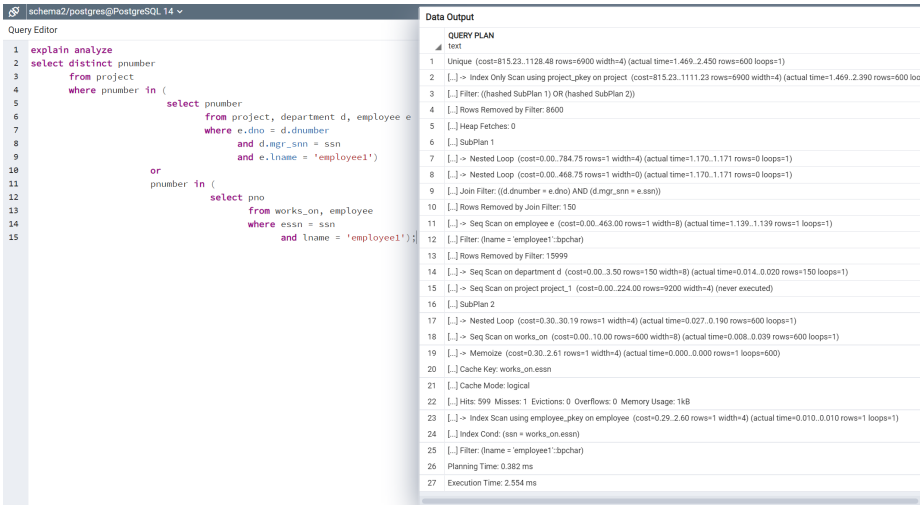
Schema 2 DB

Hussein Yasser Hussein Ebrahim

49-18120

1 Query 1

1. The Query with no Indexes



The Execution Time ranges between 2.5 msec to 2.7 msec (NOTE This Plan is the first one that the engine has generated and it uses the primary key indexes the next one i dropped the primary key constraints and had run the plan again with new plan without and indices but slower a bit)

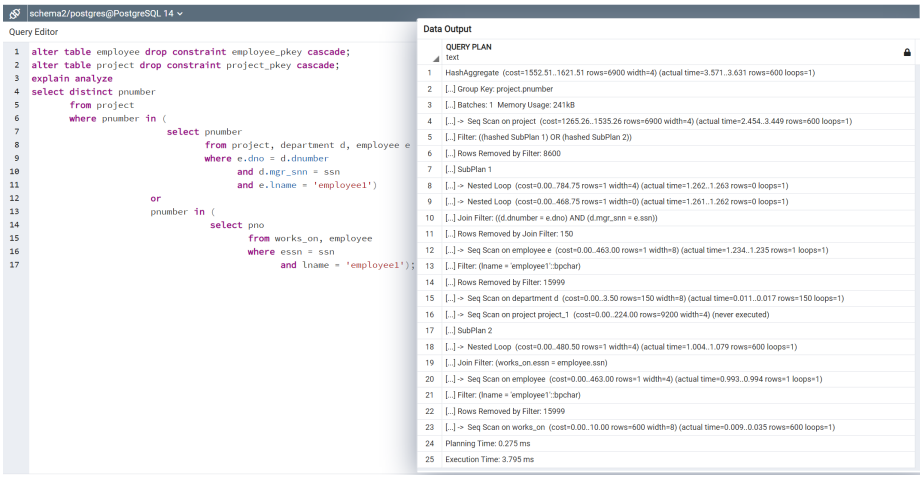
The Physical Plan :

Let's start with the first sub query , the engine makes a seq scan over the employee table making a filter such that lname = 'employee1' and a seq scan over the department table and making a nested loop join with condition that ((d.dnumber = e.dno) AND (d.mgr-snn = e.ssn)) then after that making a cartesian product with the project using a nested loop join without any conditions.

The Second Inner Query we make an index scan over the employee-pkey index (default one by postgress which is btree)applying condition lname = 'employee1', and then cache it into the memory the needed results only and seq scan over the works-on table making a nested loop join such that works-on.ssn = employee.ssn, the step of memoization helps a lot because you don't need to recompute in the join you are caching the values that you need so you get i faster.

The Whole thing , The Engine makes an index only scan over the project relation(why index only scan ? because you don't need any additional data you just want the project number and it is already in the index) and apply the filter ORING(sub-plan1 and sub-plan2) and finally applying the distnict(unique) operator.

The Estimated Cost = 1128.48



Here another one after dropping the primary key constraints to force the engine not to use any indexes at all.

Execution Time is between 3.6 msec to 4 msec.

Physical Plan :

Subquery 1 : seq scan over the employee table with applying condition lname = 'employee1' and another seqscan over the department table then applying a nested loop join on them with condition employee.ssn = department.mgr-ssn, then result of this join to be made a nested loop join with the project table with no conditions (cartesian product) between them. Subquery 2 : Seq scan over the employee table applying condition

lname = 'employee1' and another seq scan over the table works-on then a nested loop join between them with condition works-o.essn = employee.ssn . The Whole Thing : Seq scan over the project table and apply the conditions of Sub1 and Sub2 if either of them satisfied we take the row, and then it interpreted the unique(distnict) operator as a hash aggregation(group by using hash algorithm).

The Estimated Cost = 1621.51 (Greater than the original engine plan that used the primary key indexes).

2. The Query with Btree Indexes

The Execution Time is the range of 1.3 msec to 1.5 msec.

The Indexes i have created :

projectK : BTree index created on project table on attribute pnumber.

blname : BTree index created on the employee table on the attribute lname.

The Physical Plan :

The Plan goes just as the original one but with small differences. For The SubQuery 1 : It uses the blname index to filter the lname = 'employee1' then made a nest loop join with the department to satisfy the conditions(department.dnumber = employee.dno and department.mgr-ssn = employee.ssn), and then a blind nest loop join with the project. For the Subquery2 it does the same as it uses also the blname index to make the

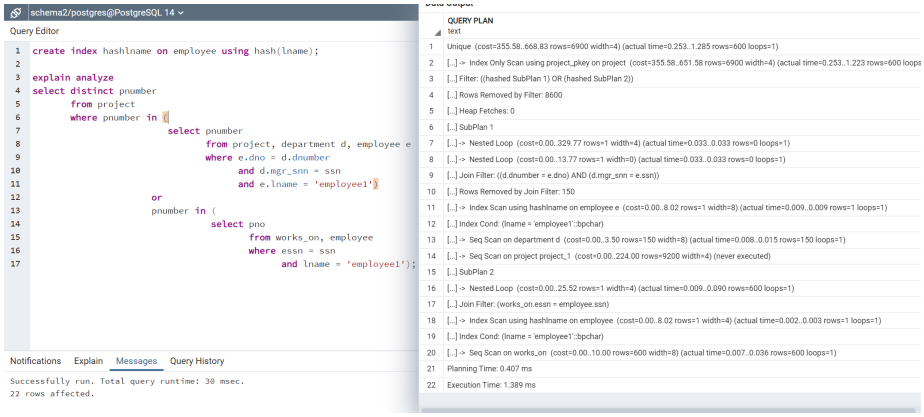
lname = 'employee1' filter and then make a nest loop with works-on table , and finally make an index only scan over the projectK index created on the project and apply for each entry in the index ORING the subplans and then the unique operator is applied.

The Estimated Cost = 669.40 (Smaller than the query without any indexes) hence a better performance.

Why The Indexes created enhanced the performance?

Because the index on the lname enhanced the search for the first condition (lname = 'employee1') before the index it was done as a seqscan and that was slower because the employee table is 16000 entries so an index scan was better in that case and also because we are not returning the whole table at the end we just need small subset so we don't have to move over the whole table costing us just time and IO, and also the BTree on the ProjectK index because the engine will make use that the data is sorted in that index created on the project table and easily can make the unique operator on it to get the final result and one other thing that when it iterates over the btree index created on the project it doesn't need to get the data from the disk anymore as everything we need to use is already existing in the index so we need one single fetch from the index :').

3. The Query with Hash Indexes



The Indexes i have created :

hashlname : Hash index on the table employee using the attribute lname.

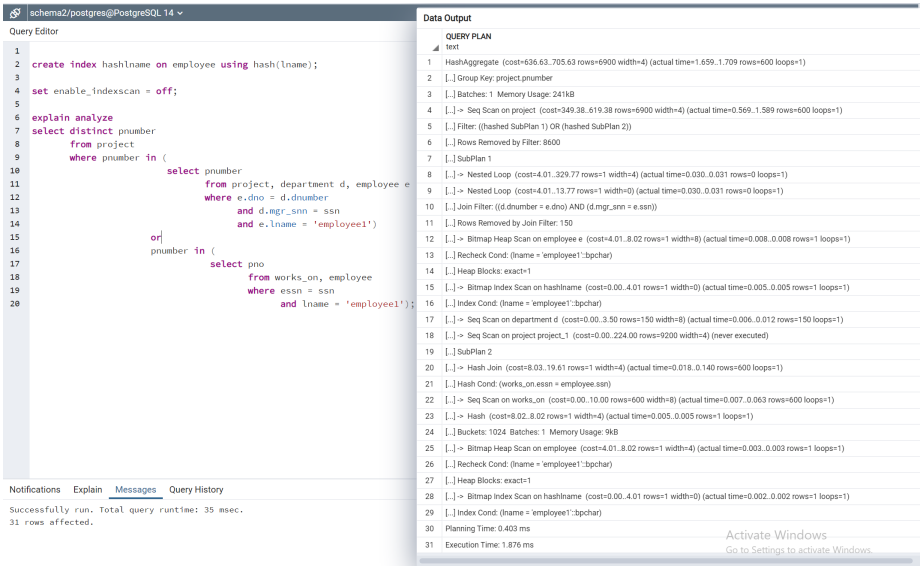
The Physical Plan :

The Same exact plan as the BTree plan but with the only difference that the index that has been used in the sub-queries 1 and 2 are the hash indices not a btree indices in this case, and for all the whole thing it used the primary key index on the project table (default by postgres btree) and applied the unique operator.

The Estimated Cost = 668 (Smaller than that of the Btree but with very small difference because the two plans hash/btree nearly have the same exact IO operations with difference of the index type (processing time of hash index is faster) but the processing time is not effective as the IO , and because both have nearly the same IO so they have nearly the same cost), The difference of this processing time may appear in the execution time that hash index plan ran in a time little bit smaller than that of the btree plan.

NOTE : The Engine decided to use the primary key index of the table project beside my hash indices that i have created , if i have disabled the primary key index (btree by default by postgresSQL) then it will do a seq scan over the project table applying the filter of sub plan 1 and sub plan 2 and then make an aggregation based on hash but of course this will increaese the cost of the whole query because index scan will be faster than the seqscan in that case because the table project has all the information we need so it is just on fetch from the index directly and making advantage of the primary key index that is sorted so just apply a unique operator on this sorted data and get faster result. The Cost after forcing it not to use the primary key index =

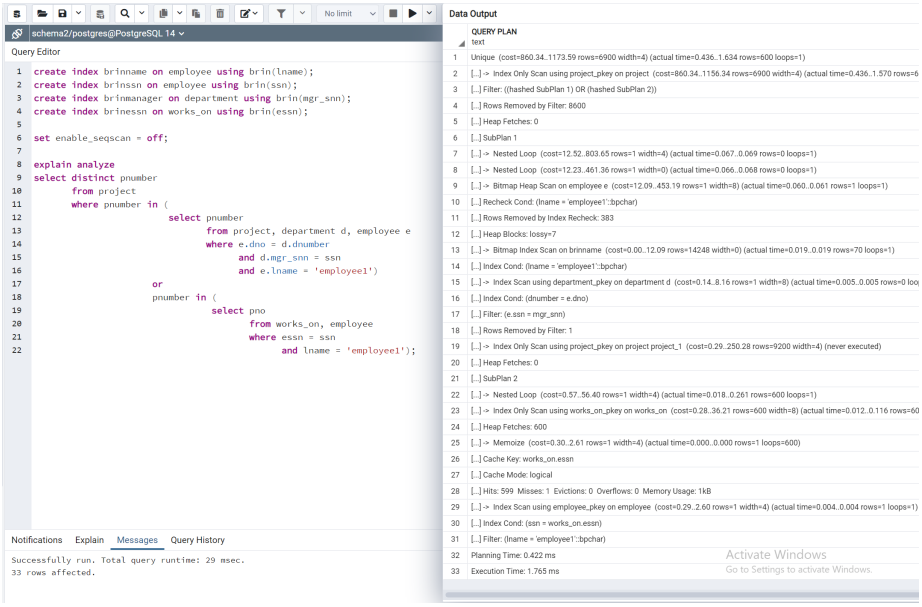
Here is the Plan after forcing even it not to use the primary key index and reside back to seq scan and now the last unique operator won't be able to use because data is not sorted then and it will have to make a hash aggregate just to be able to get the unqiue values with slower time.



Estimated Cost = 705 still better than the original query but now it is worse than the Btree index because the btree index allowed the use of sorted data and fastened the unique operator , NOTE Both hash and btree fastened the search for lname = 'employee1' but the btree gave more advantage for eliminating the duplicates so the advantage given by btree more that advantage given by hash.

4. The Query with BRIN Indexes

The Scenario goes as follows : after creating the BRIN indices the engine just ignores them and makes a normal seqscan over the tables and a nest loop join, after disabling the seqscan the engine started to use my brin indices but in addition some of the tables the engine used with it an index scan over their primary keys and make a memoize storing their intermediate results in the cache , but if i disabled the memoization then it will use more of the brin indices , but still having 3 tables yet to use the default btree index from postgresSQL. Here i report the case that i just disable only the seqscan and leave him use the default indices from itself because if i kept disabling flags it will end up with a cost that is not descriptive and i won't be able to compare the brin performance with before.



The Indexes i have created :

brinname : BRIN index created on table employee on the lname attribute.

brinssn : BRIN index created on table employee on attribute ssn.

brinmanager : BRIN index created on the table department using the attribute manager-ssn.

brinessn : BRIN index created on the table works-on using attribute essn.

The Physical Plan :

For Subquery1 it uses the brin index on the table employee to apply the condition lname='employee1' using bitmap index scan followed by a bitmap heap scan and made a nestloop join with the department table and since the seq scan is disabled it scanned the department table using an index scan using it's primary key index , then they made a nestloop blind with no conditions with the project table that also used an index scan because seq scan is already disabled. For SubQuery2 The Engine uses an index scan on table employee and memoizes the result in the cache to be used in the nestloop join with the table works-on . The Whole thing : Since the seqscan is disabled the engine will have no choice unless using the project primary key index with an index scan and the applying the unique operator.

The Estimated Cost = 1173 larger than the original query without any indices.(BRIN is not useful in this query and is just an overhead)

Why BRIN was not useful?

Because when you compare this BRIN plan with the original plan , the difference is in the join operations in the original plan it was using a seq scan then joining them , but here as we disabled seq scan to allow using the BRIN now it moved to use the index scan and here comes the issue, The brin yes improved the search for the condition lname='employee1' but also came with another problems that now iam using index scan for other tables and now for each record i will get it from the index first and then go and fetch from the disk again the original record from the table on disk and this will make a big overhead and downgrade in cost and performance , so here we had advantage faster search for lname='employee1' and downgrade when disabling the seq scan but the disadvantage effect with more than the advantage effect so overall the BRIN plan was of higher cost than the original plan. NOTE if you compare the BRIN with original query but after disabling the primary key indices you will find that the BRIN had an optimization in that query and here in this case because you lost many advantages from the primary key indices and just took a new advantage over the empty original plan for the exact value query using brin.

What if i have disabled the memoize?

The Engine would have used the BRIN of lname in the second subquery also but with worse performance because now we are not storing intermediate results in the cache.

What if i have disabled the index scan?

The Engine will change all tables and use a bitmap index scan on them using all the brin indices created unless one table only the project it will insist using the the primary key index and since we disabled it then the cost will be maximum (not descriptive). NOTE if i cahnged it to PURE PURE BRIN Trying to disable everything it will get me cost 10power 10 so this is the best to be able to compare with other plans and by intuition from the beginning the BRIN won't make any use in this query because we don't have selectivity here it is just a exact condition but the engine will insist on using the primary key indexes and try to make as the original query and cache it in the memory.

5. The Query with Mixed Indexes

The Execution Time is in the range 1.2 1.5 msec.

The Indexes i have created:

Bssn : Btree Index on table employee on the attribute ssn.

hashlname : Hash Index on table employee on the attribute lname.

Bproject : Btree Index on table Project on the attribute pnumber.

The Physical Plan :

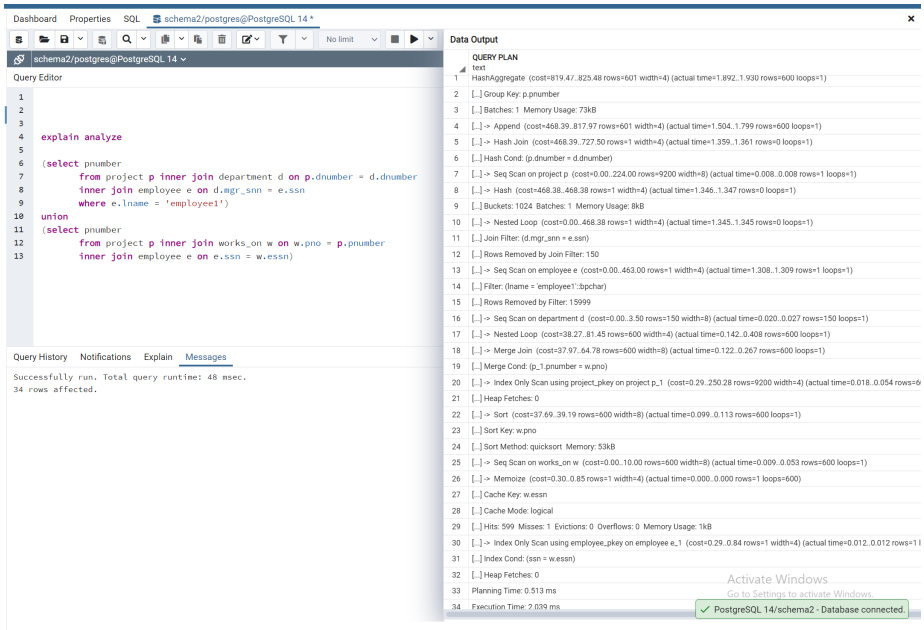
In Subquery1 : The Engines uses the hashindex created on the employee lname and make the join with the department using nestloop join and then blind join(cartesian product) with the project and the scan for department and project was seq scan for both because you need the whole record to make the join so no need to use index and go fetch once from index and once from table. In Subquery2 : The Engine uses also the hashlname index created on the lname attribute emoployee table and made a join with works-on nestloop join with seq scan over the works-on table with the same idea (seq scan is best for works-on exactly same reasoning above). The Whole thing : The Engine makes here the use of my Btree index created on the project and makes an index scan on the project making advantage that the data needed to be outputed is all in the index no need to fetch from disk so it made an index scan and applied the filters of subplan1 and subplan2 and then applied the unique operator at the end making use that the data is already sorted from the btree index.

The Estimated Cost = 668 Better from Btree alone and better from query on itself and better from PURE PURE Hash plan.

6. My Query

   Note : My New Query engine uses the primary key indices in it as the original query we are given , i haven"t disabled them because the perform ace will be downgraded very much an it won't be an optimized query so i left them and worked with them.

7. My New Query



The Execution Time Ranges between 1.8 to 2.3 msec.

The Estimated Cost = 825.48 smaller than the original query

The Physical Plan :

For the first half the engine made a seq scan on the table employee and table department and then made a nest loop join between them and hashed the result to make a hash inner join with the project table. For the Second half it uses the project primary key index and make a merge join with the with the works-on table after sorting it and all of that a nest loop join with the table employee after memoization and caching it in the memory for faster join. Lastly appending the result of both halves as the union of them

8. My Query with Btree Indexes

   The Execution Time ranges between 0.5 msec to 0.8 msec.

   The Indexes i have created :

   bName : Btree Index created on table employee using btree on lname attribute

   bMgrSNN : Btree index created on table department using the attribute manager-ssn.

   bPno : Btree index created on table works-on using the attribute pno.

   The Physical Plan :

   for the first part the engine made use of the bname index to have fast filtration over the condition lname = 'employee1' , and hashed the result to make a hash inner join with the department table to hash the result also to make a further hash join with the project table and this is for the first half. For the second half The engine used the project primary key index which is a btree index (no need to create it as it is by default) and made a merge join with the work-on table using the bPNO index that i created and nested loop it with the employee table that was memoized and cached after having an index scan on it. Last Appending the result of the two cases to make the union.

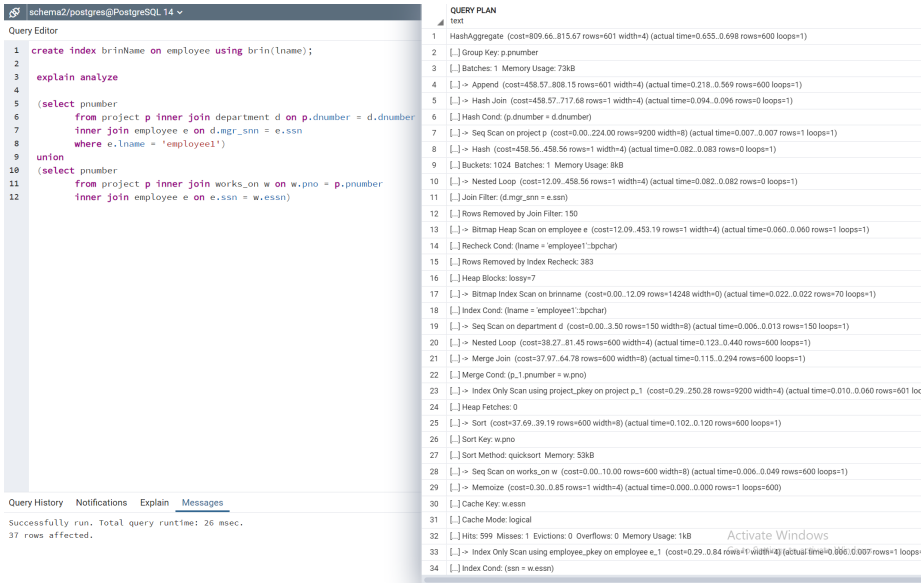
   The Estimated Cost = 362.40 which is much smaller than the original query with no indexes.

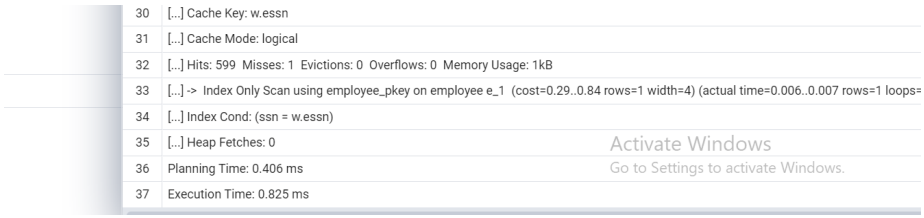
9. My Query with Hash Indexes

The Execution Time ranges between 2.2 to 2.4 msec

The Estimated Cost = 811.35

10. My Query with BRIN Indexes



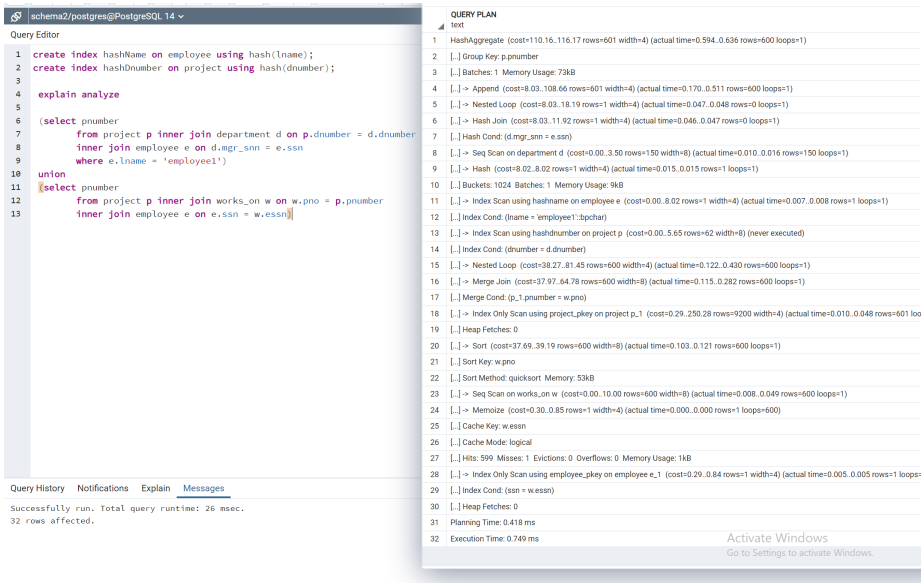


The Execution Time ranges between 0.7 to 0.9 msec.

The Physical Plan is the exact same on as the original plan but the only difference that is in the first part of the query the BRIN is used to make the filter employee.lname = 'employee1' using a bitmap index scan then followed by a bitmap heap scan and all the rest of the query is exactly the same.

The Estimated Cost = 808.15 less than the original query with no indexes.

11. My Query with Mixed Indexes



The Execution Time ranges between 0.6 to 0.8 msec.

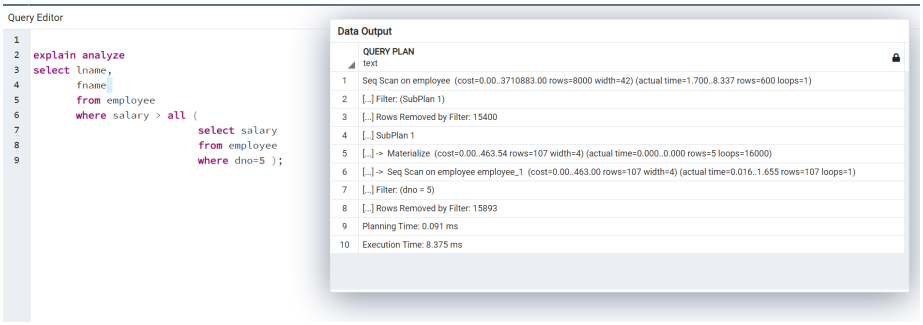
The Indexes i have created are on the screenshot and no flags are set off.

The Physical plan is the same as original but it used my indices that i have created here and by the way it still uses the primary key indices so it used also btree indices so it is a mix Indexes here.

The Estimated Cost = 116.17

2 Query 2

1. The Query without and Indexes



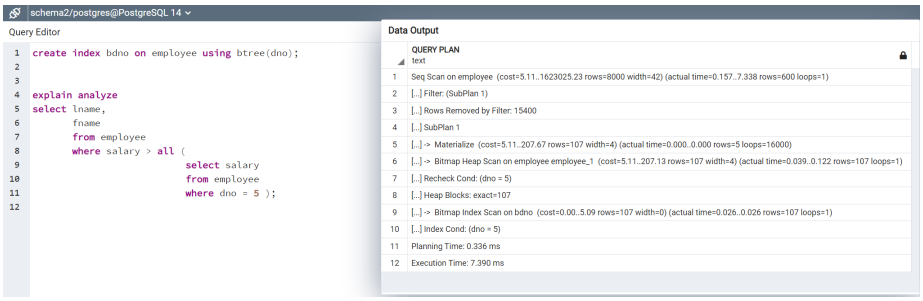
The Execution Time ranges between 8msec to 9msec.

The Physical Plan :

Let's start with the inner query , the engine made a seqscan filtering all the records just keeping that having department number = 5, and then it materialized that(keeping them in memory for further usage instead of making a rescan) , and then a seq scan on the outer query employee table and apply the filter of the inner query using the materialized intermediate results and comparing salary of current employee in the seqscan with the maximum salary from the materialized records in memory and the process of getting the maximum is not a big deal when we are having the records in the memory as long as we don't need to make an IO operation.

The Estimated Cost = 3710883

2. The Query with B+ Tree Indexes



The Execution time ranges between 6.8 msec to 7.4 msec.

The Indexes that i have used :

Bdno : Btree index created on the table employee the dno attribute.

The Physical Plan :

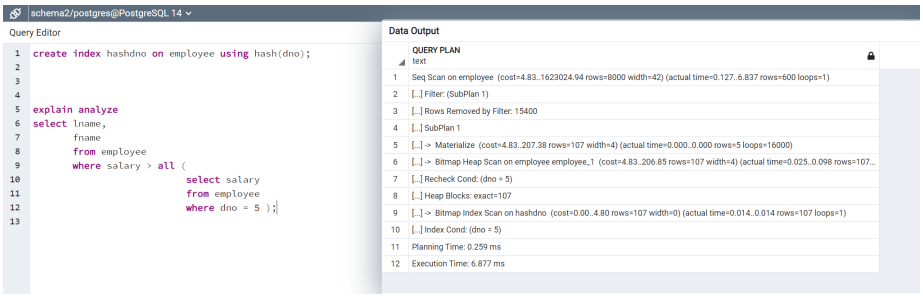
The Engine uses the same plan for the original query but with small difference that it has used the Btree in the filtration of the dno = 5 condition using a bitmap index scan followed by a bitmap heap scan.

The Estimated Cost = 1623025.23 (The Cost in smaller than the cost of the original query without the Btree index which indicating a slightly better performance)

Why an Index on the Salary won't be useful here?

Because if i have created an index on the salary first it will not make use of it in the part that it filters dno = 5 But WHY? because what will be stored in the index is just the salary so it will need to fetch row from index then from table on disk to compare the dno which doesn't exist in the index so it will choose a seq scan for the inner query , OK but what about the whole thing can it use the btree on salary and iterate over it to apply the sub plan? well, it can because the salary the one to be compared is the one stored in the index but if the condition is met then you will have to go to the disk to get fname and lname that are needed in the query output , another thing here that you don't have selectivity here you need to apply the sub plan over all the rows of the table employee then in either ways it will need to pass over the whole table so pass over the table with one fetch only is better then it will ignore it again and choose the seqscan.

3. The Query with Hash Indexes



The Execution Time ranges between 6.7 msec to 7.3 msec.

The Indexes i used :

hashdno : Hash Index created on the table employee on the dno attribute.

The Physical Plan :

The same as the original query also but with small difference only that the filter of dno = 5 is applied using the hash index with a bitmap scan followed by a bitmap heap scan in that case. which enhanced the performance of the original query.

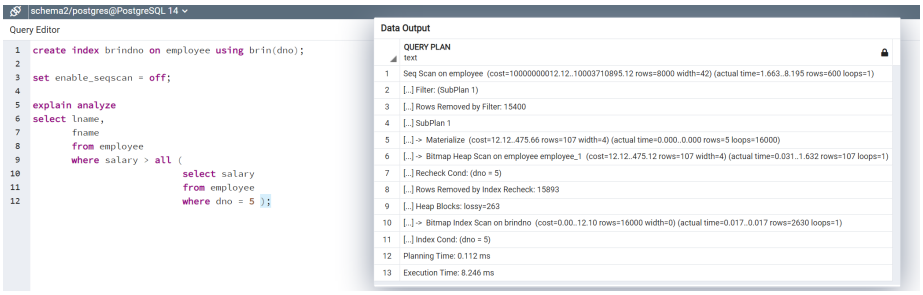
Why Hash index on the salary won't make optimization? –¿ Same argument as the one in the Btree case.

The Estimated Cost = 1623024.94 (Much Smaller than the cost of the original query and slightly smaller than the Btree index plan).

The Cost of Hash is nearly the same of The Cost of the Btree ? How and the Hash indexes are O(1) and Btree is O(log n) ?

Because in Postgress the time of creation of a hash index is bigger than that of btree (hash index takes more time to get created) , and because hash index has some problems with concurrency as it needs to deal with the deadlocks that can arise with the resizing of the hashtable, generally speaking the hash index gives nearly the same performance as a btree unless if a whole plan changes because of using a hash index example, a plan changes to hashing algorithms and changes completely when just using hash index but here in this case sample we have the same exact plan but just instead of using btree we used a hash so it is just 1 v 1 competition and the hash in 1 v 1 with Btree are the same, There is a simple test case at the end of the Schema 2 part proving this.

4. The Query with BRIN Indexes



The Execution Time ranges between 8 msec to 8.5 msec.

The Indexes i have created

brindno : BRIN index created on the table employee on the column dno.

The Engine used the same exact query but here it used a bitmap index scan then a bitmap heap scan to filter out the dno = 5, although the brin is not so good in the exact value queries but the ENGINE has insisted on using the seq scan as the same plan for the original query until i have disabled the seqscan on it so it used the brin index instead.

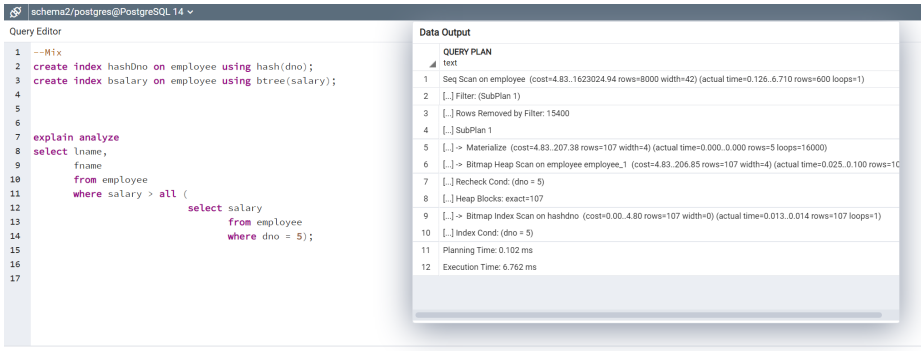
Why The Engine has ignored my BRIN and insisted on the Seq Scan?

The BRIN index is accessed by a bitmap index scan algorithm, but how does this work? it makes a scan on the index itself to indicate the rows that satisfy the condition or lies in a specific range and set them 1 in the bitmap created , and then it is followed by a bitmap heap scan , this is simply iterating over all the heaps that are set 1 in the bitmap to recheck over the condition, but why a recheck because everything is bucketized so bitmap index scan just set a whole bucket by 1 and the bitmap heap investigates inside this bucket to detect the rows satisfying the conditions. Here in this case a seq scan was better because you are just getting one scan only and also here we don't have very large range of values we are just having in the database 150 departments so the values of dno are just 150 so we don't have large range at all so seq scan here may be of same cost or even cheaper that's why my BRIN is ignored here.

The Estimated Cost = 10003710895.12 Because i have disabled the seqscan so it increased the estimated cost of the seqscan very very much which appears here, so the estimated cost here is not descriptive at all, and we can not compare the cost here with that of the original plan performance but from the time they are nearly equal to each other.

5. The Query with Mixed Indexes

For Me I think that The Best choice is just one BTree or just one Hash Index on the dno attribute of the employee table (They are the same) , we don't need more than that for this query, I have created more than one index (at least 2 to have the meaning of word mix satisfied but i want to use only one index :'( ).



The Execution Time ranges between 6.7 msec to 7.3 msec.

The Indexes that are created :

hashDno : Hash index created on the table employee on the attribute dno.

bsalary : Btree index created on the table employee on the salary attribute.

The Physical Plan : The Same exact as the part of the query using hash index.

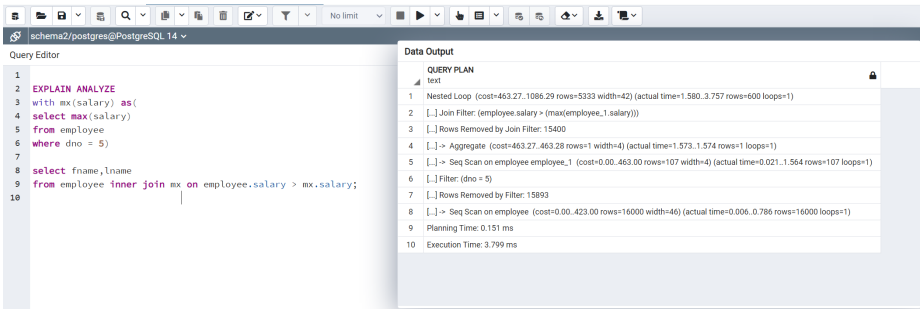
The Engine just used one of my two created indices but why it didn't use the second index?

The Second index is bsalary created on the salary attribute, and the original plane was seq scan on the employee and apply the filter of the inner query, the engine used my first index to optimize the inner query so all is good till now, but it didn't use the second for the outer table why? Because it has two options either use my index or seq scan, in either ways i need to pass by the whole table so that i can apply the filter on it and passing on the whole table using index is not effective at all because if i am now in the index and i have matched the condition then i have still to go to disk and fetch the actual row so overhead with more cost so the engine goes with the seqscan and ignore my second index.

The Estimated Cost = 1623024.94 smaller than the original plan and same as both btree plan and the hash plan.

NOTE NOTE NOTE : The Way The Engine executes this query generally speaking it has an inner query that is executed for every row of the outer table and check the inner query condition so for any mix indices i will choose it will use one of them for the inner query and whatever the second will be it will ignore it for the whole table that checks the inner query condition on it and it will just reside to the seq scan because in either ways it will need to look through the whole table so just make it with the seq scan having the row in hand is better than making it from and index and if condition met i go to get from disk(2 fetches) with more IO, this is logically speaking but even when trying to enforce it to use the second index it just refuses and makes the cost maximum value 10 power 10 because it also uses the seq scan that is already disabled.

6. My Optimized Query



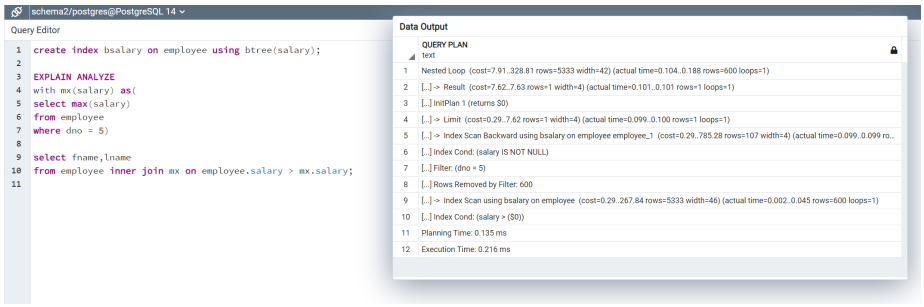
The Execution Time is around 3.2 msec to 4 msec.

The Physical Plan :

Let's start with the CTE (employee-1 relation in the screenshot) The engine made a seqscan over the table to find the maximum salary value and filtering keeping only those have dno = 5, then for the whole query it made a nested loop join between the employee table and the result of CTE getting the whole query result.

The Estimated Cost = 1086.29 Much smaller than the original query leading to better performance ¡3.

7. My Query with BTree Indexes SHOW DR WAEL



The Indexes i have created :

bsalary : Btree index on the employee table the salary attribute.

The Execution Time ranges between 0 to 0.4 msec. SHOW DR WAEL.

The Physical Plan :

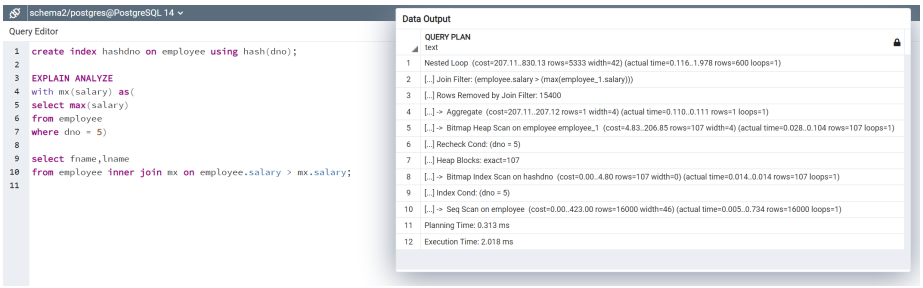
Let's start with the CTE the engine decided to calculate the maximum value in a smart way that i made a backward scan on the b+ tree index i created on the salary attribute to get the maximum value for the salary applying a filter while moving backward that the dno = 5, limiting the result to be 1 entry row this also has fastened the operation very much as it moves backward till it finds a non null salary for employee with dno = 5 then stops. why was is it an index scan because you will need to check the department number in addition to the salary so an index only scan won't do the job here it must be index scan. For the outer table it now made a nested loop join between result of CTE with employee,the scanning algorithm done on the employee is the index scan algorithm because we still have a btree on salary so the engine just used it and was able to get the values that apply with condition that employee.salary ¿¿ value of CTE which was limited to be one value , and then if a record in the index matched we go and get it from the disk from the original table.

If the Index was on the dno would it do the same optimization?

In fact no it won't, it will optimize the filtration condition keeping only the dno = 5 , but will still need to aggregate on maximizing salary and this will need to loop still over the table so the optimization in this case would be much smaller than using a btree over the salary.

The Estimated Cost = 328.81 smaller than my new Query without any indexes.

8. My Query with Hash Indexes



The Execution Time ranges between 1.9 msec to 2.5 msec.

The Indexes i have created :

hashdno : Hash index on employee table on the attribute dno.

The Physical Plan :

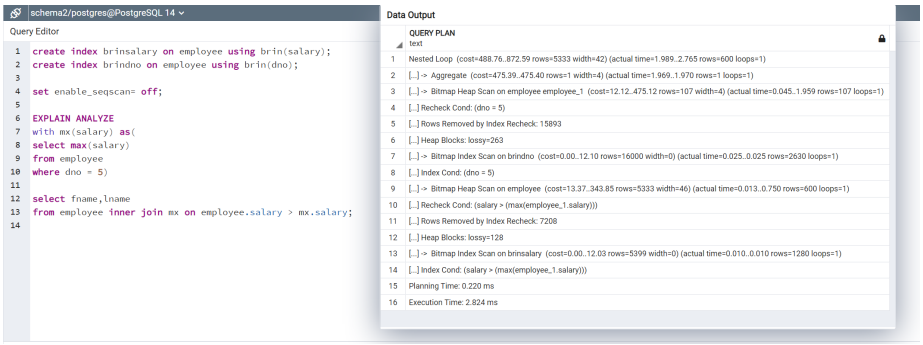
Let's start with the CTE the engine decided to make a bitmap index scan to find all the buckets that can match the condition dno = 5, then followed by a bitmap head scan to recheck within these buckets detected and then aggregation on these buckets getting maximum salary and nested loop join between CTE result and the employee table with joining condition employee.salary ¿¿ maximum salary(CTE result).

Why a Hash index on salary won't enhance performance?

Because the hash indices are not powerful in aggregate functions like the max in that case and most likely the engine will not use it at all as it can't get max from it unless it loops over the whole index and once it get max it will need to get the record from disk so it will be slower than a normal seqscan so the engine will prefer a normal seqscan in this scenario if the hash index was on the salary attribute, That is why the BTree plan is much powerful and more optimized than this hash plan, but still we are having this hash plan is much better than the original plan (benchmark).

The Estimated Cost = 830.13 (Smaller than the cost of the query with no indexes But greater than the B+ tree case).

9. My Query with BRIN Indexes



The Indexes i have created :

brinsalary : BRIN index created on employee table on salary attribute.

brindno : BRIN index created on employee table on dno attribute.

The Physical Plan

For the CTE , the Engine decided to use the BRINDNO index to enhance performance of the filtration of the condition dno = 5 using a bitmap index scan(locating buckets that meet condition) then followed by a bitmap heap scan(recheck within these buckets over the condition) then aggregating to find the maximum value of salary, then it made a nested loop join between the the CTE result and employee table and enhanced this join by using the brin on the salary attribute BRINSALARY using a bitmap scan then bitmap heap scan on condition that employee.salary ¿ max(salary).

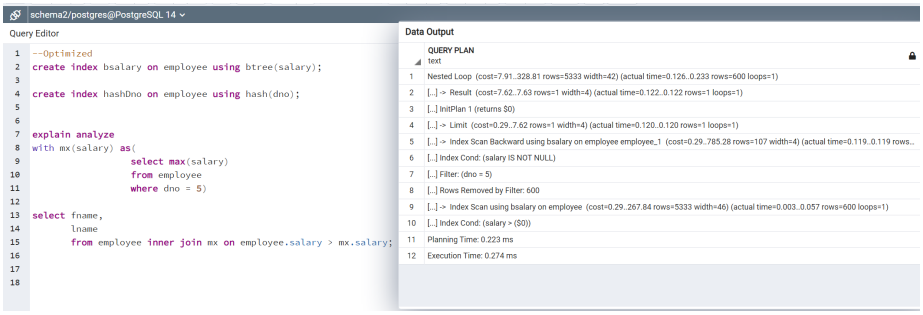
The Estimated Cost = 872.59 + 475.40 + 475.12 + 12.10 + 343.85 + 12.03 = 2191.09 (Smaller than my Query with no indexes but bigger than B+ and hash) Because it enhanced the filtration so increased performance of my query with no indexes but B+ and hash were having a better performance than this because they are better in aggregates and exacts.

What happens if i have used only one BRIN of them (Why i have disabled the seqscan)?

If one of them only is used then in this case a seq scan is used on the other column that we didn't use a brin on it, and from the engine point of view it is the best to use only the salary brin in the joining and leave the brin dno if the seq scan was enabled why is that?? because the brin is better in range queries with low selectivity but not good in the exact value queries and here it also will have more cost in case of the dno because you

will check the condition on the index and if met you still have to fetch it from disk to make the aggregate so it is better to just seqscan and have it hand and make the aggregate directly that's why i disabled the seqscan so that it can use the both brin indices created. IMPORTANT NOTE : it was better to use just one index which is the BRIN Salary why is that? Because the part of the query requiring calculating the max salary will have to pass through the table as whole in either ways to get the maximum so a direct seqscan is better than using an index and from the index we still need to go and fetch row from the original table which makes an overhead.

10. My Query with Mixed Indexes



The Execution Time ranges between 0 to 0.4 msec.

The Indexes that i have created :

hashDno : Hash Index created on table employee on the attribute dno

bsalary : BTree Index created on table Employee on the attribute salary .

The Physical Plan : The Same exact one in the BTree scenario because the engine used only the bslaary index and ignored the hashDno Index.

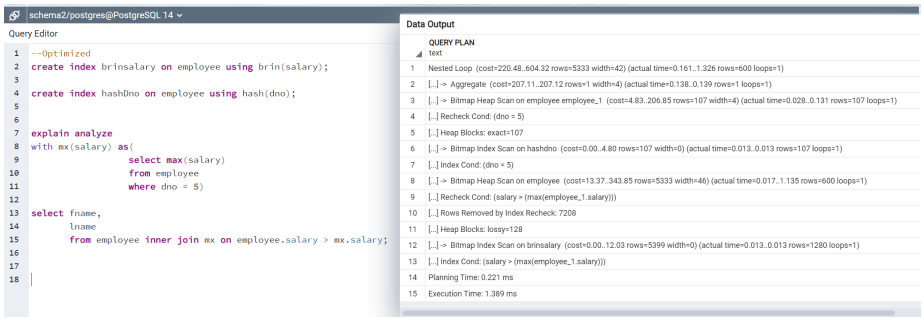
Why The Engine Ignored my second Index the Hash Index?

The Engine has Ignored it because the Query is divided into two tasks , one for the CTE getting the maximum value of salary for those dno = 5, and for this task it will be best to use the Btree index created on the salary as it will make the execution of super fat as explained in the Btree plan point, the second task is the joining between the result of the CTE and the employee , why it has used also a btree rather than the hash index in this task as well? because the joining condition is employee.salary ¿¿

result(CTE) which is a range search and it will be better to use a btree in this case because the hash indices are not efficient with the range queries as it will need to pass over the whole index and if one thing matched it will still have to go and fetch it from the disk so it will get two fetches in hash case but btree you don't need to pass through the whole index just small portion of it as the data is sorted and can answer range queries so the best case out of it is just ignoring the hash index and use the btree alone.

NOTE : The Engine will never use a mix indices if one of them is a btree it will always ignore the otter indexes if a btree exists and the reason is stated above and more clarified in the Btree plan explanation.

NOTE : The Only case to use Mix Indices(Two Different Types at same time) is making a brin index for the second task maathcing the join and hash index for the first task as it will be used for making the condition dno = 5 faster.



The Mix Indexes that are created :

brinSalary : BRIN index created on the salary attribute.

hashDno : Hash index created on the employee table on the dno attribute.

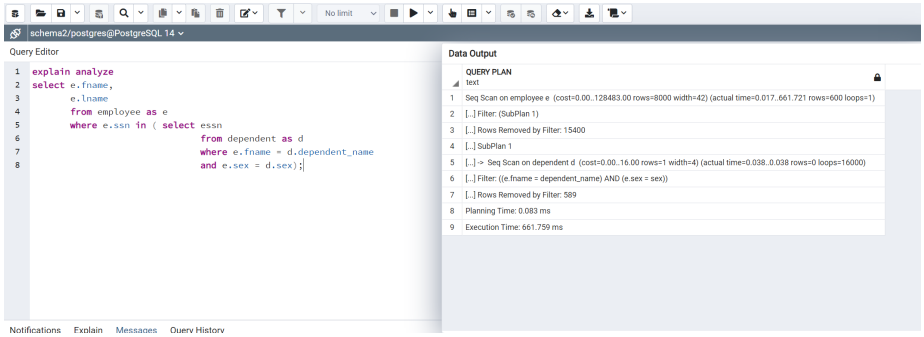
The Physical Plan :

Let's Start with the CTE the engine decided to use my index hashDno to meet the condition dno = 5 using a bitmap index scan followed by a bitmap heap scan and then makes a hash aggregation to get the maximum value for the salary for whose dno = 5, and then for the whole thing it will use a brin index over the table employee to apply the join with the result of the CTE , it uses a nestloop join and it used the BRIN directly in the joining because the join condition is that employee.salary ¿¿ result of the CTE , so it has sort of range query and selectivity and both are supported in the BRIN so the engine just directly used it.

The Estimated Cost = 604.32 (Better than my query with no indexes, better than hash indexes alone, but will never beat the btree because even in mix if a btree appears the engine will ignore whatever with it as shown in the screenshot before this).

3 Query 3

1. The Query with no Indexes



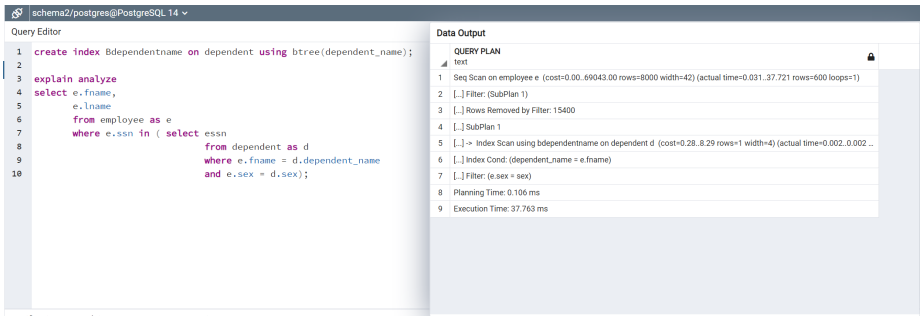
The Execution time is around 650 msec 670 msec.

The Physical Plan :

Let's start with the subplan, the engine has chosen to make a seq scan over the dependent relation and ably the filter of comparing the employee ssn with the dependent essn and employee sex with dependent sex , putting all together this subplan will be done for every tuple in the employee table , so the engine makes a seq scan over the employee table and for every row it applies the subplan from the sub-query (Generally speaking it can be thought as a nested loop in programming).

The Estimated Cost = 128483

2. The Query with B+ tree Indexes



The Execution time is around 35 msec 40 msec

The Indexes i have created :

Bdependentname : B+ tree index on the dependent table on the dependent-name attribute.

The Physical Plan :

The Engine has chosen to make a seq scan over the employee table and for every record of this table it applies a filter, this filter is the filter for the sub-plan(sub-query) in this filter it uses the B+ tree index created on the dependent-name with an Index scan access method(why? because you don't just want the dependent-name but you need more info from dependent relation so engine didn't use index only scan) and for every record matching the index scan condition (dependent-name = employee.fname) we fetch the record from the dependent table and compare the dependent-sex with employee.sex if they match then output the essn. So for every record in employee we apply this sub-plan filter using a seq scan over the outer table.

Why didn't we use a B+ tree index on the Dependent-sex attribute in addition to the created Btree?

Because if we had done that then the engine either will discard it or even if it used it the plan will be worse because you will apply the two indexes separately one will get the records satisfying(dependent-name = employee.fname) and the other satisfying the condition(dependent-sex = employee.sex) but here you will have to find a way to make intersection for both results and then apply this whole filter over each tuple in the employee table.

Why didn't we make a B+ tree index on dependent sex instead of the dependent name?

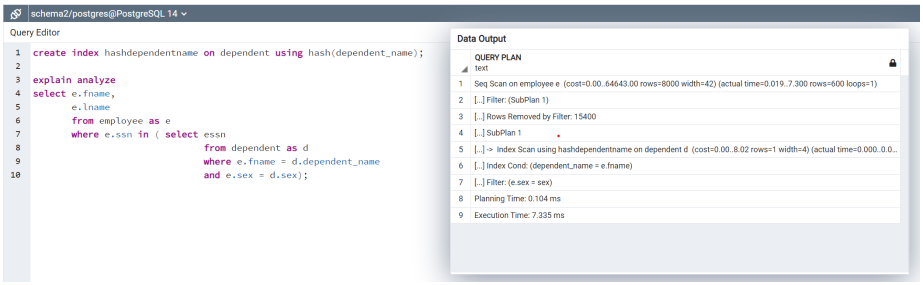
Because the attribute dependent-sex has one of only two values (M or F) so there is not a variety of values in this attribute so a choice of index on this attribute will be a poor one because most likely you will reach the first cell and seq scan for the remaining ones so it will be much slower than a B+ tree index on a dependent fname with variety of values leading to skipping many values when searching in the B+ tree with faster execution.

Why a Btree index on the employee outer table in the nested sub plan technique is not useful?

Because the engine applies the sub-query filter for every row or tuple in the employee table so in either ways it will need to pass through the whole table and apply the filter for each record in it , so an index won't be useful because if you have used an index then you will need to apply the filter of the subplan on the value in the index and if matched you will still need to go to the disk to fetch the original tuple from it's table so more IOs hence the engine will just ignore it and reside to the seq scan.

The Estimated Cost = 69043 (much smaller that the estimated cost of the original plan with no indices)

3. The Query with Hash Indexes



The Hash Indexes that i have created :

hashdependentname : Hash Index that is created on table dependent on the attribute dependent-name.

The Execution time ranges between 7 msec 8 msec

The Physical Plan :

The Engine makes a seq scan over the employee table and for each record it applies the filter of the sub-plan , the sub-query(sub-plan) the engine decided to use the hash index created which is called hashdependent-name and access it using index scan (we didn't use index only scan access method because we need some additional data with the dependent-name and we will see why) and applying index check condition(employee.fname = dependent-name) if the condition has been met we fetch the record from the table and further make a comparison between the dependent sex with sex(This is why we made an index scan so that we might need to get the sex of the dependent from the original relation). So the Plan is as follows a seq scan on employee table and for every entry we apply the sub-plan filter that was explained above.

Why we didn't use a hash index on dependent-sex beside the already created one?

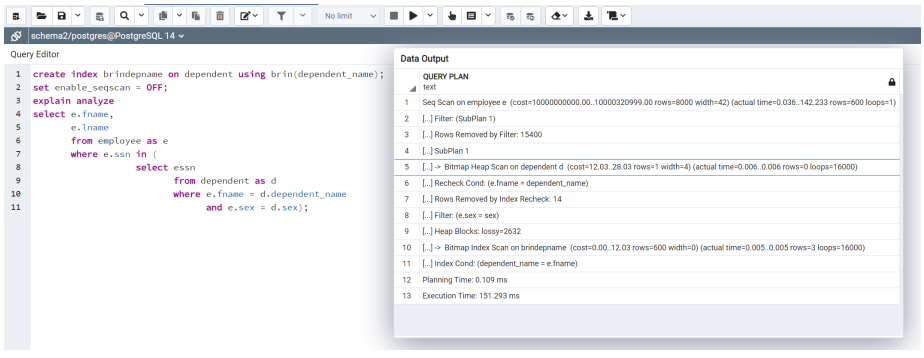
Because most likely the engine will discard it as it has an easy way that can make him the double check fast but even if it used it then we will get two result sets from the two indices and now we have to find a method to intersect them which i think will be slower so the engine just discards it.

Why we didn't use hash index on dependent-sex instead of the dependent-name?

Because the dependent-sex attribute has only two values either M or F, so making a hash index on it won't be efficient at all as we will have much much clustered buckets as we only have just two values and we will use the fast access advantage of the hash tables that we seek to make use of .

The Estimated Cost = 64643 (relatively smaller than B+ tree because the access of the hash indices is faster than B+ tree indices and much smaller than the original plan without any indices).

4. The Query with BRIN Indexes



The Index that i have created :

brindepname : BRIN index created on the table dependent on the attribute dependent-name.

The Physical Plan :

It is the same for the original query when it ran with no indexes , but the difference here that the engine has used my BRIN index in the filter of the sub-plan using a bitmap index scan to locate all the buckets or heaps that may meet the condition followed by a bitmap heap scan on the original table to make a recheck condition and get all the records that are really matching the condition (filtration condition employee.name = dependent-name) and if the condition has been met the record is fetched and we compare with the sex.

The Engine didn't use the BRIN index from the first time !

The Engine originally has chosen a seq scan for the filter of the sub-plan but it was forced then to use the BRIN when i disabled the seqscan.

The Estimated Cost = 10000320999 + 28.03 + 12.03 = 1*(1e10) THE COST IS NOT DESCRIPTIVE BECAUSE A SEQ SCAN IS FORCED TO BE OFF SO It's COST IS CHANGED TO MAXIMUM VALUE AND FOR THE OUTER TABLE THAT WE CHECK THE CONDITION ON EVERY ROW OF IT AND IT DOESN'T HAVE A REPLACEMENT OTHER THAN SEQ SCAN SO WE GOT THAT MAXIMUM COST.

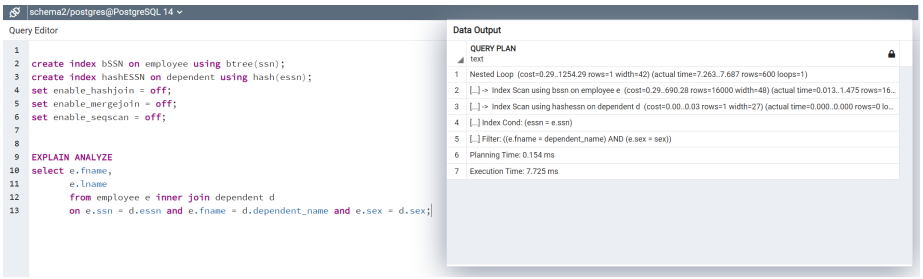
The Execution Time is 151.293 msec

When we disable a flag for the engine it can't remove it permanently it just increases it's cost for the engine to decrease the chance of choosing it , so that's why the cost of the seq scan on table employee is so huge but in fact that is because we have disabled the seq scan before.

The BRIN index has improved the performance of the query and decreased it's execution time , as the engine has used the BRIN index in comparing the condition of the filter of the sub-plan, although the BRIN is not much efficient in the exact value queries but still had an effect on the performance.

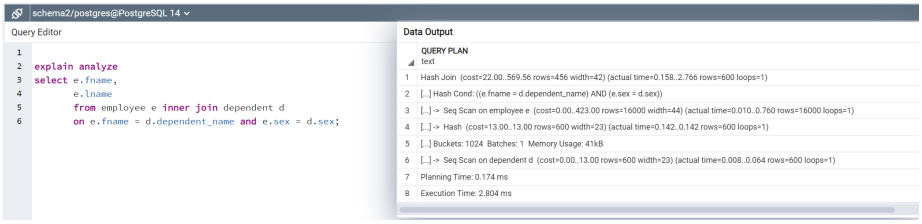
5. The Query with Mix Indexes

!!!!!!!!!!!! Don't Use mix



It Doesn't use Mix indices and just one index of the mix created because it just make a seq scan on the outer table and for every row of it it applies the filter of the subplan so it will just use one index inside the filter of the subplan and apply it for every row that is it, the filter even can not be done with more than one index because it is also made by a seq scan and just comparing we don't have even join so it is done also the filter using one index, one way of thinking is to disable the seqscan but also the table has resided to use the seqscan for the outer table because we have the row in hand instead of iterating over the whole table using an index which is not correct to use.

6. Optimized Query



The Execution Time ranges between 2.6 msec to 3.5 msec.

The Physical Plan :

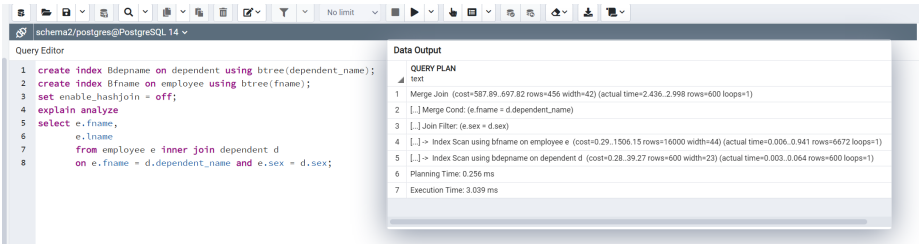
The Engine has chosen to make a seqscan on the dependent table and then hash it , and then make a seqscan over the employee relation and make a hash join between them. (Why a hash join not a semi hash join ? because you need to make a double condition one comparing sex and one comparing names and this will need to fetch the record from relation so in this case you can not apply a hash semi join that will not need any additional information).

The Estimated Cost = 569.56 much much smaller than the original query given to us.

Why my query has a better cost and optimization from the original one?

Because in my query the engine decided to make hashing for one table and then a hash join between the both tables, and hashing is so much faster than the original implementation because the original plan was using a seq scan on the table employee and for every row it applies the filter of the subplan and in the sub plan it uses a seq scan to filter the condition of the join here.

7. My Query with B+ tree Index



The Engine didn't use my indexes until i have disabled the hash join , that is because usually the hashing algorithms are faster than sorting algorithms as the hashing algorithms has a O(1) effect mostly.

The BTree Indexes i have created :

Bdepname : BTree on dependent table on attribute dependent-name.

Bfname : BTree on employee table created on the fname attribute.

The Execution Time ranges between 2.8 msec to 3.4 msec. the reported number is 3.04 msec.

The Physical Plan :

The Engine has made an index scan on both Btrees created on both tables employee and dependent , and then applied a merge join over them with condition that employee.fname = dependent.name , but why the engine has used an index scan? because it needs more data other than the fname and dependent-name that's why it didn't use index only scan , these additional data required for the another condition employee.sex = dependent.sex.

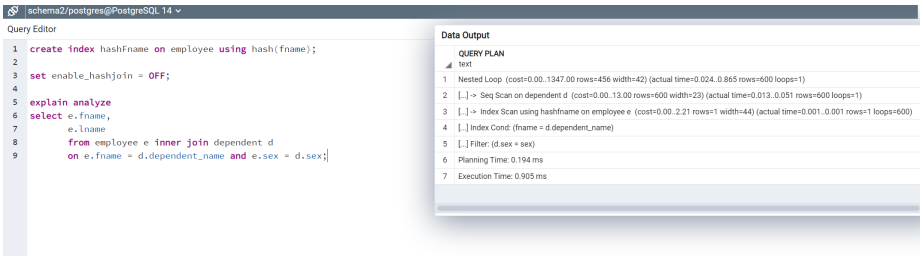
BTree Indexes on sex attribute is inefficient because it has only two unique values M or F so the data would be clustered with no variation and nearly will act as a seqscan with slower performance as in that case you will need more data so you will pay a fetch from the disk also.(More cost worse performance).

Why The Engine Ignored my BTree indices ?

The Engine ignored it because it has an alternative way which is seq scan the table and hash it using a hash table on the fly and this hash table kan be multidimensional one hashing on more than one value , but in the case of the merge join the btree index is created on one attribute only so we will join on one attribute only effectively but the other values will need a recheck on them still so the hashing method here was faster with lower IOs and because the case of the index it will be index scan requiring additional fetches from the disk to check over the additional attributes.

The Estimated Cost = 697.82 (Another reason why the engine has chosen the hash inner join algorithm because the estimated cost in case of this one was much larger than estimated cost on the normal execution and the engine chooses the smallest estimated cost that it estimates even if it is not exactly equal to the actual cost).

8. My Query with Hash Indexes (Very Important Note at the end of this point)



The Indexes i have created :

hashFname : Hash index on the table Employee on the attribute fname.

The Physical Plan :

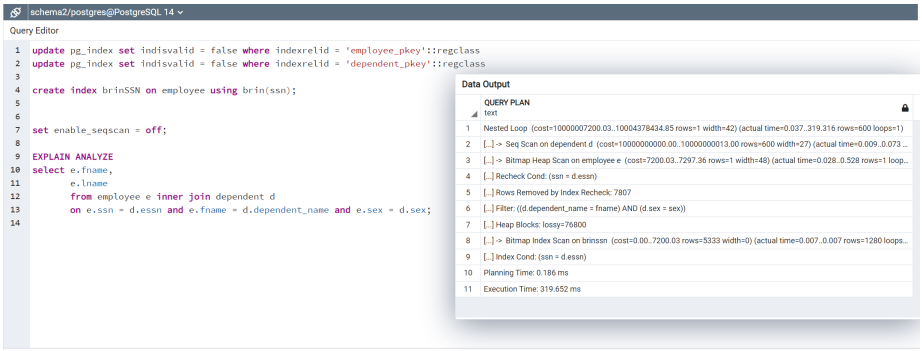
The engine decided to make a seqscan over the dependent table and then use the hash index made on the employee table , and then made a nested join such that it uses the hash index to match the tuples.

The Estimated Cost = 1347 More than My Query without any indices , The Hash Indexes didn't improve my query.

VERY IMPORTANT NOTE

The Engine first used a hashing algorithm then hash join , after disabling it the engine started to use my index, I THINK that the engine always search for a plan satisfying the order of join it is given so first it tried to use the hash join best choice for the order of join introduced in the query , then disabling it it will try the indices, when i used an index on the dependent using dependent-name attribute it discarded the hash index of the employee and has chosen the plan satisfying the order of join but when i have used the hash index on the employee ONLY without any other indices the engine didn't find any other way to make the join rather than using my index and changed the order of join and the dependent table became as the outer query and seqscan now became on the dependent and used the hash index on the employee for lookups (this will have extra fetch from the disk because when a lookup succeeds we want to get fname,lname from employee so then we want to get the record from the disk in that case) but still seqscan on dependent(600 rows) is much better than seqscan on employee(16000 rows) , so the best workout here in this point is using a hash index on the employee only to force engine to change the order of join and have a better performance.

9. My Query with BRIN Indexes



The Execution Time ranges between 318 msec to 320 msec.

The Indexes that i have created :

brinSSN : BRIN index on the table employee on the attribute ssn.

The Physical Plan :

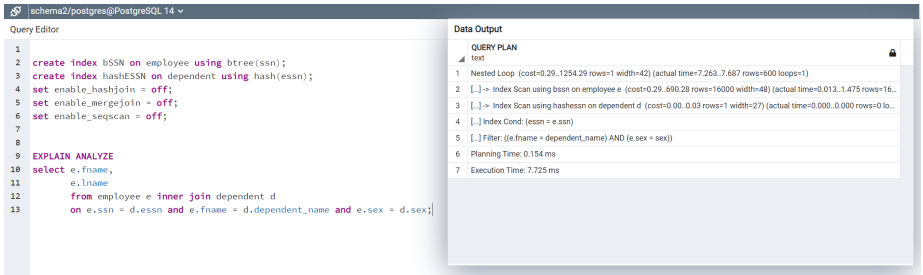
The Engine made a bitmap index scan in the employee table buckets and detect what of them that match the dependent.essn in the join operation and set to 1 in the bitmap and then makes a bitmap heap scan on the heaps that are detected to be 1 to make a recheck condition on the ssn = dependent.essn and make further comparisons for the fname= dependent-name and employee.sex = dependent.sex and a seq scan done on the dependent table to make the join with the employee.

I have disabled the seqscan and all the primary keys indexes that are created by default with PostgreSQL WHY?

Of course the original plan without any indexes is much better than this one , in the original plan it makes a seqscan hash and hash join so i disabled the seqscan and it was advantage because you have the row in hand but in case of brin you need to get the instance from the index then go and fetch it from the disk to make further comparisons for the sex and the name of employee and dependent, secondly when the seqscan is disabled the engine now decided to go and use the primary key indexes and just make a merge join on them and this is better because the primary key indexes are btree indexes created by default so it has used it to enhance the exact value query and to make advantage that it is already sorted so it can make a merge join easily that is why it favored this plan over my brin lastly i have disabled the primary key indexes and finally it used my indexes and made a nestloop join but in nest loop join one of them is a seqscan so it didn't have any other way except to use it and since the seqscan is already set off before then we got a cost maximum value.

The Estimated Cost = 10004378434.85 Not Descriptive because the use of the banned seqscan.

10. My Query with Mixed Indices



The Execution Time ranges between 7 msec to 8 msec.

The Indexes i have created :

bSSN : btree index on table employee using the ssn attribute

hashESSN : hash index on table dependent on the attribute essn.

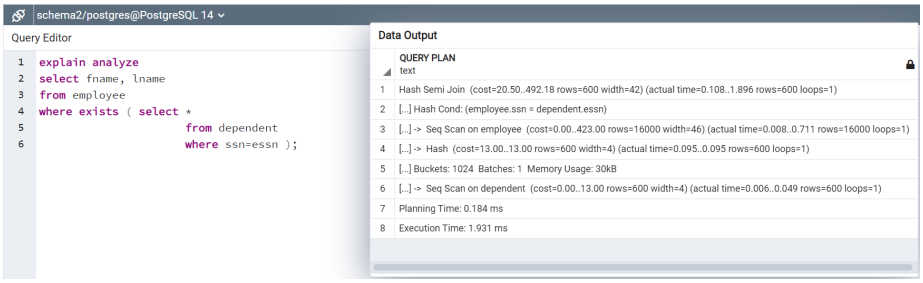
The Scenario is as follows :

First of all the Engine used the same plan as there is no indices and ignored all my indices and that is because he creates a hash-table on the fly on of the tables and makes a hash-join which is of the lowest cost (BEST CASE), but i have disabled the hashjoin so it has shifted it's direction to merge join and sorting algorithms and here it has used my bSSN btree index and ignored the other hash index i created and just used the primary key index of the dependent table to have data sorted and able to make a mergejoin (SECOND BEST CASE), but i have disabled the merge join so that it has now no choice rather than to take my indices and make a nestloop join between them.

The Estimated Cost = 1254.29 Larger than my Query with no indices which is logic because the plan of the query with no indices is the best over them all.

4 Query 4

1. The Query with no Indexes



The Execution time ranges in the range 1.8 msec 2.4 msec

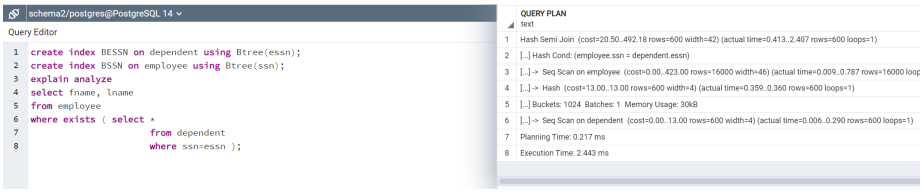
The Physical Plan :

The engine has chosen to transform this query to a simple join problem , with the outer relation of the join to be the employee and the inner relation of the join being the dependent , so the engine made a seq scan on the dependent table and hashed it, then started the semi hash join operation,

it started a seq scan over the the employee table and for every value it searches in the hashed dependent if it is found. So the whole query engine changed to a simple join using the semi hash technique (Results are taken only from the left table of the join we don't care about the right one we only just check if there is a match) with hashing condition employee.ssn = dependent.essn .

The Estimated Cost = 492.18

2. The Query with B+ tree Indexes

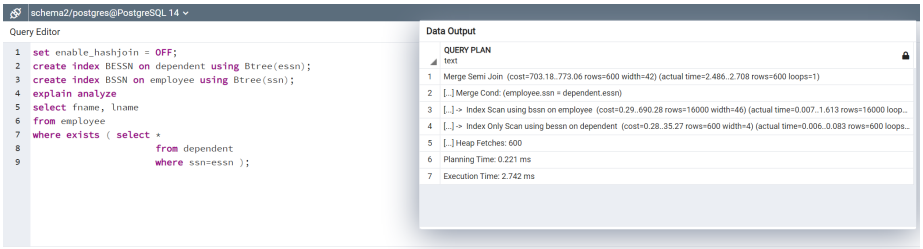


I have created 2 B+ tree Indexes :

BSSN : B+ tree on employee on the attribute ssn.

BESSN : B+ tree on dependent table on the attribute essn.

The Engine didn't use the B+ tree indexes that i have created !!! BUT WHY?



The Only way to force the Engine to use my B+ indexes is by disabling the hash join , so the engine don't have the option of making a hash join, so The engine decided to make Index only scan (It doesn't need to get data from the original table just fetching data from the index only) on table dependent(this makes sense because the output has nothing to do with the dependent table so we don't need to fetch from the original table just the essn from the B+ index will do the job) and made also an INDEX SCAN on table employee(here we need the data also from the table as ssn

only won't be enough because we have fname,lname in the output from the employee table) and then we had a semi merge join making advantage that the data is already sorted because the use of the btree indices and finally it is a semi merge join because we only get the output from one side of it which is the employee only.

Why the Engine insisting on the Hash join here and didn't use my Indexes?

The Engine favored the seq scan and hash join because in scanning the employee table in the first case we just had a seq scan on both tables hashed one of them and the other was used as outer relation in the join , in the second case the engine will need to access the dependent table it accessed it using index only scan and accessed the employee table with an index scan and here is the difference the index scan in that case will take more time because you will need to fetch from the index first then go and fetch from the original table and the seq scan of the first case would be faster in that case so the whole merge join in this case was slower than the hash join specifically that the employee table was quite large 16000 entries , so searching in the index is efficient yes bet yet you still have to go to the original table on disk and size is LARGE.

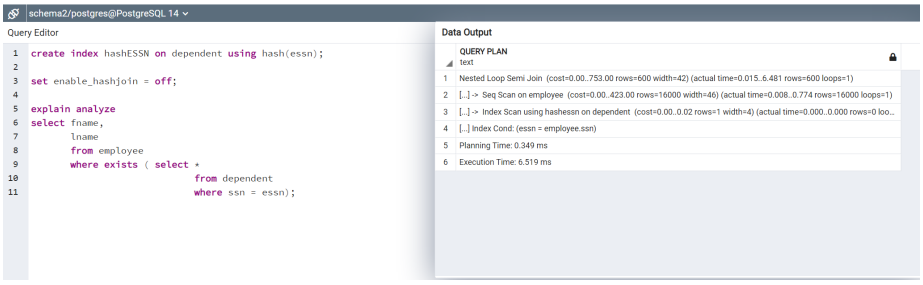
The Estimated Cost = 773.06

This Estimated Cost is bigger than the Cost of the First case Plan using the hash semi join which was the exact one with the query without any indices.

Conclusion :

The B+ index here is not useful as it is had a bit slower plan.

3. The Query with Hashing Indexes



The Execution Time ranges between 6.2 msec to 7 msec.

The Indexes that i have created :

hashESSN : hash index created on table dependent using the attribute essn.

The Physical Plan :

The Engine makes a seq scan over the table employee and makes a nestloop join and for every row inside the table it searches inside the hash index created (hashESSN) for the value that satisfies the joining condition and use the advantage of the already existing index.

Why is the query ignoring my indexes ?

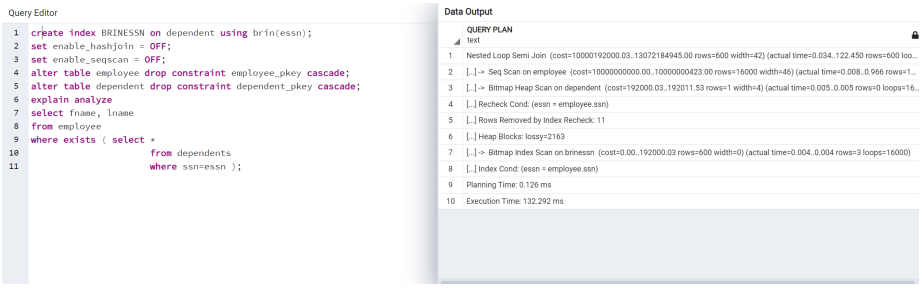
Because it originally makes a seq scan and then hash the dependent table and makes a hash full join which is much faster because the use of the hash-table on the fly, better than using the index why is that? because using the hash index will require you to get the record from the index first and then go to the disk and fetch the row from the table and this is an overhead as in the original plan you just have the row in your hand and join it directly , but in the index you still have go to the original table first and then make the join. more IOs more cost more time.

The Estimated Cost = 753 more than that of the original query.

4. The Query with BRIN Indexes

I think the BRIN Index is not useful at all in this query because BRIN is not that much powerful in the exact value queries and also here we don't have a range selectivity query, so i think that the original plan that was chosen to the Query with no indexes will be much better and faster.

The Engine Doesn't use BRIN index at all on the normal state, i disabled the hash-join and still doesn't use the BRIN used the seq scan then sorting and merge join and this is slower than the original plan with no indexes , then i disabled the seqscan and then it used the primary key Indexes that are created by postgress by default and they are Btrees , then i dropped the primary key constraint of the tables and finally it used my BRIN index.(As expected it won't be useful at all).



Execution Time around 130 msec 140 msec

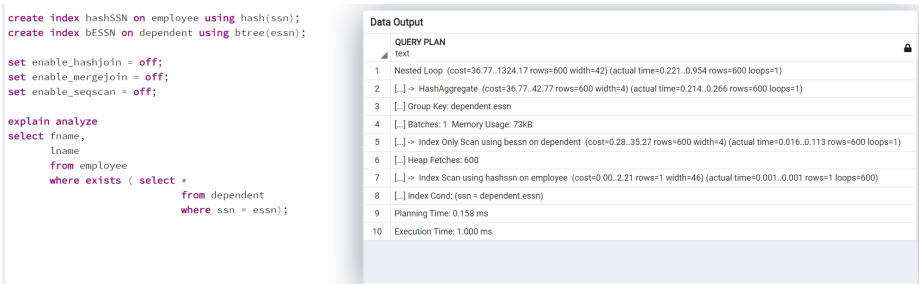
Here when the engine used the BRIN index created on the dependent table , it made a bitmap scan on this index (BRINESSN index) to locate the pages satisfying the joining condition (employee.ssn = dependent.essn) and then it made a bitmap heap scan on these located blocks as they have multiple rows so it have to check inside these blocks (recheck condition) , then it made a seq scan on the employee table !!(i disabled it already!) and then a nested loop semi join between the relations (semi join means we only get result from the left table of the join and we don't care about the right table of the join).

From this plan one can notice that the performance has been downgraded too much and the cost increased in a very bad way.

The Estimated Cost = 13072184945

The Difference in the Estimated Cost is so huge because here the engine insisted on the seq scan although it is disabled before.

5. The Query with Mixed Indexes



The Execution Time ranges between 0.8 msec to 1.5 msec.

The Physical Plan :

The Engine makes a nestloop join between the dependent and the employee tables , it uses the btree index created on the dependent table and makes an index scan over it and then aggregates it(group by the dependent essn) and it used index only scan because it doesn't need any additional information as the essn (part of the joining condition) is already stored in the index, and for the other table it uses index scan over the employee table because it has some attributes that are needed in further selections as the fname and lname so it is index scan that will need further disk access.

The Scenario has gone as follows :

First of all the Engine used a hash inner join between the two tables , and it ignored both of my indices why? because making a hashtable on the fly is much faster than making index scan that may require additional IOs from the disk as the case here, then it went to use the btree i created with another primary key index which is also a btree and make a merge join and this is also faster than my indices because the btrees are sorted and it just will make a join based on that the data is sorted which will be faster, then i went and disabled the merge join to make it use my indices but still the engine used a seq scan with materialization to store the intermediate results in memory for further rescans and this is also better than my indices because here we get the advantage of using the memory in storing intermediate results and would have faster access with less IOs and also because in seq scan we have the whole row from the original table in our hand but a case of an index you may not have all the information that you need in the value that is stored in the index and then you will need to go and fetch it from the disk and this will give you more IOs specially when the table size is large you will have many more IOs.

The Estimated Cost = 1324.17 much bigger than the original query with no indices created on it.

6. My Alternative Query : THE ORIGINAL QUERY HAS NO OPTIMIZATION

Why The Query provided has no optimization ?

Because the Engine itself has interpreted it as a join and not as a nested query and directly used a hashtable on the fly and hash inner join for the joining between the two tables , so any other query will not pass this

one because this one already has the best performance as it uses hashing algorithms with O(1). Here is another alternate query that runs in the same exact cost and same execution time and also the engine interprets it the same and uses the same physical plan as the original query.

My Alternate Query :

```
explain analyze
select fname,
            lname
            from employee e inner join dependent d on e.ssn = d.essn;
```

Data Output

	QUERY PLAN text
1	Hash Join (cost=20.50..509.50 rows=600 width=42) (actual time=0.115..1.916 rows=600 loops=1)
2	[...] Hash Cond: (e.ssn = d.essn)
3	[...] -> Seq Scan on employee e (cost=0.00..423.00 rows=16000 width=46) (actual time=0.009..0.716 rows=16000 loops=1)
4	[...] -> Hash (cost=13.00..13.00 rows=600 width=4) (actual time=0.101..0.102 rows=600 loops=1)
5	[...] Buckets: 1024 Batches: 1 Memory Usage: 30kB
6	[...] -> Seq Scan on dependent d (cost=0.00..13.00 rows=600 width=4) (actual time=0.006..0.052 rows=600 loops=1)
7	Planning Time: 0.170 ms
8	Execution Time: 1.953 ms

```
explain analyze
select fname,
            lname
            from employee e, dependent d
            where e.ssn = d.essn;
```

	QUERY PLAN text
1	Hash Join (cost=20.50..509.50 rows=600 width=42) (actual time=0.107..2.058 rows=600 loops=1)
2	[...] Hash Cond: (e.ssn = d.essn)
3	[...] -> Seq Scan on employee e (cost=0.00..423.00 rows=16000 width=46) (actual time=0.008..0.795 rows=16000 loops=1)
4	[...] -> Hash (cost=13.00..13.00 rows=600 width=4) (actual time=0.094..0.095 rows=600 loops=1)
5	[...] Buckets: 1024 Batches: 1 Memory Usage: 30kB
6	[...] -> Seq Scan on dependent d (cost=0.00..13.00 rows=600 width=4) (actual time=0.006..0.048 rows=600 loops=1)
7	Planning Time: 0.152 ms
8	Execution Time: 2.091 ms

```
explain analyze
select fname,
            lname
            from employee
            where ssn in ( select essn
                           from dependent);
```

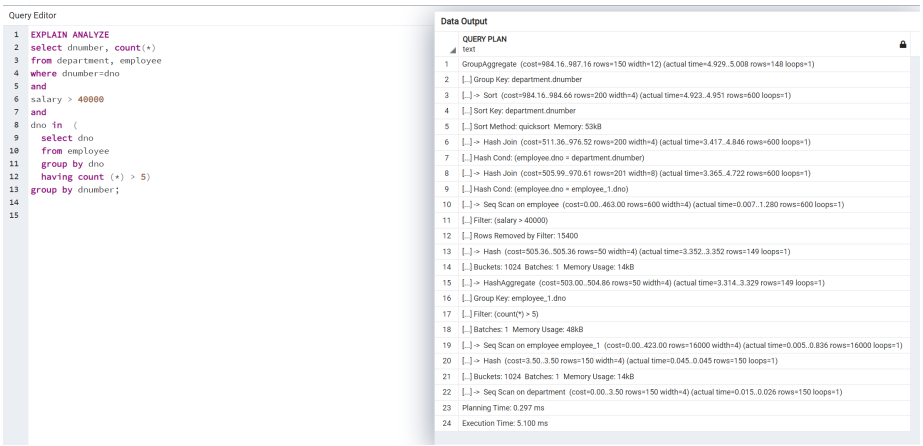
	QUERY PLAN text
1	Hash Semi Join (cost=20.50..492.18 rows=600 width=42) (actual time=0.113..2.171 rows=600 loops=1)
2	[...] Hash Cond: (employee.ssn = dependent.essn)
3	[...] -> Seq Scan on employee (cost=0.00..423.00 rows=16000 width=46) (actual time=0.008..0.811 rows=16000 loops=1)
4	[...] -> Hash (cost=13.00..13.00 rows=600 width=4) (actual time=0.099..0.099 rows=600 loops=1)
5	[...] Buckets: 1024 Batches: 1 Memory Usage: 30kB
6	[...] -> Seq Scan on dependent (cost=0.00..13.00 rows=600 width=4) (actual time=0.006..0.050 rows=600 loops=1)
7	Planning Time: 0.170 ms
8	Execution Time: 2.206 ms

All The Previous Queries run with the same cost as the Given Query and all of them has the cost 492.18 except the one with the cartesian product

because it applies the condition after the joining other wise all the others are the same also all of them gets the same exactly physical plan hence in adding the indices it will be the exactly the same as the original query so no need to repeat it in writing more.

5 Query 5

1. The Query with no Indexes



The Execution time is around 5.1 msec and has small range it changes approximately around 4.8 5.1 msec.

The Physical Plan :

Let's start with the sub-query first, The Engine has made sequential scan on the employee table to make the Hash aggregation grouping by the dno attribute from employee table with key dno, and then filtered count(*) greater than 5 and then hashed the result this is for the sub-query. For the outer query we made a sequential scan for the employee table and filtered out all rows with salary ¿ 40000 , after then we made a hash join with the Hashed result of the sub-query. Now with the department table we made a sequential scan to hash it so now we also have the department relation hashed and then we make hash join with the result of(sub-query with employee table) with condition employee.dno = department.dnumber . Now what is all left is the final aggregation of the whole result the engine has chosen to do aggregation based on sorting so it has sorted the result from the last hash and then made the aggregation based on aggregation key department.dnumber.

The Estimated Cost:

The Estimated cost = 987.23

2. The Query with B+ Indexes

Query Editor

```
1  select indexdef from pg_indexes where tablename = 'employee';
```

Data Output

	indexdef text
1	CREATE UNIQUE INDEX employee_pkey ON public.employee USING btree (ssn)
2	CREATE INDEX bsalary ON public.employee USING btree (salary)
3	CREATE INDEX employee_dno_idx ON public.employee USING btree (dno)

The indexes that i have used a B+ tree index on the Employee table

Bsalary : B+ tree on salary attribute of employee table

Bdno : B+ tree on dno attribute of the employee table

schema2/postgres@PostgreSQL 14

Query Editor

```
1  create index bsalary on employee using btree(salary);
2  create index bDno on employee using btree(dno);
3  explain analyze
4  select dnumber,
5         count(*)
6         from department , employee
7         where dnumber = dno
8         and salary > 40000
9         and dno in (
10                  select dno
11                         from employee
12                         group by dno
13                         having count(*) > 5)
14          group by dnumber;
15
16
17
18
19
20
21
22
23
24
25
26
27
```

Notifications Explain Messages Query History

Data Output

	QUERY PLAN text
1	HashAggregate (cost=453.88..455.38 rows=150 width=12) (actual time=2.119..2.133 rows=148 loops=1)
2	[...] Group Key: department.dnumber
3	[...] Batches: 1 Memory Usage: 48kB
4	[...] -> Hash Join (cost=392.93..452.87 rows=203 width=4) (actual time=1.802..2.041 rows=600 loops=1)
5	[...] Hash Cond: (employee.dno = department.dnumber)
6	[...] -> Hash Join (cost=387.56..446.95 rows=204 width=8) (actual time=1.755..1.923 rows=600 loops=1)
7	[...] Hash Cond: (employee.dno = employee_1.dno)
8	[...] -> Index Scan using bsalary on employee (cost=0.29..58.03 rows=609 width=4) (actual time=0.007..0.067 rows=600 loops=1)
9	[...] Index Cond: (salary > 40000)
10	[...] -> Hash (cost=386.65..386.65 rows=50 width=4) (actual time=1.742..1.742 rows=149 loops=1)
11	[...] Buckets: 1024 Batches: 1 Memory Usage: 14kB
12	[...] -> GroupAggregate (cost=0.29..386.15 rows=50 width=4) (actual time=0.040..1.726 rows=149 loops=1)
13	[...] Group Key: employee_1.dno
14	[...] Filter: (count(*) > 5)
15	[...] -> Index Only Scan using bdno on employee employee_1 (cost=0.29..304.29 rows=16000 width=4) (actual time=0.014..0.857 ro...
16	[...] Heap Fetches: 0
17	[...] -> Hash (cost=3.50..3.50 rows=150 width=4) (actual time=0.039..0.040 rows=150 loops=1)
18	[...] Buckets: 1024 Batches: 1 Memory Usage: 14kB
19	[...] -> Seq Scan on department (cost=0.00..3.50 rows=150 width=4) (actual time=0.010..0.019 rows=150 loops=1)
20	Planning Time: 0.332 ms
21	Execution Time: 2.221 ms

The Execution Time is approximately in the range 1.8 2.4 msec

The Physical Plan :

Let's start with the sub-query , in the sub-query the engine used the B+ tree on the employee-dno-idx and made an index scan over the relation employee and made aggregation based on sorting taking the advantage that data is sorted in the index and has made the filtration of count(*)

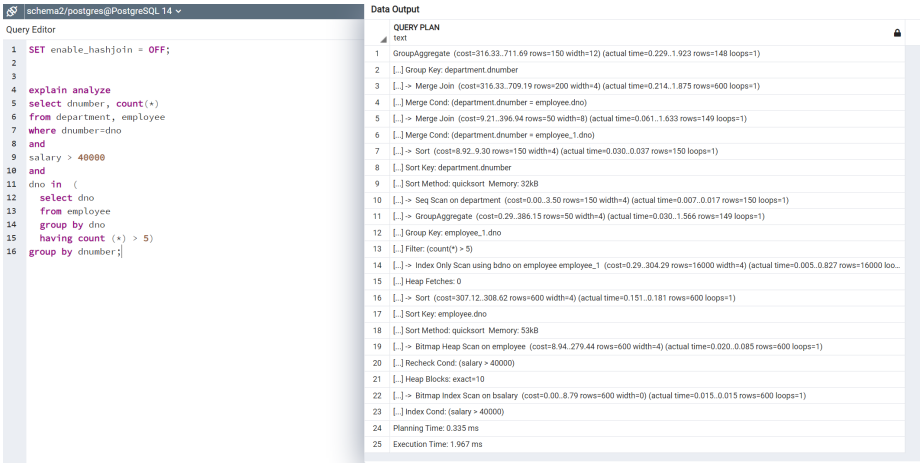
greater than 5 then hashed it. Now with the outer query for the employee relation the engine also has used the B+ tree built on the salary attribute and made an index scan to filter out all salaries more than 40000 this is so much faster than the seq scan that was used before, and the we made inner hash join between the sub-query result and the employee table after filtration. Now for the department relation we made a seq scan to hash the table and now we hashed it. Then we made a hash join between the department and the result of hash joining sub-query with employee and lastly we made aggregation of the whole query and it was a hash based aggregation with aggregation key the department.dnumber.

Why Don't make a B+ tree on department.dnumber ?

I didn't have to because in either way the department table needs to be read to get hashed so in a way or another i will need to read the whole table to hash it so it won't give me advantage in this case and just making overhead.

The Estimated Cost = 455 which is smaller than the original query with no indexes

Another Implementation with B+ Trees After disabling Hash Join



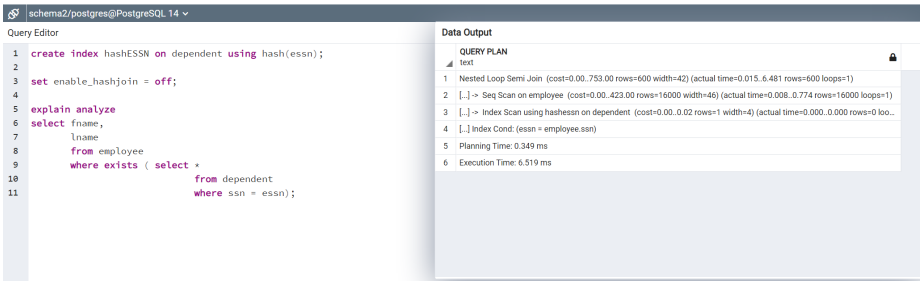
Here I have disabled the Hash Join , so the Engine didn't have the choice of hashing any more and started using sorting algorithms.

Let's start with the inner query the engine has used the Btree created on the dno attribute of employee relation and used it in the aggregation with key of dno and filtered the count(*) greater than 5, and made a seq scan on the department table and sorted it (It didn't use the BTree of the primary key of the department table because it is just a table of 150 entries so index will be slower to read it then get the data from the table itself to memory) and sorted the department relation in memory. then it made a merge join between the sub-query and the department table, regarding the employee table the engine used the BTree on salary attribute to have a faster filtration removing salary more than 40000 using bitmap index scan then sorted it in memory using quick sort and then merge join with the result of joining (sub-query and department) and finally the final group aggregation based on sorting on the key dnumber.

The Execution time is nearly 1.8 ms 2.2 ms as the first implementation using the BTrees as well.

The Estimated Cost = 709.19 Still Better than original query but worse than B+ tree with hash join plan.

3. The Query with Hash Indexes



The Execution Time ranges between 5.2 msec to 5.6 msec.

The Physical Plan :

For the Sub Query the engine decided to make a seq scan over the employee table and make aggregation based on hashing with aggregate key the dno to make the group by command, and then hashed the result(grouped by dno) and then it has made a hash inner join with the employee table itself (The grouped by version with the employee) so that he can get the number of the employees making more than 40000 in the department and lastly it made a hash inner join with the department but here instead of making a seq scan on the department table it made an index scan over it making use

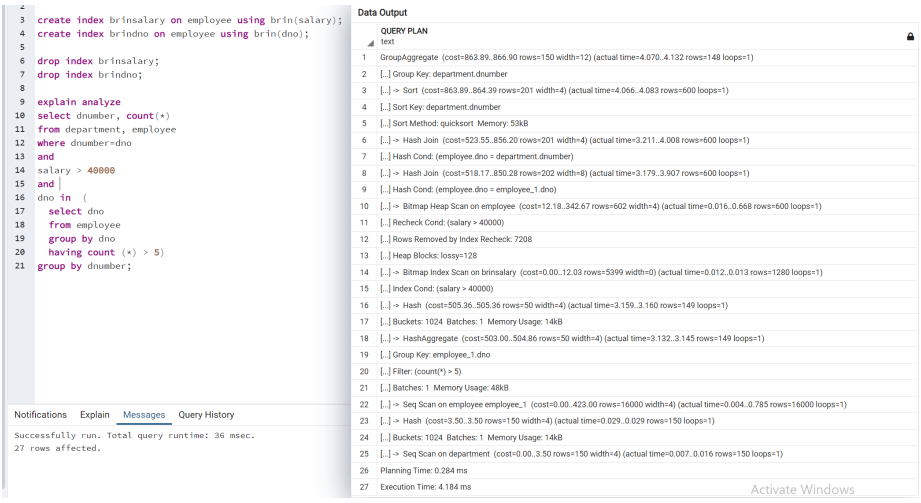
of the hash index that is present here, after all these joins now it sorted the result and made an aggregation based on sorting and used sorting here as it sorted the result in memory using quick sort technique which is fast as it is done in memory without much IOs required.

The Estimated Cost = 992.84 Worse than the original query !

Why This Plan is worse than the original plan with no indexes?

Because Here we didn't make use of the hash index in the filtration of salary ¿ 40000 and hash is not efficient for range queries , also the hash index we didn't use in the aggregation process because in either ways you need to pass through the whole table to make the aggregation and using an index scan won't be efficient as you will need to pass over the scan and for each iteration you will need a fetch from the disk so till now the hash is of no use, until the last Port of the join when joining the sub query with the department and here the engine used my hash index but it turned out with more cost , this may be reason of the data insertions that may differ here the data insertions made the join has many tuples meet the condition nearly the whole department table meet the condition and i made insertions like that so that i meet the doctor requirements , so nearly the whole table has met the condition so i have to get all rows from the index first then go and fetch it also from disk to make the join and here is the drawback with data insertions the hash index is overhead but with another data set when we have less rows meeting the condition we can get better cost then using the hash index. Hope i have introduced the drawback in a simple way.

4. The Query with BRIN Indexes



```
create index brinsalary on employee using brin(salary);
create index brindno on employee using brin(dno);

drop index brinsalary;
drop index brindno;

explain analyze
select dnumber, count(*)
from department, employee
where dnumber=dno
and
salary > 40000
and
dno in  (
  select dno
  from employee
  group by dno
  having count (*) > 5)
group by dnumber;
```

Notifications | Explain | Messages | Query History

```
Successfully run. Total query runtime: 36 msec.
27 rows affected.
```

Data Output

	QUERY PLAN text
1	GroupAggregate (cost=863.89..866.90 rows=150 width=12) (actual time=4.070..4.132 rows=148 loops=1)
2	[...] Group Key: department.dnumber
3	[...] -> Sort (cost=863.89..864.39 rows=201 width=4) (actual time=4.066..4.083 rows=600 loops=1)
4	[...] Sort Key: department.dnumber
5	[...] Sort Method: quicksort Memory: 53kB
6	[...] -> Hash Join (cost=523.55..856.20 rows=201 width=4) (actual time=3.211..4.008 rows=600 loops=1)
7	[...] Hash Cond: (employee.dno = department.dnumber)
8	[...] -> Hash Join (cost=518.17..850.28 rows=202 width=8) (actual time=3.179..3.907 rows=600 loops=1)
9	[...] Hash Cond: (employee.dno = employee_1.dno)
10	[...] -> Bitmap Heap Scan on employee (cost=12.18..342.67 rows=602 width=4) (actual time=0.016..0.668 rows=600 loops=1)
11	[...] Recheck Cond: (salary > 40000)
12	[...] Rows Removed by Index Recheck: 7208
13	[...] Heap Blocks: lossy=128
14	[...] -> Bitmap Index Scan on brinsalary (cost=0.00..12.03 rows=5399 width=0) (actual time=0.012..0.013 rows=1280 loops=1)
15	[...] Index Cond: (salary > 40000)
16	[...] -> Hash (cost=505.36..505.36 rows=50 width=4) (actual time=3.159..3.160 rows=149 loops=1)
17	[...] Buckets: 1024 Batches: 1 Memory Usage: 14kB
18	[...] -> HashAggregate (cost=503.00..504.86 rows=50 width=4) (actual time=3.132..3.145 rows=149 loops=1)
19	[...] Group Key: employee_1.dno
20	[...] Filter: (count(*) > 5)
21	[...] Batches: 1 Memory Usage: 48kB
22	[...] -> Seq Scan on employee employee_1 (cost=0.00..423.00 rows=16000 width=4) (actual time=0.004..0.785 rows=16000 loops=1)
23	[...] -> Hash (cost=3.50..3.50 rows=150 width=4) (actual time=0.029..0.029 rows=150 loops=1)
24	[...] Buckets: 1024 Batches: 1 Memory Usage: 14kB
25	[...] -> Seq Scan on department (cost=0.00..3.50 rows=150 width=4) (actual time=0.007..0.016 rows=150 loops=1)
26	Planning Time: 0.284 ms
27	Execution Time: 4.184 ms

The Execution time is around 3.8 msec 4.4 msec The Index i have created are :

brinsalary : BRIN index on salary attribute in the employee table

brindno : BRIN index on dno attribute in the employee table

The Physical Plan :

Starting with the sub-query the engine has chosen to do hash aggregation after a seq scan , but why?! although we have made a brin index on the dno attribute , in fact the BRIN index is not useful in the aggregation that is why the engine discarded that index and didn't use it and after the hash aggregation the engine hashed the result. Now for the outer query for the Employee relation the engine has used the BRIN index we created(brinsalary) using bitmap index scan as it is useful here for the range selectivity better than the seq scan filtration that was used before , and then engine has used a hash-join between the sub-query and employee after filtration,for the department table the engine seq scanned it to hash it and then hash joined with the result of joining sub-query and employee , Now for the very last aggregation of the whole query the engine made a quick sort in memory and made a sort aggregation as this was faster than hashing and making a hashed aggregation.

The BRINDNO index is useless

It is useless because the BRIN index is not useful in aggregation and the engine decided to go along with hash aggregation faster than sorting the data and make a sort aggregation , The BRIN was used only in the salary selectivity. This BRINDNO can be discarded as it is useless.

The Estimated Cost = 866.90 + 864.39 + 856.20 + 850.28 + 342.67 + 12.03 + 505.36 + 504.86 + 423 + 3.5 + 3.5 = 5232.69

Here the BRIN Index only improved very small part of the query which was the range selectivity part , that's why the difference in execution time and estimated cost of the original query is not that much powerful.

5. The Query with Mixed Indexes

```sql
create index bsalary on employee using btree(salary);
create index bdno on employee using btree(dno);
create index hashDnumber on department using hash(dnumber);

set enable_hashjoin = off;


explain analyze
select dnumber,
       count(*)
       from department , employee
       where dnumber = dno
       and salary > 40000
       and dno in (
                select dno
                       from employee
                       group by dno
                       having count(*) > 5)
        group by dnumber;
```

Data Output

	QUERY PLAN text
1	GroupAggregate (cost=86.48..492.13 rows=150 width=12) (actual time=0.194..2.172 rows=148 loops=1)
2	[...] Group Key: department.dnumber
3	[...] -> Nested Loop (cost=86.48..489.61 rows=203 width=4) (actual time=0.176..2.119 rows=600 loops=1)
4	[...] -> Merge Join (cost=86.48..478.06 rows=204 width=8) (actual time=0.171..1.786 rows=600 loops=1)
5	[...] Merge Cond: (employee.dno = employee_1.dno)
6	[...] -> Sort (cost=86.20..87.72 rows=609 width=4) (actual time=0.142..0.167 rows=600 loops=1)
7	[...] Sort Key: employee.dno
8	[...] Sort Method: quicksort Memory: 53kB
9	[...] -> Index Scan using bsalary on employee (cost=0.29..58.03 rows=609 width=4) (actual time=0.007..0.086 rows=600 loops=1)
10	[...] Index Cond: (salary > 40000)
11	[...] -> GroupAggregate (cost=0.29..386.15 rows=50 width=4) (actual time=0.027..1.549 rows=149 loops=1)
12	[...] Group Key: employee_1.dno
13	[...] Filter: (count(*) > 5)
14	[...] -> Index Only Scan using bdno on employee employee_1 (cost=0.29..304.29 rows=16000 width=4) (actual time=0.004..0.758 ro...
15	[...] Heap Fetches: 0
16	[...] -> Index Scan using hashdnumber on department (cost=0.00..0.06 rows=1 width=4) (actual time=0.000..0.000 rows=1 loops=600)
17	[...] Index Cond: (dnumber = employee.dno)
18	Planning Time: 0.516 ms
19	Execution Time: 2.216 ms

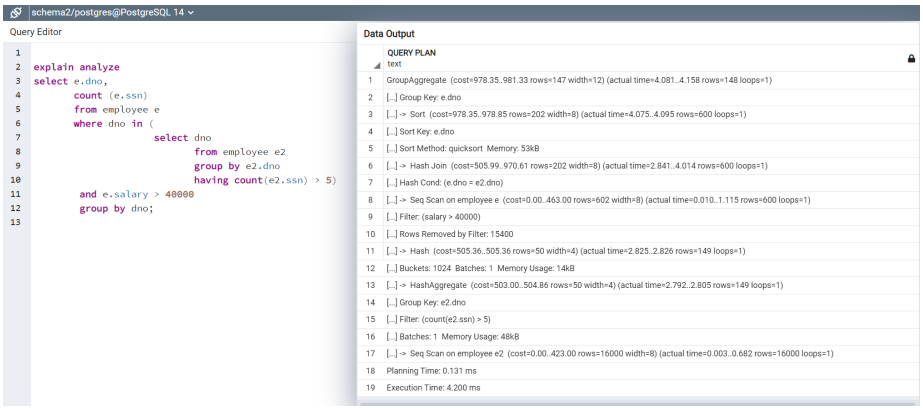
The Execution Time ranges between 2 msec to 2.4 msec.

The Physical Plan :

For the Sub Query The Engine decided to make an index scan the department table and make an aggregation based on sorting making advantage that the data is sorted in the index , and then after the aggregation it makes a merge join with the table employee , and for thus table it makes an index scan using the btree created on the salary attribute so that it can have a faster filtration on the salary with the use of the btree index and then sort the table but it is already sorted but it is sorted in the index according to salary we want to join on dno so it sorted with sorting key is the dno, and then merge sort between result of grouping and the employee and then it is also given nest loop join with the table department using the hash index created on the table department and then the final result of all the joins are aggregated on the attribute dnumber using hash algorithms.

The Estimated Cost = 492.13 Smaller than the original query as The Mixed Indices were efficient here and made an optimization.

6. My Optimized Query



The Execution Time ranges between 4 msec to 4.4 msec.

The Physical Plan :

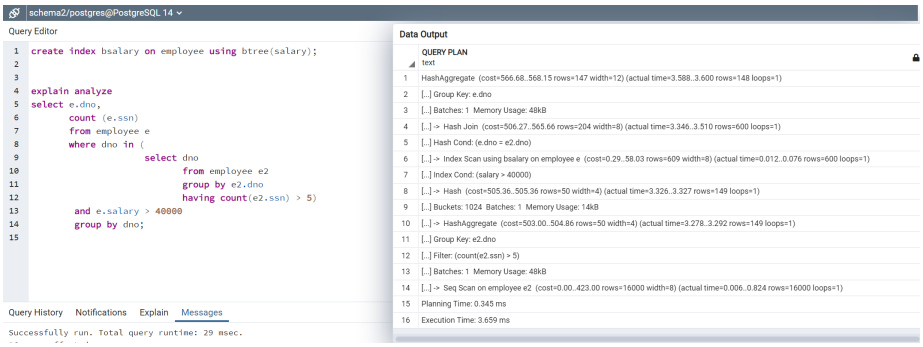
For the Sub Query the Engine decides to make a seq scan over the table employee and make an aggregation on the dno attribute with hashing algorithms (aggregation based on hash) and then hashed the result, Now for the Whole Thing the engine makes a seq scan over the table employee and makes then an inner hash join with the result of the sub-query and finally sorted the result and made an aggregation on the dno based on the sorted result he made.

What is The Difference between my Query and the Query given?

My New Query is more efficient because it uses less tables in the plan here we don't use even the table department as it is redundant in the join operations and this will appear on the cost, yes the cost difference is not the WOW but because the most costy operations are still existing which are the group by and can not be reduced because this will affect the logic of the query itself.

The Estimated Cost = 981.33 (less than the original query)

7. My Query with BTree Indexes



The Execution Time ranges between 3.4 to 3.7 msec.

The Indexes that i have created :

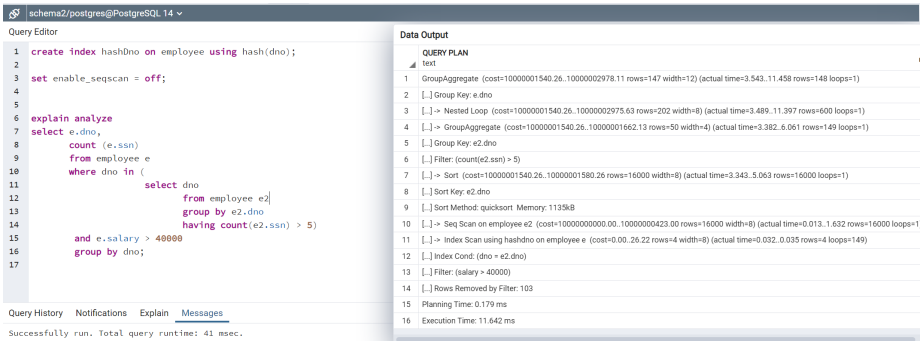
bSalary : Btree index on the table employee using the attribute salary;

The Physical Plan :

For the sub-Query The engine decides to make a seq scan over the table employee and make a hash aggregate over the attribute dno (why a btree in this case is not useful because hash aggregate is faster than looping over the index then fetch row from disk and then aggregate it and because in aggregation in either ways you will need to loop over the whole table) and then it hashed the result from the aggregation, for the whole thing the engine decided to make use of the bsalary that is created on the salary attribute to enhance the range query condition salary¿40000 , and then makes a hash inner join with the hashed result of the inner sub query and finally makes a hash aggregation over the whole result on the dno to get the final result.

The Estimated Cost = 568.15 better than my query with no indexes

8. My Query with Hash Indexes



The Indexes i have created :

hashDno : Hash Index created on table employee using the dno attribute.

The Physical Plan :

For the inner query the engine decided to make a seqscan over table employee and make a hash aggregation over the dno attribute to make the group by part and filter the count(dno) ¿ , then the result of group by is joined with the table employee with nest loop join with the table employee and here the engine used my hash index that i have created and then hash aggregate to make the last group by and make the output.

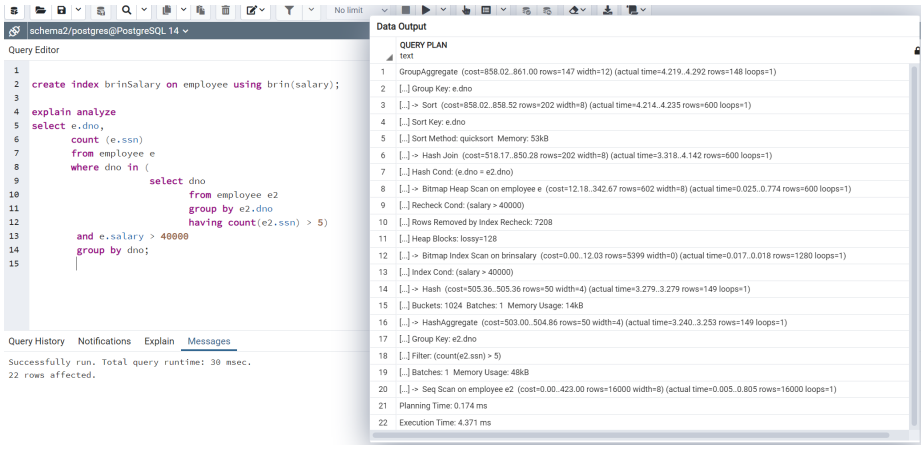
Why I have disabled the Seq Scan?

because the engine was insisting on using it as the using the index in aggregation is not efficient because you need to pass by the whole table so index passing through the whole table and for every row you will get a fetch from original table then this is more IOs, and also it was finding that seq scan is also better in the joining because he have his row in hand and because we have many rows matching here so even the result of the join is we need to nearly pass over the whole table so i have disabled the seqscan but it also insisted on it even after disabling in the aggregation.

The Estimated Cost = 10000002978.11 max default because the use of seq scan that is already disabled (Not Descriptive)

The Execution Time ranges between 11.7 to 12.3 msec indicating worse performance than the original query.

9. My Query with BRIN Indexes



The Execution Time ranges between 4.2 msec to 4.5 msec.

The Indexes that i have created :

brinSalary : BRIN Index created on table employee on attribute salary

The Physical Plan :

For The SubQuery the Engine has decided to make a seq scan and then make a hash aggregation on the key e.dno and apply the condition count(e.ssn) ¿ 5 then hash the result, and for the whole thing the result of the sub query will be joined with the employee table and making a bitmap index scan over the table employee using the created BRIN index why ? Because here we have selectivity and the data insertions ahd the salary with wide range so we have selectivity in large range and BRIN will be useful in it and then followed by a bitmap heap scan with recheck condition salary ¿ 40000, and finally the hash inner join is done and the final result is sorted to make the last group aggregation by the dno.

The Estimated Cost = 861 better than My Query with no Indexes.

10. My Query with Mix Indexes

The Execution Time ranges between 6.8 msec to 7.1 msec.

The Indexes i have created :

brinSalary : brin Index created on the employee table on salary

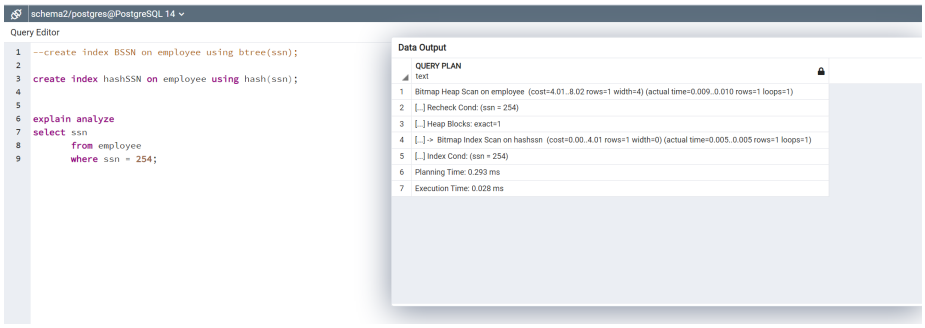
bDNO : Btree index on employee using dno attribute.

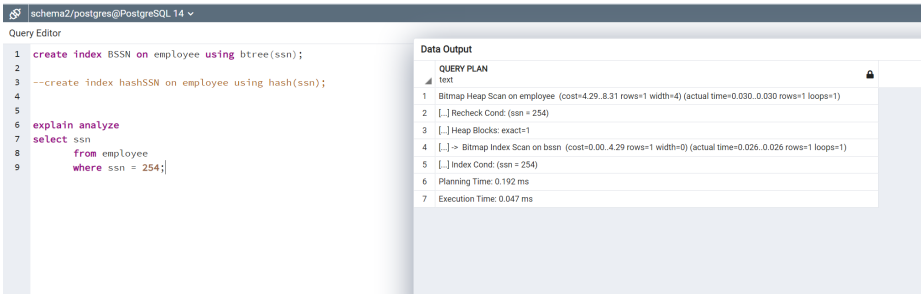
The Physical Plan :

For the subquery i have disabled the hashaggregate so that force the engine to use my btree index and it now has used it and made a sort aggregation making advantage that the data is sorted in the index and filtered count(ssn) ¿ 5 , and it took the result of the subquery and made it a hash inner join with the emoployee table, fot the employee table it used the brin index that i have created to make the low selectivity part salary ¿ 40000 and it is low selectivity because the data insertions have very large variation of salary values and finally the final aggregation to make the output.

The Estimated Cost = 1794.28 Lower than the original query with no indexes

6 The HASH VS BTREE





Here in Both Cases they nearly have the same cost because what i mentioned before some concurrency problems regarding the hashing in PostgreSQL specially when doubling the size of the hashtable when exceeding the load factor and this may cause some deadlocks so there are some mechanisms to avoid this that consumes some more time that makes the hash nearly equal to BTree in 1 Vs 1 Competition on the same exact plane just hash instead btree, but of course if the plan differs this will have an impact on the performance of the whole plan.