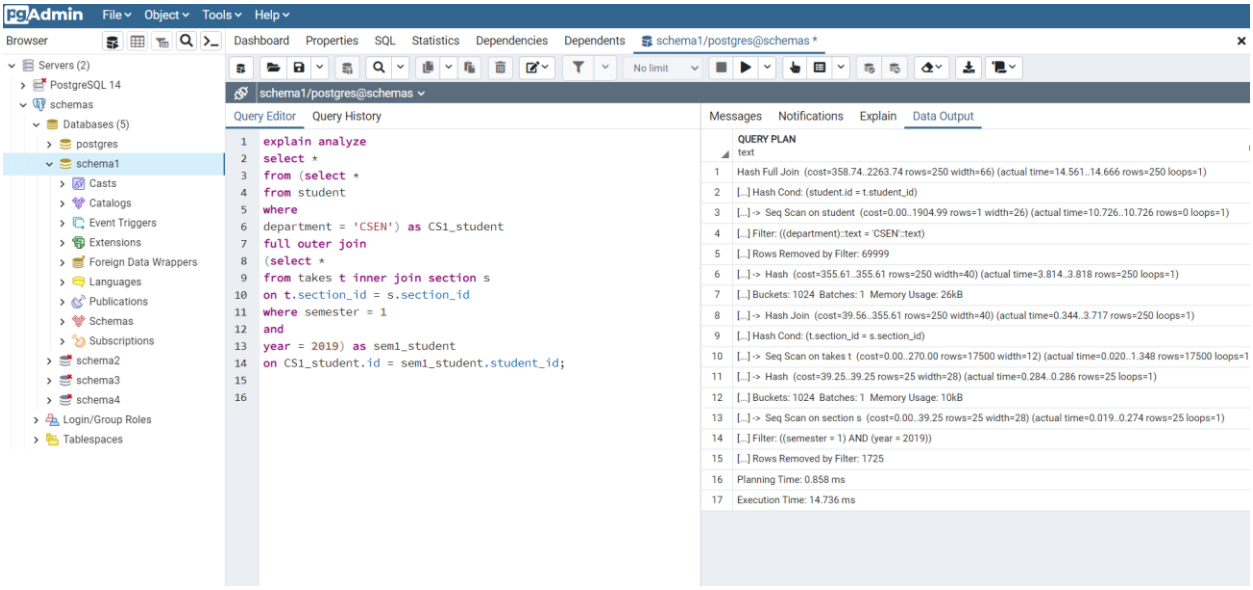
Q1: original

=>Without indices (only seqscans)

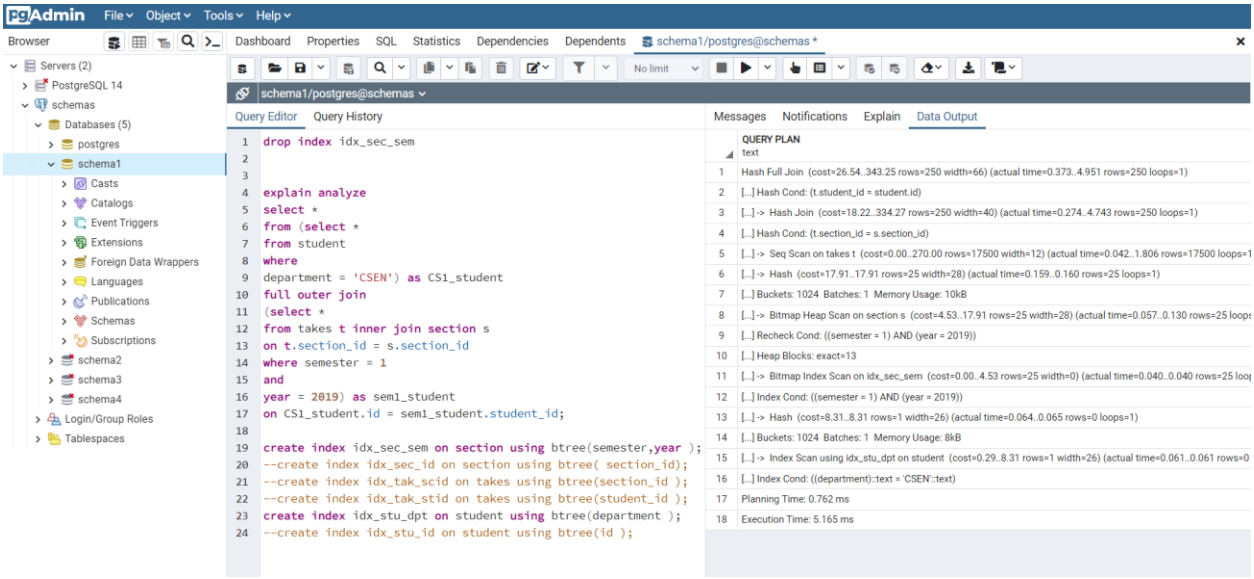


Cost=2263, time=14.7ms

Notes:

All relation access are seqscans only, no indeces created

⇨ Using Btrees: on section(semester,year), student(department)



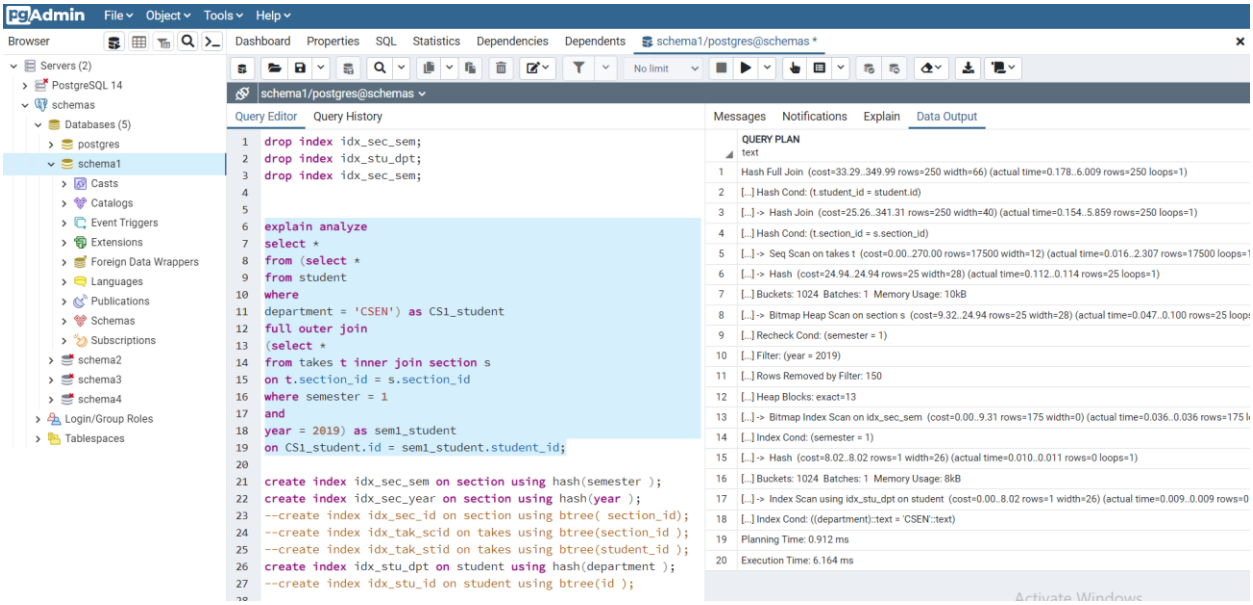
Cost=343.25, time=5.16ms

Notes:

The btree indices reduced the cost to 15.16% of original cost, time to 35% of original time.

The indeces made the search for semester=1,year=2019,department='CSEN' much faster, so instead of doing seqscan, we do index scan.

⇨ Using Hash: on section(semester),section(year), student(department)

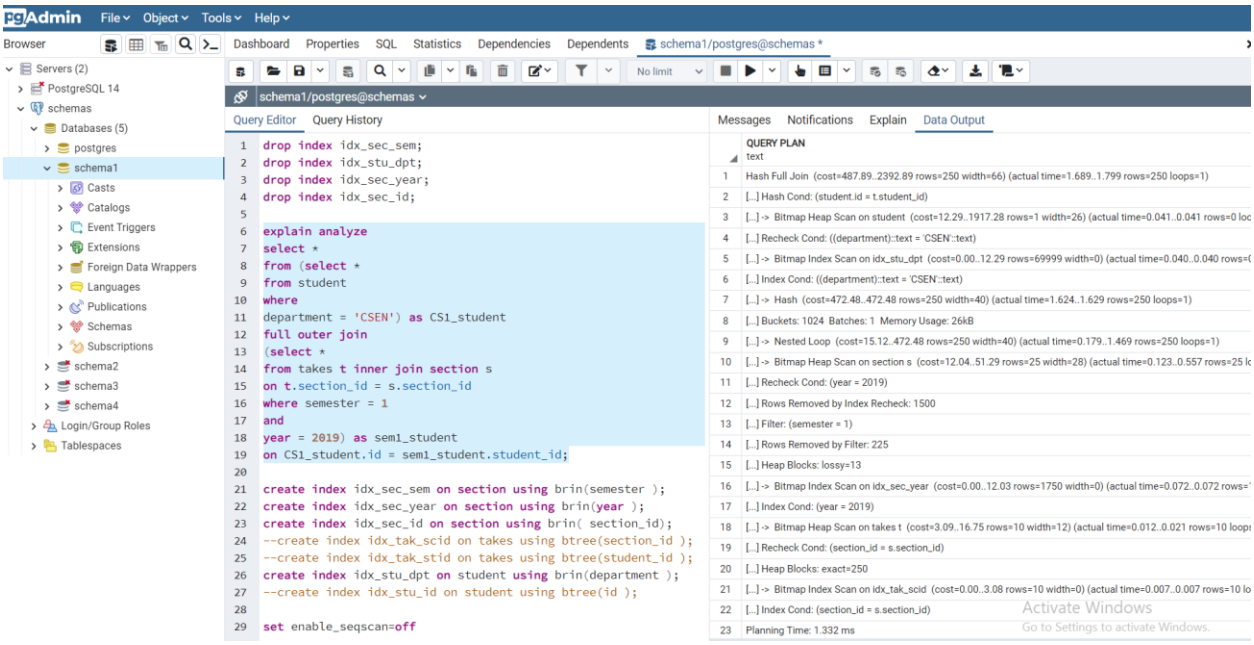


Cost=349.99, time=6.1ms

Notes:

The hash is usually alittle faster than btrees, but here it's not the case, because btree can have multicolumns, so it can check both semester=1 and check year =2019, but hash has to chose one and filter the other which takes some more time.

⇨ Using BRIN: on section(year), student(department), takes(section_id)

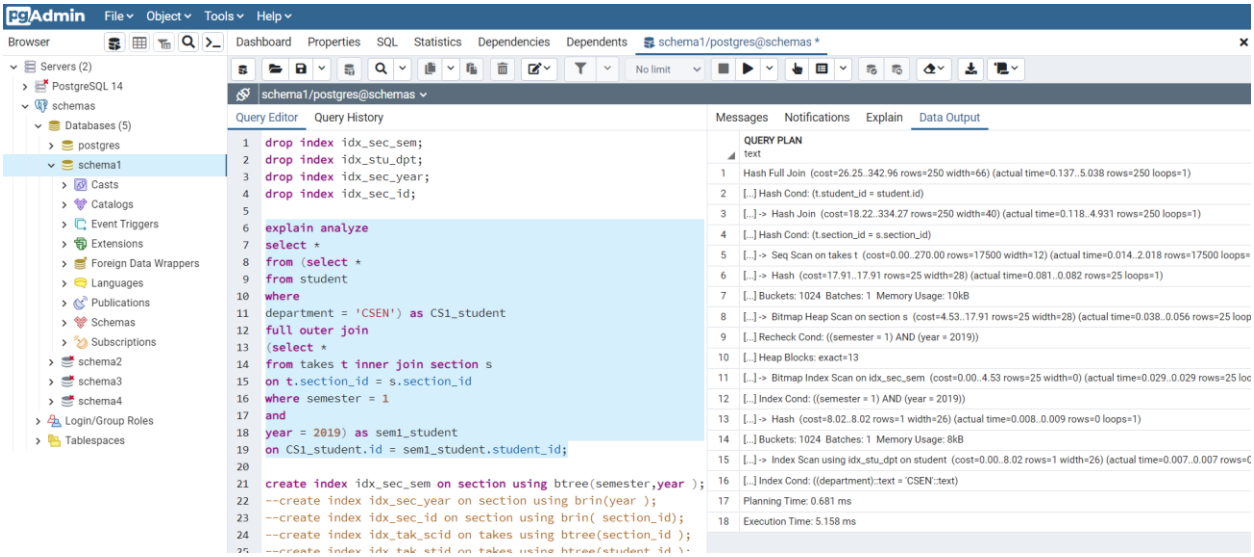


Cost=2392, time=2ms

Notes:

Since the brin is not good in exact values, the engine wasn't willing to use any of those indeces, but we turned seqscan off so it used the indeces mention above the screenshot, it increased the cost. The reason why brin is not good at this type of queries in explained in another query analysis.

⇨ Using mixed indeces: btree on section(semester,year), hash on student(department)



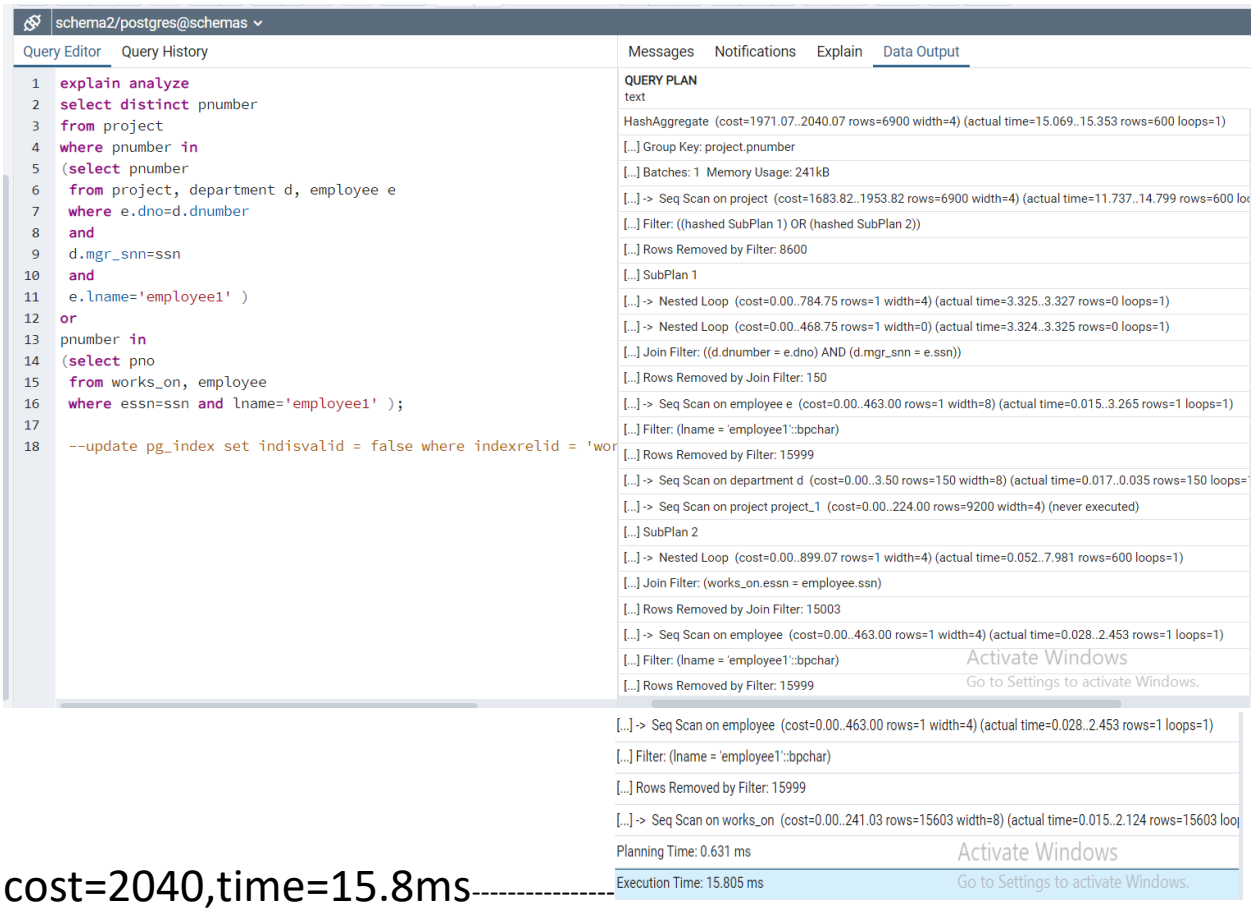
Cost=342.96, time=5.15ms

Notes:

In this mixed types, we make use of ability to have multicolumn of btree to check both year and semester, hash is alittle bit faster so we used it to search about department (single column) so the cost decreased a little compared to the btree section above as the difference in performance between btree and hash index

Q2:Original

=>Without indices (only seqscans)



[...] -> Seq Scan on employee (cost=0.00..463.00 rows=1 width=4) (actual time=0.028..2.453 rows=1 loops=1)
[...] Filter: (lname = 'employee1'::bpchar)
[...] Rows Removed by Filter: 15999
[...] -> Seq Scan on works_on (cost=0.00..241.03 rows=15603 width=8) (actual time=0.015..2.124 rows=15603 loo
Planning Time: 0.631 ms
Execution Time: 15.805 ms
Activate Windows
Go to Settings to activate Windows.

cost=2040,time=15.8ms----------

Flags:

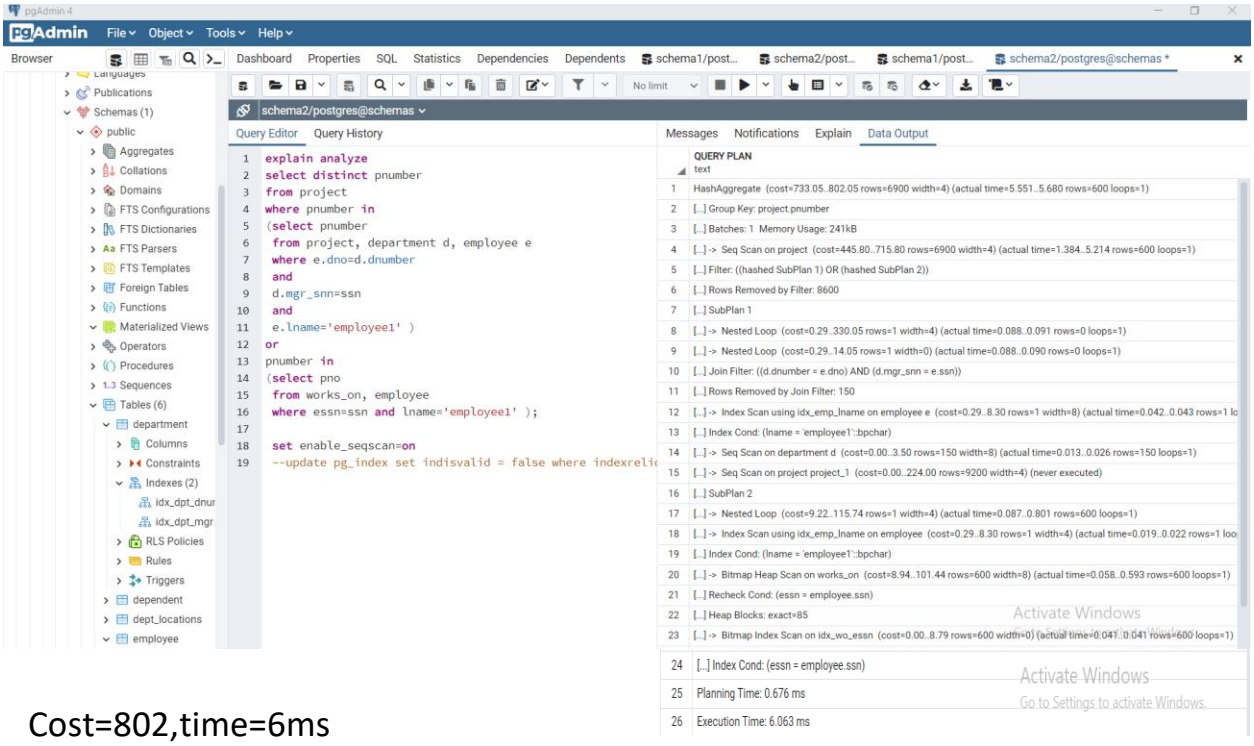
Turn off all index scan flags and turn on seqscan only

Notes:

All scans over all tables are done using seqscan

=>Using B+Trees:

-Btrees indices are used on employee(lname), works_on(essn)



Cost=802,time=6ms

Notes:

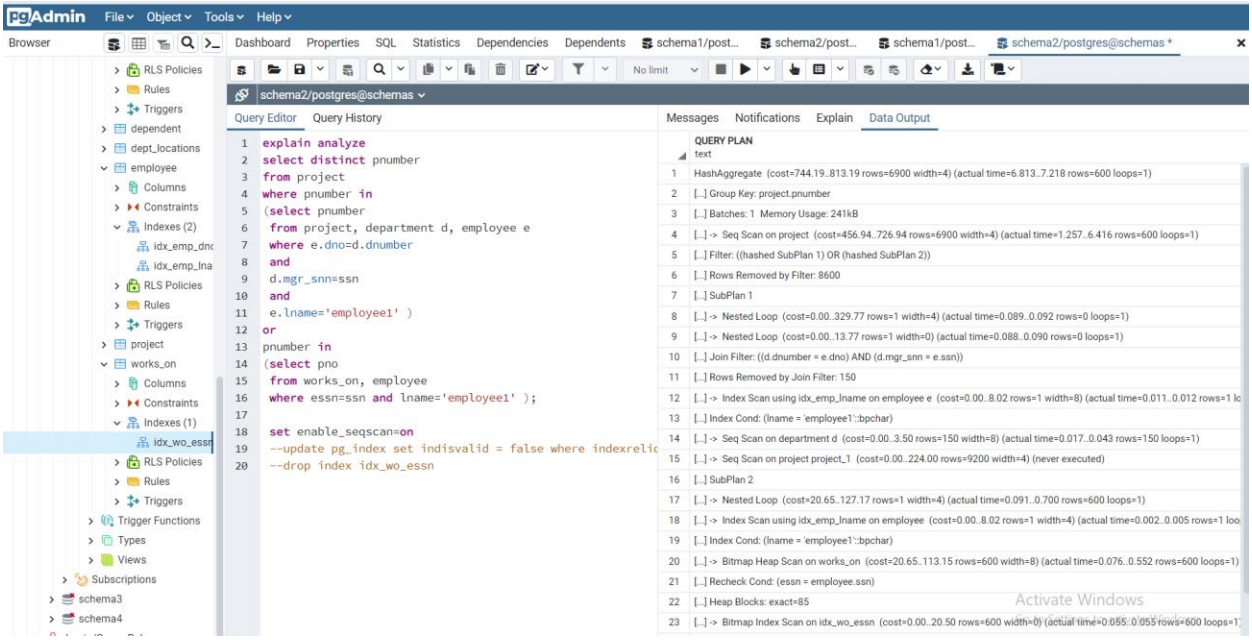
Using Btree indices decreased the cost to 39.3% of initial cost (W/O indices),

Time to 40% of initial time(W/O indices)

Difference in cost is refered to the difference in performance between seqscan to search for lname=’employee1’ , search for matching essn from works_on, and the index scan over those relations

=>Using Hash indeces:

-Hash indices are used on employee(lname), works_on(essn)



Cost=813,time=7ms

Notes:

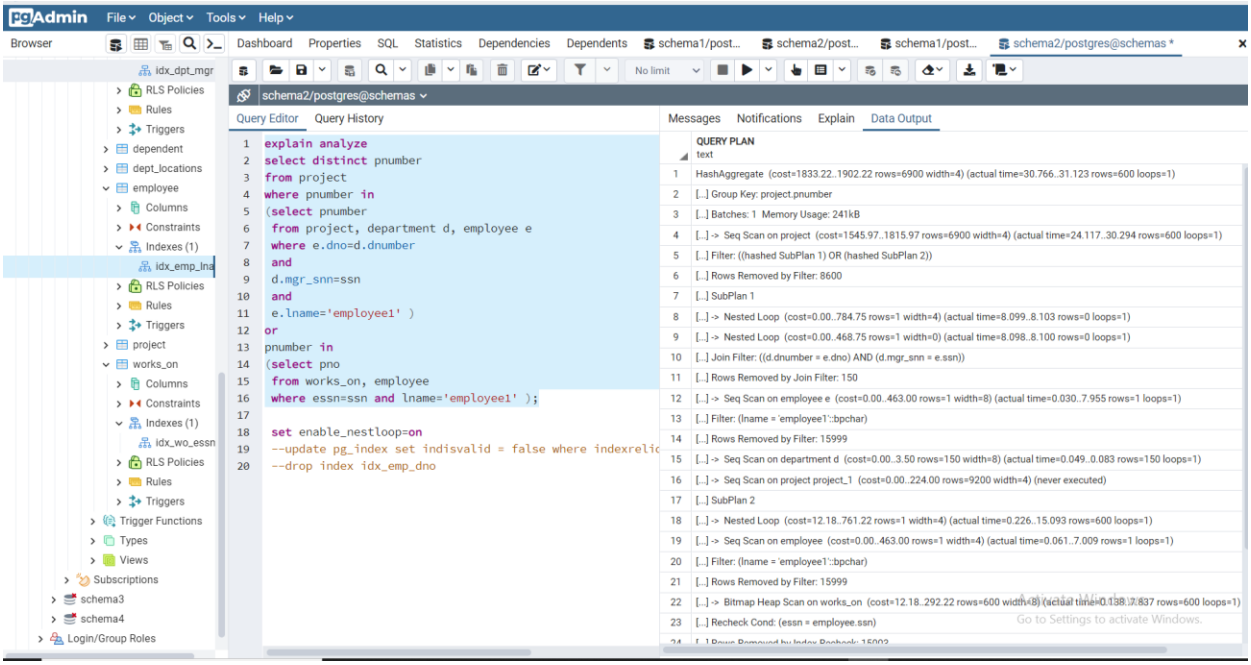
Using Hash indices decreased the cost to 39.8% of initial cost (W/O indices),

Time to 50% of initial time(W/O indices)

Difference in cost is refered to the difference in performance between seqscan to search for lname=’employee1’ , search for matching essn from works_on, and the index scan over those relations

=>Using BRIN indeces:

-BRIN indices are used on works_on(essn)



Cost=1902,time=31ms

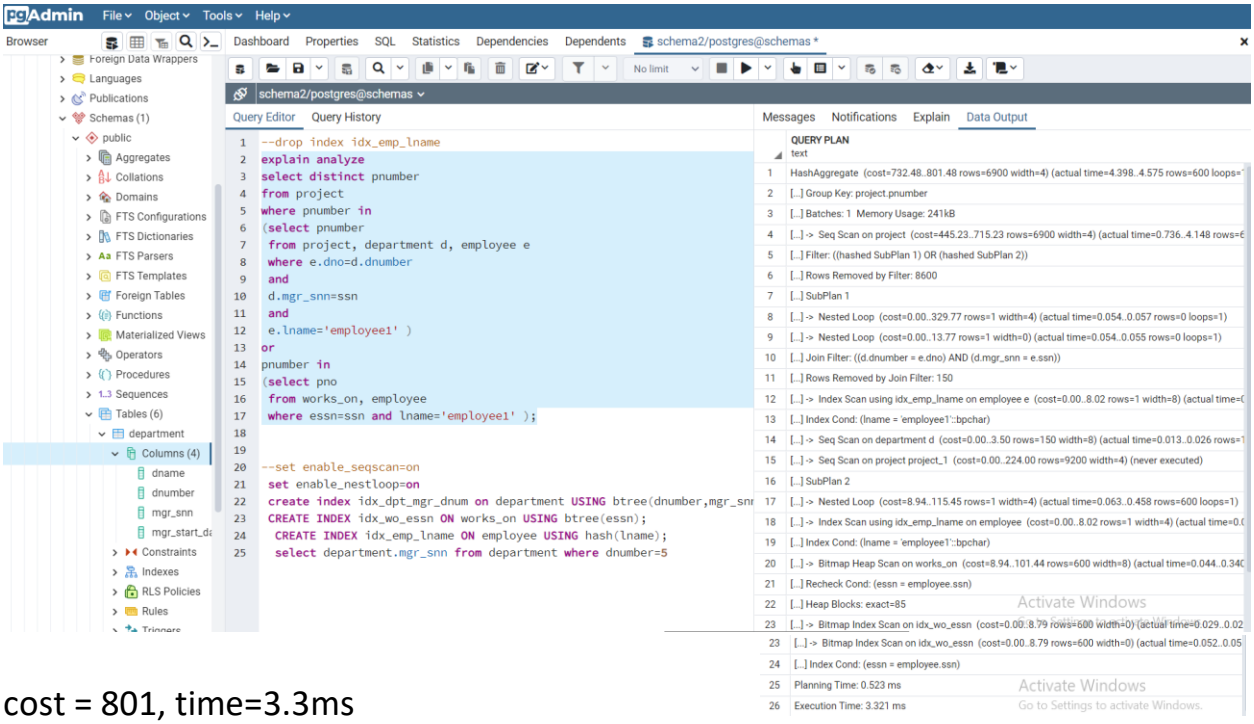
Notes:

Using BRIN indices decreased the cost to 93% of initial cost (W/O indices),

Time to 200% of initial time(W/O indices)

The reason of this bad cost and time reduction is that brin uses several sparse levels to reach data, but still better that seqscans only specially when working on works_on(essn)as it helps the join to find the match to each tuple of employee

=>Using Mixed indices: hash on employee(lname), btree on works_on(essn)



cost = 801, time=3.3ms

Notes:

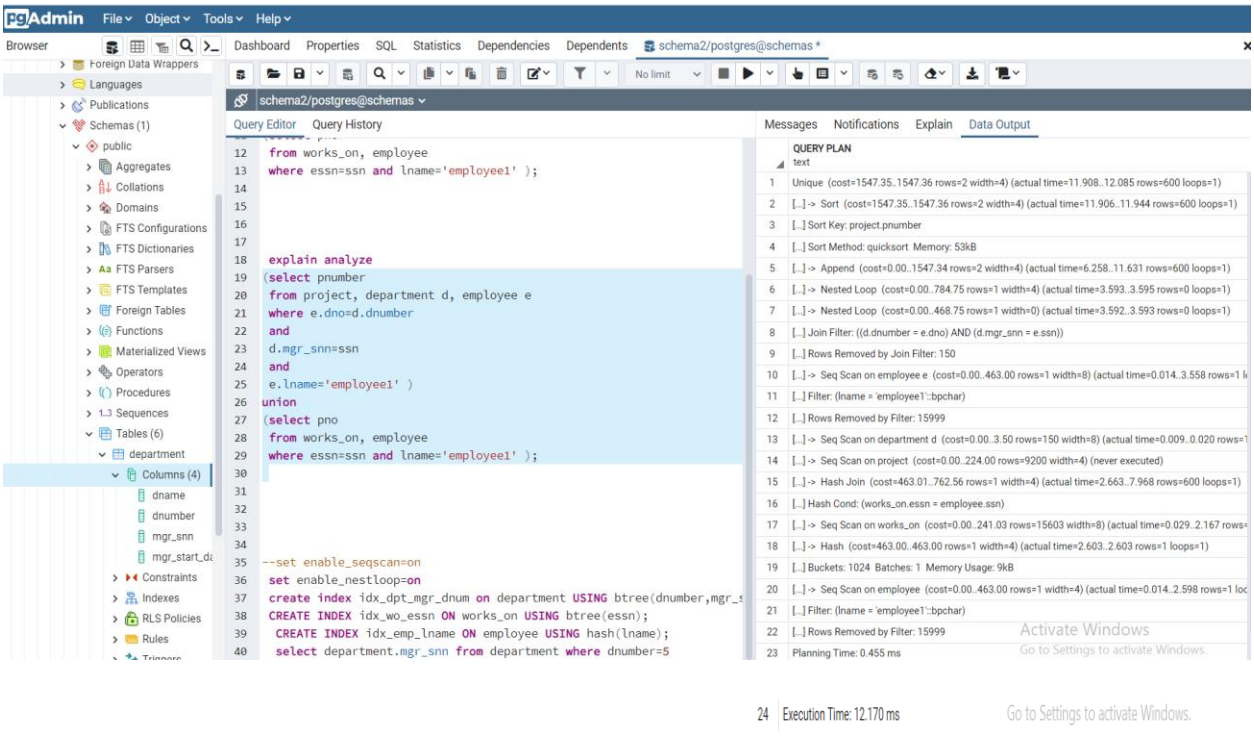
The mixed indices is the optimal solution as it gives the lowest cost among all previous, it's not far away from btree indices, but it gives less cost anyway.

Mixed indices reduces the cost to be 39.2% of the one w/o indices and time to 20.88% of initial time.

Both plans of mixed and btree indices runs the same, as we get the filtered version of employee relation where lname='employee1' and try to match it whith the relation department. The other way around is not accepted as it takes the whole relation of department and search for each tuple in employee then check that the lname ='employee1', so the first does much fewer searches, the difference between the mixed indices version and the btree version is that the hash

Q2: Optimized

=>Without indices (seqscans only):



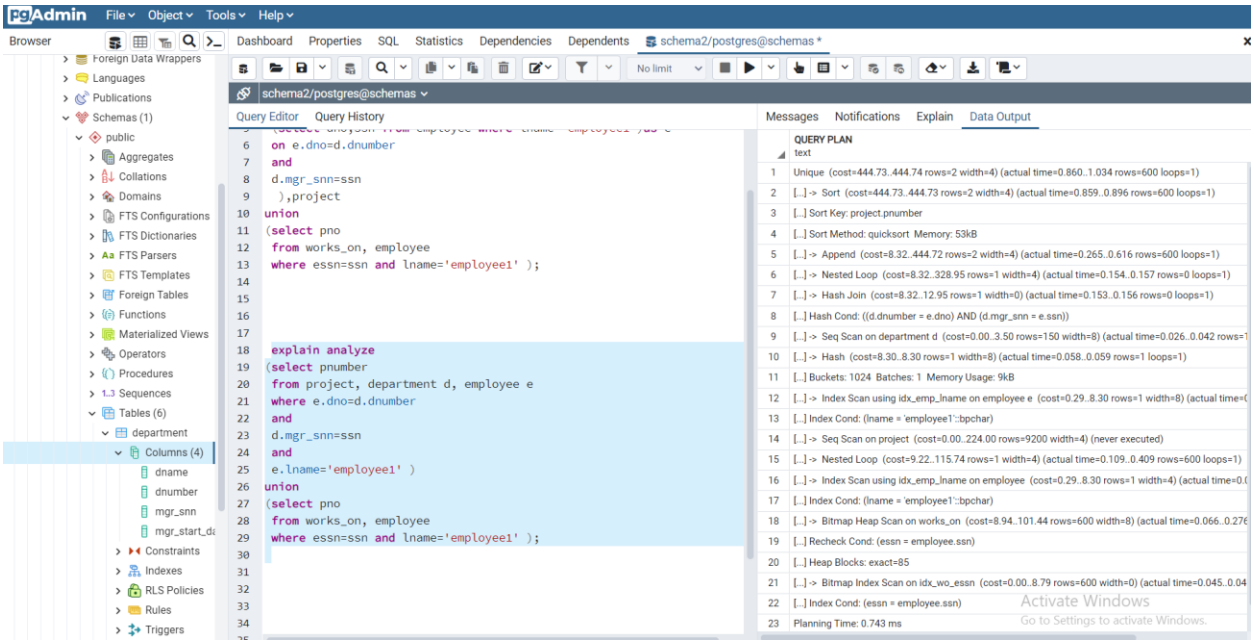
Cost=1547,time=12ms

Notes:

Optimization reduced cost 75.8% of original, the reason is the removal of “or” operator that requires “distinct” operator to ensure no-duplicates and used union that remove duplicates in more efficient way

Time hasn’t been affected that much (became 75% of original time)

=>Using B+Tree: on employee(lname), works_on(essn)



Cost=444.74, time=1.2ms

Notes:

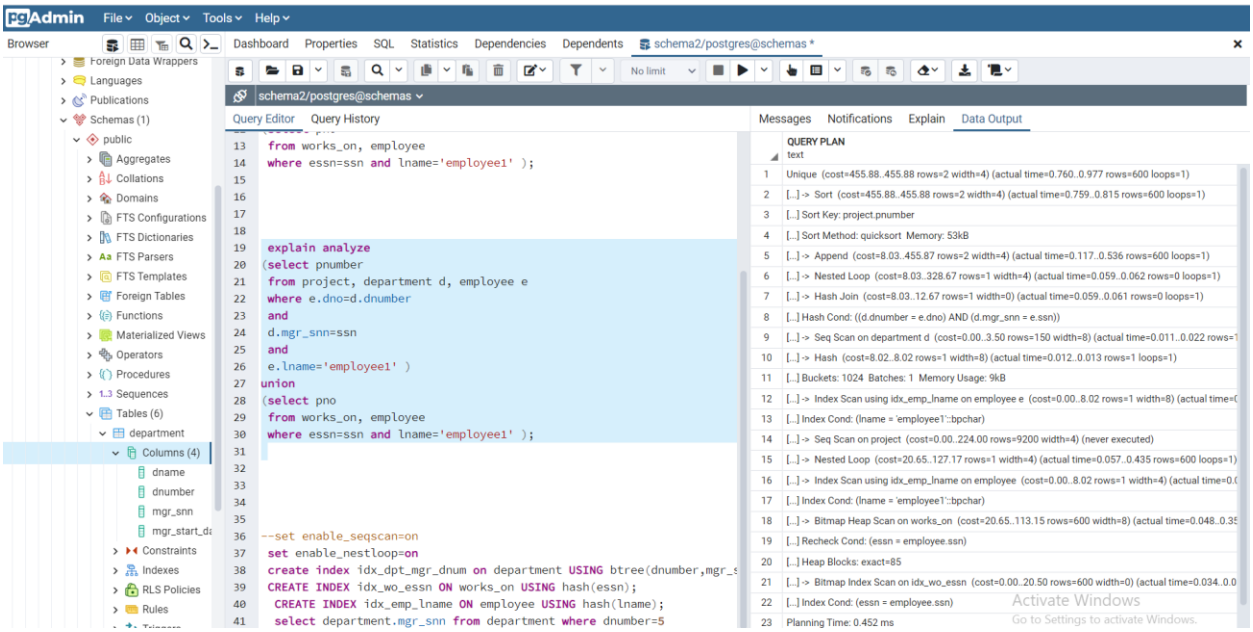
Using Btree indices decreased the cost to 28.75% of initial cost (W/O indices),

Time to 10% of initial time (W/O indices)

The reason behind this is instead of seqscan on relation to do nested loop join(to find matching tuples) the indices made it much faster

Performance improvement here is good as btree is good regardless there is duplicates (like works_on(essn)) or being with not much duplicates (like employee(lname))

=>Using Hash: on employee(lname), works_on(essn)



Cost=455, time=1ms

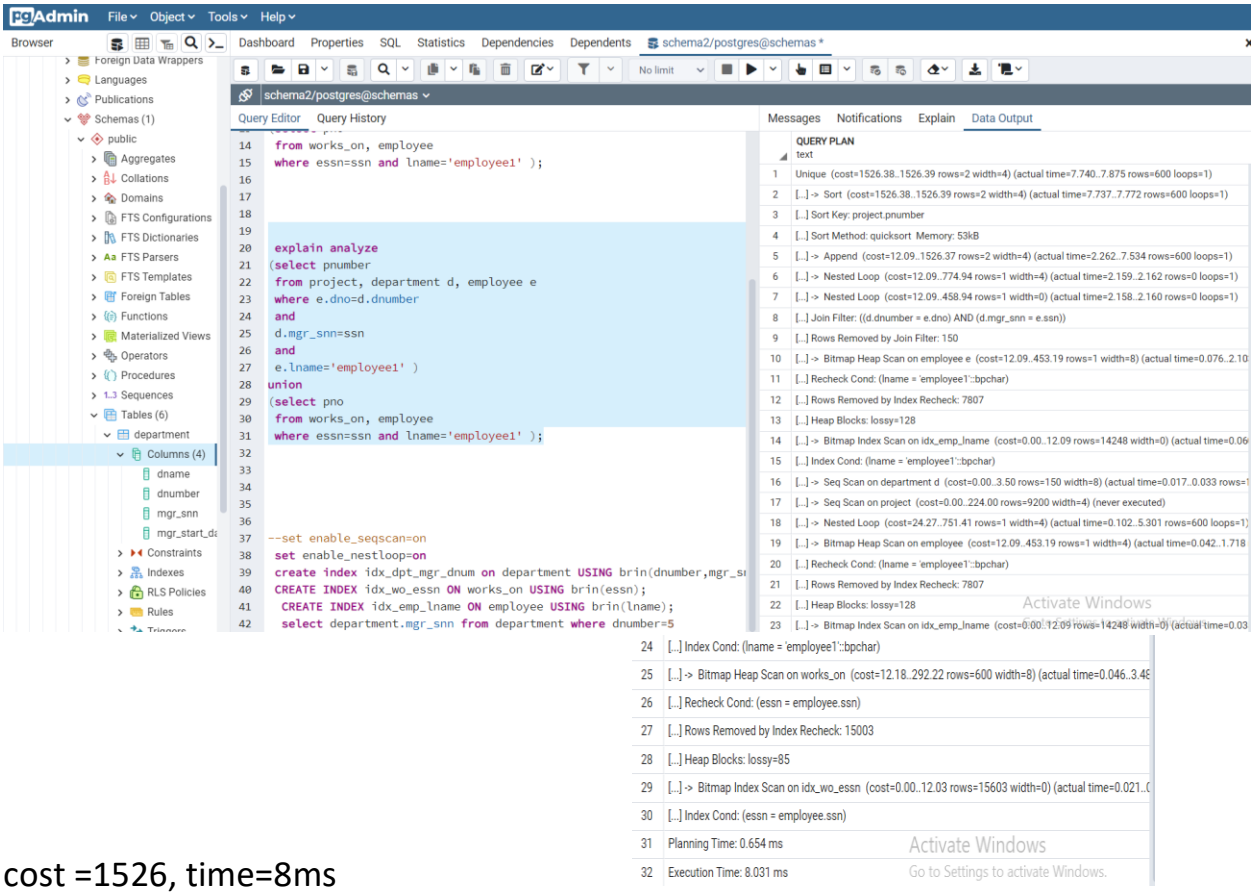
Notes:

Using hash indices decreased the cost to 29.4% of initial cost (W/O indices),

Time to 8.33% of initial time (W/O indices)

Performance improvement here is not good nor bad as hash is bad if there is duplicates (like works_on(essn)) but works well if column has not much duplicates (like employee(lname))

⇨ Using BRIN : on employee(lname), works_on(essn)



cost =1526, time=8ms

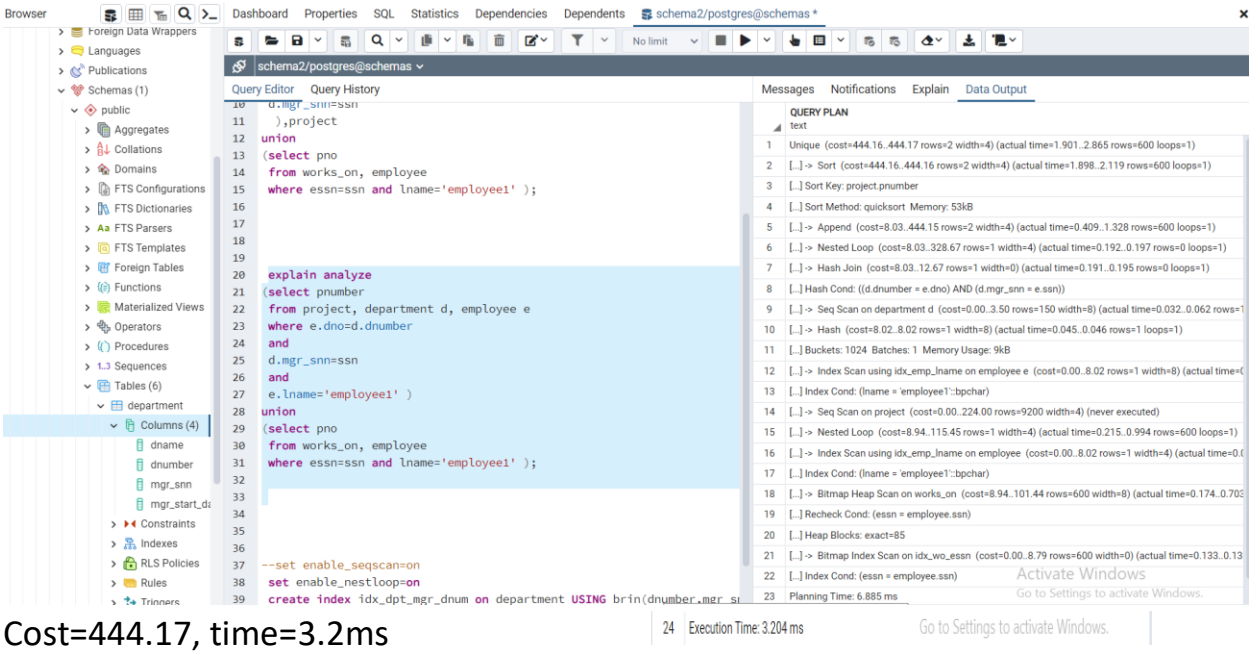
Notes:

Using BRIN indices decreased the cost to 98.6% of initial cost (W/O indices),

Time to 66.6% of initial time (W/O indices)

BRIN as other indeces in this query are aiming to serve the joins, because the brin are not the best to get exact values, it's not the best indix to use here, however it does better than nothing.

⇨ Mixed indices: btree on works_on(essn), hash on employee(lname)



Cost=444.17, time=3.2ms

Notes:

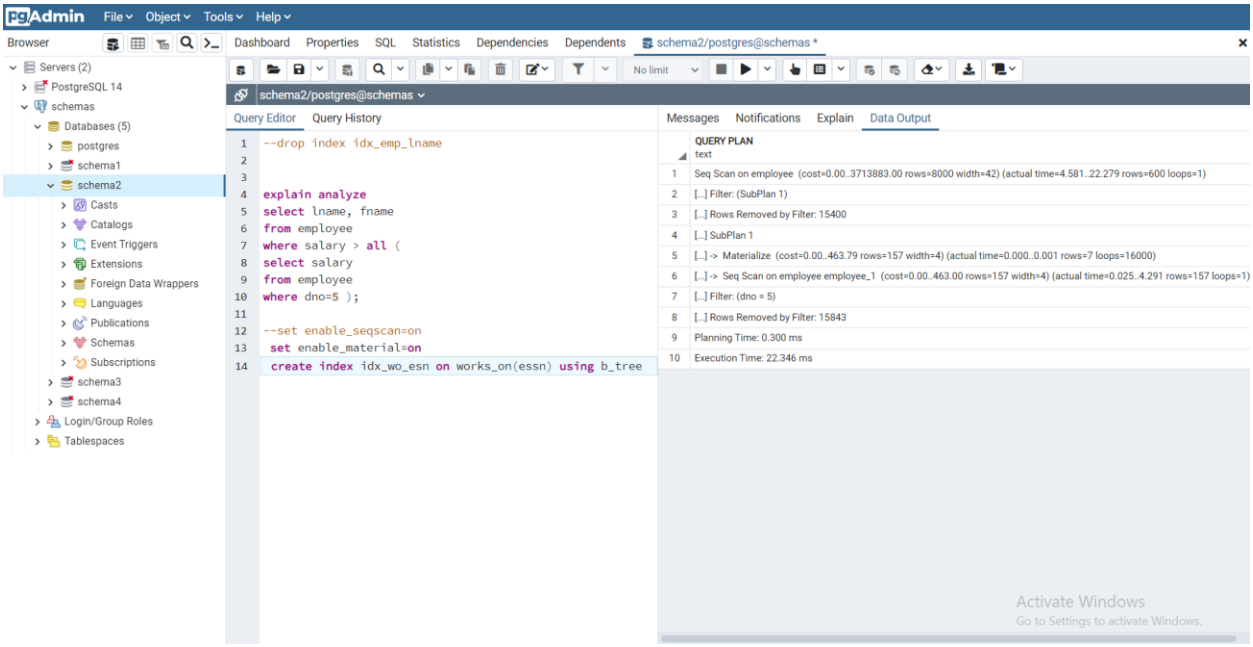
The mixed indices is the optimal solution as it gives the lowest cost among all previous, it's not far away from btree indices, but it gives less cost anyway.

Mixed indices reduces the cost to be 28.71% of the one w/o indices and time to 26.6% of initial time.

Both plans of mixed and btree indices runs the same, but when used btree in the one that has many duplicates(works_on(essn)) that hash is bad at, and used hash in employee(lname)that doesn't have many duplicates, the performance improved alittle (according to the little difference in access time between btree and hash indices)

Q3:

=>Without indices (seqscans only)(with enable_materialize=on):



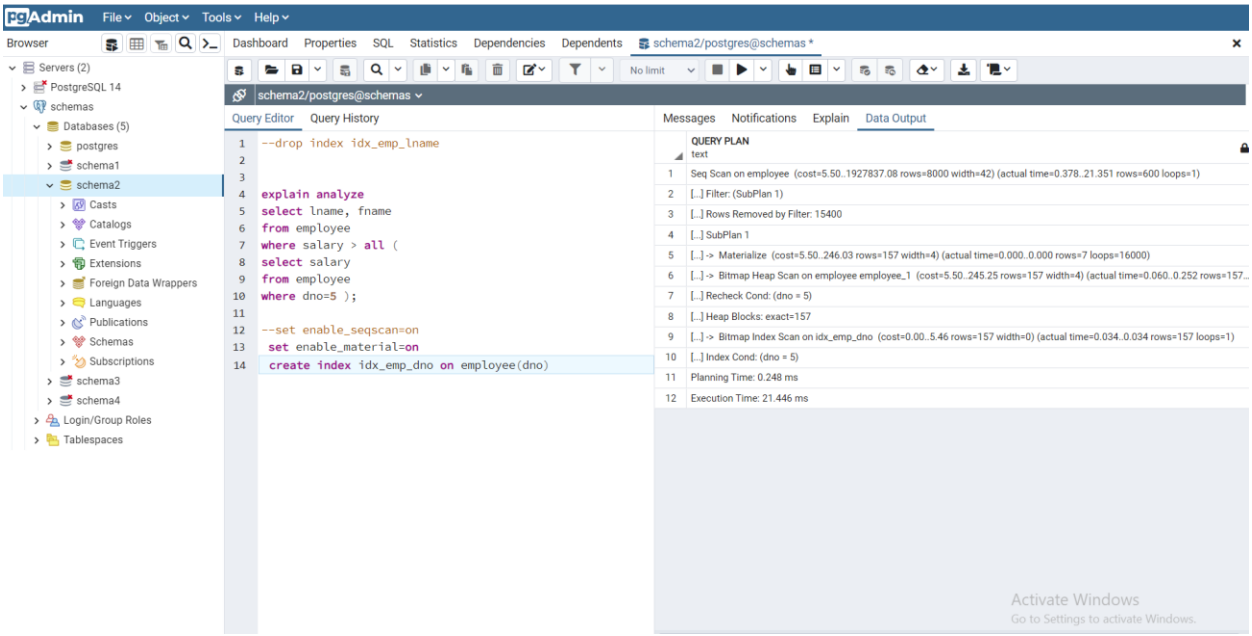
Cost=3713883, time=22ms

Without materialize: cost=3707603, time=4372ms

Notes:

One relation only used which is employee, materialization takes alittle more cost but reduces the time so much

=>Using BTree: on employee(dno)

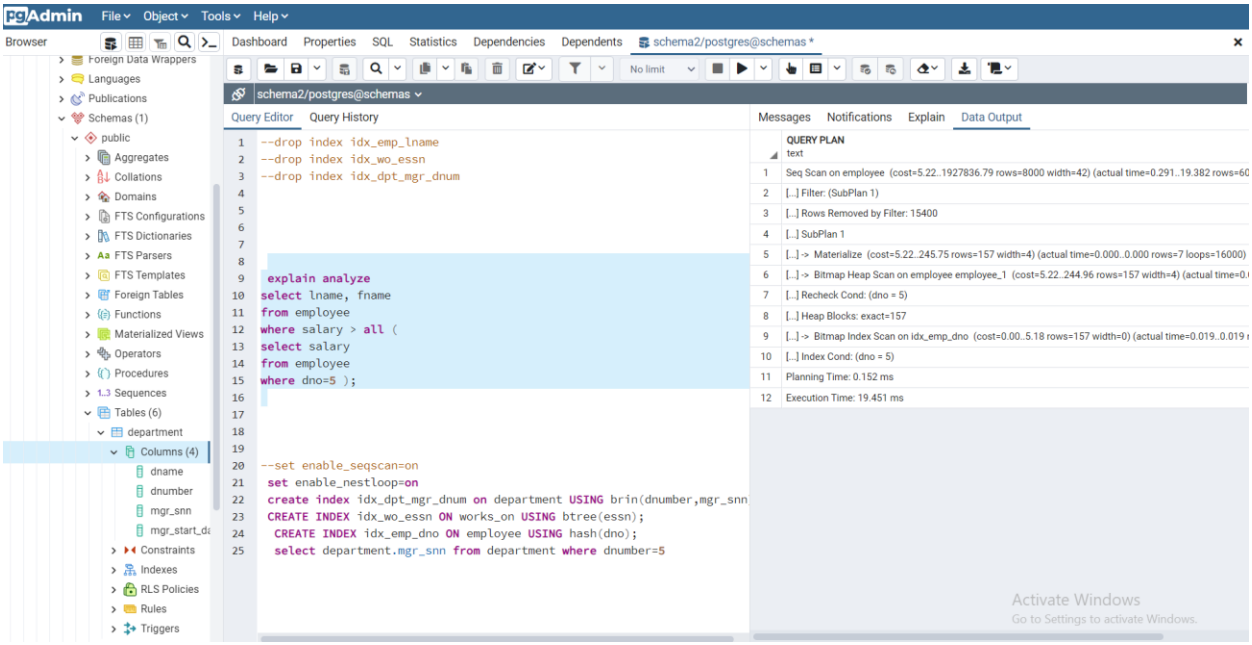


Cost=1927837, time=21ms

Notes:

Index is used to select the exact value(dno=5) and reduced the cost to 51% but time hasn’t been affected that much

=>Using Hash: on employee(dno)

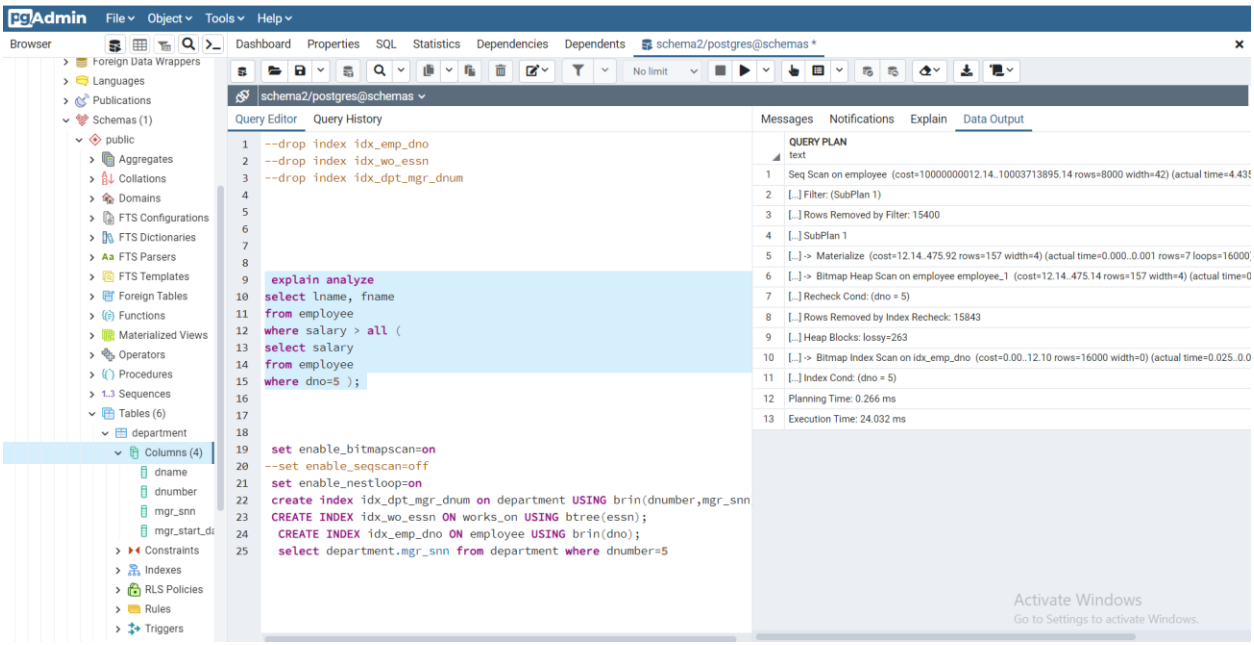


Cost=1927836, time =19.45ms

Notes:

Index is used to select the exact value(dno=5) and reduced the cost to 51% but time hasn't been affected that much

=>Using BRIN: on employee(dno) enable_seqscan=off

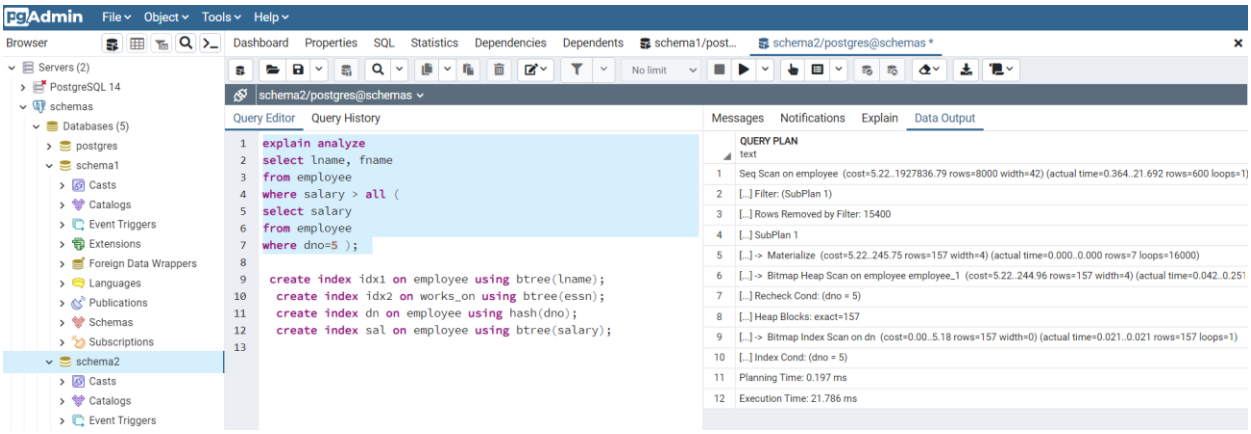


Cost = 10003713895, time=24ms

Notes:

Index is used to select the exact value(dno=5) but the performance of brin is not good because of it passes by several layers of sparce and it requires a seqscan inside the target page

=>Using mixed:hash on employee(dno),btree on employee(salary)



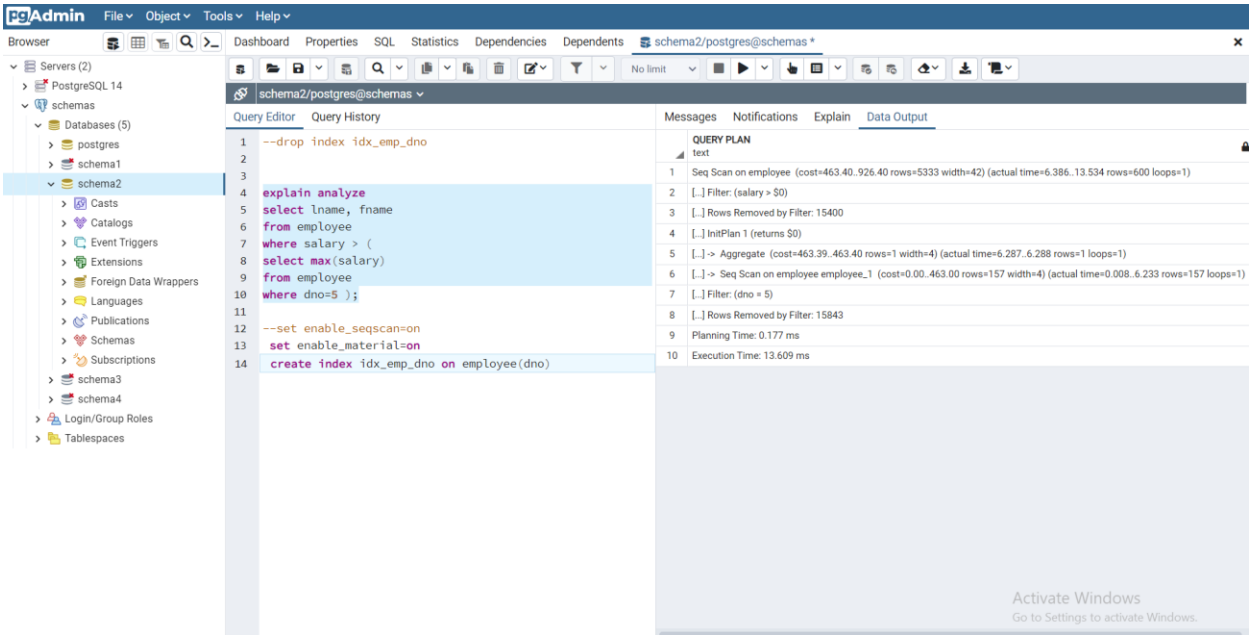
Cost=1927836, time=21ms

Notes:

Same as hash section since the engine could use one index only which is the hash index.

Q3: optimized

=>Without indices (seqscans only)(with enable_materialize=on):



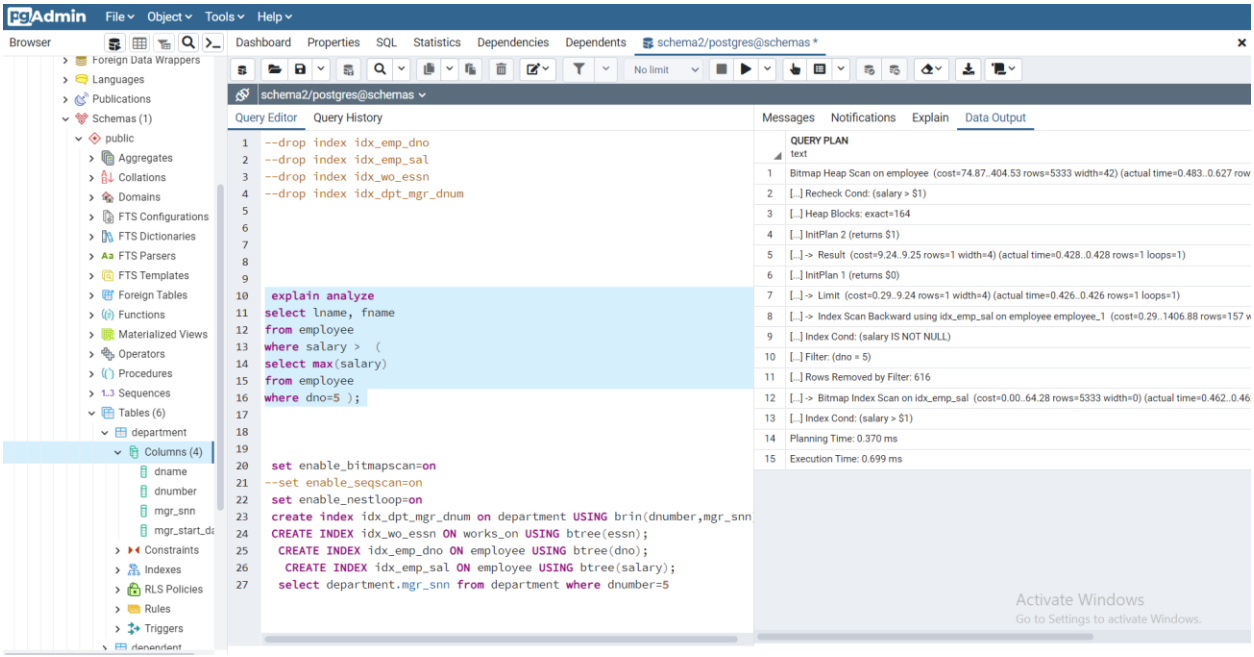
Cost=926, time=13ms

Notes:

Optimization is based on instead of going checking “all” salaries of employees of dno=5 to check is the current employee is greater than all of their salaries, it’s enough to check that the current employee gets more salary than the max of dno=5 employee salary

Optimization reduced cost to 0.025% of the original cost, time to 61.9% of original time

=>Using B+Tree: on employee(salary)



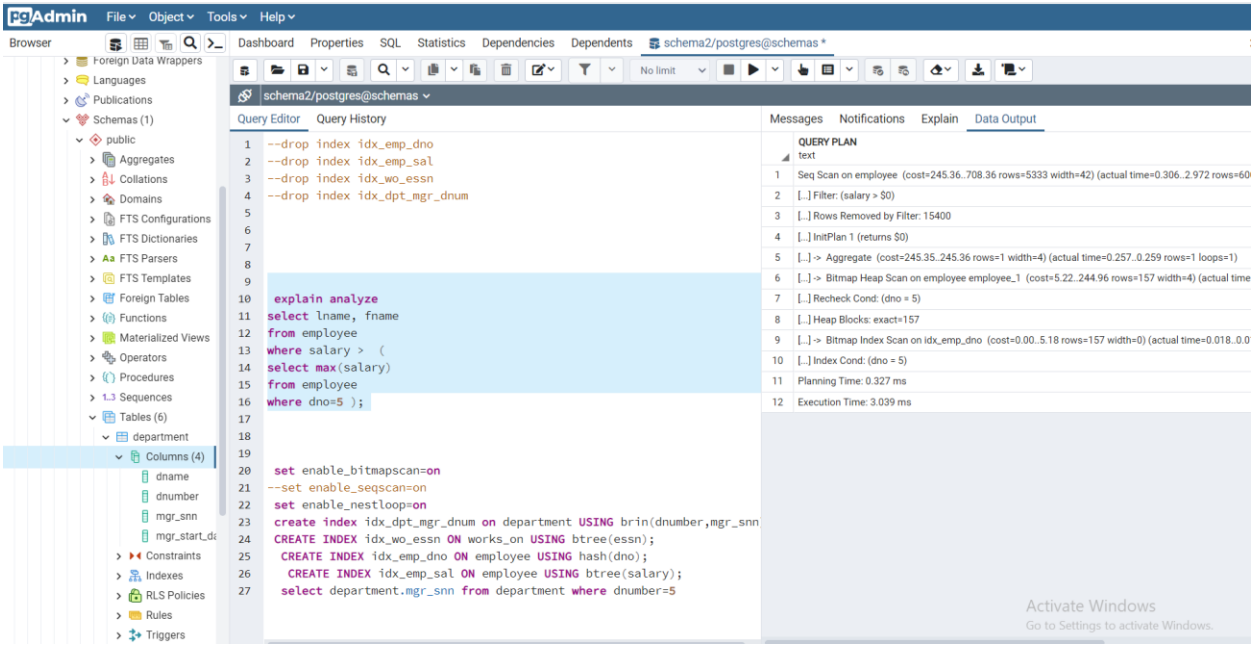
Cost=404.5, time=0.7ms

Notes:

Using Btree indices decreased the cost to 43.68% of initial cost (W/O indices),

Time to 5.38% of initial time(W/O indices)

⇨ Using Hash: on employee(dno)



Cost=708, time=3ms

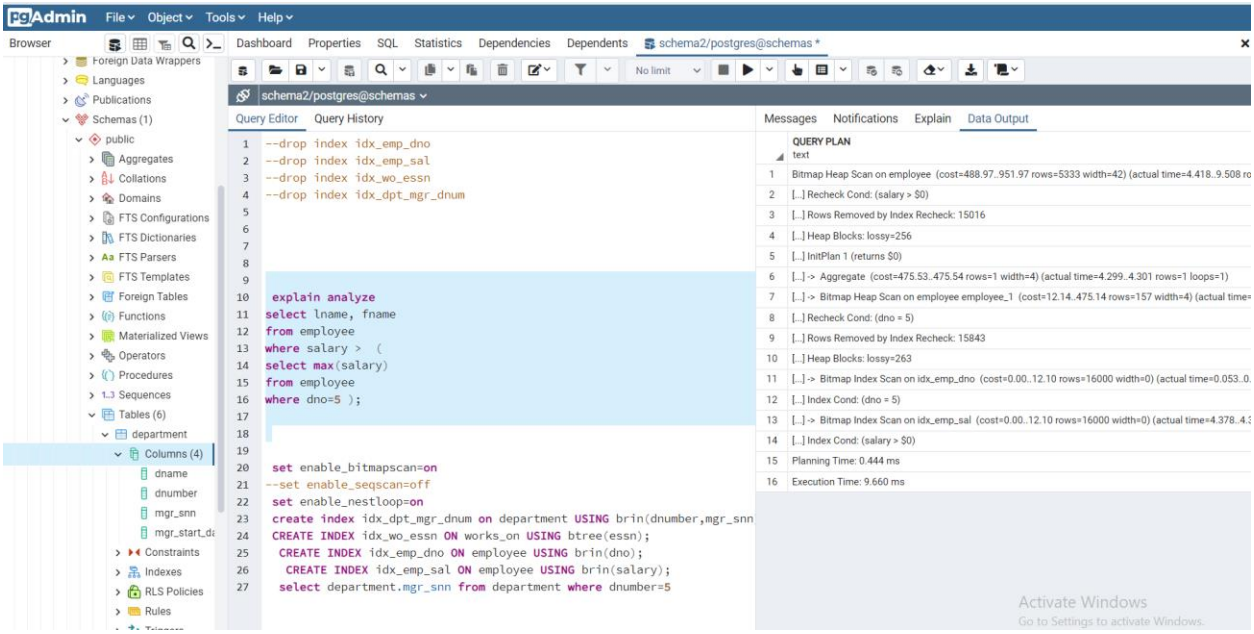
Notes:

Using Hash indices decreased the cost to 76.5% of initial cost (W/O indices),

Time to 23% of initial time (W/O indices)

The hash is not that efficient as btree in this query because btree support ranges so it can scan the part of employee table with salaries > max of dno=5 salaries, unlike the hash that is going to check every single tuple

⇨ Using BRIN: on employee(dno),employee(salary)



Cost =951, time =9.6ms

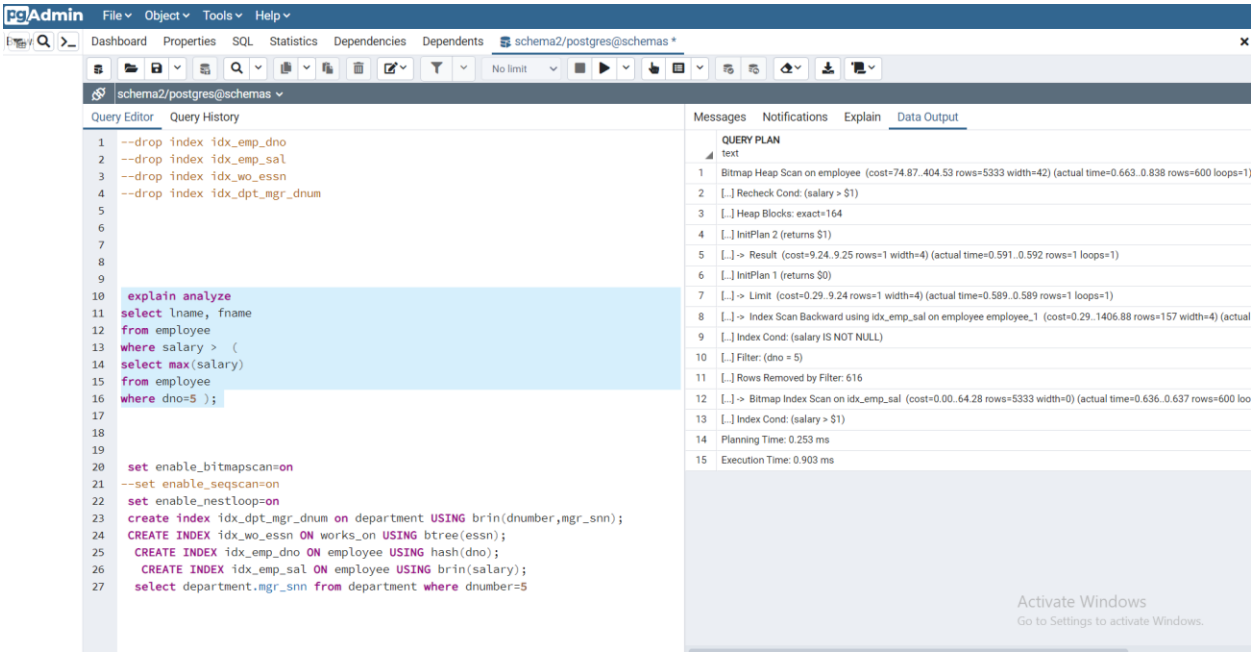
Notes:

Using BRIN indices increased the cost to 102.7% of initial cost (W/O indices),

Decreased Time to 73% of initial time (W/O indices)

The brin do support ranges, but the high cost operation was after detecting the entry in idx, we needed to go retrieve it from employee relation to continue executing the query, even brin doesn't support included columns that gives the chance to carry the attributes that you may need not to go back to relation.

⇨ Mixed Indices: btree on employee(salary), hash on employee(dno)



Cost=404.5, time= 0.9ms

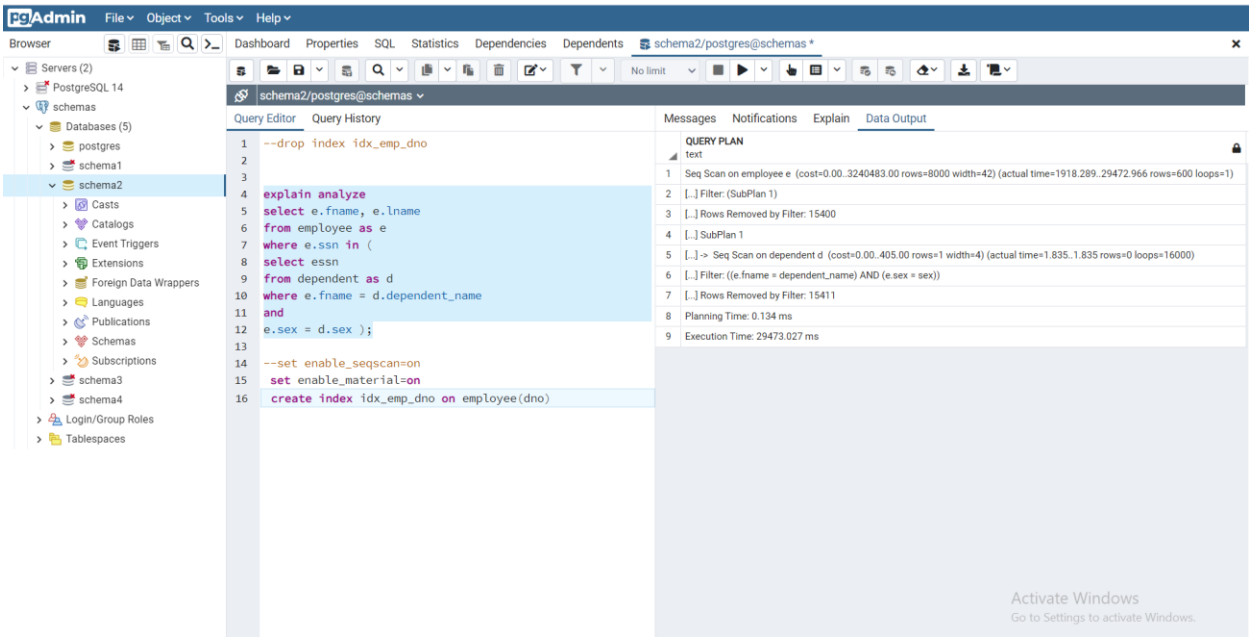
*only one index was used which is employee(salary) as the query can use only one idx, so the chosen one is the one giving least cost

Notes:

The results are exactly the same as btree section, because the query plan uses only one idx which is btree on employee(salary)

Q4:

=>Without indices (seqscans only):

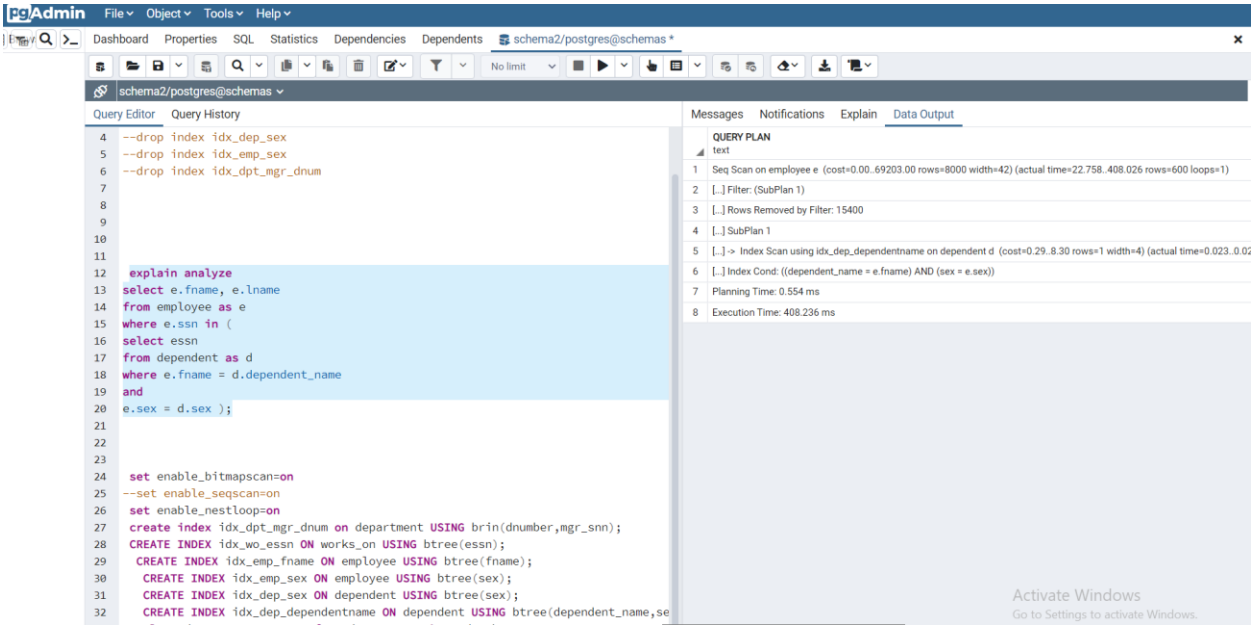


Cost=3240483, time=29473ms

Notes:

The cost is too high, the time is high as well.

=>Using BTree: on dependent(dependent_name,sex)



Cost=69203, time=408ms

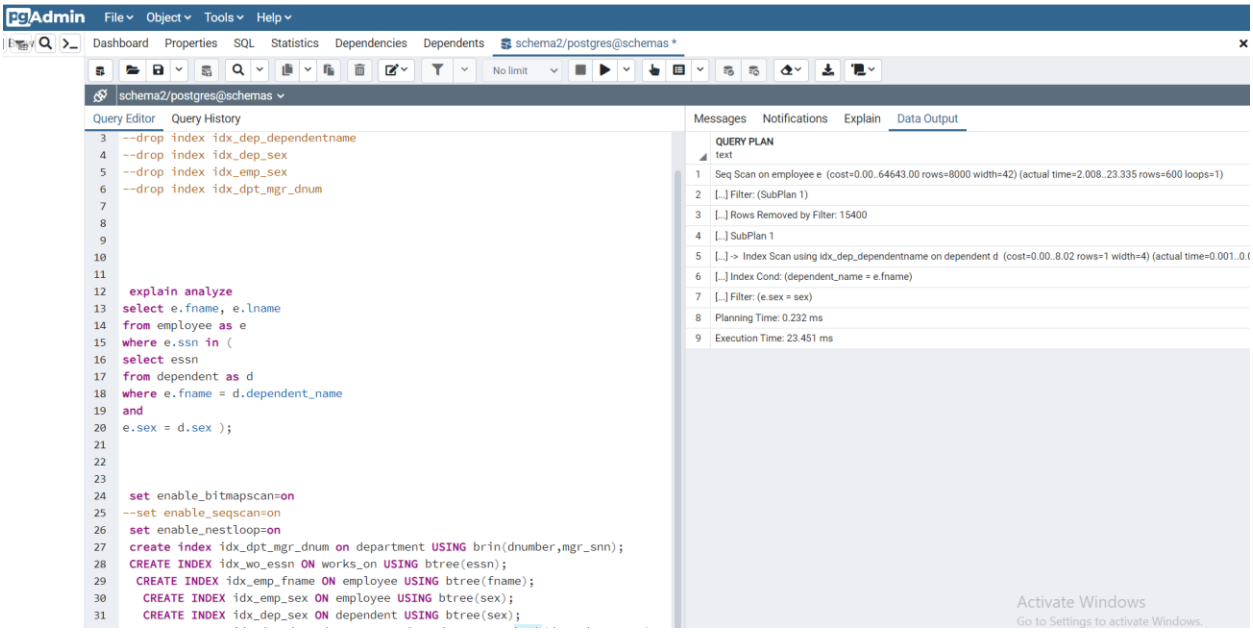
Notes:

Using Btree indices decreased the cost to 2.1% of initial cost (W/O indices),

Time to 1.38% of initial time (W/O indices)

The reason behind this is instead of seqscan on relation to find matching tuples, the indices made it much faster

⇨ Using Hash:



Cost =64643, time=23ms

Notes:

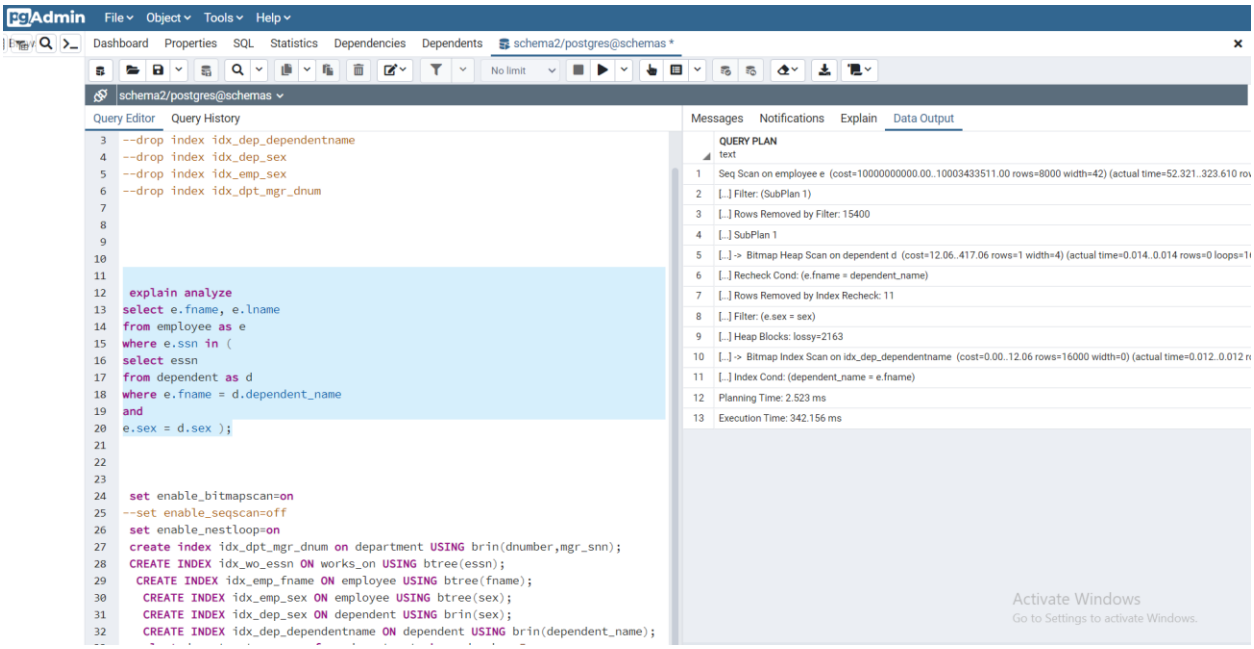
Using Hash indices decreased the cost to 1.9% of initial cost (W/O indices),

Time to 0.078% of initial time (W/O indices)

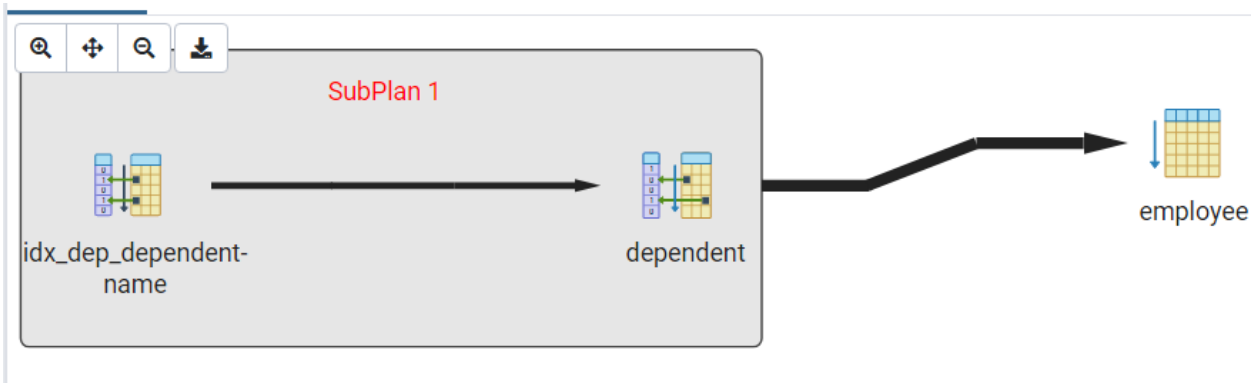
The reason behind this is instead of seqscan on relation to find matching tuples, the indices made it much faster.

Hash index here is even much more efficient than btree on this query because it's exact values query that is accessible in O(1) with hashing

=>Using BRIN:



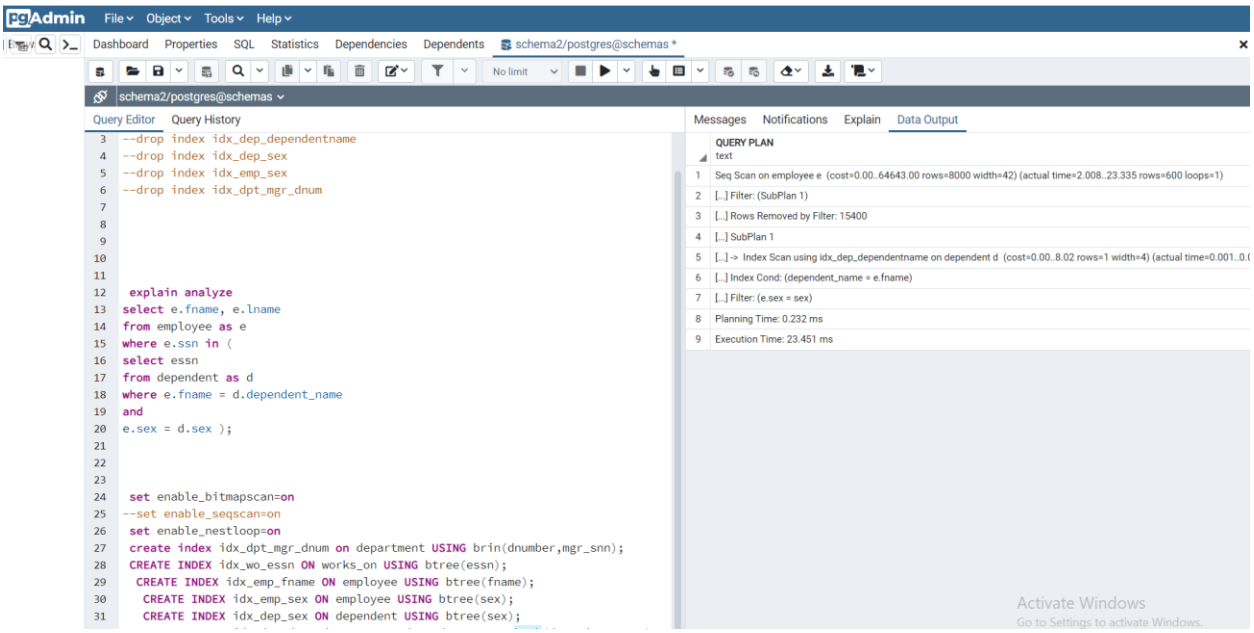
Cost is too high, time=342ms



Notes:

The reason of this very high cost is that we are forcing to use brin index which is not much good in exact value queries, It takes time to reach the data of relation due to the layers of sparse, in addition, it doesn't support multicolumn indix, so it checks the first name and then goes to the relation to check the sex of dependent

=>Using mixed indices: hash on dependent(dependent_name),btree on dependent(sex)



Cost=64643, time= 23.45ms

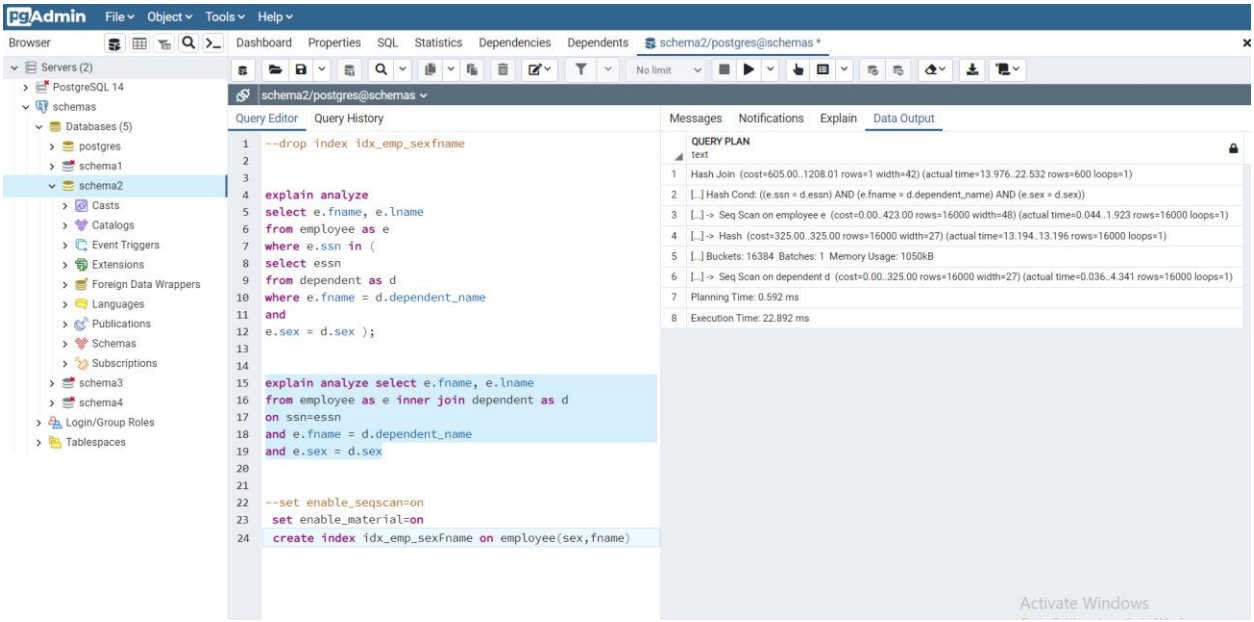
*only one index was used which is dependent(dependent_name) as the query can use only one idx, so the chosen one is the one giving least cost

Notes:

The results are exactly the same as Hash section, because the query plan uses only one idx which is btree on dependent(dependent_name)

Q4: optimized

=>Without indices (seqscans only):



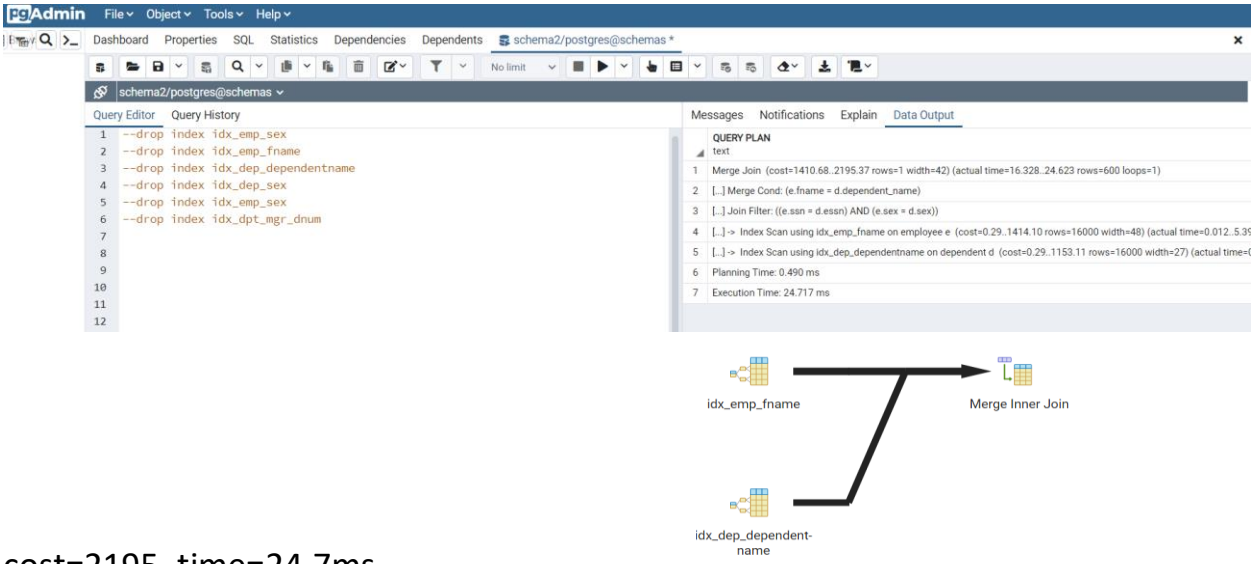
Cost=1208, time=22ms

Notes:

Optimization is based on instead of seqscan of the whole relation dependent for every tuple in the relation employee to check the where clause conditions, we can inner join(hash join) on the same conditions

Optimization reduced cost to 0.037% of the original cost, time to 0.075% of original time

=>Using Btree:

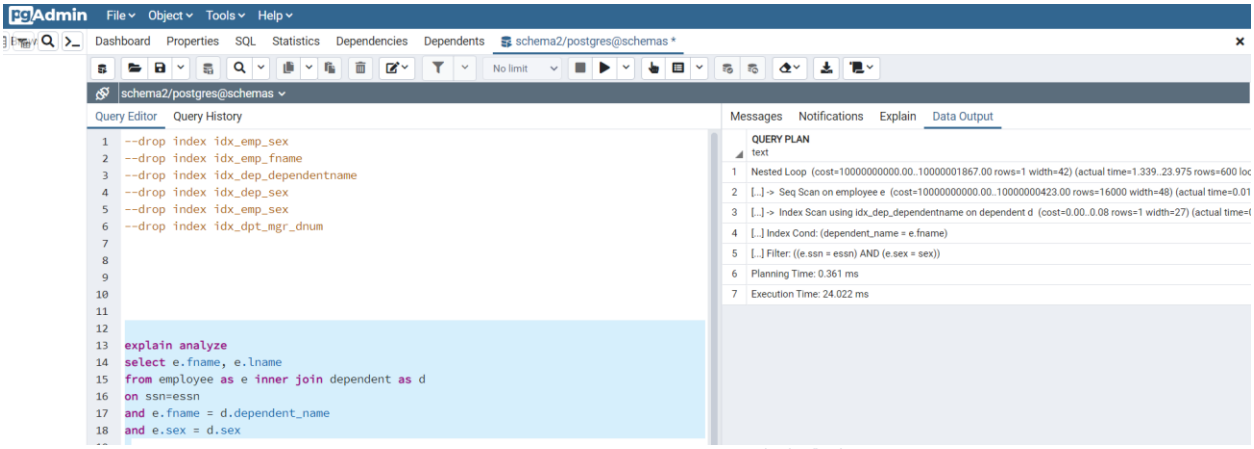


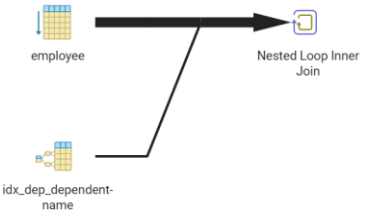
cost=2195, time=24.7ms

Notes:

The cost increased to 181.7% of the cost without indeces, the reason is that the hash join was the optimal plan, but it doesn't require indeces, so to force the engine to use my btrees, seqscan was turned off, so the type of join done is merge join.

=>Using Hash: on dependent(dependent_name)



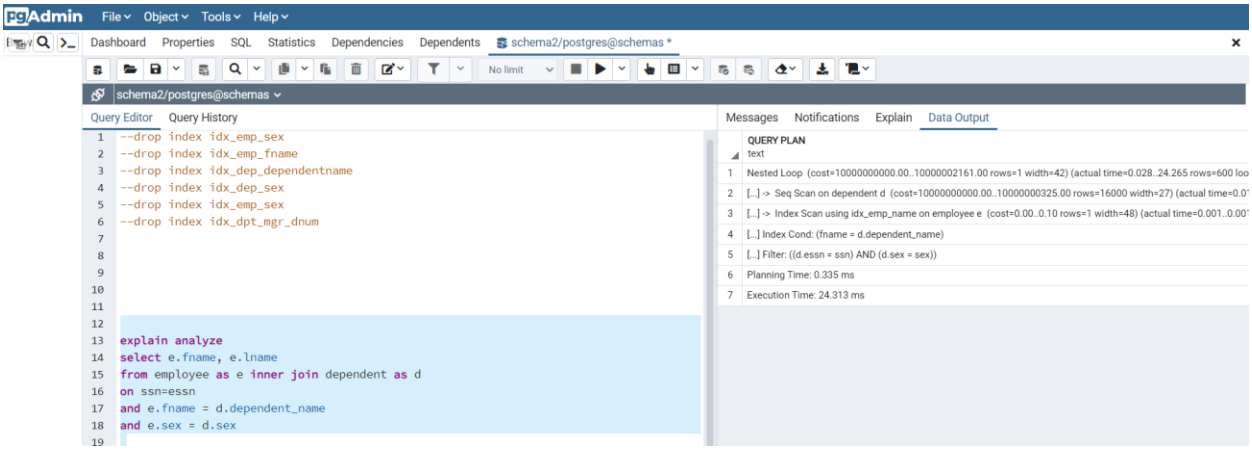


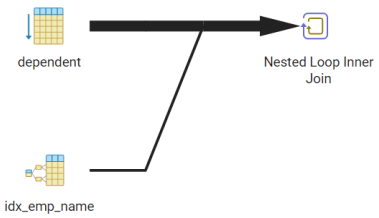
cost=10000001867,time=24ms

Notes:

The cost increased too much, the reason is that the hash join was the optimal plan, but it doesn't require indeces, so to force the engine to use my hash indeces, seqscan was turned off, so the type of join done is nested loop join, which gets one by one from employee and searches in dependent if there is matching tuple.

=>Using BRIN: on employee(name)



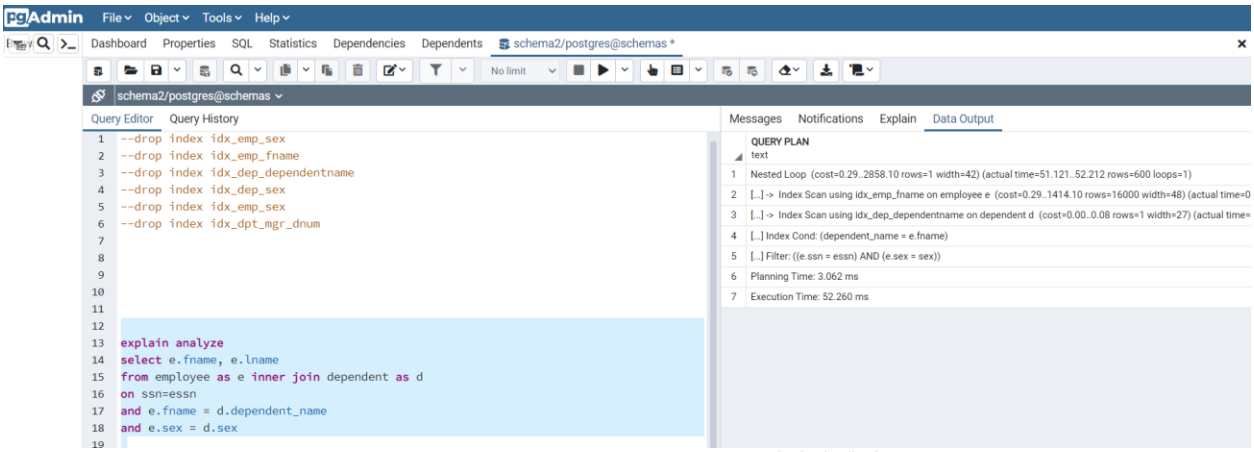


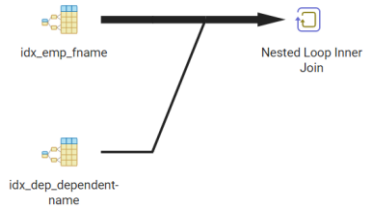
time=24ms

Notes:

The cost increased too much, the reason is that the hash join was the optimal plan, but it doesn't require indeces, so to force the engine to use my brin indeces, seqscan was turned off, so the type of join done is nested loop join, which gets one by one from employee and searches in dependent if there is matching tuple using brin index which is bad in exact values query.

=>Using Mixed: btree on employee(fname), hash on depenent(depenedent_name)





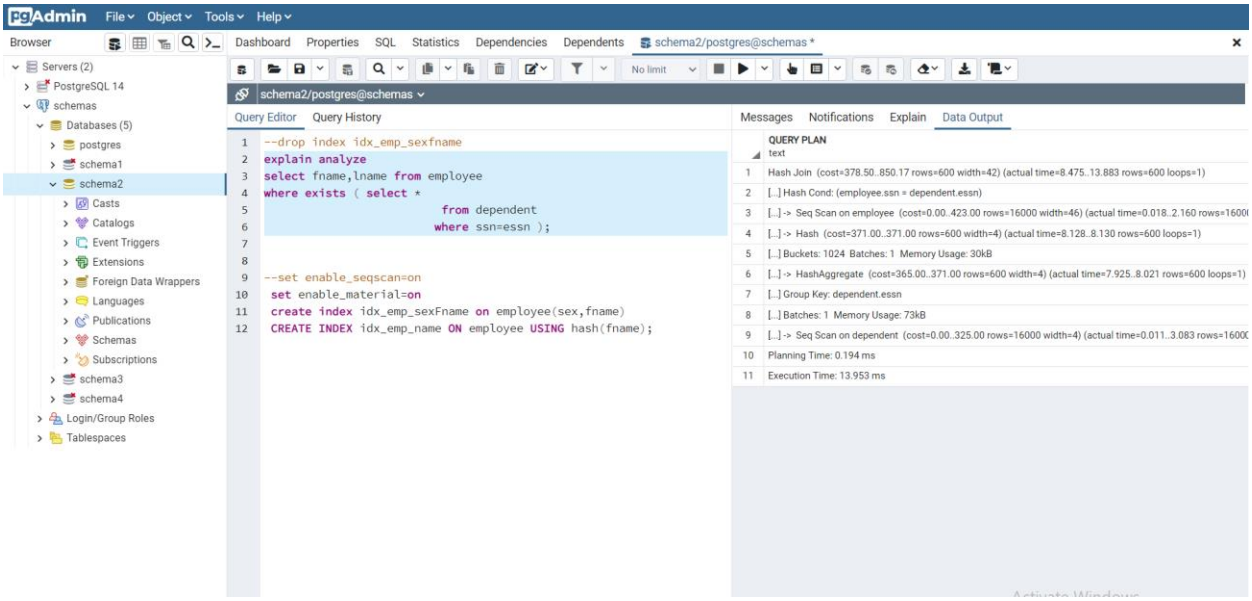
cost=2858,time=52ms

Notes:

The cost is not too high, but it's higher the one w/o indeces. The engine was not willing to use any indeces to to hash join, so we turned the seqscan off. The cost is not far away from btree version .

Q5:

=>Without indices (seqscans only):

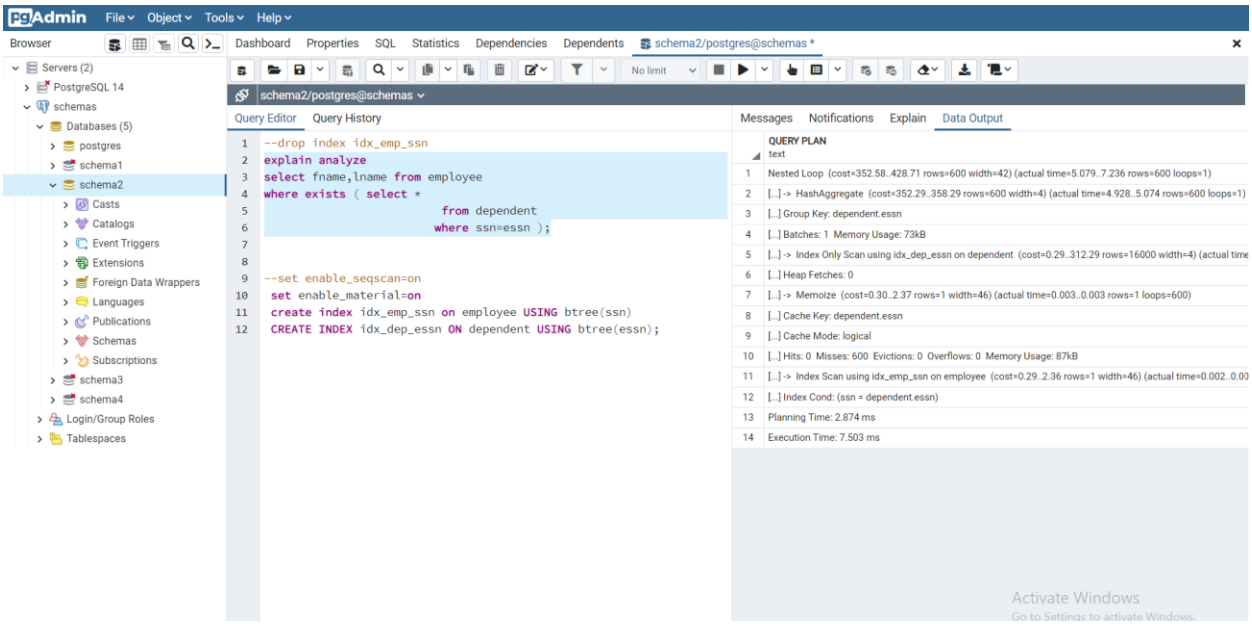


Cost=850, time=13.9ms

Notes:

One relation only used which is employee, materialization takes alittle more cost but reduces the time so much

=>Using BTree: on employee(ssn), dependent(essn)



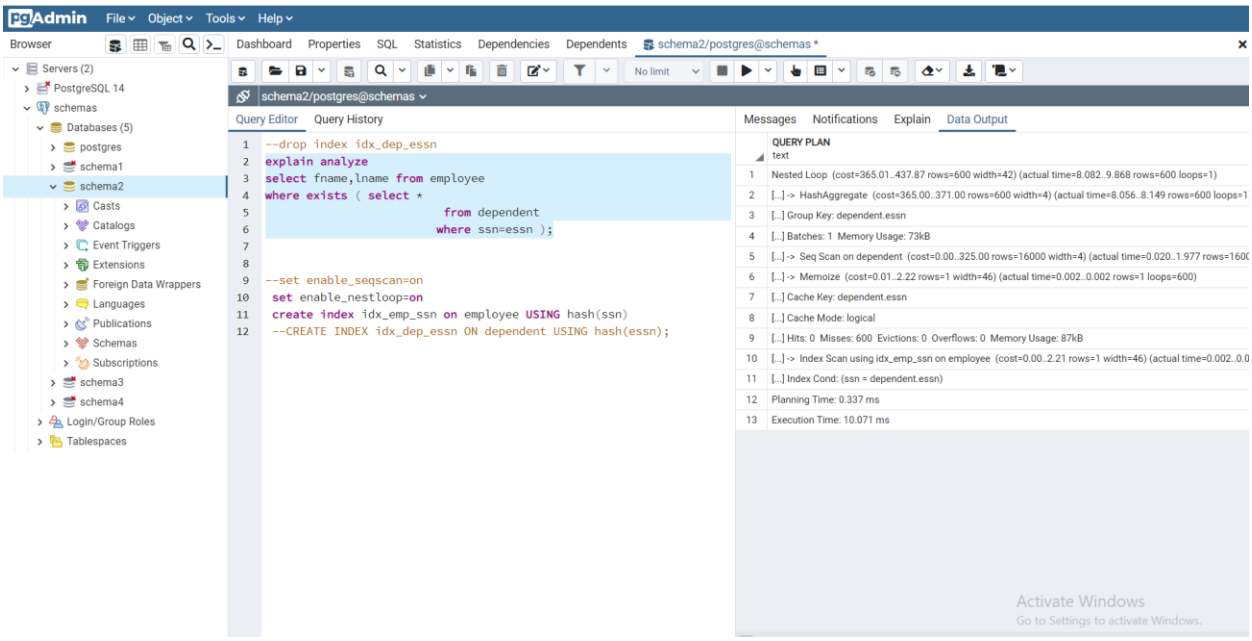
Cost=428, time=7ms

Notes:

Btree indeces reduce the cost to 50.3% of same query W/O indeces, time to 50% of previous time.

Instead of hashing to do hash join, the btree indices provides fast way to search for the condition of nested loop join, so we are searching for ssn=essn, so we can reach both of them faster

=>Using Hash: on employee(ssn)



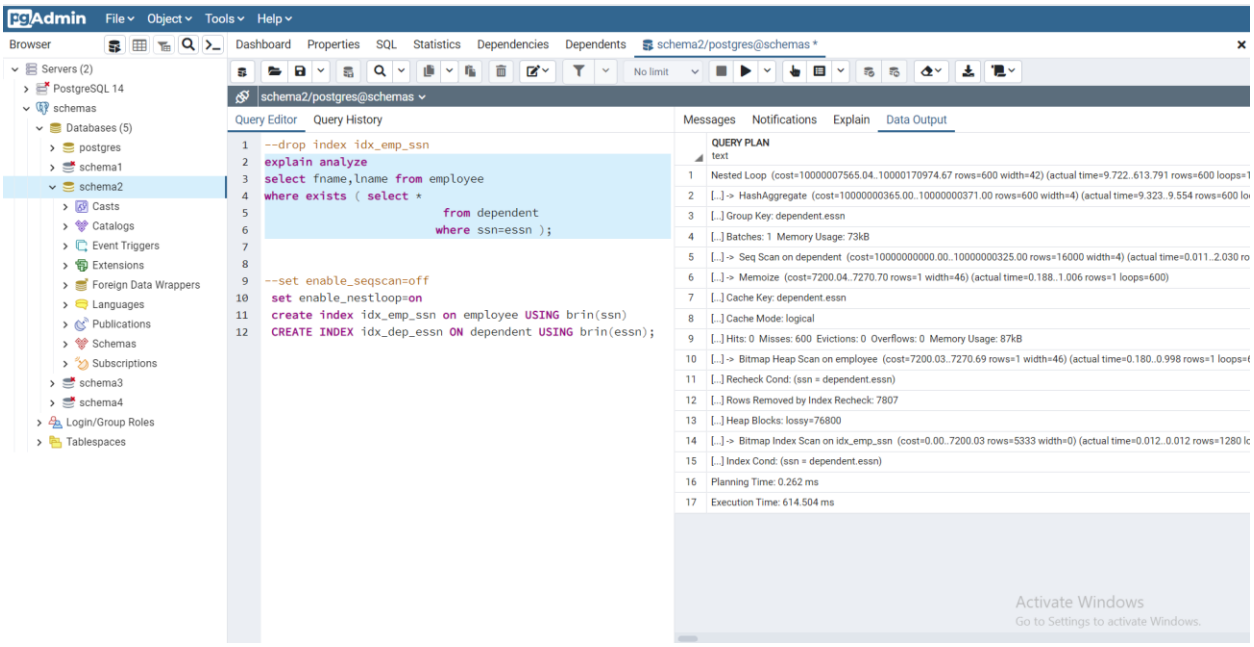
Cost=437, time=10ms

Notes:

Hash indeces reduce the cost to 51.4% of same query W/O indeces, time to 71.9% of previous time.

Since ssn is primary key (unique), but essn is not unique, so searching for unique key using hash is much easier so employee(ssn) is chosen to be the key which is being searched for, for each tuple of dependent relation.

=>with BRIN: on employee(ssn)



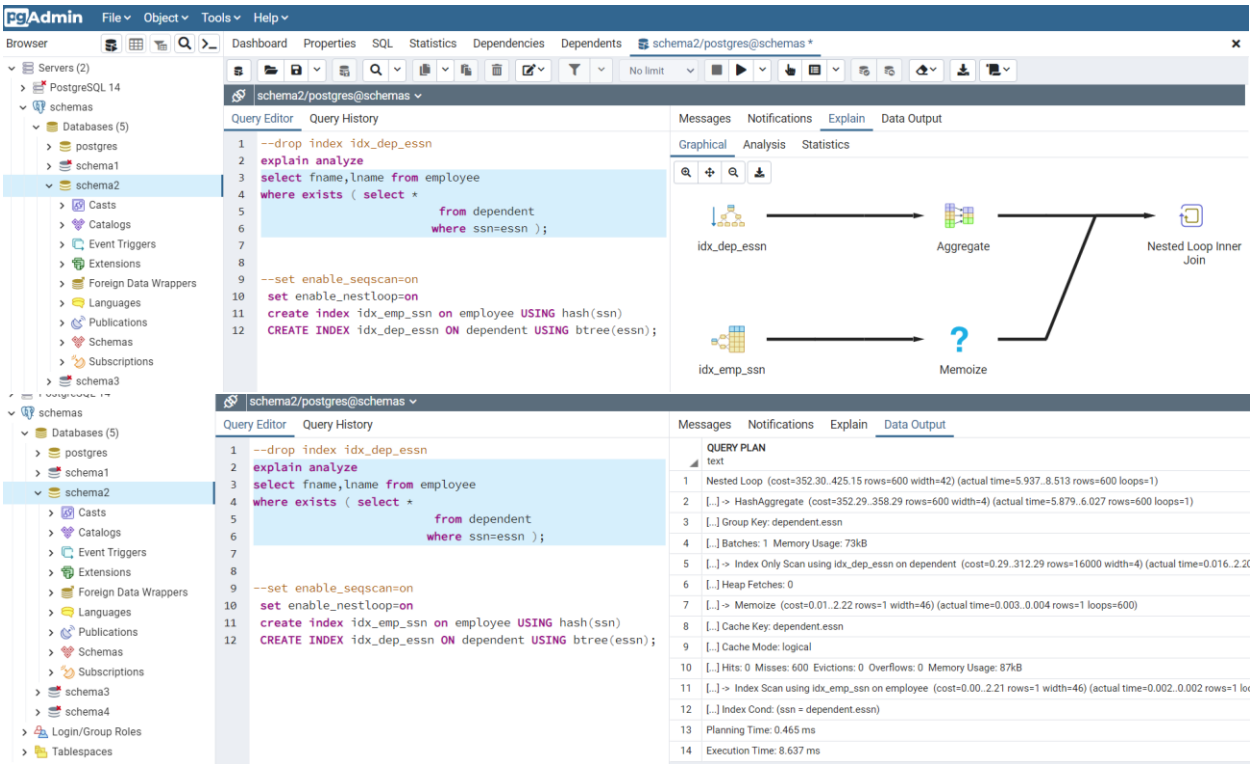
Cost=10000170974, time=614ms

Notes:

The plan can use one brin indix only to search for each key of the other relation, the the options are employee(ssn) or dependent(essn), postgres has a feature of chaching repetitve searches, so when chosing employee to have the seqscan to search in dependent, it's a bad choice because we won't repeat any search as ssn in unique, but the other side will have 600 unique search only instead of 16000.

It's obvious that brin performance is much worse than the one w/o indix because the search of brin is not that easy, it passes by several sparse layers, however the efficiancy of brin appears when searchingt in small range of big domain which is not the case here.

=>mixed indices: hash on employee(ssn),btree on dependent(essn)



Cost=425, time=8.6ms

Notes:

The mixed indices is the optimal solution as it gives the lowest cost among all previous, it's not far away from btree indices, but it gives less cost anyway.

Mixed indices reduces the cost to be 50.0% of the one w/o indices and time to 61% of initial time.

Both plans of mixed and btree indices runs the same, as we get tuple of dependent from its index and search for matching tuple in employee relation using the index on ssn, but hash indix is a bit faster than the btree index in searching specially in exact value in a unique column.