Javier Palomares

Professor Miranker

EE382 Data Engineering

11/9/2018

Query Plans in SQLite and Postgres

I chose to investigate the query plans SQLite generated for the 20 queries from the assignment. I observed that for several of the join queries, SQLite chooses to scan tables using an automatic index. For the remaining join queries, the behavior was to use full table scans over all tuples of the relations. I investigated the automatic index in SQLite's documentation and found that SQLite might create an automatic index that lasts only for the duration of a single SQL statement. The cost of constructing the automatic index is O(T(R)*logT(R)) and the cost of doing a full table is O(T(R)), so an automatic index will be created if SQLite expects that the lookup will be run more than logT(R) times during the course of the SQL statement. Moreover, the current implementation on SQLite only includes nested joins (no merge-join or hash-join). This behavior explains why I observed no difference when joining relations over join keys where secondary indexes exists compared to joining with secondary indexes. Therefore, the best set of indexes in SQLite for the workload is to have no indexes. Because of this, I do not recommend choosing SQLite when join queries are frequent.

After the difficulties I observed with SQLite, I migrated the experiment to PostgreSQL, which does implement merge-join and hash-join, and was able to observe interesting results that allowed me to conclude the best set of indices is secondary indices on the `ot`,`hund`, and `ten` attributes. The following are the observations I made that formed this conclusion:

1. Query 3 `A'.ht = B'.ht` does not make use of the secondary index on `ht`. The query plan is to use a hash join just like Query 2 where no secondary index is available.

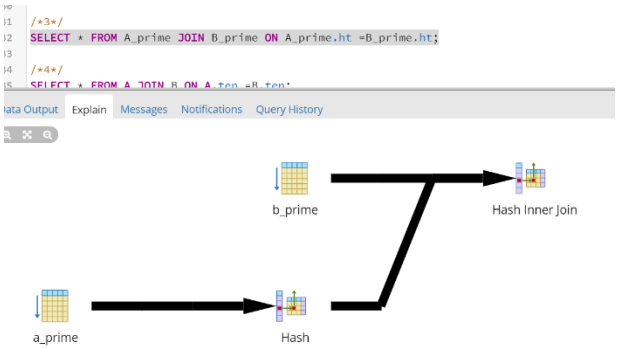


Figure 1. Adding a secondary index on ht does not change the query plan

2. In contrast, Query 5 `A'.ten = B'.ten` produces in a merge sort using the index to avoid a sort. The secondary index is beneficial for Query 5. Query 4, with no available secondary index, must do a sort before the merge join.

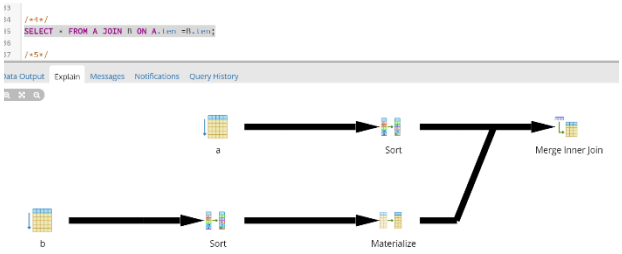


Figure 2. Query 4 A.ht=B.ht. With no secondary index on ten, a sort must be done to use merge join.

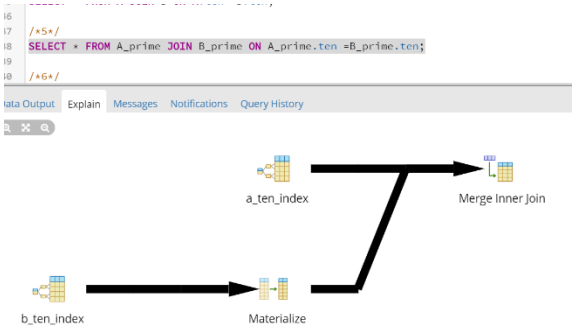


Figure 3. Query 5. A'.ht=B'.ht. With a secondary index, no sorting is needed prior to merge join.

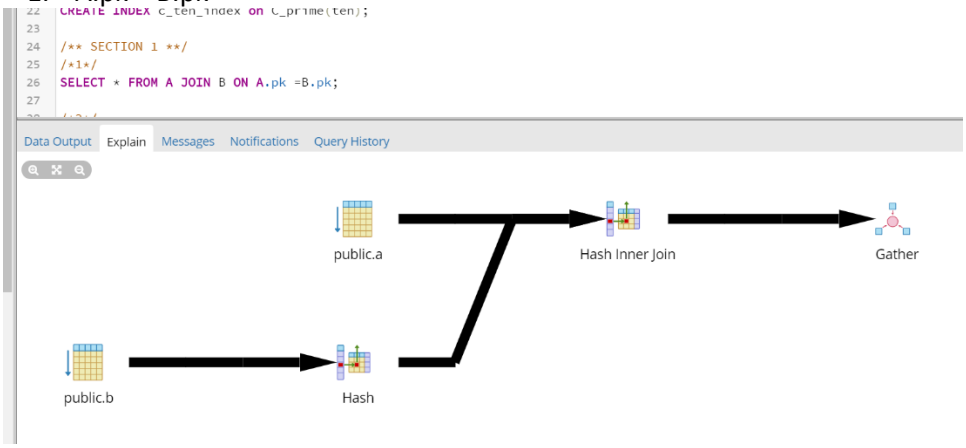
3. Three way joins on attributes with `ht` and `tt` use a hash-join regardless of whether a secondary index exists. In contrast, the query plan is to use merge join when joining on `ot`, `hund`, or `ten` attributes.

Taking these 3 observations in place, I concluded that having a secondary index on `ot`, `hund`, and `ten` is beneficial in reducing the costs of joins on these attributes, and secondary indices on `ht` and `tt` are not beneficial.

The rest of the paper serves as an appendix. I include screenshots and observations for all of the queries from the assignment.

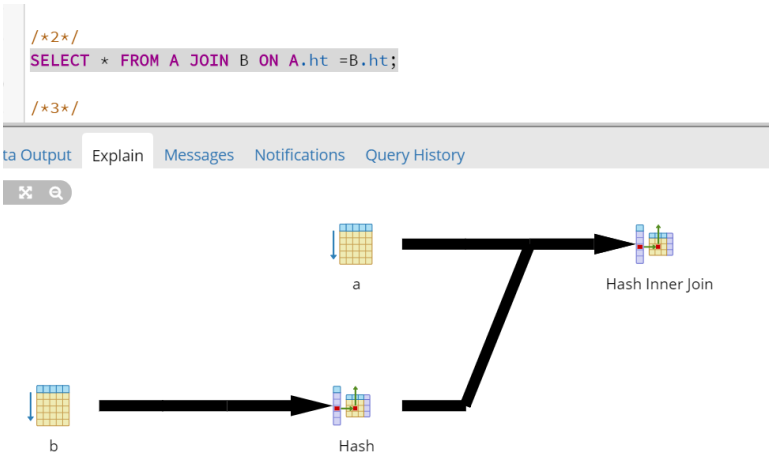
Section 1: Explain Query Plans

1. A.pk = B.pk



System Chooses Hash Join. Hashes B and scans over A to do hash inner join
Expected Time:1338135.43
Expected Size:5000016 rows
T(A) = 5000000 V(A,pk) = 5000000
T(B) = 5000000 V(B,pk) = 5000000
The selected plan is hash join.

2. A.ht = B.ht

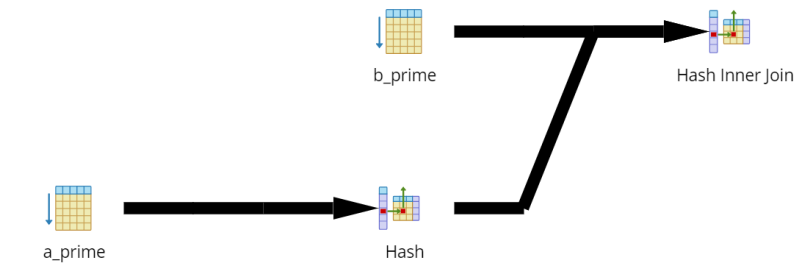


T(A) = 5000000 V(A,ht) = 100000
T(B) = 5000000 V(B,ht) = 100000

3. A'.ht = B'.ht

```
/*3*/
SELECT * FROM A_prime JOIN B_prime ON A_prime.ht =B_prime.ht;

/*4*/
SELECT * FROM A JOIN B ON A.ten =B.ten;
```

Data Output | Explain | Messages | Notifications | Query History



same query plan as #2

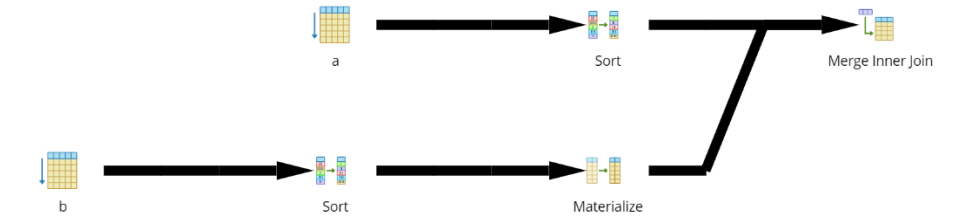
T(A') = 5000000 V(A',ht) = 100000
T(B') = 5000000 V(B',ht) = 100000

V(R,attrib) did not change the query plan across 1,2,3. The secondary index has no impact on the selected query plan.

4. A.ten = B.ten

```
/*4*/
SELECT * FROM A JOIN B ON A.ten =B.ten;

/*5*/
```

Data Output | Explain | Messages | Notifications | Query History

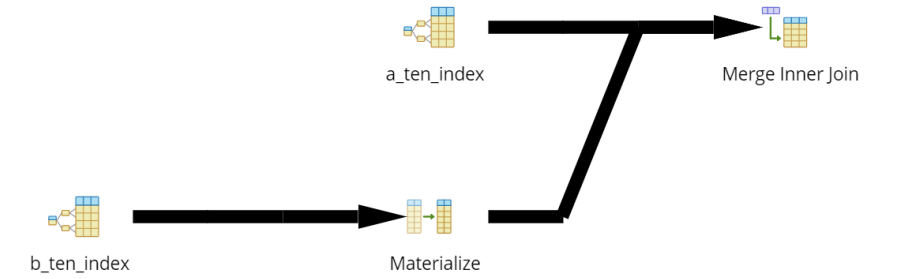


T(A) = 5000000 V(A,ten) = 10
T(B) = 5000000 V(B,ten) = 10
Query plan selects Merge join

5. A'.ten = B'.ten

```
/*5*/
SELECT * FROM A_prime JOIN B_prime ON A_prime.ten =B_prime.ten;

/*6*/
```

Data Output | Explain | Messages | Notifications | Query History

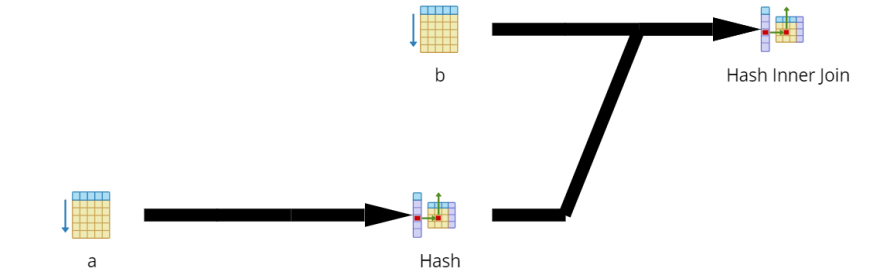


T(A) = 5000000 V(A,ten) = 10
T(B) = 5000000 V(B,ten) = 10
Query plan selects Merge join using the ten_index to shortcut data sorting

6. A.ht = B.ten

```
/*6*/
SELECT * FROM A JOIN B ON A.ht =B.ten;

/*7*/
```

Data Output | Explain | Messages | Notifications | Query History

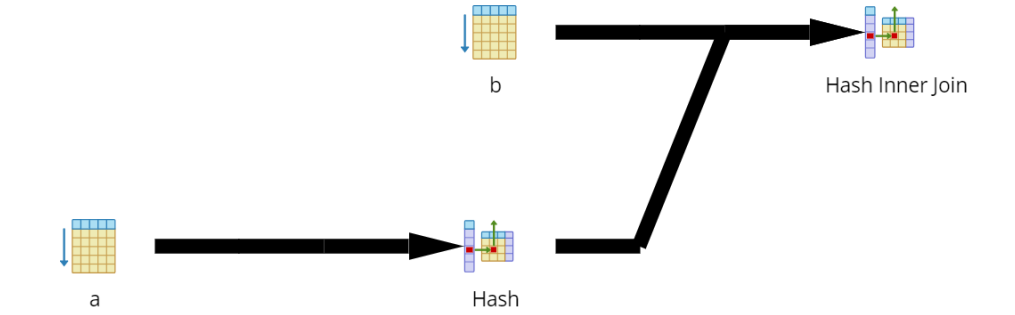


T(A)= 5000000 V(A,ht) = 100000
T(B)= 5000000 V(B,ten) = 10
Selected plan is hash join

7. B.ten = A.ht

```
42
43  /*7*/
44  SELECT * FROM B JOIN A ON B.ten =A.ht;
45
46  /*8*/
```

Data Output | Explain | Messages | Notifications | Query History



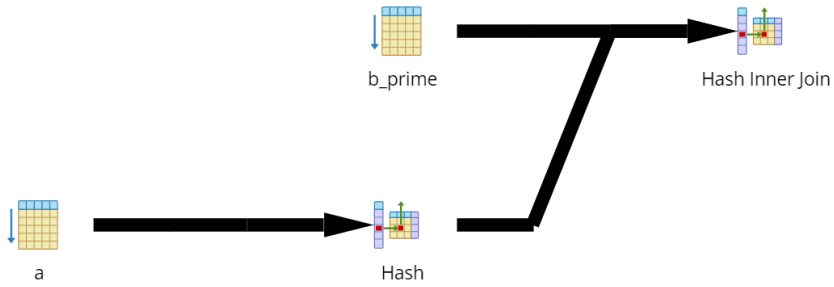
T(A)= 5000000 V(A,ht) = 100000
T(B)= 5000000 V(B,ten) = 10

Selected plan is hash join again. The optimizer chose to hash a, then join b regardless of the order that a,b appear in the JOIN statement

8. A.ht = B'.ten

```
/*8*/
SELECT * FROM A JOIN B_prime ON A.ht =B_prime.ten;

/*9*/
```

ta Output | Explain | Messages | Notifications | Query History

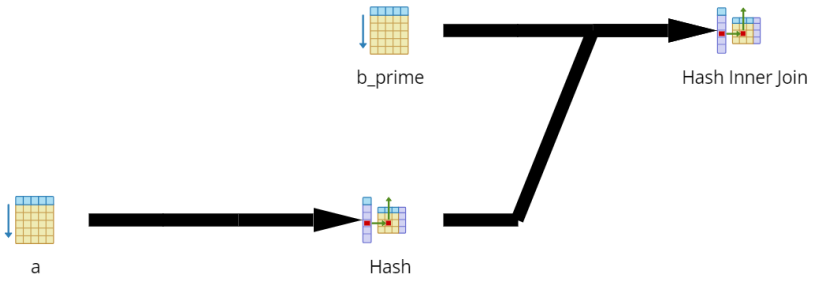


T(A) = 5000000 V(A,ht) = 100000
T(B') = 5000000 V(B',ten)= 10
Selected plan is hash join

9. B'.ten = A.ht

```
48
49  /*9*/
50  SELECT * FROM B_prime JOIN A ON B_prime.ten = A.ht;
51
52  /* 3 way joins */
53  /*10*/
54  SELECT * FROM A JOIN B ON A.ht = B.ht JOIN C ON B.ht = C.ht;
```

Data Output | Explain | Messages | Notifications | Query History

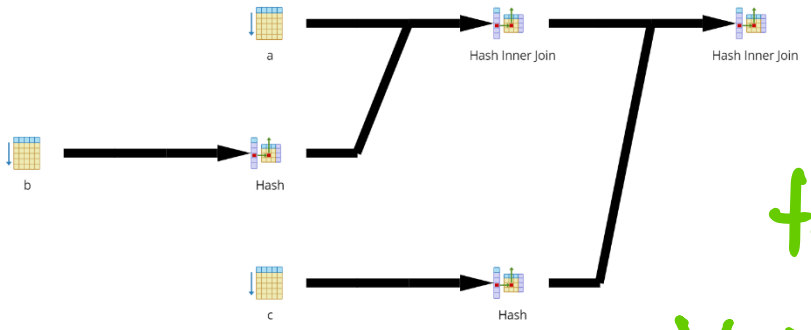


T(A) = 5000000 V(A,ht) = 100000
T(B') = 5000000 V(B',ten)= 10
Selected plan is hash join. The optimizer again chose its own order regardless of how a,b appear in the JOIN statement

10. A.ht = B.ht = C.ht

```
/* 3 way joins */
/*10*/
SELECT * FROM A JOIN B ON A.ht = B.ht JOIN C ON B.ht = C.ht;

/*11*/
SELECT * FROM A JOIN B ON A.ten = B.ten JOIN C ON B.ten = C.ten;
```

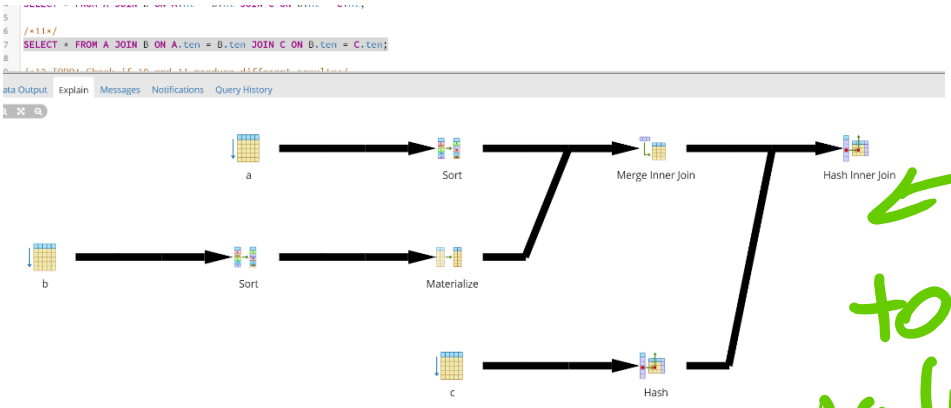
Data Output | Explain | Messages | Notifications | Query History



optimizer has decided it's better to use hashjoin than to sort the 100,000 values in ht.

T(A) = 5000000 V(A,ht) = 100000
T(B) = 5000000 V(B,ht) = 100000
T(c)= 5000000 V(C,ht) = 100000
Selected plan: hash join A,b, hash join c

11. A.ten = B.ten = C.ten



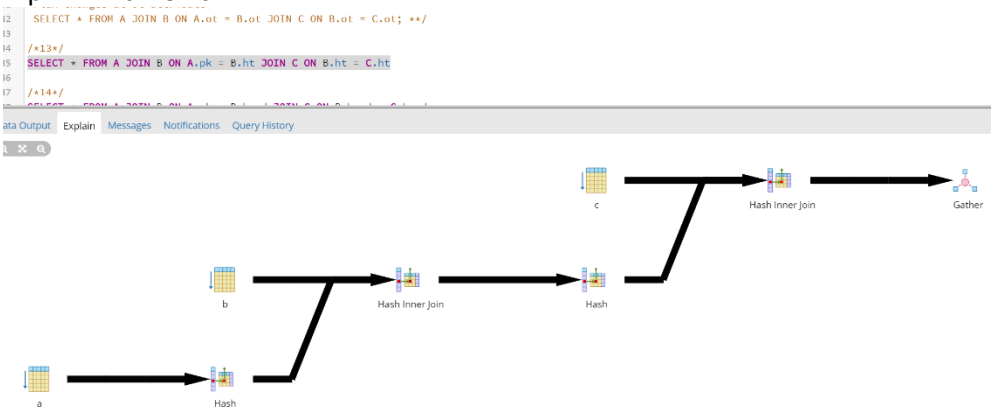
cheaper to sort 10 values? use merge join

T(A) = 5000000 V(A,ten) = 10
T(B) = 5000000 V(B,ten) = 10
T(c)= 5000000 V(C,ten) = 10
Selected plan: merge join A,B, hash join c

12. Do they produce different plans?

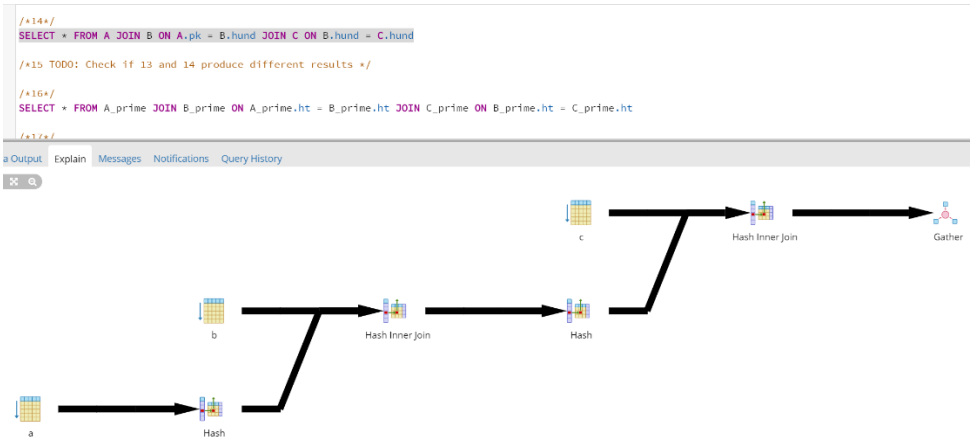
Yes, the plans are different. I found the plan changes on the ot attribute when V(R,ot) = 1000
A.ot = B.ot = C.ot

13. A.pk = B.ht = C.ht



Selected pla n is to hash join a,b then hash join c
T(A) = 5000000 V(A,pk) = 5000000
T(B) = 5000000 V(B,ht) = 100000
T(C) = 5000000 V(C,ht) = 100000

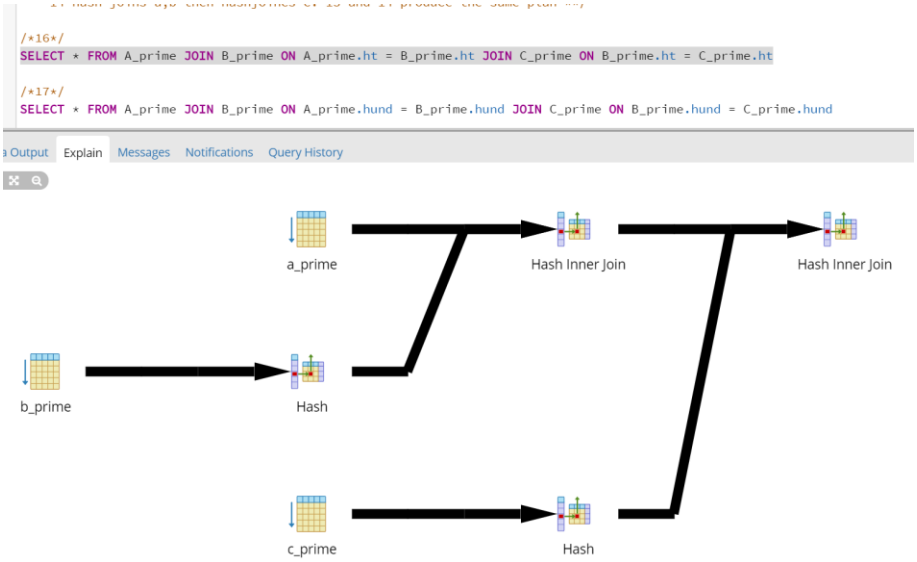
14. A.pk = B.hund = C.hund



Selected pla n is to hash join a,b then hash join c
T(A) = 5000000 V(A,pk) = 5000000
T(B) = 5000000 V(B,hund) = 100
T(C) = 5000000 V(C,hund) = 100

15. Do they produce different plans?
    No, 13 and 14 produce the same plan.

16. A'.ht = B'.ht = C'.ht

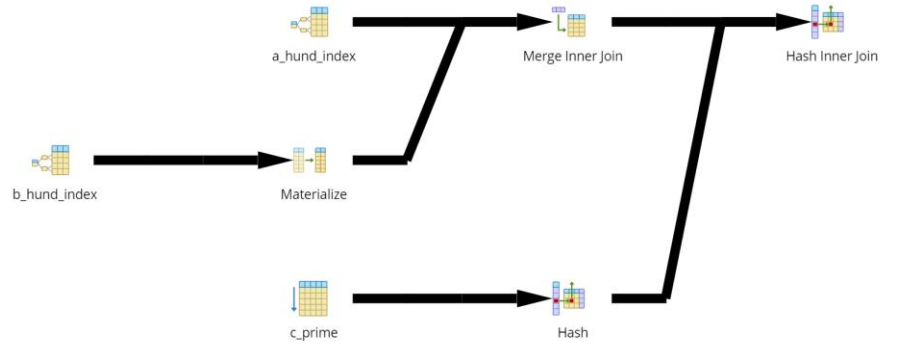


T(A) = 5000000 V(A,ht) = 100000
T(B) = 5000000 V(B,ht) = 100000
T(c)= 5000000 V(C,ht) = 100000
Selected plan: hash join A,b, hash join c. The query plan does not change from 10, so the secondary index has no effect on the plan

17. A'.hund = B'.hund = C'.hund

```
/*17*/
SELECT * FROM A_prime JOIN B_prime ON A_prime.hund = B_prime.hund JOIN C_prime ON B_prime.hund = C_prime.hund

/*18 */
SELECT * FROM A_prime JOIN B_prime ON A_prime.ht = B_prime.ten JOIN C_prime ON B_prime.ten = C_prime.ht
```

ata Output | Explain | Messages | Notifications | Query History



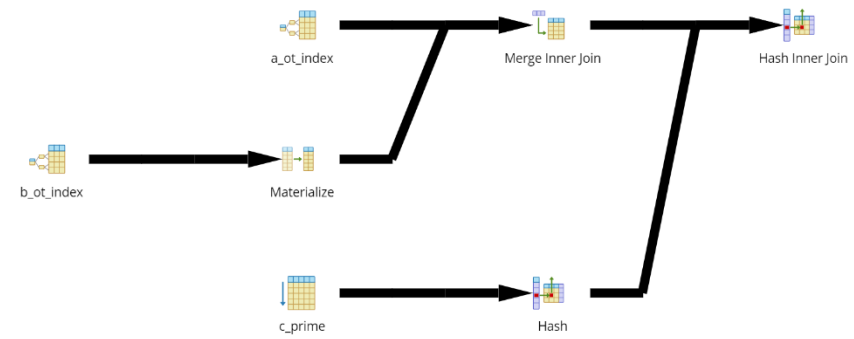
T(A) = 5000000 V(A,hund) = 100
T(B) = 5000000 V(B,hund) = 100
T(c)= 5000000 V(C,hund) = 100
Selected plan: merge join A,b, hash join c. The query plan changes from 16.

18. Do 16 and 17 produce different results?
    Yes. 16 chooses hash join to join a,b. 17 chooses merge join. The transition point is at the ot attribute

hw7_pk on postgres@PostgreSQL 11

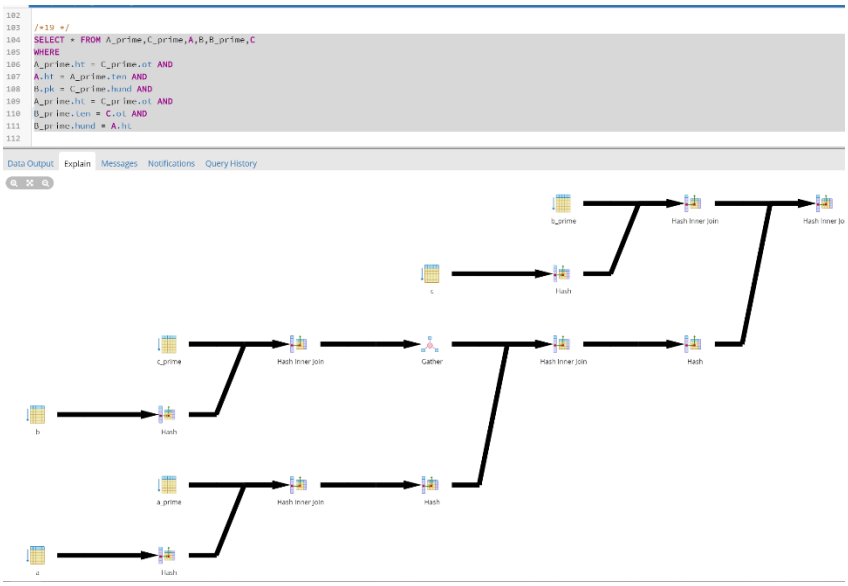
```
/*Transition point*/
SELECT * FROM A_prime JOIN B_prime ON A_prime.ot = B_prime.ot JOIN C_prime ON B_prime.ot = C_prime.ot

/*19 */
SELECT * FROM A_prime,C_prime,A,B,B_prime,C
WHERE
A_prime.ht = C_prime.ot AND
A.ht = A_prime.ten AND
B.pk = C_prime.hund AND
```

ata Output | Explain | Messages | Notifications | Query History



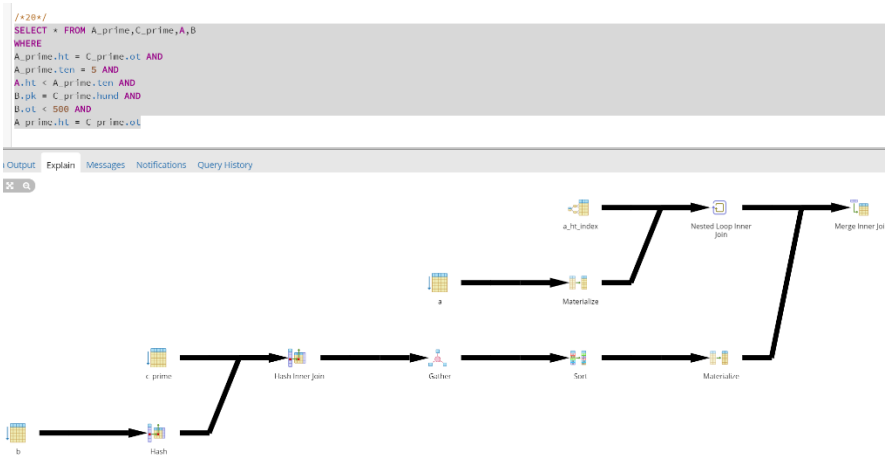
Section 2:

19. A'.ht = C'.ot AND
    A.ht = A'.ten AND
    B.pk = C'.hund AND
    A'.ht = C'.ot AND
    B'.ten = C.ot AND
    B'.hund = A.ht



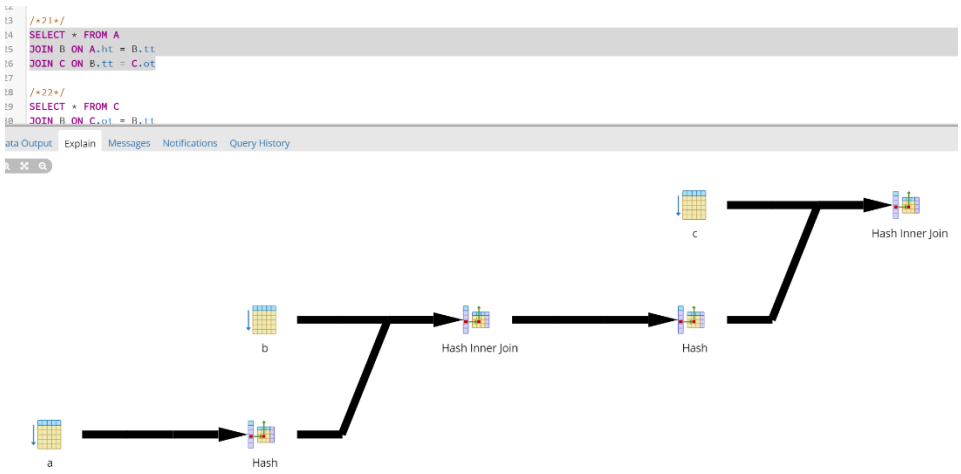
The estimated size in the last hash inner join is over 3 quadrillion rows (3,172,245,629,574,755)

20. A'ht = C'.ot AND
    A'.ten = 5 AND
    A.ht < A'.ten AND
    B.pk = C'.hund AND
    B.ot < 500 AND
    A'.ht = C'.ot



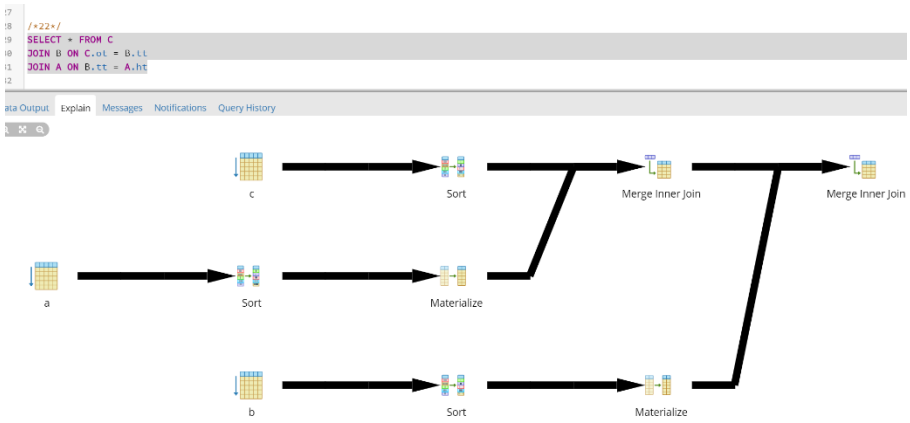
The estimated size of the last merge inner join is: over 21 trillion rows(21,554,197,796,542)! The compiler did not notice the equivalent predicate.

21. A.ht = B.tt = C.ot



T(A) = 5000000 V(A,ht) = 100000
T(B) = 5000000 V(B,tt) = 10000
T(C) = 5000000 V(C,ot) = 1000

22. C.ot = B.tt = A.ht

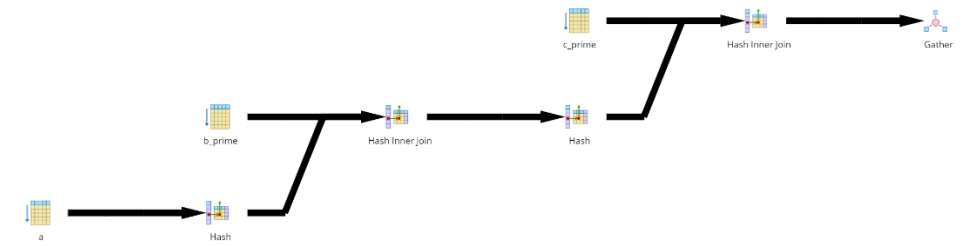


The query plan changes between 21 and 22 even though the only difference is the order of the tables in the query.
T(A) = 5000000 V(A,ht) = 100000
T(B) = 5000000 V(B,tt) = 10000
T(C) = 5000000 V(C,ot) = 1000

23. A.pk = B'.tt = C'.ot

```
/*23*/
SELECT * FROM A
JOIN B_prime ON A.pk = B_prime.tt
JOIN C_prime ON B_prime.tt = C_prime.ot

/*24*/
SELECT *FROM C_prime
JOIN B_prime ON C_prime.ot = B_prime.tt
JOIN A ON B_prime.tt = A.pk
```

Data Output | Explain | Messages | Notifications | Query History

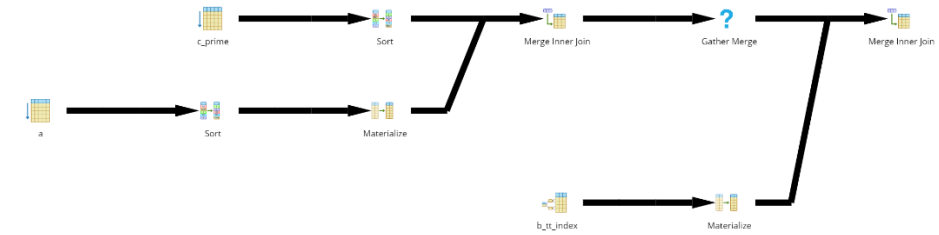


T(A) = 5000000 T(A,pk) = 5000000
T(B) = 5000000 T(B,tt) = 10000
T(C) = 5000000 T(C,ot) = 1000

24. C'.ot = B'.tt = A.pk

```
/*24*/
SELECT *FROM C_prime
JOIN B_prime ON C_prime.ot = B_prime.tt
JOIN A ON B_prime.tt = A.pk
```

Data Output | Explain | Messages | Notifications | Query History



T(A) = 5000000 T(A,pk) = 5000000
T(B) = 5000000 T(B,tt) = 10000
T(C) = 5000000 T(C,ot) = 1000
Again, the query plan changes between 23 and 24 even though the only difference is the order of the tables in the query