Relational algebra and query execution

CSE 444, fall 2010 — section 8 worksheet

November 18, 2010

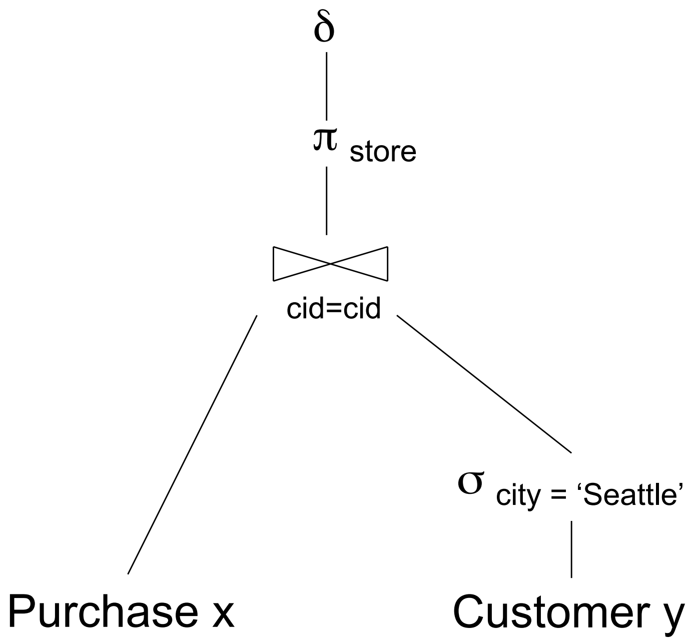
1 Relational algebra warm-up

1. Given this database schema:

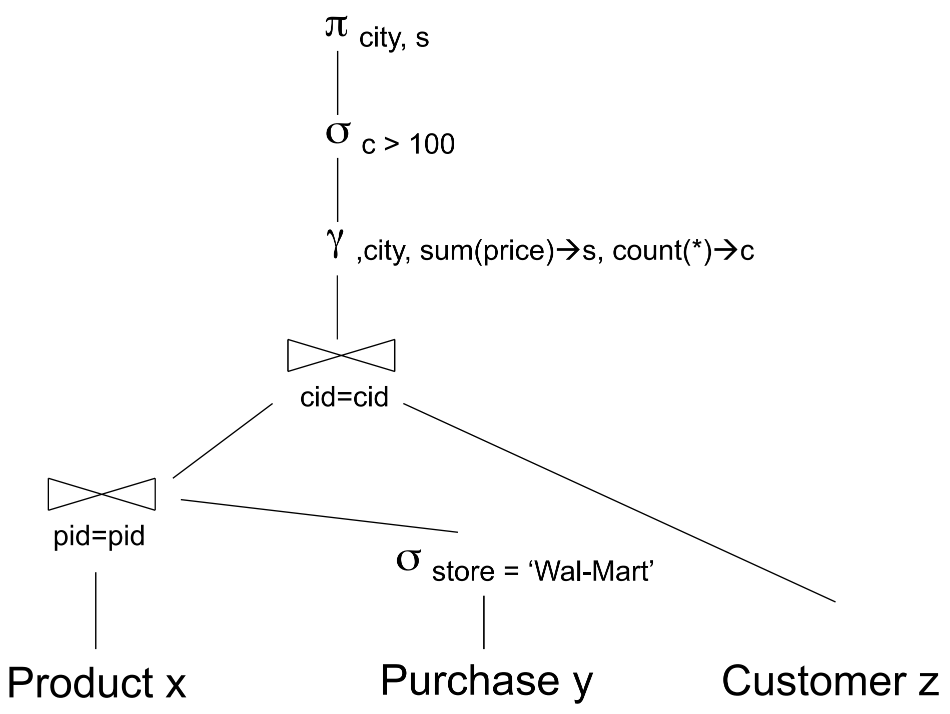
   - Product (pid, name, price)
   - Purchase (pid, cid, store)
   - Customer (cid, name, city)

   draw the logical query plan for each of the following SQL queries.

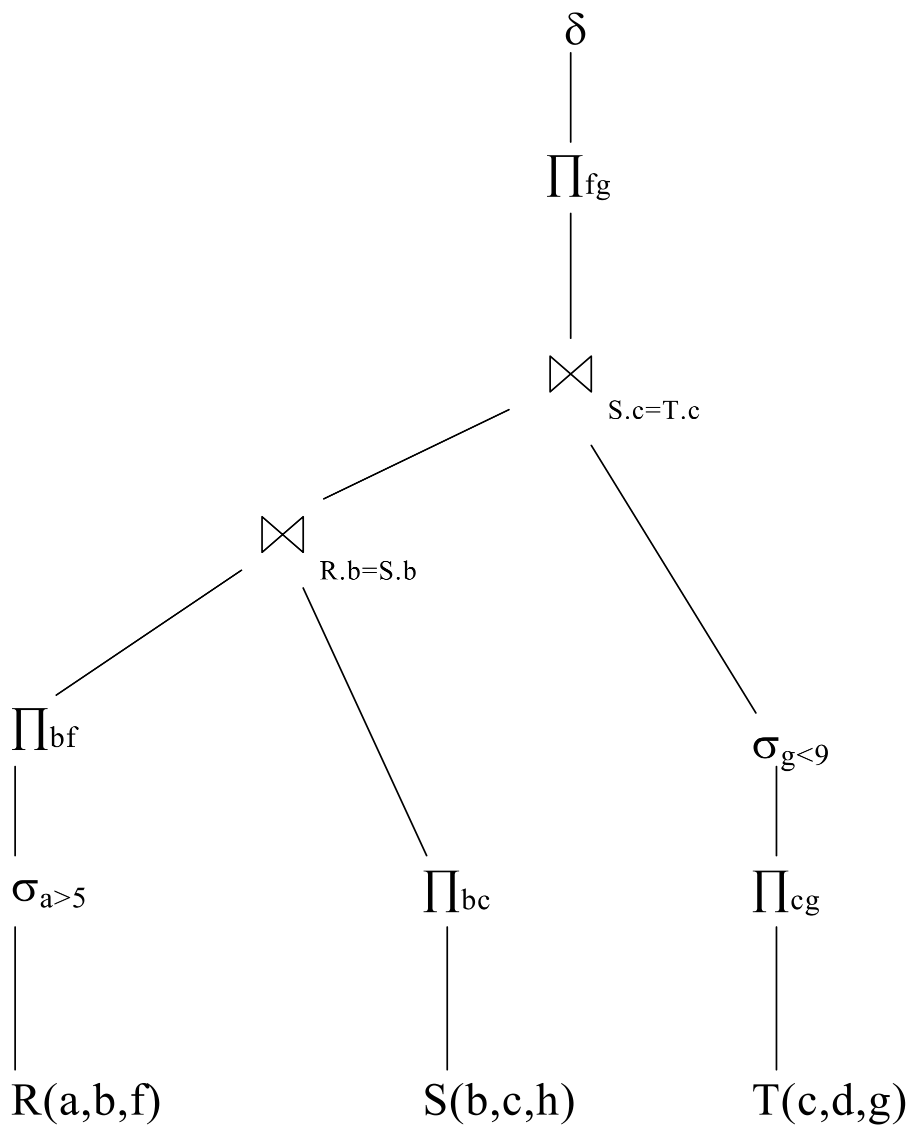
   (a)
```
SELECT DISTINCT x.store
FROM Purchase x, Customer y
WHERE x.cid = y.cid
      and y.city = 'Seattle'
```
   Solution:



(b)
```
SELECT z.city, sum(x.price)
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid
     and y.store = 'Wal-Mart'
GROUP BY z.city
HAVING count(*) > 100
```
Solution:



2. Write a SQL query that is equivalent to the logical plan below:



Solution:

```
SELECT DISTINCT R.f, T.g
FROM R, S, T
WHERE R.a > 5 and T.g < 9 and
      R.b = S.b and S.c = T.c
```

3. Consider two tables $R(A, B, C)$ and $S(D, E, F)$, and the logical query plan P:

$$P = \sigma_{A>9}\left(\gamma_{A,\text{sum}(F)}\left(R \bowtie_{C=D} S\right)\right)$$

Indicate which of the following three query plans are equivalent to P.

$$P_1 = \gamma_{A,\text{sum}(F)}\left((\sigma_{A>9}R) \bowtie_{C=D} \left(\gamma_{D,\text{sum}(F)}S\right)\right)$$
$$P_2 = \gamma_{A,\text{sum}(F)}\left((\sigma_{A>9}R) \bowtie_{C=D} \left(\gamma_{D,E,\text{sum}(F)}S\right)\right)$$
$$P_3 = \gamma_{A,\text{sum}(F)}\left((\sigma_{A>9}R) \bowtie_{C=D} \left(\gamma_{D,E,\text{sum}(F)}\left((\sigma_{A>9}R) \bowtie_{C=D} (S)\right)\right)\right)$$

Hint: Draw out each plan in tree form before solving.

Solution:
P_1 and P_2 are equivalent, but P_3 is not.

As an aside, you can tell the difference between P, P_2, and P_3 by running them on the following example tables:

Table 1: R

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1

Table 2: S

D	E	F
c_1	e_1	f_1
c_2	e_1	f_2
c_1	e_1	f_3

2 Spring 2009 final, problem 3 (parts a-b)

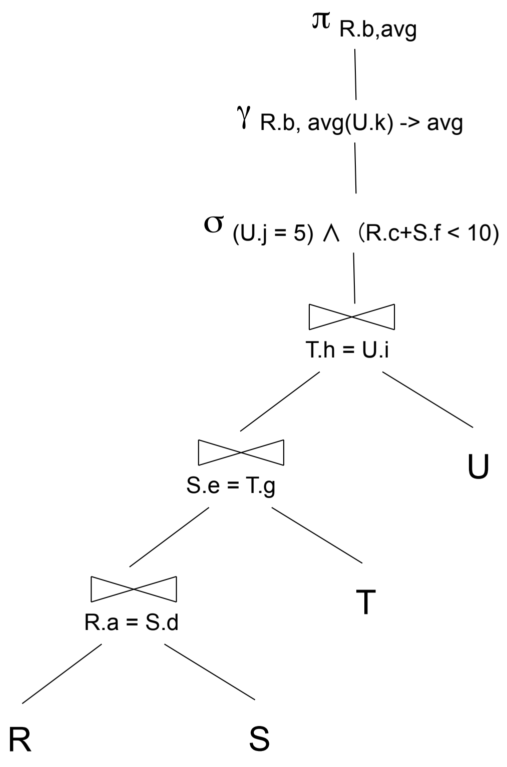
Consider four tables $R(a, b, c)$, $S(d, e, f)$, $T(g, h)$, $U(i, j, k)$.

(a) Consider the following SQL query:

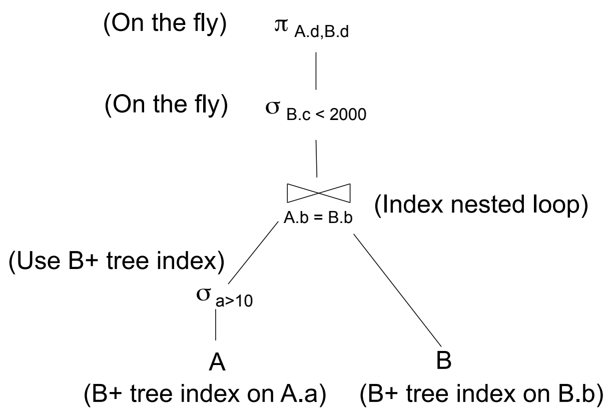
```
SELECT   R.b, avg(U.k) as avg
FROM     R, S, T, U
WHERE    R.a = S.d
         AND S.e = T.g
         AND T.h = U.i
         AND U.j = 5
         AND (R.c + S.f) < 10
GROUP BY R.b
```

Draw a *logical* plan for the query. You may choose any plan as long as it is correct (i.e. no need to worry about efficiency).

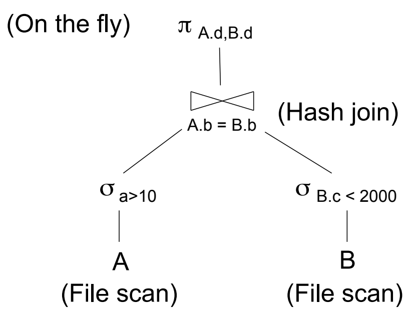
Solution:
Many solutions were possible including:



(b) Consider the following two physical query plans. Give **two** reasons why plan B may be faster than plan A. **Explain** each reason.



Plan A



Plan B

Solution:
The following are three possible reasons why plan B could be faster than plan A:

- The low selectivity of the selection predicate ($a > 10$) and an unclustered index on A.a may make a file scan of A faster than using the index.
- If there are lots of matches, hash joins may be faster than index nested loops. The hash join reads its input only once (unless the input is too large). The index-nested loop may end-up reading the same pages of B multiple times.
- Pushing the selection ($B.c < 2000$) down can get rid of lots of B tuples before the join, reducing the cost of that operation.

3 Summer 2009 final, problem 2

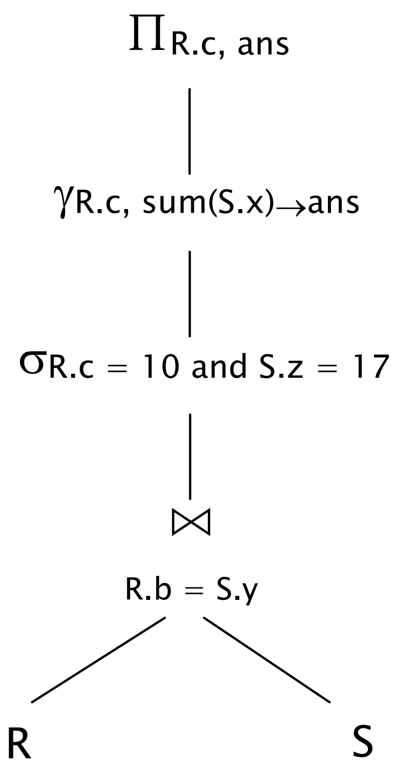
This problem and the next concern two relations $R(a, b, c)$ and $S(x, y, z)$ that have the following characteristics:

$B(R) = 600$	$B(S) = 800$
$T(R) = 3000$	$T(S) = 4000$
$V(R, a) = 300$	$V(S, x) = 100$
$V(R, b) = 100$	$V(S, y) = 400$
$V(R, c) = 50$	$V(S, z) = 40$

We also have $M = 1000$ (number of memory blocks).

Relation R has a clustered index on attribute a and an unclustered index on attribute c. Relation S has a clustered index on attribute x and an unclustered index on attribute z. All indices are B+ trees.

Now consider the following logical query plan for a query involving these two relations.


$\Pi_{\text{R.c, ans}}$
$\gamma_{\text{R.c, sum(S.x)} \to \text{ans}}$
$\sigma_{\text{R.c = 10 and S.z = 17}}$
$\bowtie$
R.b = S.y
R
S

(Answer the questions about this query on the following page.)

(a) Write a SQL query that is equivalent to the logical query plan on the previous page.

Solution:

```
SELECT R.c, sum(S.x) AS ans
FROM R, S
WHERE R.b = S.y AND R.c = 10 AND S.z = 17
GROUP BY R.c
```

(b) Change or rearrange the original logical query plan to produce one that is equivalent (has the same final results), but which is estimated to be significantly faster, if possible. Recall that logical query optimization does not consider the final physical operators used to execute the query, but only things at the logical level, such as the sizes of relations and estimated sizes of intermediate results.

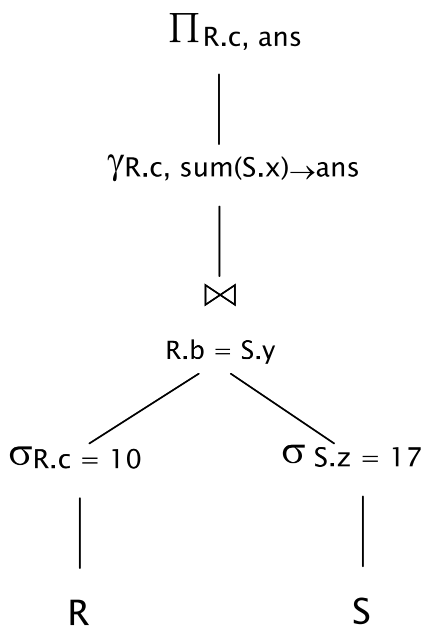
You should include a brief but specific explanation of how much you expect your changes to improve the speed of the query and why.

Draw your new query plan and write your explanation below. *If* you can clearly and easily show the changes on the original diagram, you may do so, otherwise draw a new diagram here.

Solution:

In the original plan, the estimated size of the join result would be $T(R)T(S)/\max(V(R,b),V(S,y)) =$ 30,000 tuples, which then have to be passed through the select operation.

The most useful change would be to push the select operations down below the join (as shown below). If that is done, the estimated result size of $\sigma_{R.c=10}(R)$ is $T(R)/V(R,c) = 60$ and of $\sigma_{S.z=17}(S)$ is $T(S)/V(S,z) = 100$. The estimated size of the join is then $60 \times 100/400$ or 15. The amount of work done by the group and project operations would remain the same as in the original plan.



4 Summer 2009 final, problem 3

This problem uses the same two relations and statistics as the previous one, but for a different operation not related to the previous query.

As before, the relations are $R(a,b,c)$ and $S(x,y,z)$. Here are the statistics again:

$B(R) = 600$ $B(S) = 800$
$T(R) = 3000$ $T(S) = 4000$

$V(R,a) = 300$ $V(S,x) = 100$
$V(R,b) = 100$ $V(S,y) = 400$
$V(R,c) = 50$ $V(S,z) = 40$
$M = 1000$ (number of memory blocks)

Also as before, relation R has a clustered index on attribute a and an unclustered index on attribute c. Relation S has a clustered index on attribute x and an unclustered index on attribute z. All indices are B+ trees.

Your job for this problem is to specify and justify a good physical plan for performing the join

$$R \bowtie_{a=z} S \qquad \text{(i.e., join } R \text{ and } S \text{ using the condition } R.a = S.z\text{)}$$

Your answer should specify the physical join operator used (hash, nested loop, sort-merge, or other) and the access methods used to read relations R and S (sequential scan, index, etc.). Be sure to give essential details: i.e., if you use a hash join, which relations(s) are included in hash tables; if you use nested loops, which relation(s) are accessed in the inner and outer loops, etc.

Give the estimated cost of your solution in terms of number of disk I/O operations needed (you should ignore CPU time and the cost to read any index blocks).

You should give a brief justification why this is the best (cheapest) way to implement the join. You do not need to exhaustively analyze all the other possible join implementations and access methods, but you should give a brief discussion of why your solution is the preferred one compared to the other possibilities.

(Write your answer on the next page. You may remove this page for reference if you wish.)

Give your solution and explanation here.

Solution:
A simple hash join where both R and S are scanned sequentially will be as good as anything. The cost of that is $B(R) + B(S)$. We have enough memory to store either R or S in a hash table, so there is particular advantage to picking either one if we view this query in isolation. From the system's standpoint, it would be better to hash R and then read blocks of S to perform the join. R is smaller, so storing it in the hash table will tie up less memory.

There is no advantage to using indices to read the blocks of the relations compared to a simple scan, and there can be a significant cost. We can scan the blocks of either relation at a cost of $B(R)$ or $B(S)$. If we look for tuples during the join using an index the costs will be higher, even with the clustered index, since we need to find matching tuples in one relation depending on the attribute values of the current tuple in the other, and those may be in any order.

A nested loop join is at least as expensive. We could hold all of one relation in memory and scan it repeatedly as we scan the other a block at a time, but the I/O cost will still be at least $B(R) + B(S)$ and we will be repeatedly scanning the relation stored in memory. A naïve loop join where we read the blocks of the inner relation for each block of the outer one would cost $B(R) + B(R) \times B(S)$ or $B(S) + B(S) \times B(R)$, depending on which relation we scan in the outer loop, which would be more expensive.

A standard sort-merge join would be worse since we will need to sort one relation, write that to disk, sort the other, then read the first relation back a block at a time to join it to the sorted tuples of the other one. If we sort R first to minimize the number of blocks in the temporary file, the total cost of a sort-merge join would be $3B(R) + B(S)$. A naïve sort-merge join that writes both sorted relations to temporary files would be even worse, with a cost of $3B(R) + 3B(S)$.

If we were clever, we could do a sort-merge join where we sort S in memory at the cost $B(S)$, then read the corresponding tuples of R using the clustered index on a at a cost of $B(R)$ for a total cost of $B(S) + B(R)$. But this has the same I/O costs as the hash join, and requires an extra sort operation.

Note: On the actual test, answers were not expected to be this long or detailed, but did need to pick the right algorithm, analyze the cost, and give at least a general argument about why this was the best choice.