Middle East Technical University

Department of Computer Engineering

CENG 553

Database Management Systems

Spring 2020–2021

Written Assignment 2 - Sample Answers

1 Query Processing

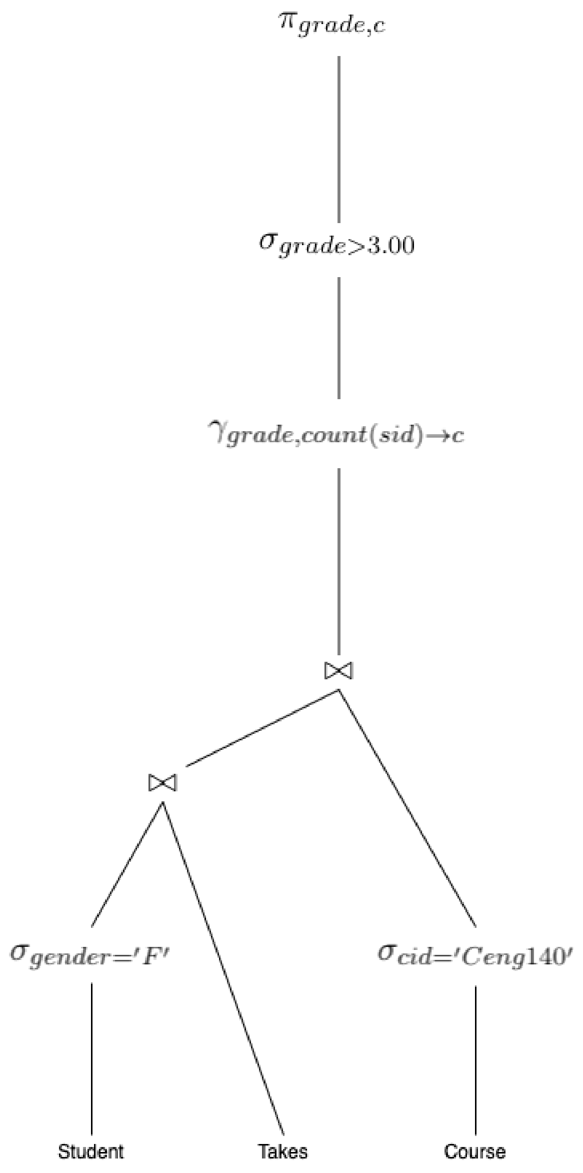
1.1 Part 1

a. The query first eliminates actors over age 30. Then grouping actors by their age, for each age group that has more than 2 actors (eliminates other groups) finds the maximum networth of each age group.
The minimal search key for B+ tree index that allows index only plan is **age**.

b. First query can be evaluated, however, second query can not be evaluated using index only plan with composite index on (year, budget) since the second query contains the attribute **genre** on the Where clause.

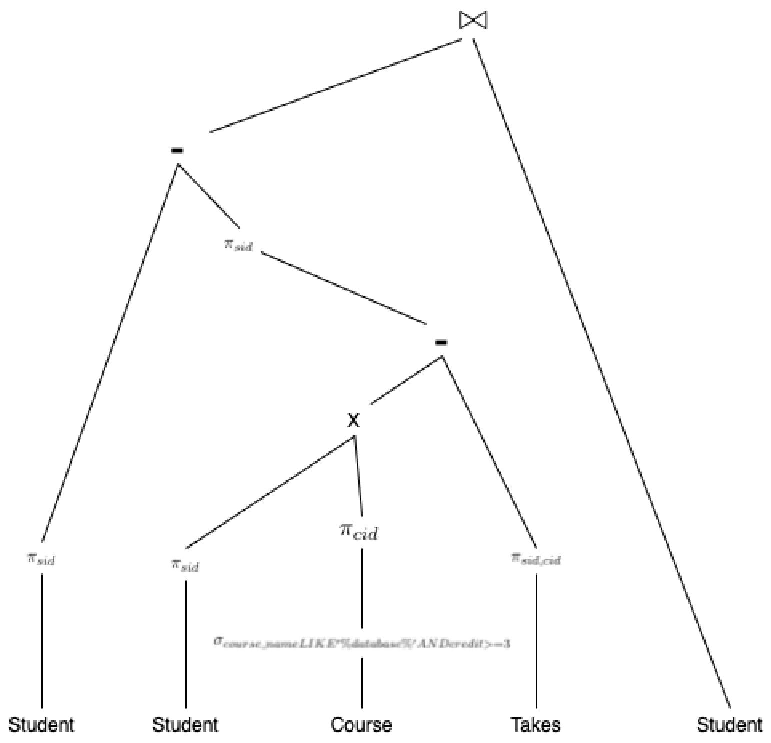
1.2 Part 2

a.



$\pi_{grade,c}$
$\sigma_{grade>3.00}$
$\gamma_{grade,count(sid)\rightarrow c}$
$\bowtie$
$\bowtie$
$\sigma_{gender='F'}$
$\sigma_{cid='Ceng140'}$
Student
Takes
Course

b.


⋈
-
π_{sid}
-
x
π_{sid}
π_{sid}
π_{cid}
$\pi_{sid,cid}$
$\sigma_{course\_name LIKE '\%database\%' AND credit >= 3}$
Student
Student
Course
Takes
Student

2 Join Algorithms

We know the following:

```
T(R) = 40000
B(R) = 40000/10 = 4000

T(S) = 8000
B(S) = 8000/15 = 533.33 (by ceiling) = 534

M = 62
```

a. Block nested loops join can be performed when we have 3 memory pages available (page from R, page from S and output page). Since we satisfy this condition M¿3 (M = 62) we can perform block nested loops join. The cost can be calculated using the formula:
Cost = B(R) + B(R)*B(S) / (M-2) = 4000 + 4000 * 534 / 60 = 39600 IO.

b. Similarly, we can use the following formula:
Cost = B(S) + B(S)*B(R) / (M-2) = 534 + 4000 * 534 / 60 = 36134 IO.

c. As in general, we can not apply one-pass algorithm for this example (since R and S are too large for our memory). However, different from the lectures we can not even make two-pass algorithm since R is again too large for our memory ($B(R) > M^2 \rightarrow 4000 > 3844$), we need to perform 3 pass.
First, sorting S will give 534/62 = 8.6 → 9 runs. These runs will fit in the memory we don't need a second pass for that. The cost of this operation is 2B(S) (read+write).
Sorting R will give 4000/62 = 64.5 → 65 runs. This operation will cost 2B(R).
We need to merge these runs into bigger least number of runs. Again we will sort R as we did before into 2 runs(33/32). This operation will cost 2B(R).
Now we have runs of R and S which can fit in the memory separately or together. We need to merge them. The cost of the merge will be B(R) + B(S).
By adding all of them together we will get → 2B(S) + 2B(R) + 2B(R) + B(R) + B(S) = 5B(R) + 3B(S) = 4000 * 5 + 534 * 3 = 21602.

d. We will use our first hash function in order to create partitions of R:
Read R, use the hash function, write partitions to the disk 2B(R).
Read S, use the hash function, write partitions to the disk 2B(S).
Read partitions from R by using the second hash function, hash them in the memory, read partitions from S by using the second hash function find matchings. The cost is B(R) + B(S).
We are able to apply these since min(B(R), B(S)) ¡ M2 → 534 ¡ 62*62 (3844)
The total cost will be 3B(R) + 3B(S) = 3*4000 + 3*534 = 13602.

e. For each tuple of R, we need to find corresponding match of it using the index on the S.
Read R, for each tuple, check the index and find the matches.
For the calculations we will need V(S,y). Since y is the primary key of S all will be unique and V(S,y) = 8000.
If it is clustered the cost will be: B(R) + T(R)B(S)/V(S,y) = 4000 + 40000*534/8000 = 4000 + 2670 = 6670.
If it is unclustered the cost will be: B(R) + T(R)T(S)/V(S,y) = 4000 + 40000*8000/8000 = 4000 + 40000 = 44000.

f. V(S,z) = 20.

   i. For clustered index → Cost = B(S)/V(S,z) = 534/20 = 26.7 (ceiling) = 27.

   ii. For unclustered index → Cost = T(S)/V(S,z) = 8000/20 = 400.

   iii. No index → Cost = B(S) = 534. Only clustered index is giving good results. Unclustered index and no index are similar in results, so the best option is clustered index. If you are not able to use clustered index, choosing no index might be better (depends on the situation).

3 Physical query plan

a. Since the attribute hashtag has unclustered index, both the result size and cost will be equal to T(T)/V(T,hashtag) = 600000/50000 = 12.

b. Since there are approximately 12 tweets and 100 users with corona hashtag:
   Result size = 12*100 = 1200.
   Cost: B(R)/V(R,hashtag) = 3 pages of users. For each user there are 12 tweets, so cost will be 3*12 = 36

c. Since it is on the fly, cost = 0. There are 1200 results coming from part b, since there are 1000 different users. The cardinality of the 'Selin' users will be 1200/1000 = 1.2 = 2.

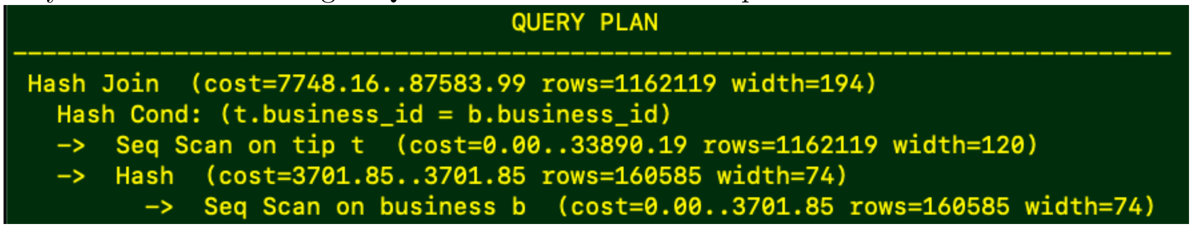
d. Again as part a, both the cost and result size will be equal to T(R)/V(R,user) = 3000000/1000 = 3000.

e. Using the unlimited memory assumption, the join operation will not have any cost.
   From part a, we know that there are 12 tweets. So, there are 12 * 3000 / min(V(R,hashtag), V(T,hashtag)) = 12 * 3000/30000 = 1.2 = 2

f. The cost of the plan 1: 12 + 36 + 0 = 48.
   The cost of the plan 2: 12 + 3000 + 0 = 3012.
   Plan 1 seems a better option.

4 Experiments

a. As you can see that PostgreSQL favors Hash Join and Sequential Scan:



```
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=7748.16..87583.99 rows=1162119 width=194)
   Hash Cond: (t.business_id = b.business_id)
   ->  Seq Scan on tip t  (cost=0.00..33890.19 rows=1162119 width=120)
   ->  Hash  (cost=3701.85..3701.85 rows=160585 width=74)
         ->  Seq Scan on business b  (cost=0.00..3701.85 rows=160585 width=74)
```

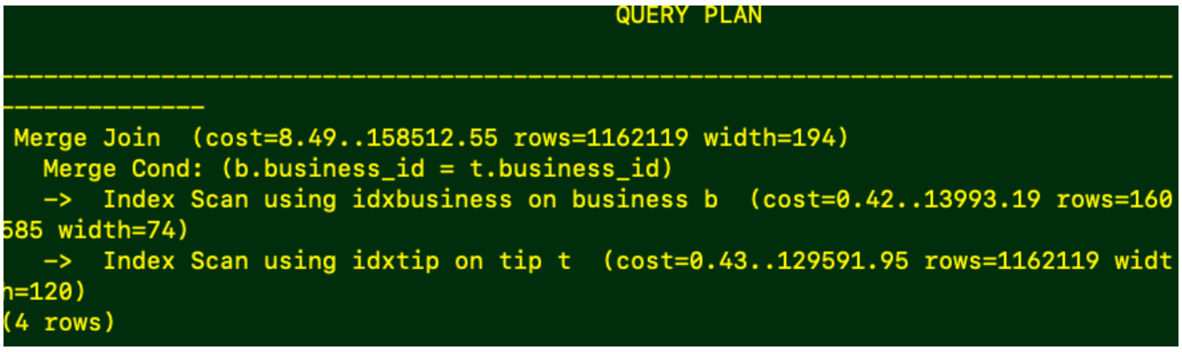
b. Creating and clustering indexes on business_id will not change the execution times. Since the Postgresql always favors sequential scan when joining the whole relation (all tuples). Also, business_id is already primary key on the table business. By default, postgresql creates indexes on the primary key.

c. If you perform not equality on the tables. PostgreSQL performs Nested Loop join this time.

d. I will only explain part a. If you disable the Hash join using the following command:

```
set enable_hashjoin to off;
```

The result will be:



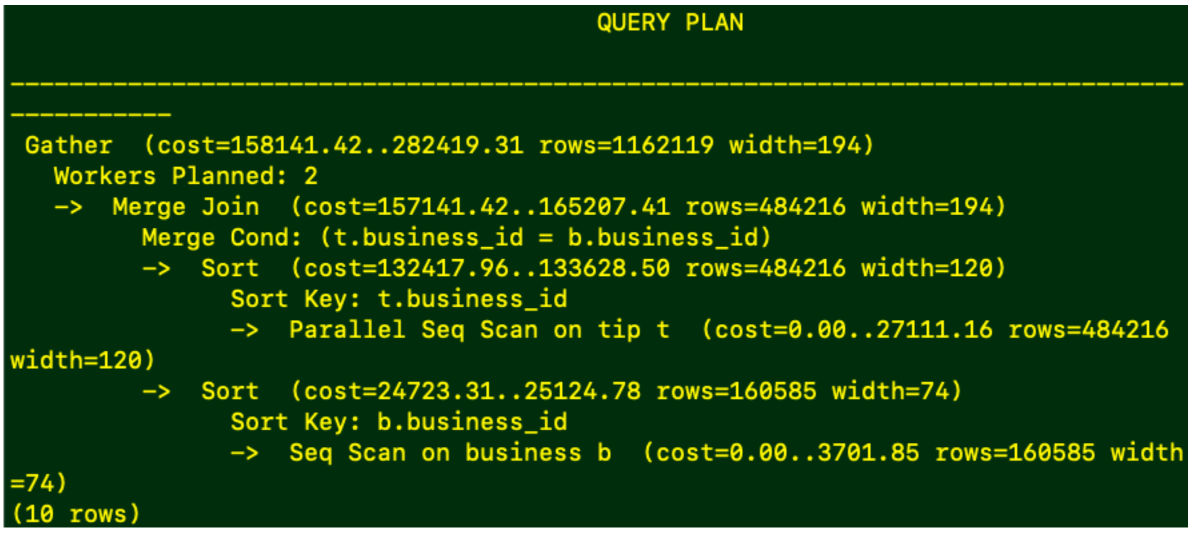
```
                                          QUERY PLAN

--------------------------------------------------------------------------------
--------------
 Merge Join  (cost=8.49..158512.55 rows=1162119 width=194)
   Merge Cond: (b.business_id = t.business_id)
   ->  Index Scan using idxbusiness on business b  (cost=0.42..13993.19 rows=160
585 width=74)
   ->  Index Scan using idxtip on tip t  (cost=0.43..129591.95 rows=1162119 widt
h=120)
(4 rows)
```

Then if you disable index scan using:

```
set enable_indexscan to off;
```

The result will be:



```
                                                          QUERY PLAN
--------------------------------------------------------------------------------
-----------
 Gather  (cost=158141.42..282419.31 rows=1162119 width=194)
   Workers Planned: 2
   ->  Merge Join  (cost=157141.42..165207.41 rows=484216 width=194)
         Merge Cond: (t.business_id = b.business_id)
         ->  Sort  (cost=132417.96..133628.50 rows=484216 width=120)
               Sort Key: t.business_id
               ->  Parallel Seq Scan on tip t  (cost=0.00..27111.16 rows=484216
width=120)
         ->  Sort  (cost=24723.31..25124.78 rows=160585 width=74)
               Sort Key: b.business_id
               ->  Seq Scan on business b  (cost=0.00..3701.85 rows=160585 width
=74)
(10 rows)
```

e. You can create an index on (state, avg_stars) on the relation.

f. The indexes created before is helpful for their own purposes. However, changing the clustered index (for example part e) will affect all other queries and slow their execution times.