Database System Implementation

Solutions for Chapter 7

Solutions for Section 7.1

Exercise 7.1.1(a)

There are many ways to handle the `DISTINCT` keyword. Given the grammar rules defined so far, the easiest might be to treat it as an option at the beginning of a select-list. That is, we change the rules for <SelList> to

```
<SelList> ::= <OptDistinct><SelList1>
<OptDistinct> ::= DISTINCT
<OptDistinct> ::= emptyString
<SelList1> ::= <Attribute> , <SelList1>
<SelList1> ::= <Attribute>
```

That is, a <SelList1> is a list of attributes, as the old <SelList> used to be. The syntactic category <OptDistinct> represents an ``optional `DISTINCT`, and is replaced by either the keyword `DISTINCT` or the empty string (i.e., it disappears).

Exercise 7.1.2(a)

```
<Condition> ::= <Condition> OR <Condition>
<Condition> ::= NOT <Condition>
```

Return to Top

Solutions for Section 7.2

Exercise 7.2.1

If the index on *S* helps us find tuples that satisfy the condition *C*, then surely we should push the selection to *S*. Let the result of the selection on *S* be a relation *S'*, which is a subset of *S*. Assuming there is no index on *R* that helps with condition *C*, then if we push the selection to *R* as well, we still

have to intersect that result with *S*'. However, if we do *not* push the selection to *R*, then all that remains is intersecting *R* with *S*'. Performing either the selection or the intersection on *R* requires that we read all of *R*, so we are better off performing just the intersection rather than the selection plus a (smaller) intersection.

Exercise 7.2.2(a)

Let *R(A,B) = {(1,2)* and *S(A,B) = {(1,3)}*. Then *pi_A(R union_S S) = pi_A({(1,2), (1,3)} = {(1), (1)}*. But *pi_A(R) union_S pi_B(S) = {(1)}*, since the set union removes duplicates from the union.

Exercise 7.2.4(a)

Any example where *R* is nonempty works. For example, let *R = {(1)}*. Then *R union_B R = {(1), (1)}*, which is not equal to *R*.

Exercise 7.2.6(a)

We cannot push down *b+c -&> x*, because both *b* and *c* are used in the join. Notice that if we replaced *b+c* by their sum before joining, we would join tuples whose sum *b+c* agreed, but that had different values of *b* and *c*. What we can push down is:

1. The elimination of *a* from *R*.
2. The elimination of *e* from *S*.
3. The replacement of *c+d* by *y* in *S*.

Thus, the answer is:

pi_{b+c->x, y}(pi_{b,c}(R) JOIN pi_{b, c, c+d->y}(S))

Exercise 7.2.9(a)

The law holds. In proof, a tuple of *R* is in the result of the lefthand expression if it:

1. Joins with some tuple of *S*, and
2. Satisfies condition *C*.

A tuple of *R* is in the result of the righthand expression it it:

1. Satisfies condition *C*, and
2. Joins with some tuple of *S*.

Since the order in which the two tests are applied doesn't matter, the two sides of the expression are the same.

Exercise 7.2.9(b)

This law fails to hold. For example, let the condition *C* be *A=B*. Let *R(A,B) = {(1,2)}* and *S(B,C) = {(2,3)}*. Then the outerjoin *R OJ S = {(1,2,3)}*, and the result of the selection on this outerjoin is empty, since *A <> B* in this one tuple.

However, if we perform the selection on *R* first, the result is the empty set. When we that the outerjoin *emptySet OJ S*, we have to pad the tuple of *S*, and the result is *{(NULL,2,3)}*.

Exercise 7.2.9(f)

This law also fails to hold. Here is a simple example. Let our relations be *R(A,B) = {(1,1)}*, *S(C,D) = {(2,2)}*, and *T(B,C) = {(3,3)}*.

If we outerjoin *R* and *S* first, it is really a product, and we get the one tuple *{(1,1,2,2)}*. When this is outerjoined with *T*, nothing matches, so we pad and get *{(1,1,2,2), (NULL,3,3,NULL)}*.

However, if we outerjoin *S* and T first, we get no matches, and so pad to produce the relation *{(3,3,NULL), (NULL,2,2)}*. When this is outerjoined with *R*, there are again no matches, and the result is *{(1,1,NULL,NULL), (NULL,3,3,NULL), (NULL,NULL,2,2)}*.

Return to Top

Solutions for Section 7.3

Exercise 7.3.1(a)

PI_{a,R.b,c,d}[R(a,b) TH-JOIN_{R.b=S.b} S(b,c) TH-JOIN_{S.c>T.c} T(c,d)]

If we write the join with the projection outside, rather than applied to the first join only, then we can see the joins as one three-way theta-join of *R(a,b)*, *S(b,c)*, and *T(c,d)*, where the condition of the theta-join is that *R.b=S.b AND S.c>T.c*. In a sense, this join can be ordered in any way, although if we choose to join *R* and *T* first, the join is really a product.

Exercise 7.3.3(a)

The trick is to form an expression *Q* for the subquery, then project it only the empty list of attributes, and apply the delta (distinct) operator. We next take the product of this relation, which can only be empty or contain a single copy of the empty tuple, with *R*. That is, we compute *R TIMES DELTA[PI_{empty}(Q)]*. If *Q* is empty, then this product is empty, and if *Q* is not empty, then the product is *R*.

Return to Top

Solutions for Section 7.4

Exercise 7.4.1(a)

Start by taking the product of the sizes of the relations: 100*200*300*400 = 2,400,000,000. Since attribute *b* appears in two relations, divide by the larger of *V(W,b)* and *V(X,b)*, which is 60. Similarly, for *c* divided by 100, and for *d* divide by 50. The result is 8000 tuples.

Exercise 7.4.1(b)

The estimate is *T(W)/V(W,a) = 100/20 = 5*.

Exercise 7.4.1(g)

Start with the number of tuples of *W*, which is 100. We must then multiply by the estimated probability that any one tuple will satisfy both of the conditions *a=1* and *b=2*. Since there is no reason to believe these conditions are not independent, we divide 100 by *V(W,a)*V(W,b)*. The result is the fraction 1/12. That is, the estimated size of the result is less than one tuple. This conclusion

makes sense, since there are 1200 possible tuples that could be constructed from 20 *a*-values and 60 *b*-values; (1,2) is only one of them.

Exercise 7.4.2

Start by multiplying the sizes of the relations, which is 1000*2000*3000*4000 = 24,000,000,000,000. Attribute *a* appears three times, so we divide by the two largest of *V(E,a)*, *V(F,a)*, and *V(G,a)*. These numbers are 1000, 50, and 50, so we divide by 50,000.

Similarly, for *b* we divide by the two largest of 50, 100, and 40, or 5000. For *c* we divide by the two largest of 20, 300, and 100, or 3000, and for *d* we divide by the two largest of 200, 500, and 400, or 200,000.

The result is a fraction: 8/50000. That is, the expected number of tuples is under 1. That should not be surprising, since there are so many equalities that must be satisfied, in order for four tuples, one from each relation, to join.

Return to Top

Solutions for Section 7.5

Exercise 7.5.2

We know that the 100 tuples from *R* that have *b=0* all join with the 200 tuples from *S* with *b=0*. Thus, there are a minimum of 20,000 tuples in the join.

Since we are not given the number of different positive and negative *b*-values for the two relations, we cannot perform an accurate estimate of the contributions of the tuples that do not have *b=0*. However, it is likely that these do not contribute a significant number of additional tuples. For example, if there were as many as 100 different *b*-values in the positive and in the negative ranges of both *R* and *S*, then the estimate of the number of tuples in the join that have negative *b* would be 500*300/100 = 1500, and the number of join tuples with positive *b* would be 400*500/100 = 2000.

Exercise 7.5.3(a)

Changing *V(R,b)* has no effect on our estimates of the sizes of intermediate relations in plan (a), although it could reduce the estimated size of the result if it were increased above *V(S,b)*, which is 200. **Note: one might wonder how *V(R,b)* could affect the number of *b*-values in the result of an expression like *DELTA[SIGMA_{a=10}(R)]*, since there can be at most 50 different such values. However, when taking the join of that relation with *S*, we know that the *b*-values that appear were chosen from a set of *V(R,b)* different values, and our assumption about how values are chosen relates all of these values to the values from which *S.b* was chosen. Thus, we can estimate the sizes of a join whose relations are obtained by processing stored relations in some way (as long as the proceessing doesn't involve a selection on a join attribute), as if the statistics for the processed relation were the same as for the stored relations.**

Returning to the question at hand: in plan (b), changing *V(R,b)* can only decrease the size of the intermediate join. Thus, there is no way to make plan (a) become cheaper than plan (b) by changing *V(R,b)*.

Exercise 7.5.4(a)

We are really asking what is the probability that a tuple is in none of the four relations. That

probability is (800/1000)(700/1000)(600/1000)(500/1000) = 21/125. The remaining fraction, 104/125, *will* be in the union, which will thus have an estimated 832 tuples.

Exercise 7.5.4(c)

One way to take the union is to use a left-deep tree. If we do, then the cost of the intermediate relations is determined by the first three relations in the union, so we may as well take the three smallest, with sizes 200, 300, and 400. Using the technique from part (a), we determine that the union of all three has 664 tuples. To this number, we must add the cost of the first intermediate, which is the union of two of these three. We may as well take the two smallest to union first; that is relations of sizes 200 and 300, whose union has an estimated 440 tuples. The estimated cost of the best left-deep tree is thus 664+440 = 1104.

Now, we must consider bushy plans, where we union the relations in pairs and last take the union of the two results. There are three combinations, but none of them happens to have a cost less than that of the best left-deep plan. For instance, if we group them (200 union 300) and (400 union 500), then the two intermediates have sizes 440 + 700 = 1140.

Exercise 7.5.6(a)

First, we can only join *R JOIN S* or *S JOIN T* first, since a product, as in *R JOIN T*, has been ruled out. We have not resticted consideration to left-deep trees, so each of these two joins can be either the left or right argument in the final join. There are thus four possible join orders.

If *R* and *S* are joined first, then we must consider both the output in which the result is unsorted, and the output sorted on *b*, because that order is certainly ``interesting''; the entire expression winds up sorted by *b*. We could also consider the ``interesting'' order in which the result is sorted by *c*, since there is a join still to be taken with *c* as the join attribute. That might be advantageous if *S* were initially sorted by *c* (e.g., there was a B-tree index on *c*), or if *R JOIN S* were much bigger than *S*, and therefore sorting *S* before the join would be an efficient way to perform a sort-join of *R JOIN S* with *T*.

If we compute *S JOIN T* first, then we should consider only the unsorted order for the result and the interesting order in which the result is sorted by *b*. The latter attribute is an interesting sort, because it is the join attribute of a subsequent join.

Return to Top

Solutions for Section 7.6

Exercise 7.6.4

The cheapest initial join is *S JOIN T*, with a result size of 100*100/10 = 1000. (Well actually, this is tied for cheapest, so the greedy algorithm *might* make this choice.) Next, we could join *S JOIN T* with either *R* or *U*; the result size is the same, at 1000*1000/100 = 10,000. Thus, the greedy algorithm produces a plan with a cost of 11,000.

However, the best strategy is to start by joining *R JOIN S*, and *T JOIN U*, each of which have a cost of 1000*100/max(100,10) = 1000. This plan has a cost of 2000.

Exercise 7.6.5(a)

The formula gives 7!(T(1)T(6) + T(2)T(5) + T(3)T(4) + T(4)T(3) + T(5)T(2) + T(6)T(1)) = 5040*

(1*42 + 1*14 + 2*5 + 5*2 + 14*1 + 42*1) = 5040*132 = 665,280. Since the number of left-deep trees is 7! = 5040, the number of right-deep trees is the same, and no left-deep tree of this size can also be right-deep, the number of bushy join orders is 665,280 - 10,080 = 655,200.

Exercise 7.6.6(a)

The question leaves open the possibility that we might join *R JOIN S* by a method other than one in which *R* is first read into memory as a build relation, since *R* is not an ``intermediate'' relation. However, assuming that we do so, we shall have at one time all of *R* and *R JOIN S* in memory, and at another time all of *R JOIN S* and *(R JOIN S) JOIN T* in memory. Thus, *M* must be at least *B(R JOIN S)* + min*(B(R), B[(R JOIN S) JOIN T])*.

Exercise 7.6.7

There is an entry in the dynamic-programming table for every subset of the *k* attributes, with the exception of the empty set. Thus, we need *2^k - 1* entries in the table.

Return to Top

Solutions for Section 7.7

Exercise 7.7.1(a)

Use an index-scan using the nonclustering index on *c*. Since *V(R,c)* = 5000, only one block should be retrieved. Filter the retrieved tuples for *a=1* and *d=3*. The expected disk I/O cost is 1.

Exercise 7.7.2(a)

The number of blocks we expect to read using the nonclustering index is *T(R)/V(R,x)*, while the expected number of blocks using the clustering index is *B(R)/V(R,y)*. Thus, the first is less whenever *T(R)V(R,y)* < *B(R)V(R,x)*.

Exercise 7.7.4(a)

***B(T)* <= 61. In this case, we can divide each of *R* and *S* into 20 sorted sublists each. These sublists are created and sorted before we bring *T* into memory, and thus can each have length 100 (or 101, although it doesn't make a difference in this example). Now, we bring *T* into 61 buffers, and we use the remaining 40 buffers for the 40 sublists from *R* and *S*.**

We join the tuples of *R* and *S*, and for each tuple generated (it need not be copied anywhere; we just need its value for *a*), we search memory for matching *T*-tuples, i.e., tuples with the same value of *a*. Note that if there are many tuples from *R* and *S* that share a common *a*-value, we shall need extra buffers. Thus, it might be wise to make sure that *B(T)* is somewhat less than the theoretical limit 61 --- how much less depends on the statistics of *R* and *S* --- or be willing to accept extra disk I/O's if we need to shuffle blocks of the sorted sublists in and out of memory, occasionally.

Return to Top