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:

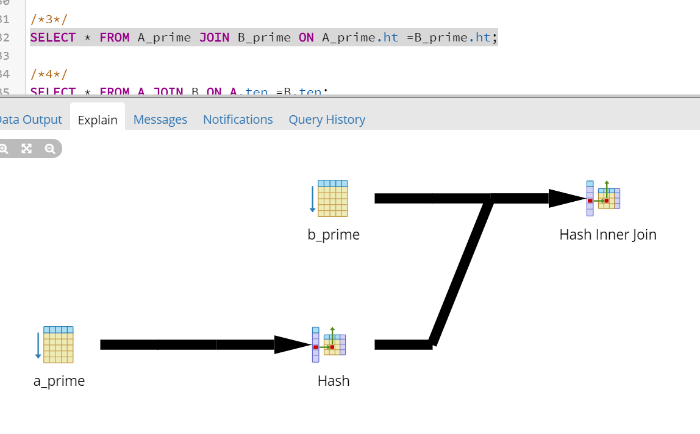
1. Query 3 A’.ht = B’.ht does not make use of the secondary index on ht. The query plan is to use a hash join just like Query 2 where no secondary index is available.

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

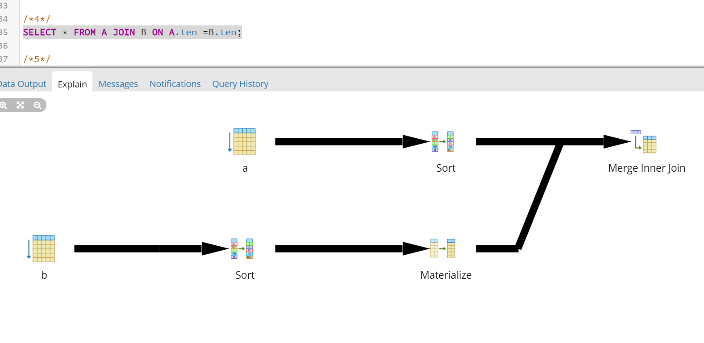
1. In contrast, Query 5 A’.ten = B’.ten produces in a merge sort using the index to avoid a sort. The secondary index is beneficial for Query 5. Query 4, with no available secondary index, must do a sort before the merge join.

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

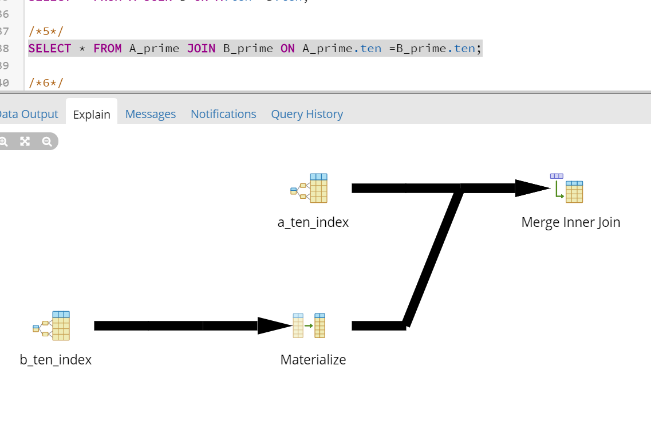


Figure . Query 5. A'.ht=B'.ht. With a secondary index, no sorting is needed prior to merge join.

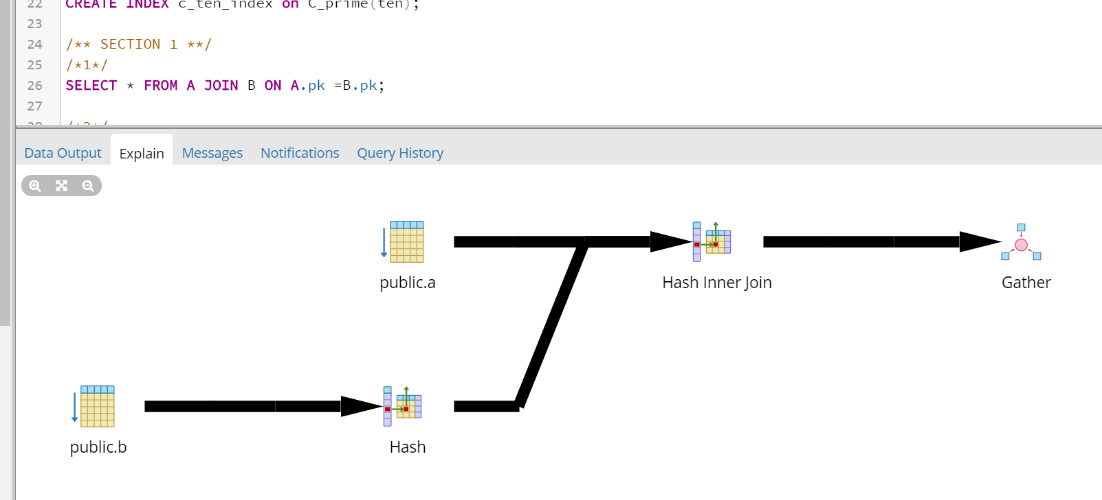
1. Three way joins on attributes with ht and tt use a hash-join regardless of whether a secondary index exists. In contrast, the query plan is to use merge join when joining on ot, hund, or ten attributes.

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

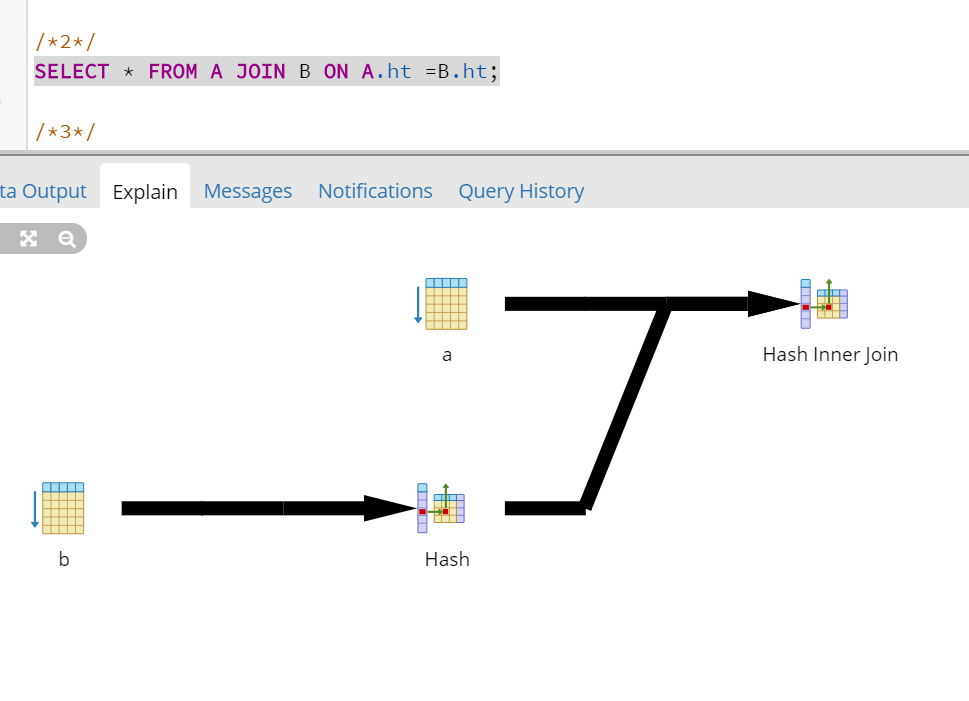
Expected Size:5000016 rows

T(A) = 5000000 V(A,pk) = 5000000

T(B) = 5000000 V(B,pk ) = 5000000

The selected plan is hash join.

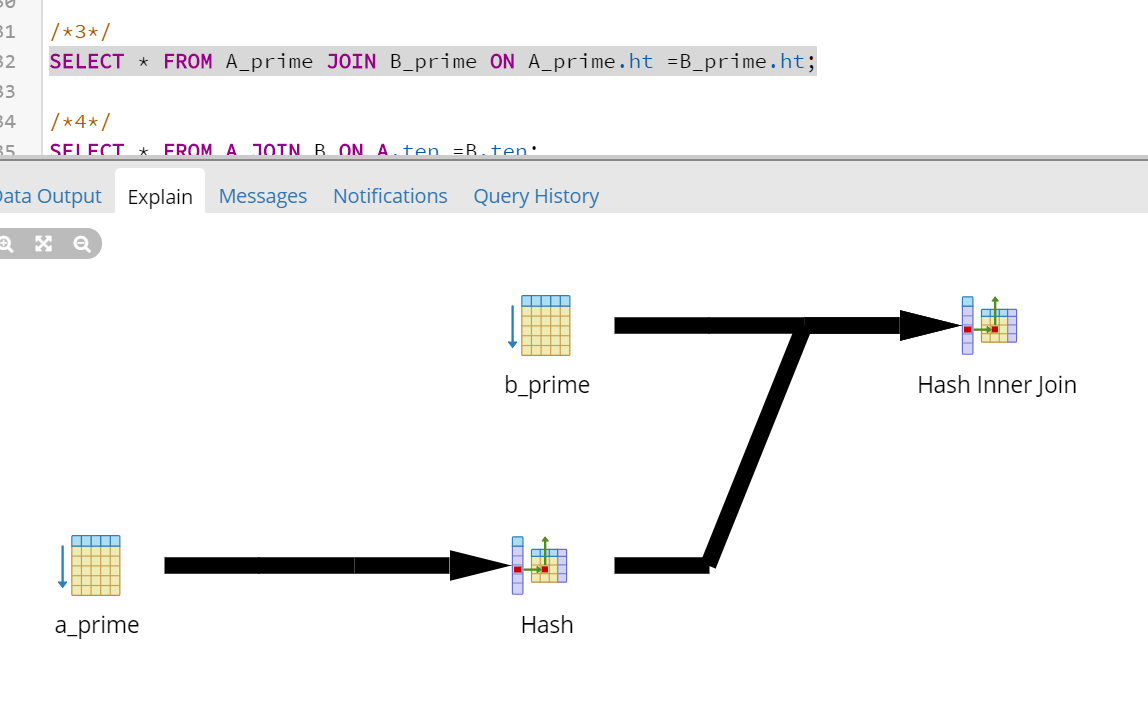
1. A.ht = B.ht



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

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

1. A'.ht = B'.ht



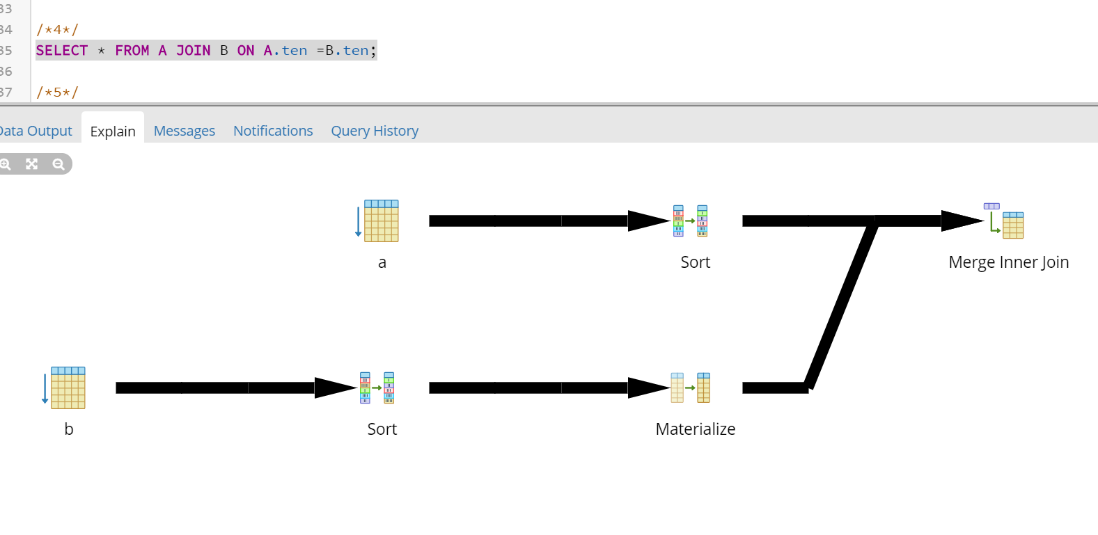


T(A') = 5000000 V(A',ht) = 100000

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.

1. A.ten = B.ten

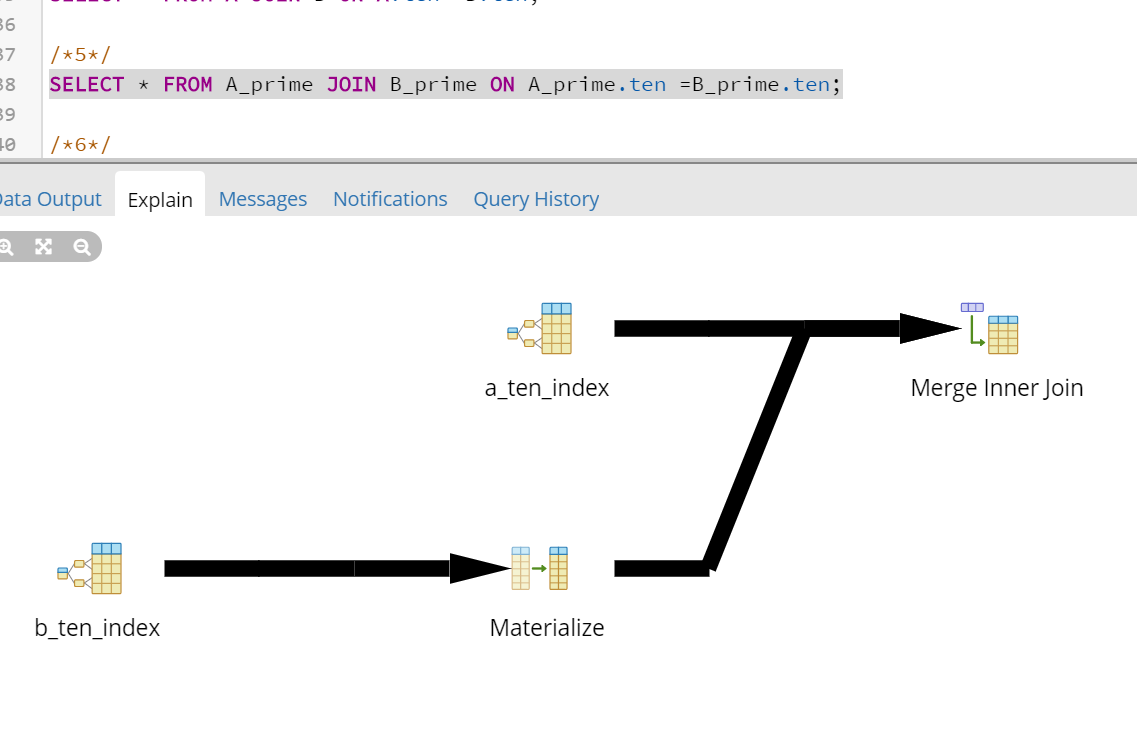


T(A) = 5000000 V(A,ten) = 10

T(B) = 5000000 V(B,ten) = 10

Query plan selects Merge join

1. A'.ten = B'.ten

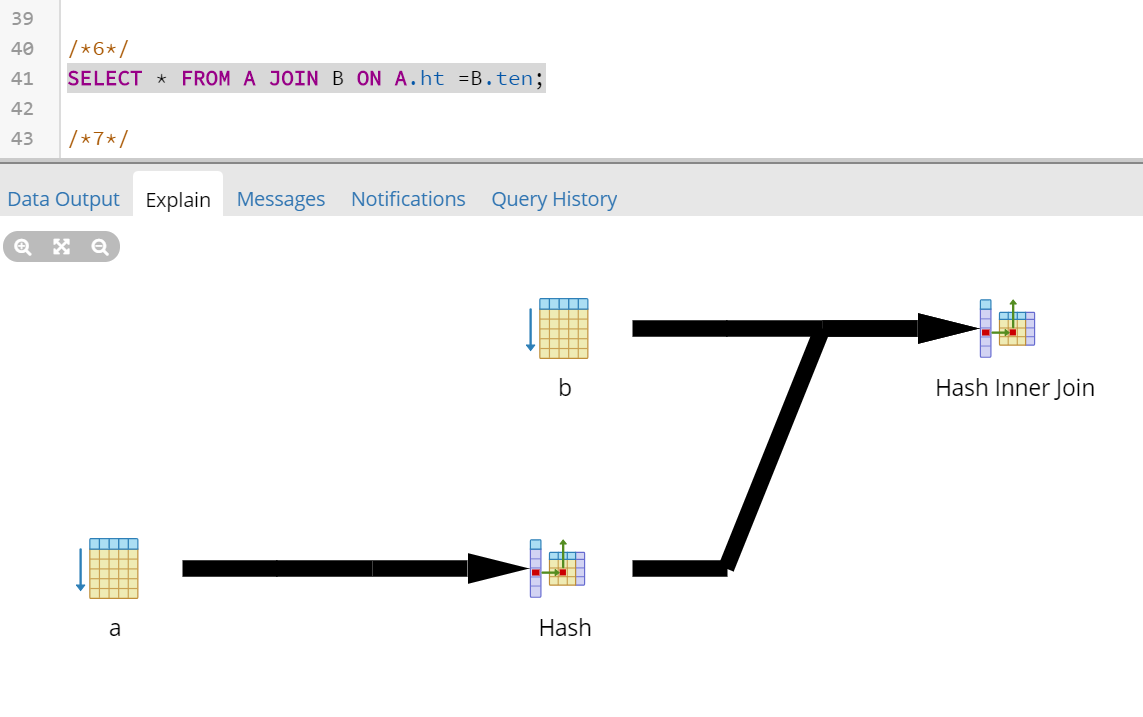


T(A) = 5000000 V(A,ten) = 10

T(B) = 5000000 V(B,ten) = 10

Query plan selects Merge join using the ten\_index to shortcut data sorting

1. A.ht = B.ten

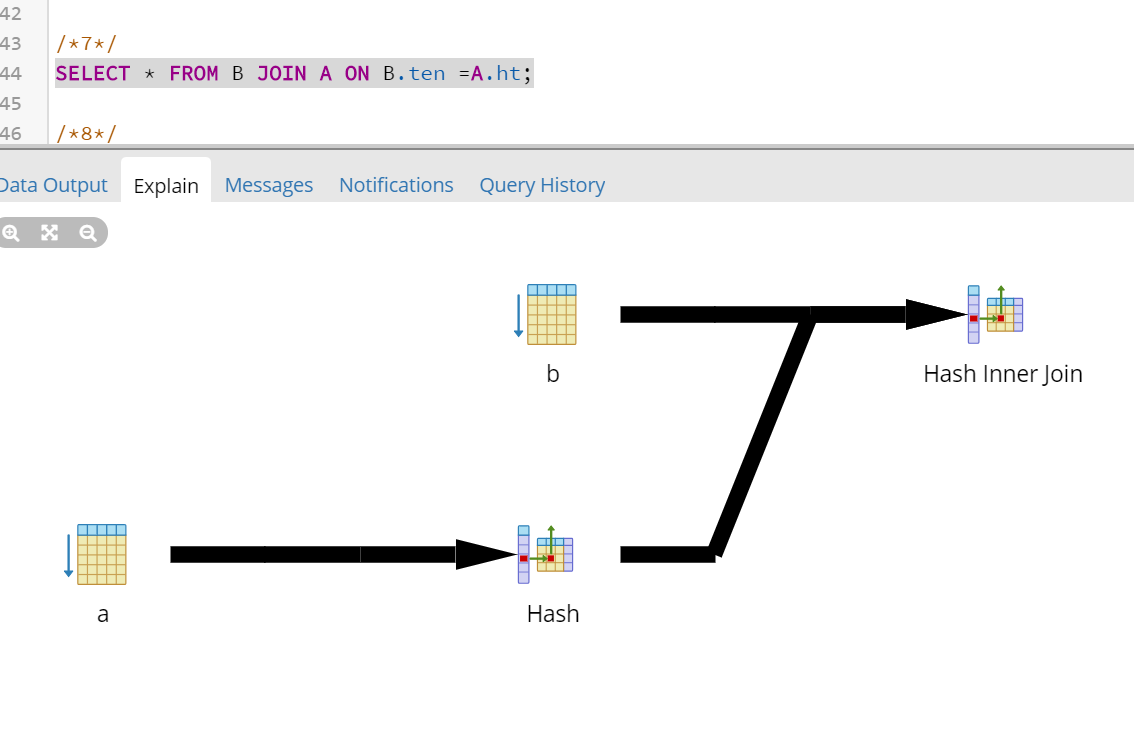


T(A)= 5000000 V(A,ht) = 100000

T(B)= 5000000 V(B,ten) = 10

Selected plan is hash join

1. B.ten = A.ht

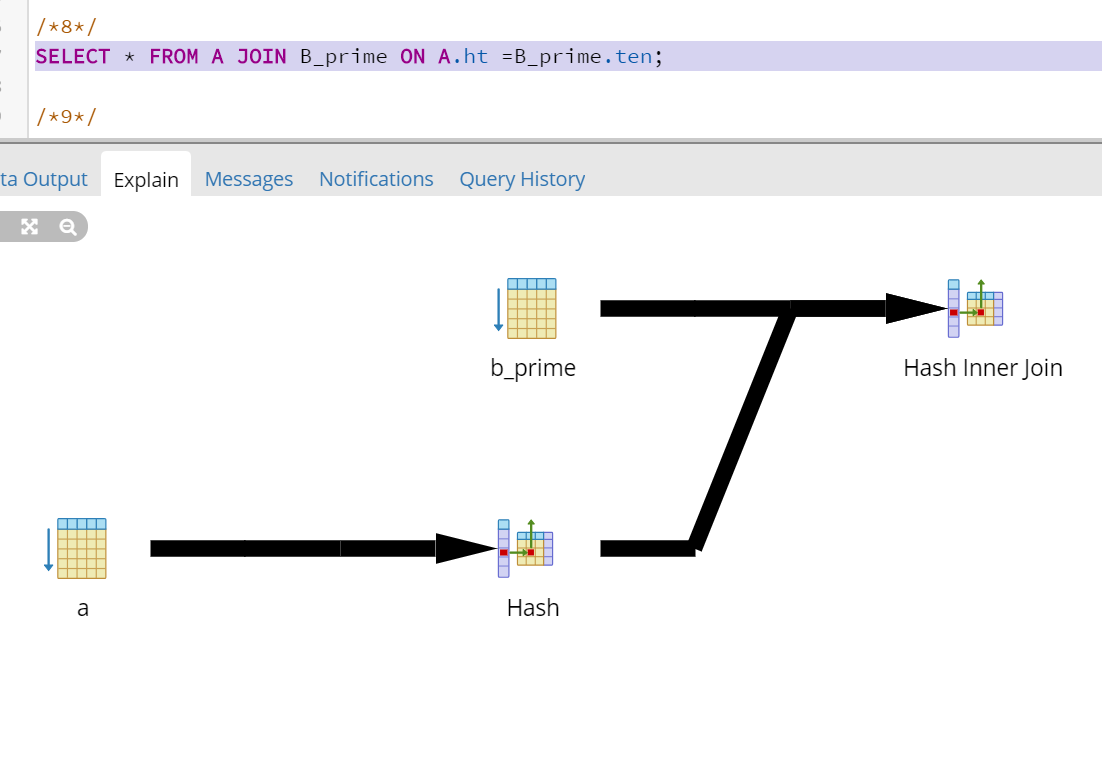


T(A)= 5000000 V(A,ht) = 100000

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

1. A.ht = B'.ten

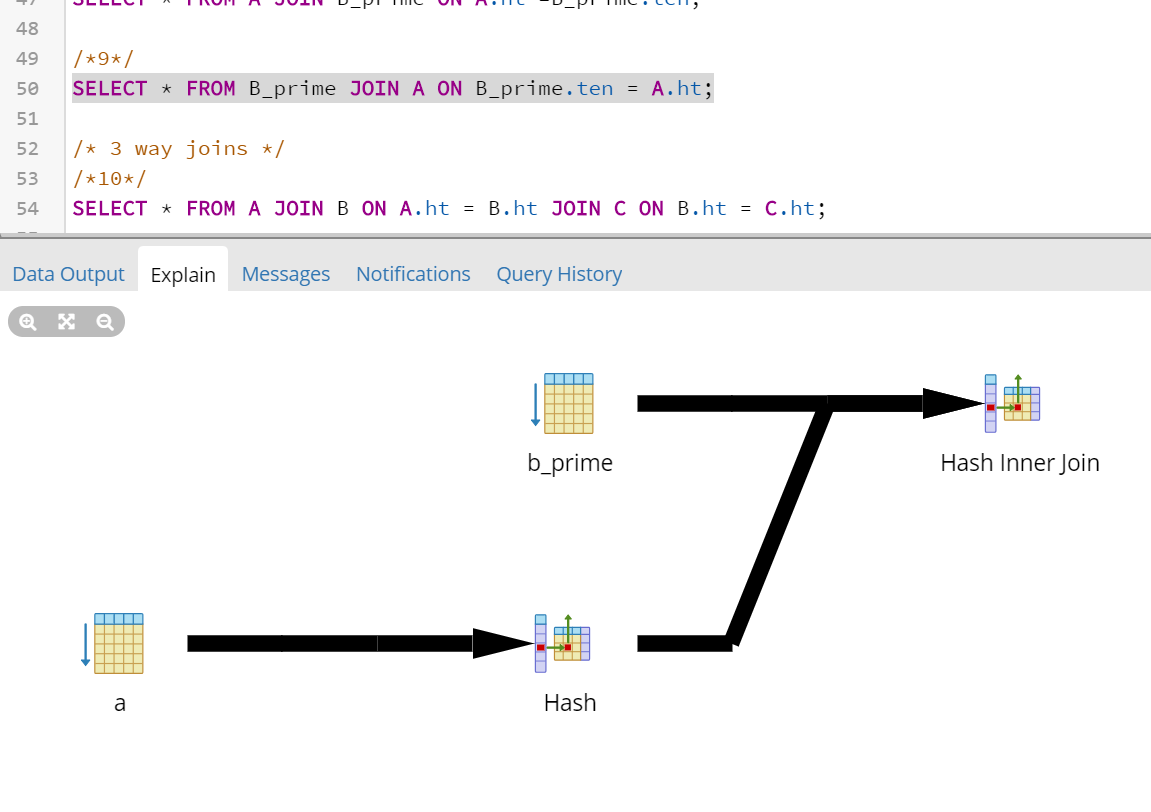


T(A) = 5000000 V(A,ht) = 100000

T(B') = 5000000 V(B',ten)= 10

Selected plan is hash join

1. B'.ten = A.ht



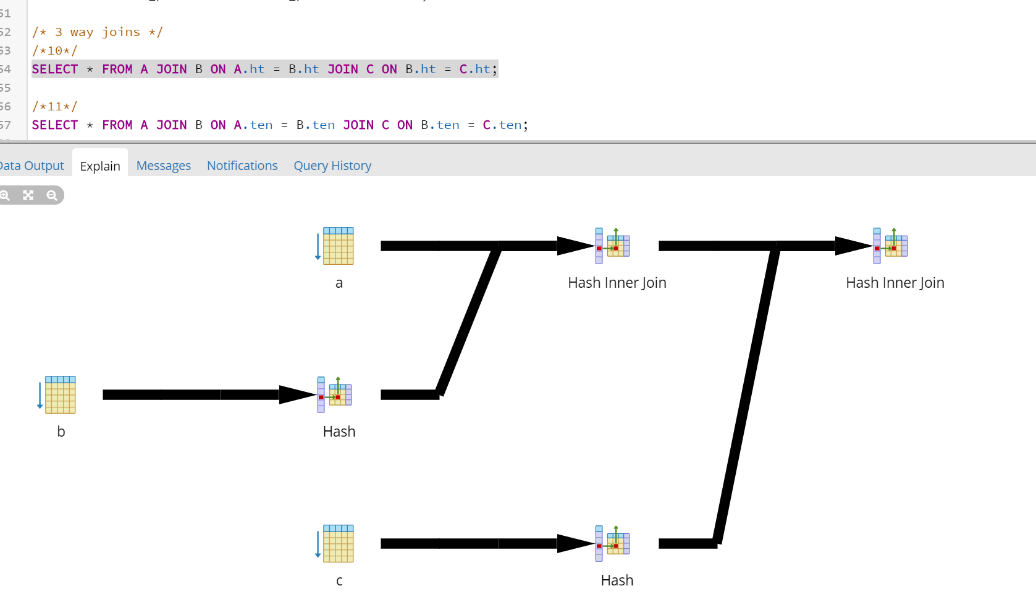
T(A) = 5000000 V(A,ht) = 100000

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

1. A.ht = B.ht = C.ht





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



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



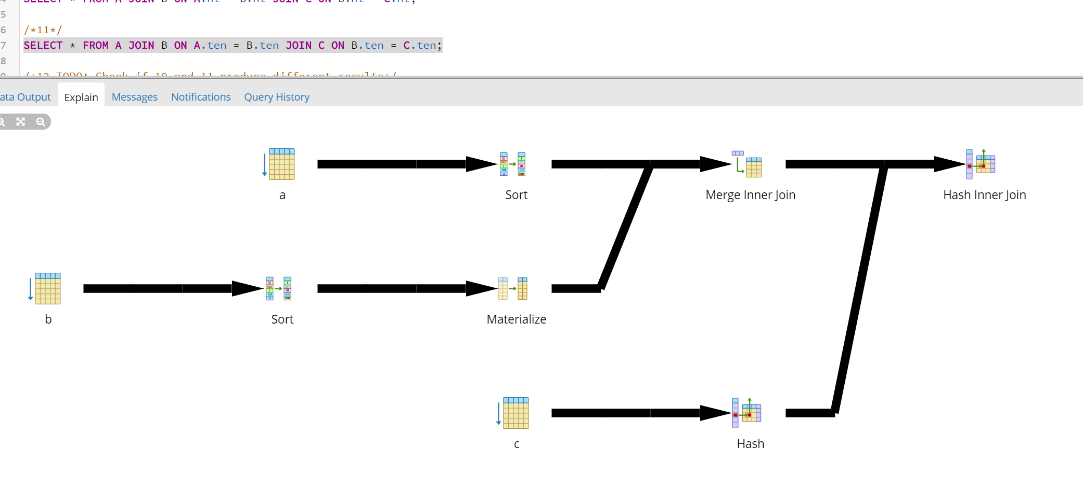
T(c )= 5000000 V(C,ht) = 100000



Selected plan: hash join A,b, hash join c

1. A.ten = B.ten = C.ten







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

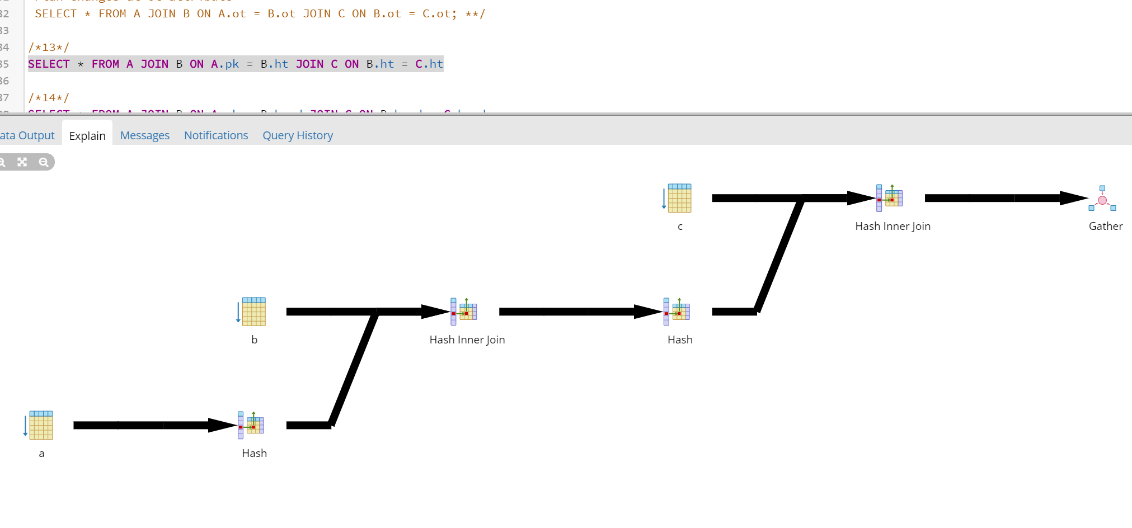


1. Do they produce different plans?

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

1. A.pk = B.ht = C.ht



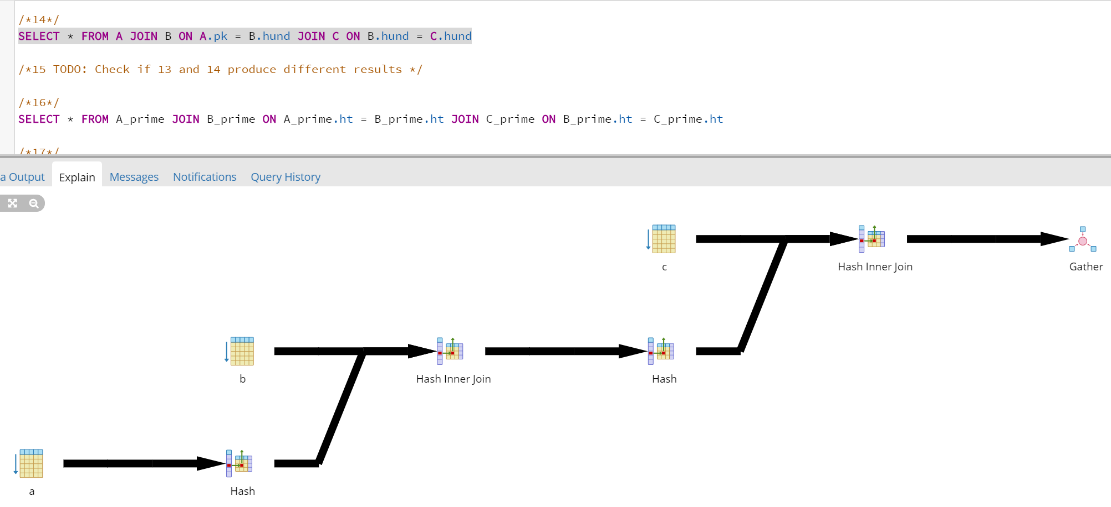
Selected pla n is to hash join a,b then hash join c

T(A) = 5000000 V(A,pk) = 5000000

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

T(C ) = 5000000 V(C,ht) = 100000

1. 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

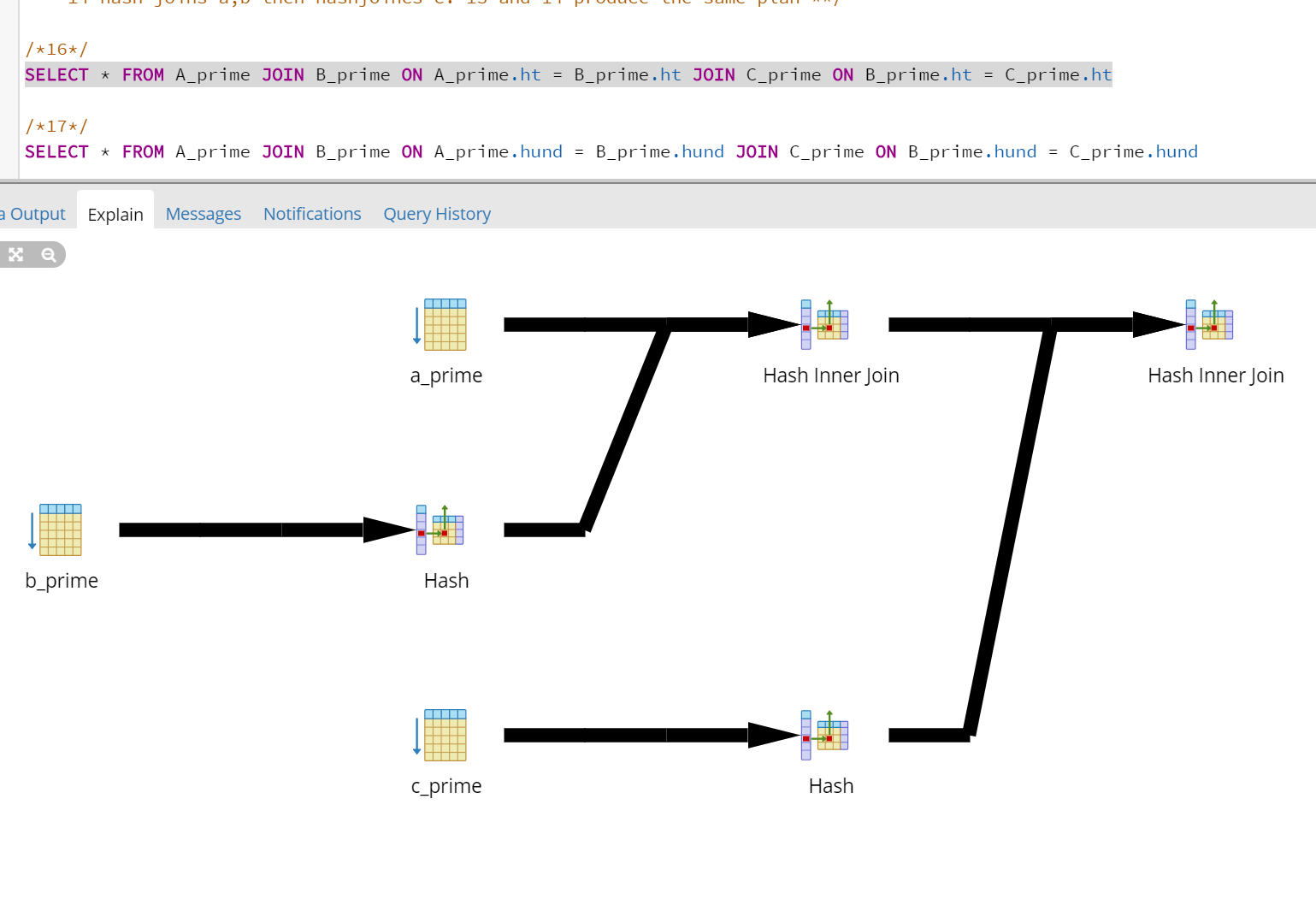
T(B) = 5000000 V(B,hund) = 100

T(C ) = 5000000 V(C,hund) = 100

1. Do they produce different plans?

No, 13 and 14 produce the same plan.

1. A'.ht = B'.ht = C'.ht



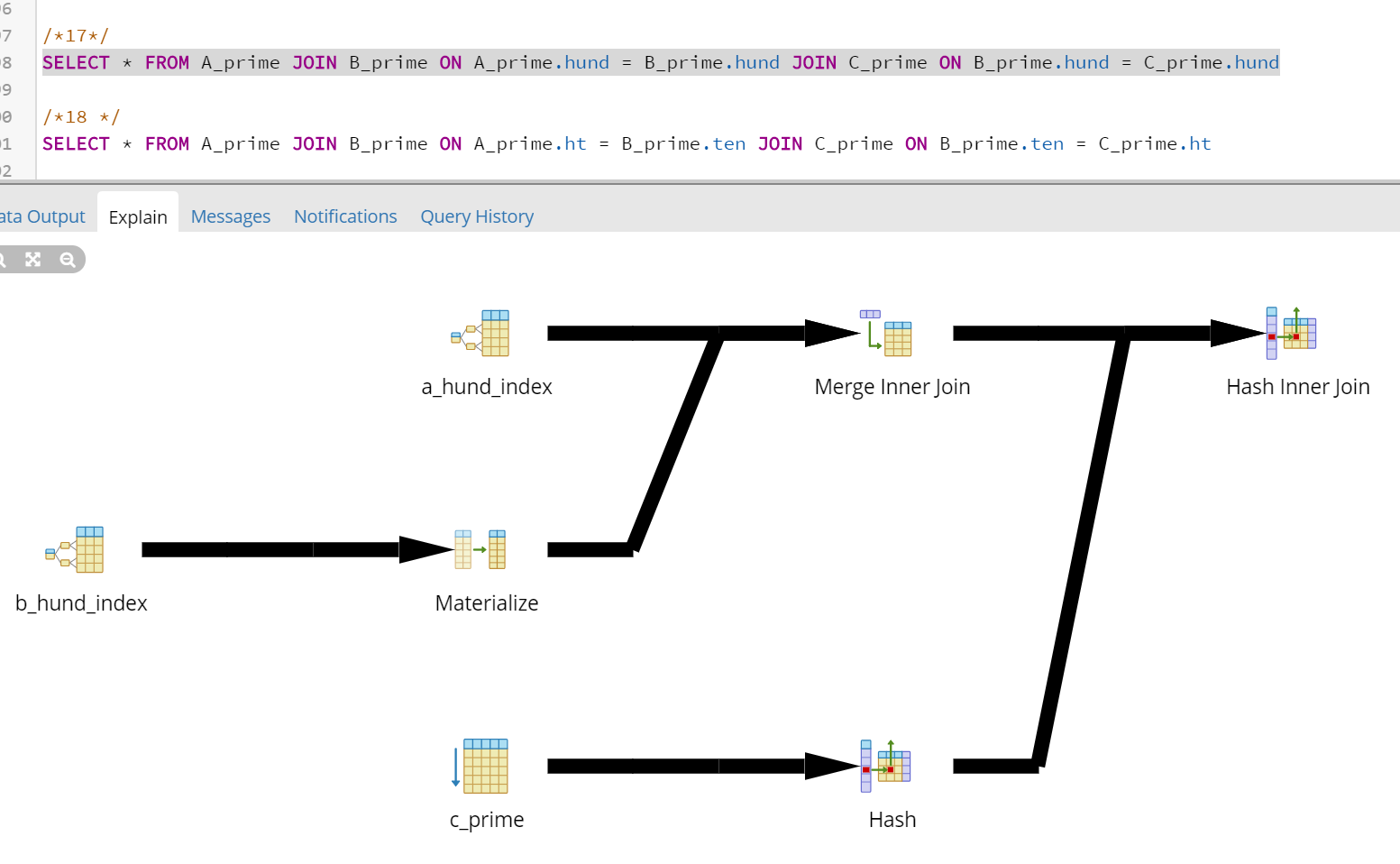
T(A) = 5000000 V(A,ht) = 100000

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

1. A'.hund = B'.hund = C'.hund



T(A) = 5000000 V(A,hund) = 100

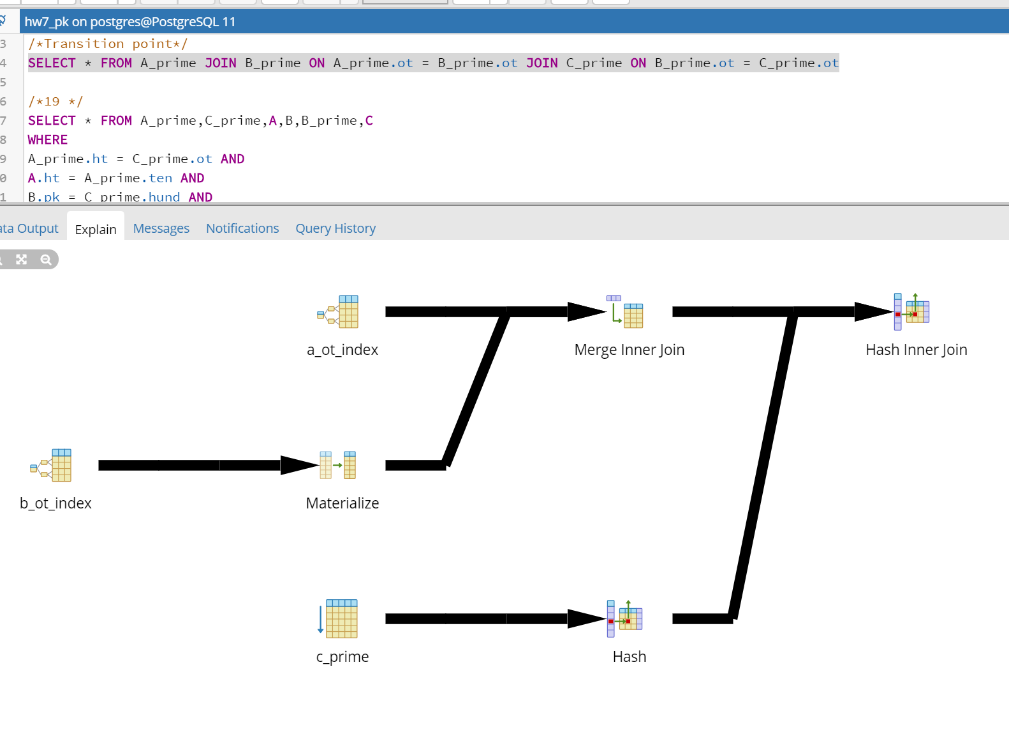
T(B) = 5000000 V(B,hund) = 100

T(c )= 5000000 V(C,hund) = 100

Selected plan: merge join A,b, hash join c. The query plan changes from 16.

1. 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



Section 2:

1. A'.ht = C'.ot AND

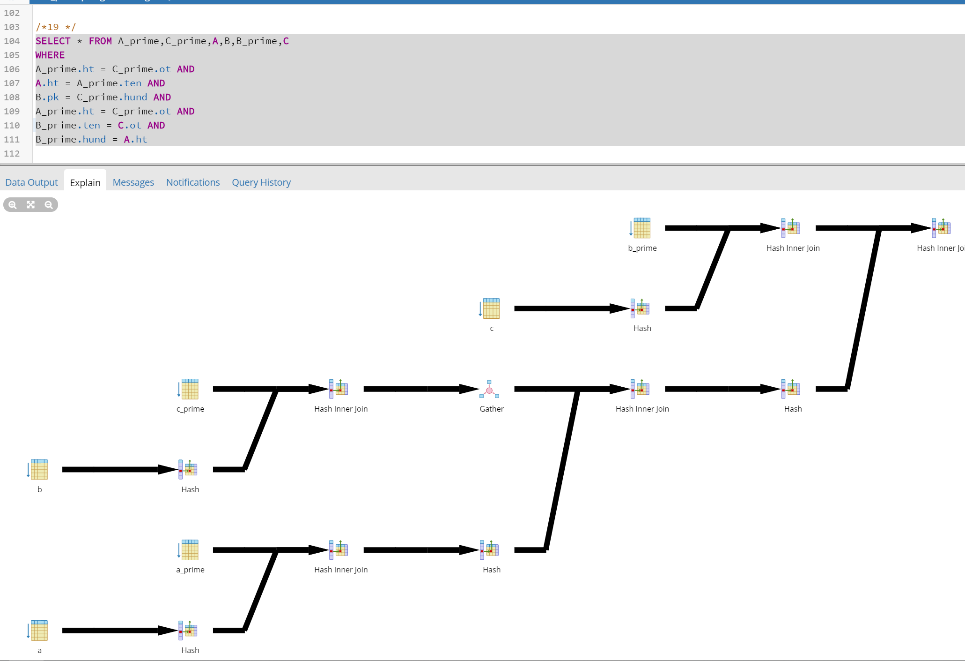
A.ht = A'.ten AND

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)

1. A'ht = C'.ot AND

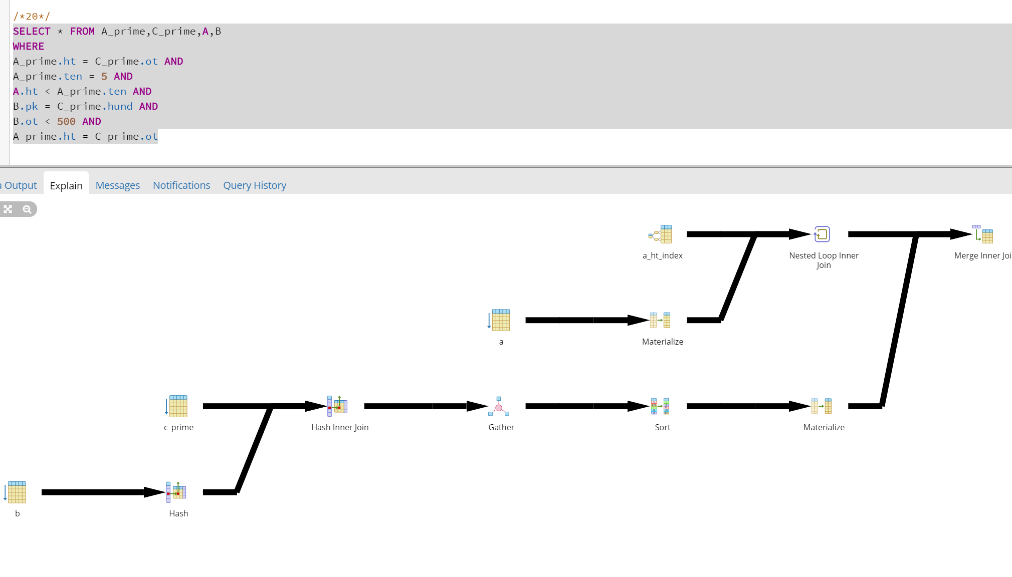
A'.ten = 5 AND

A.ht < A'.ten AND

B.pk = C'.hund AND

B.ot < 500 AND

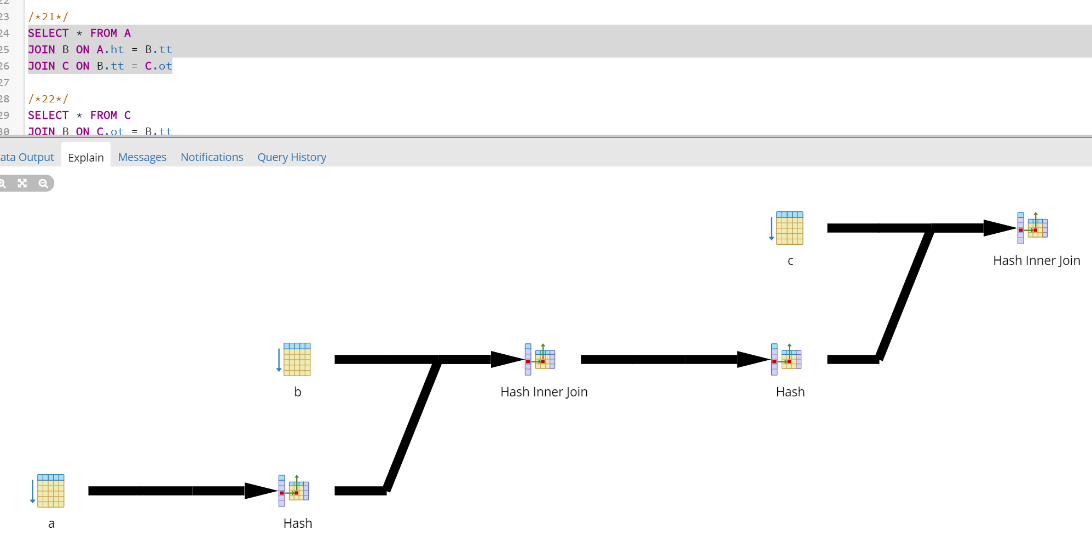
A'.ht = C'.ot



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.

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

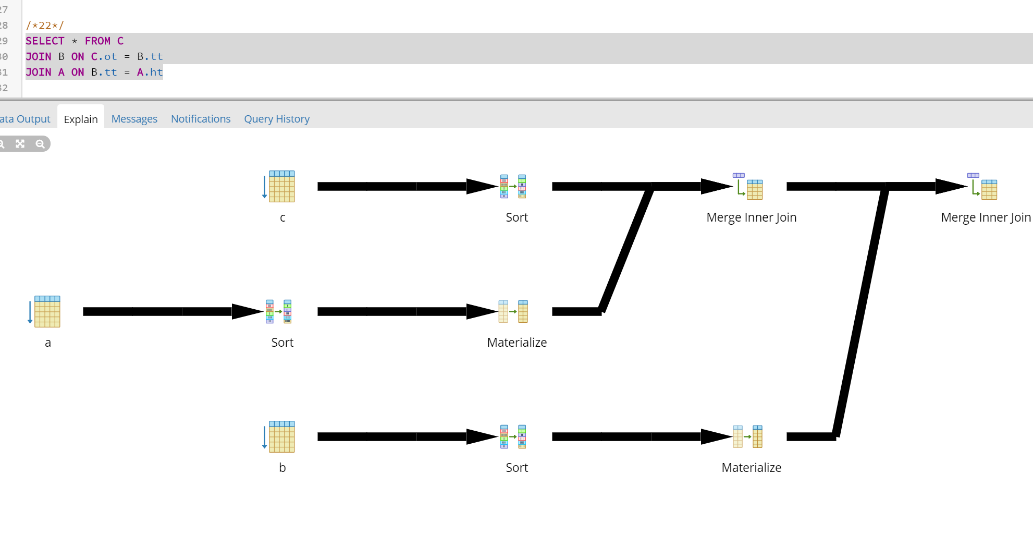


T(A) = 5000000 V(A,ht) = 100000

T(B) = 5000000 V(B,tt) = 10000

T(C ) = 5000000 V(C,ot) = 1000

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



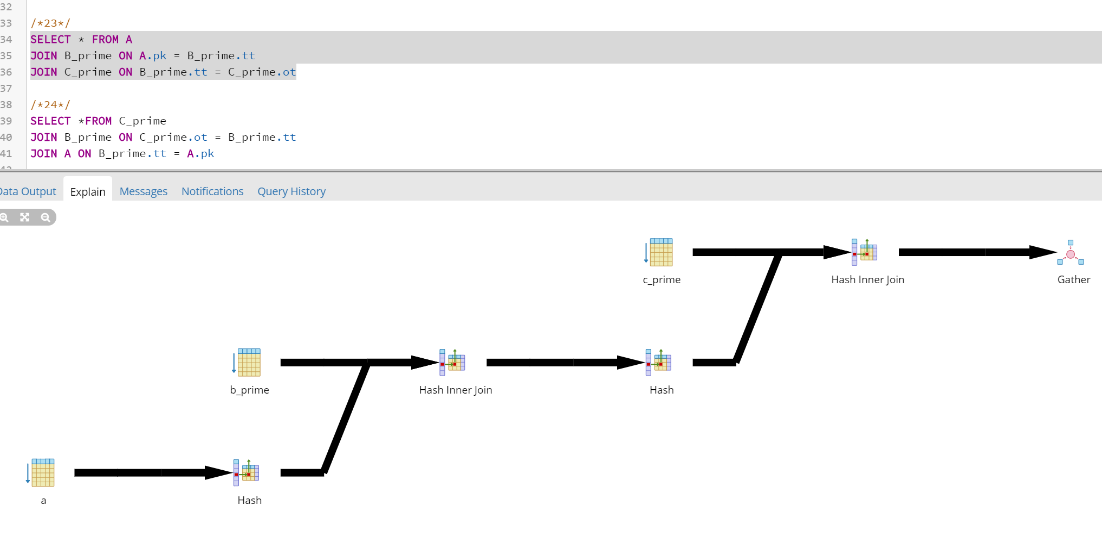
The query plan changes between 21 and 22 even though the only difference is the order of the tables in the query.

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

T(B) = 5000000 V(B,tt) = 10000

T(C ) = 5000000 V(C,ot) = 1000

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

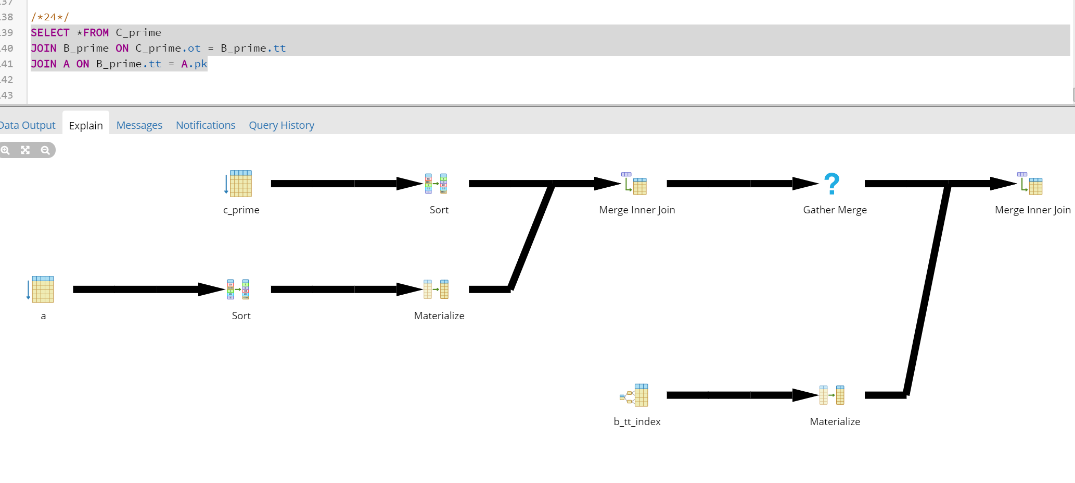


T(A) = 5000000 T(A,pk) = 5000000

T(B) = 5000000 T(B,tt) = 10000

T(C ) = 5000000 T(C,ot) = 1000

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



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