Database System Implementation

Solutions for Chapter 6

Solutions for Section 6.1

Exercise 6.1.1(a)

Technically, the U_S operator assumes that the arguments are sets, as well as the result. Since the arguments are really bags, the best we can do is use the SQL union operator, which removes duplicates from the output even if there are duplicates in the input. The answer is {(0,1), (2,3), (2,4), (3,4), (2,5), (0,2)}.

Exercise 6.1.1(i)

{(1,0,1), (5,4,9), (1,0,1), (6,4,16), (7,9,16)}.

Exercise 6.1.1(k)

{(0,1), (0,1), (2,4), (3,4)}. Note that all five tuples of *R* satisfy the first condition, that the first component is less than the second. The last two tuples satisfy the condition that the sum of the components is at least 6, and the tuple (0,1) satisfies the condition that the sum exceed the product. Only the tuple (2,3) satisfies neither of the latter two conditions, and therefore doesn't appear in the result.

Exercise 6.1.2(a)

{(0,1,2), (0,1,2), (0,1,2), (0,1,2), (2,3,5), (2,4,5), (3,4,5)}.

Exercise 6.1.2(e)

{(0,1,4,5), (0,1,4,5), (2,3,2,3), (2,4,2,3)}.

Exercise 6.1.2(g)

{(0,2), (2,7), (3,4)}.

Exercise 6.1.3(a)

One way to express the semijoin is by projecting *S* onto the list *L* of attributes in common with *R* and then taking the natural join: *R* JOIN pi_*L(S)*.

A second way is to take the natural join first, and then project onto the attributes of *R*.

Exercise 6.1.3(c)

First, we need to find the tuples of *R* that don't join with *S* and therefore need to be padded. Let *T* = *R* - *pi_M(R JOIN S)*, where *M* is the schema, or list of attributes, of *R*. Then the left outerjoin is *(R JOIN S) UNION (T x N)*, where *N* is the suitable ``null'' relation with one tuple that has `NULL` for each attribute of *S* that is not an attribute of *R*.

Exercise 6.1.5(a)

When a relation has no duplicate tuples, DELTA has no effect on it. Thus, DELTA is idempotent.

Exercise 6.1.5(b)

The result of *PI_L* is a relation over the list of attributes *L*. Projecting this relation onto *L* has no effect, so *PI_L* is idempotent.

Return to Top

Solutions for Section 6.3

Exercise 6.3.1(a)

The iterator for *pi_L(R)* can be described informally as follows:

Open():
    `R.Open()`

GetNext():
    Let *t* = `R.GetNext()`. If `Found = False`, return. Else, construct and return the tuple consisting of the elements of list *L* evaluated for tuple *t*.

Close():
    `R.Close()`

Exercise 6.3.1(b)

Here is the iterator for *delta(R)*:

Open():

First perform `R.Open()`. Then, initialize a main-memory structure *S* that will represent a set of tuples of *R* seen so far. For example, a hash table could be used. Initially, set *S* is empty.

GetNext():

Repeat `R.GetNext()` until either `Found = False` or a tuple *t* that is not in set *S* is returned. In the later case, insert *t* into *S* and return that tuple.

Close():

`R.Close()`

Exercise 6.3.1(d)

For the set union *R1 union R2* we need both a set *S* that will store the tuples of *R1* and a flag `CurRel` as in Example 6.13. Here is a summary of the iterator:

Open()

Perform `R1.Open()` and initialize to empty the set *S*.

GetNext()

If `CurRel = R1` then perform `t = R1.GetNext()`. If `Found = True`, then insert *t* into *S* and return *t*. If `Found = False`, then *R1* is exhausted. Perform `R1.Close()`, `R2.Open()`, set `CurRel = R2`, and repeatedly perform `t = R2.GetNext()` until either `Found = False` (in which case, just return), or *t* is not in *S*. In the latter case, return *t*.

Close():

Perform `R2.Close()`

Exercise 6.3.3

A group entry needs a value of `starName`, since that is the attribute we are grouping on. It also needs an entry for the sum of lengths, since that sum is part of the result of the query of Exercise 6.1.6(d). Finally, the group entry needs a count of the number of tuples in the group, since this count is needed to accept or reject groups according to the `HAVING` clause.

Exercise 6.3.5(a)

Read *R* into main memory. Then, for each tuple *t* of *S*, see if *t* joins with any of the tuples of *R* that are in memory. Output any of those tuples that do join with *t*, and delete them from the main-memory set.

Exercise 6.3.5(b)

Read *S* into memory. Then, for each tuple *t* of *R*, see if *t* joins with any tuple of *S* in memory; output *t* if so.

Exercise 6.3.5(e)

Read *R* into main memory. Then, for each tuple *t* of *S*, find those tuples in memory with which *t* joins, and output the joined tuples. Also mark as ``used" all those tuples of *R* that join with *t*. After *S* is exhausted, examine the tuples in main memory, and for each tuple *r* that is not marked ``used," pad *r* with nulls and output the result.

Return to Top

Solutions for Section 6.4

Exercise 6.4.2

The formula in Section 6.4.4 gives 10000 + 10000*10000/999 or 110,101.

Return to Top

Solutions for Section 6.5

Exercise 6.5.2(a)

An iterator for *delta(R)* must do the first sorting pass in its `Open()` function. Here is a sketch:

Open():
: Perform `R.Open()` and repeatedly use `R.GetNext()` to read memory-sized chunks of *R*. Sort each into a sorted sublist and write out the list. Each of these lists may be thought of as having its own iterator. Open each list and initialize a data structure into which the ``current" element of each list may be read into main memory. Use `GetNext()` for each list to initialize the main-memory structure with the first element of each list. Finally, execute `R.Close()`.

GetNext():
: Find the least tuple *t* in the main-memory structure, and output one copy of it. Delete all copies of *t* in main memory, using `GetNext()` for the appropriate sublist, until all copies of *t* have disappeared, then return. If there was no such tuple *t*, because all the sublists were exhausted, set Found = False and return.

Close():
: Close all the sorted sublists.

Exercise 6.5.2(c)

For *R1 intersect R2* do the following:

Open():
: Open *R1* and *R2*, use their `GetNext()` functions to read all their tuples in main-memory-sized chunks, and create sorted sublists, which are stored on disk. Initialize a main-memory data structure that will hold the ``current" tuple of each sorted sublist, and use the `Open()` and `GetNext()` functions of an iterator from each sublist to initialize the structure to have the first tuple from each sublist. Finally, close *R1* and *R2*.

GetNext():
: Find the least tuple *t* among all the first elements of the sorted sublists. If *t* occurs on a list from *R1* and a list from *R2*, then output *t*, remove the copies of *t*, use `GetNext` from the two sublists involved to replentish the main-memory structure, and return. If *t* appears on only one of the sublists, do not output *t*. Ratherm remove it from its list, replentish that list, and repeat these steps with the new least *t*. If no *t* exists, because all the lists are exhausted, set `Found = False` and return.

Close():
: Close all the sorted sublists.

Exercise 6.5.3(b)

The formula of Fig. 6.16 gives 5*(10000+10000) = 100,000. Note that the memory available, which is 1000 blocks, is larger than the minimum required, which is sqrt(10000), or 100 blocks. Thus, the method is feasible.

Exercise 6.5.5(a)

We effectively have to perform two nested-loop joins of 500 and 250 blocks, respectively, using 101 blocks of memory. Such a join takes 250 + 500*250/100 = 1500 disk I/O's, so two of them takes 3000. To this number, we must add the cost of sorting the two relations, which takes four disk I/O's per block of the relations, or another 6000. The total disk I/O cost is thus 9000.

Exercise 6.5.7(a)

The square root of the relation size, or 100 blocks, suffices.

Return to Top

Solutions for Section 6.6

Exercise 6.6.1(a)

To compute *delta(R)* using a ``hybrid hash'' approach, pick a number of buckets small enough that one entire bucket plus one block for each of the other buckets will just fit in memory. The number of buckets would be slightly larger than *B(R)/M*. On the first pass, keep all the distinct tuples of the first bucket in main memory, and output them the first time they are seen. On the second pass, do a one-pass distinct operation on each of the other buckets.

The total number of disk I/O's will be one for *M/B(R)* of the data and three for the remaining *(B(R)-M)/B(R)* of the data. Since the data requires *B(R)* blocks, the total number of disk I/O's is *M + 3(B(R) - M) = 3B(R) - 2M*, compared with *3B(R)* for the standard approach.

Exercise 6.6.4

This is a trick question. Each group, no matter how many members, requires only one tuple in main memory. Thus, no modifications are needed, and in fact, memory use will be pleasantly small.

Return to Top

Solutions for Section 6.7

Exercise 6.7.1(a)

Our option is to first copy all the tuples of *R* to the output. Then, for each tuple *t* of *S*, retrieve the tuples of *R* for which *R.a* matches the *a*-value of *t*. Check whether *t* appears among them, and output *t* if not.

This method can be efficient if *S* is small and *V(R,a)* is large.

Exercise 6.7.2(a)

The values retrieved will fit on about 10000/*k* blocks, so that is the approximate cost of the selection.

Exercise 6.7.5

There is an error (see The Errata Sheet in that the relation `Movie` should be `StarsIn`. However, it doesn't affect the answer to the question, as long as you believe the query can be answered at all.

The point is that with the index, we have only to retrieve the blocks containing `MovieStar` records for the stars of ``King Kong." We don't know how many stars there are, but it is safe to assume that their records will occupy many more blocks than there are stars in the two ``King Kong" movies. Thus, using the index allows us to take the join while accessing only a small part of the `MovieStar` relation and is therefore to be preferred to a sort- or hash-join, each of which will access the entire `MovieStar` relation at least once.

Return to Top

Solutions for Section 6.8

Exercise 6.8.1(a)

One of the relations must fit in memory, which means that min(*B(R)*,*B(S)*) >= *M/2*.

Exercise 6.8.1(b)

We can only assume that there will be *M/2* blocks available during the first pass, so we cannot create more than *M/2-1* buckets. Each pair of matching buckets must be joinable in a one-pass join, which means one of each pair of corresponding buckets must be no larger than *M/2*, as we saw in part (a). Since the average bucket size will be *1/(M/2 - 1)* of the size of its relation, we require min*(B(R),B(S))/(M/2 - 1)* >= *M/2*, or approximately min(*B(R)*,*B(S)*) >= *M^2/4*. This figure should be no surprise; it simply says we must make the most conservative assumption at all times about the size of memory.

Return to Top

Solutions for Section 6.9

Exercise 6.9.1(a)

The way we have defined the recursive algorithm, it always divides relations into *M-1* pieces (100 pieces in this exercise) if it has to divide at all. In practice, we may have options to use fewer pieces at one or more levels or the recursion, and in this exercise there are many such options. However, we'll describe only the one that matches the policy given in the text. In what follows, all sorting and merging is based on the values of tuples in the join attributes only, since we are describing a join operation.

Level 1: *R* and *S* are divided into 50 sublists each. The sizes of the lists are 400 and 1000 blocks, for *R* and *S*, respectively. Note that no disk I/O's are performed yet; the partition is conceptual.

Level 2: Each of the sublists from Level 1 are divided into 100 subsublists; the sizes of these lists are 4 blocks for *R*, and 10 blocks for *S*. Again, no disk I/O's are performed for this conceptual partition.

Level 3: We sort each of the subsublists from Level 2. Each block is read once from disk, and an equal number of blocks are written to disk as the sorted subsublists.

Return to Level 2: We merge each of the 100 subsublists that comprise one sublist, using the standard multiway merging algorithm. Every block is read and written once during this process.

Return to Level 1: We merge each of the 50 sorted sublists from *R* and the 50 sorted sublists from *S*, thus joining *R* and *S*. Each block is read once during this part.

Return to Top

Solutions for Section 6.10

Exercise 6.10.1(a)

If the division of tuples from *R* is as even as possible, some processors will have to perform 13 disk I/O's, but none will have to perform more. The 13 disk I/O's take 1.3 seconds, so the speedup is 10/1.3 or about 7.7.

Return to Top