashs on sailors s (cost=0.00..0.07 rows=1 width=25) (actual time=0.001..0.002 rows=1 loops=3750)
15	Index Cond: (sid = r.sid)
16	Planning time: 8.859 ms
17	Execution time: 15.556 ms

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=on;
```

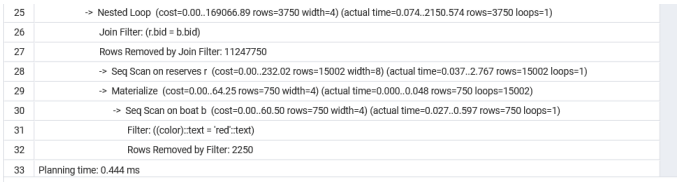
Explanation and justification: In the above query the most efficient scenario was the hash based index which was used on the s.sid, r.bid, and b.color. The hash-based index's average Cost and Execution time were 1165.29 and 16.226ms respectively. As mentioned previously, whenever a query has an equality in any of its where statements, the hash-based index works most efficiently in executing it. This query had the following equality: `where b.color = 'red'));` The scenario that was least efficient with this query specially because it was a nested one, was the sequential scan one. It took the engine 6 secs on average, which is 300x slower than the hash and the btree index.

Query 9:

1. Without indexes:



```
1  Nested Loop  (cost=675500.65..1266725.82 rows=1563 width=21) (actual time=14824.053..16633.449 rows=1500 loops=1)
2    Join Filter: (r.sid = s.sid)
3    Rows Removed by Join Filter: 13498500
4    ->  Seq Scan on sailors s  (cost=0.00..165.00 rows=9000 width=25) (actual time=0.029..2.005 rows=9000 loops=1)
5    ->  Materialize  (cost=675500.65..1055559.73 rows=1563 width=12) (actual time=1.246..1.616 rows=1500 loops=9000)
6          ->  Nested Loop  (cost=675500.65..1055551.92 rows=1563 width=12) (actual time=11212.725..13651.719 rows=1500 loops=1)
7                Join Filter: (r.sid = s2.sid)
8                Rows Removed by Join Filter: 14064750
9                ->  HashAggregate  (cost=675500.65..675538.15 rows=3750 width=8) (actual time=9152.853..9154.668 rows=3751 loops=1)
10                     Group Key: s2.sid
11                     ->  Nested Loop  (cost=0.00..675491.27 rows=3750 width=8) (actual time=2291.056..9148.998 rows=3751 loops=1)
12                           Join Filter: (r2.sid = s2.sid)
13                           Rows Removed by Join Filter: 33755249
14                           ->  Seq Scan on sailors s2  (cost=0.00..165.00 rows=9000 width=4) (actual time=0.013..2.802 rows=9000 loops=1)
15                           ->  Materialize  (cost=0.00..169085.64 rows=3750 width=4) (actual time=0.000..0.501 rows=3751 loops=9000)
16                                 ->  Nested Loop  (cost=0.00..169066.89 rows=3750 width=4) (actual time=0.859..2287.516 rows=3751 loops=1)
17                                       Join Filter: (b2.bid = r2.bid)
18                                       Rows Removed by Join Filter: 11247749
19                                       ->  Seq Scan on reserves r2  (cost=0.00..232.02 rows=15002 width=8) (actual time=0.014..3.199 rows=15002 loops=1)
20                                       ->  Materialize  (cost=0.00..64.25 rows=750 width=4) (actual time=0.000..0.047 rows=750 loops=15002)
21                                             ->  Seq Scan on boat b2  (cost=0.00..60.50 rows=750 width=4) (actual time=0.017..0.527 rows=750 loops=1)
22                                                   Filter: ((color)::text = 'green'::text)
23                                                   Rows Removed by Filter: 2250
24         ->  Materialize  (cost=0.00..169085.64 rows=3750 width=4) (actual time=0.000..0.791 rows=3750 loops=3751)
```



```
25               ->  Nested Loop  (cost=0.00..169066.89 rows=3750 width=4) (actual time=0.074..2150.574 rows=3750 loops=1)
26                     Join Filter: (r.bid = b.bid)
27                     Rows Removed by Join Filter: 11247750
28                     ->  Seq Scan on reserves r  (cost=0.00..232.02 rows=15002 width=8) (actual time=0.037..2.767 rows=15002 loops=1)
29                     ->  Materialize  (cost=0.00..64.25 rows=750 width=4) (actual time=0.000..0.048 rows=750 loops=15002)
30                           ->  Seq Scan on boat b  (cost=0.00..60.50 rows=750 width=4) (actual time=0.027..0.597 rows=750 loops=1)
31                                 Filter: ((color)::text = 'red'::text)
32                                 Rows Removed by Filter: 2250
33   Planning time: 0.444 ms
```

Apologizes for the none displayed execution time due to technical problems. It was 15 secs & 235 ms

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=off;
set enable_indexscan=off;
set enable_bitmapscan=off;
```

2. Btree index on sailors s.sid, reserves r.bid, r.sid, & boat b.color, b.bid:

```
Nested Loop  (cost=1.70..6160.10 rows=1563 width=21) (actual time=0.555..61.608 rows=1500 loops=1)
 -> Nested Loop  (cost=0.57..929.13 rows=3750 width=4) (actual time=0.156..6.689 rows=3750 loops=1)
    -> Index Scan using btreeb on boat b  (cost=0.28..112.26 rows=750 width=4) (actual time=0.094..0.435 rows=750 loops=1)
       Index Cond: ((color)::text = 'red'::text)
    -> Index Scan using btreer on reserves r  (cost=0.29..1.04 rows=5 width=8) (actual time=0.004..0.007 rows=5 loops=750)
       Index Cond: (bid = b.bid)
 -> Nested Loop Semi Join  (cost=1.14..1.38 rows=1 width=33) (actual time=0.014..0.014 rows=0 loops=3750)
    -> Index Scan using btrees on sailors s  (cost=0.29..0.33 rows=1 width=25) (actual time=0.002..0.003 rows=1 loops=3750)
       Index Cond: (sid = r.sid)
    -> Nested Loop  (cost=0.85..1.05 rows=1 width=8) (actual time=0.011..0.011 rows=0 loops=3750)
       -> Nested Loop  (cost=0.57..0.73 rows=1 width=12) (actual time=0.005..0.006 rows=2 loops=3750)
          Join Filter: (s2.sid = r2.sid)
          -> Index Only Scan using btrees on sailors s2  (cost=0.29..0.35 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=3750)
             Index Cond: (sid = s.sid)
             Heap Fetches: 3750
          -> Index Scan using btreer2 on reserves r2  (cost=0.29..0.35 rows=2 width=8) (actual time=0.002..0.003 rows=2 loops=3750)
             Index Cond: (sid = r.sid)
       -> Index Scan using btreeb2 on boat b2  (cost=0.28..0.31 rows=1 width=4) (actual time=0.002..0.002 rows=0 loops=6001)
          Index Cond: (bid = r2.bid)
          Filter: ((color)::text = 'green'::text)
          Rows Removed by Filter: 1
Planning time: 9.274 ms
Execution time: 61.931 ms
```

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

3. Hash index on sailors s.sid, reserves r.bid, r.sid, & boat b.color, b.bid:

```
Nested Loop  (cost=0.00..1847.63 rows=1563 width=21) (actual time=0.087..53.385 rows=1500 loops=1)
 -> Nested Loop  (cost=0.00..824.25 rows=3750 width=4) (actual time=0.036..6.734 rows=3750 loops=1)
    -> Index Scan using hashb on boat b  (cost=0.00..129.13 rows=750 width=4) (actual time=0.024..0.548 rows=750 loops=1)
       Index Cond: ((color)::text = 'red'::text)
    -> Index Scan using hashr on reserves r  (cost=0.00..0.88 rows=5 width=8) (actual time=0.002..0.007 rows=5 loops=750)
       Index Cond: (bid = b.bid)
 -> Nested Loop Semi Join  (cost=0.00..0.26 rows=1 width=33) (actual time=0.012..0.012 rows=0 loops=3750)
    -> Index Scan using hashs on sailors s  (cost=0.00..0.05 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=3750)
       Index Cond: (sid = r.sid)
    -> Nested Loop  (cost=0.00..0.21 rows=1 width=8) (actual time=0.009..0.009 rows=0 loops=3750)
       -> Nested Loop  (cost=0.00..0.17 rows=1 width=12) (actual time=0.004..0.005 rows=2 loops=3750)
          Join Filter: (s2.sid = r2.sid)
          -> Index Scan using hashs on sailors s2  (cost=0.00..0.07 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=3750)
             Index Cond: (sid = s.sid)
          -> Index Scan using hashr2 on reserves r2  (cost=0.00..0.07 rows=2 width=8) (actual time=0.002..0.002 rows=2 loops=3750)
             Index Cond: (sid = r.sid)
       -> Index Scan using hashb2 on boat b2  (cost=0.00..0.03 rows=1 width=4) (actual time=0.002..0.002 rows=0 loops=6002)
          Index Cond: (bid = r2.bid)
          Filter: ((color)::text = 'green'::text)
          Rows Removed by Filter: 1
Planning time: 0.737 ms
Execution time: 53.569 ms
```

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=off;
```

4. Gin index on sailors s.sid, reserves r.bid, r.sid, & boat b.color, b.bid:

```
1  Nested Loop  (cost=14.13..64142.55 rows=1563 width=21) (actual time=0.267..235.614 rows=1500 loops=1)
2    ->  Nested Loop  (cost=14.03..3291.63 rows=3750 width=4) (actual time=0.138..15.061 rows=3750 loops=1)
3          ->  Bitmap Heap Scan on boat b  (cost=13.81..46.19 rows=750 width=4) (actual time=0.110..0.289 rows=750 loops=1)
4                Recheck Cond: ((color)::text = 'red'::text)
5                Heap Blocks: exact=23
6                ->  Bitmap Index Scan on ginb  (cost=0.00..13.63 rows=750 width=0) (actual time=0.098..0.098 rows=750 loops=1)
7                      Index Cond: ((color)::text = 'red'::text)
8          ->  Bitmap Heap Scan on reserves r  (cost=0.21..4.28 rows=5 width=8) (actual time=0.007..0.010 rows=5 loops=750)
9                Recheck Cond: (bid = b.bid)
10               Heap Blocks: exact=3750
11               ->  Bitmap Index Scan on ginr  (cost=0.00..0.21 rows=5 width=0) (actual time=0.005..0.005 rows=5 loops=750)
12                     Index Cond: (bid = b.bid)
13   ->  Nested Loop Semi Join  (cost=0.11..16.22 rows=1 width=33) (actual time=0.058..0.058 rows=0 loops=3750)
14         ->  Bitmap Heap Scan on sailors s  (cost=0.02..4.04 rows=1 width=25) (actual time=0.004..0.004 rows=1 loops=3750)
15               Recheck Cond: (sid = r.sid)
16               Heap Blocks: exact=3750
17               ->  Bitmap Index Scan on gins  (cost=0.00..0.02 rows=1 width=0) (actual time=0.003..0.003 rows=1 loops=3750)
18                     Index Cond: (sid = r.sid)
19         ->  Nested Loop  (cost=0.08..12.17 rows=1 width=8) (actual time=0.046..0.046 rows=0 loops=3750)
20               ->  Nested Loop  (cost=0.07..8.13 rows=1 width=12) (actual time=0.025..0.026 rows=2 loops=3750)
21                     Join Filter: (s2.sid = r2.sid)
22                     ->  Bitmap Heap Scan on sailors s2  (cost=0.04..4.05 rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=3750)
23                           Recheck Cond: (sid = s.sid)
24                           Heap Blocks: exact=3750
25                           ->  Bitmap Index Scan on gins  (cost=0.00..0.04 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=3750)
26                                 Index Cond: (sid = s.sid)
27                     ->  Bitmap Heap Scan on reserves r2  (cost=0.03..4.06 rows=2 width=8) (actual time=0.005..0.006 rows=2 loops=3750)
28                           Recheck Cond: (sid = r.sid)
29                           Heap Blocks: exact=6000
30                           ->  Bitmap Index Scan on ginr2  (cost=0.00..0.03 rows=2 width=0) (actual time=0.004..0.004 rows=2 loops=3750)
31                                 Index Cond: (sid = r.sid)
32               ->  Bitmap Heap Scan on boat b2  (cost=0.01..4.03 rows=1 width=4) (actual time=0.004..0.004 rows=0 loops=6001)
33                     Recheck Cond: (bid = r2.bid)
34                     Filter: ((color)::text = 'green'::text)
35                     Rows Removed by Filter: 1
36                     Heap Blocks: exact=6001
37                     ->  Bitmap Index Scan on ginb2  (cost=0.00..0.01 rows=1 width=0) (actual time=0.002..0.002 rows=1 loops=6001)
38                           Index Cond: (bid = r2.bid)
39 Planning time: 5.906 ms
40 Execution time: 235.986 ms
```

```
set enable_seqscan=off;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=on;
```

5. All 3 indexes:

```
1  Nested Loop  (cost=13.81..1764.69 rows=1563 width=21) (actual time=0.244..48.535 rows=1500 loops=1)
2    -> Nested Loop  (cost=13.81..741.31 rows=3750 width=4) (actual time=0.215..6.144 rows=3750 loops=1)
3      -> Bitmap Heap Scan on boat b  (cost=13.81..46.19 rows=750 width=4) (actual time=0.201..0.416 rows=750 loops=1)
4        Recheck Cond: ((color)::text = 'red'::text)
5        Heap Blocks: exact=23
6        -> Bitmap Index Scan on ginb  (cost=0.00..13.63 rows=750 width=0) (actual time=0.189..0.189 rows=750 loops=1)
7          Index Cond: ((color)::text = 'red'::text)
8      -> Index Scan using hashr on reserves r  (cost=0.00..0.88 rows=5 width=8) (actual time=0.002..0.006 rows=5 loops=750)
9        Index Cond: (bid = b.bid)
10   -> Nested Loop Semi Join  (cost=0.00..0.26 rows=1 width=33) (actual time=0.011..0.011 rows=0 loops=3750)
11     -> Index Scan using hashs on sailors s  (cost=0.00..0.05 rows=1 width=25) (actual time=0.002..0.002 rows=1 loops=3750)
12       Index Cond: (sid = r.sid)
13     -> Nested Loop  (cost=0.00..0.21 rows=1 width=8) (actual time=0.008..0.008 rows=0 loops=3750)
14       -> Nested Loop  (cost=0.00..0.17 rows=1 width=12) (actual time=0.004..0.005 rows=2 loops=3750)
15         Join Filter: (s2.sid = r2.sid)
16         -> Index Scan using hashs on sailors s2  (cost=0.00..0.07 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=3750)
17           Index Cond: (sid = s.sid)
18         -> Index Scan using hashr2 on reserves r2  (cost=0.00..0.07 rows=2 width=8) (actual time=0.001..0.002 rows=2 loops=3750)
19           Index Cond: (sid = r.sid)
20       -> Index Scan using hashb2 on boat b2  (cost=0.00..0.03 rows=1 width=4) (actual time=0.002..0.002 rows=0 loops=6002)
21         Index Cond: (bid = r2.bid)
22       Filter: ((color)::text = 'green'::text)
23       Rows Removed by Filter: 1
24 Planning time: 1.621 ms
```

```
set enable_seqscan=on;
set enable_hashjoin=off;
set enable_nestloop=on;
set enable_mergejoin=off;
set enable_indexonlyscan=on;
set enable_indexscan=on;
set enable_bitmapscan=on;
```

In this query there was a typo, in the last where statement the equality was on ‘red’ instead of ‘green’, so that’s the only thing that was changed in the sql query.

Explanation and Justification: The last executed query we had the indexes made on the following columns: sailors s.sid. reserves r.bid, r.sid, boat b.color, b.bid. The above screenshots also show that the most efficient index type was the hash-bashed one, with Cost and Execution time 1847.63 and 53.569ms respectively. Again, the reason behind this is that this query had equalities and the hash-based index is an index that is mostly efficient when there are qualities due to the hash function which makes the query much faster. The two equalities that were in this query were: `b.color = 'red'` & `b2.color = 'red');` The least efficient scenario was the sequential scan which took almost 15 secs to display the output.

When all three scenarios were executed, the engine’s most used index was the hash-based one which seconds the above justification.