Database Systems

Instructors: Hao-Hua Chu
Winston Hsu
Fall Semester, 2007

Assignment 6 Solutions

Questions

I. Consider a disk with an average seek time of 10ms, average rotational delay of 5ms, and a transfer time of 1ms for a 4K page. Assume that the cost of reading/writing a page is the sum of these values (i.e., 16ms) unless a sequence of pages is read/written. In this case, the cost is the average seek time plus the average rotational delay (to find the first page in the sequence) plus 1ms per page (to transfer data). You are given 320 buffer pages and asked to sort a file with 10,000,000 pages.

1. Why is it a bad idea to use the 320 pages to support virtual memory, that is, to *new* 10,000,000 * 4K bytes of memory, and to use an in-memory sorting algorithm such as Quicksort?

2. Assume that you begin by creating sorted runs of 320 pages each in the first pass. Evaluate the cost of the following approaches for the subsequent merging passes:

   (a) Do 319-way merges.
   (b) Create 256 'input' buffers of 1 page each, create an 'output' buffer of 64 pages, and do 256-way merges.
   (c) Create 16 'input' buffers of 16 pages each, create an 'output' buffer of 64 pages, and do 16-way merges.
   (d) Create eight 'input' buffers of 32 pages each, create an 'output' buffer of 64 pages, and do eight-way merges.
   (e) Create four 'input' buffers of 64 pages each, create an 'output' buffer of 64 pages, and do four-way merges.

Ans:

In Pass 0, 31250 sorted runs of 320 pages each are created. For each run, we read and write 320 pages sequentially. The I/O cost per run is
2 * (10+5+1* 320) = 670ms. Thus, the I/O cost for Pass 0 is 31250 * 670 = 20937500ms. For each of the cases discussed below, this cost must be added to the cost of the subsequent merging passes to get the total cost. Also, the

calculations below are slightly simplified by neglecting the effect of a final read/written block that is slightly smaller than the earlier blocks.

1. For 319-way merges, only 2 more passes are needed. The first pass will produce ceiling(31250/319) = 98 sorted runs; these can then be merged in the next pass. Every page is read and written individually, at a cost of 16ms per read or write, in each of these two passes. The cost of these merging passes is therefore 2∗(2∗16)∗10000000 = 640000000ms.
   (The formula can be read as 'number of passes times cost of read and write per page times number of pages in file'.)

2. With 256-way merges, only two additional merging passes are needed. Every page in the file is read and written in each pass, but the effect of blocking is different on reads and writes. For reading, each page is read individually at a cost of 16ms. Thus, the cost of reads (over both passes) is
   2 ∗ 16 ∗ 10000000 = 320000000ms. For writing, pages are written out in blocks of 64 pages. The I/O cost per block is 10 + 5 + 1 ∗ 64 = 79ms. The number of blocks written out per pass is 10000000/64 = 156250, and the cost per pass is 156250∗79 = 12343750ms. The cost of writes over both merging passes is therefore 2 ∗ 12343750 = 24687500ms. The total cost of reads and writes for the two merging passes is 320000000 + 24687500 = 344687500ms.

3. With 16-way merges, 4 additional merging passes are needed. For reading, pages are read in blocks of 16 pages, at a cost per block of 10 + 5 + 1 ∗ 16 = 31ms. In each pass, 10000000/16 = 625000 blocks are read. The cost of reading over the 4 merging passes is therefore 4 ∗ 625000 ∗ 31 = 77500000ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 4 merging passes is 4 ∗ 12343750 = 49375000ms, and the total cost of the merging passes is 77500000+ 49375000 = 126875000ms.

4. With 8-way merges, 5 merging passes are needed. For reading, pages are read in blocks of 32 pages, at a cost per block of 10 + 5 + 1 ∗ 32 = 47ms. In each pass, 10000000/32 = 312500 blocks are read. The cost of reading over the 5 merging passes is therefore 5∗ 312500 ∗ 47 = 73437500ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before. The cost of writes over 5 merging passes is 5 ∗ 12343750 = 61718750ms, and the total cost of the merging passes is 73437500+ 61718750 = 135156250ms.

5. With 4-way merges, 8 merging passes are needed. For reading, pages are read in blocks of 64 pages, at a cost per block of 10 + 5 + 1 ∗ 64 = 79ms. In each pass, 10000000/64 = 156250 blocks are read. The cost of reading over the 8 merging passes is therefore 8∗ 156250 ∗ 79 = 98750000ms. For writing, pages are written in 64 page blocks, and the cost per pass is 12343750ms as before.

The cost of writes over 8 merging passes is 8 ∗ 12343750 = 98750000ms, and the total cost of the merging passes is 98750000+ 98750000 = 197500000ms.

II. Consider the join $R \infty_{R.a=S.b} S$, given the following information about the relations to be joined. The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

Relation R contains 10,000 tuples and has 10 tuples per page.
Relation S contains 2000 tuples and also has 10 tuples per page.
Attribute b of relation S is the primary key for S.
Both relations are stored as simple heap files.
Neither relation has any indexes built on it.
52 buffer pages are available.

1. What is the cost of joining R and S using a page-oriented simple nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
2. What is the cost of joining R and S using a block nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
3. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?
4. What is the cost of joining R and S using a hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?
5. What would be the lowest possible I/O cost for joining R and S using any join algorithm, and how much buffer space would be needed to achieve this cost? Explain briefly.
6. How many tuples does the join of R and S produce, at most, and how many pages are required to store the result of the join back on disk?
7. Would your answers to any of the previous questions in this exercise change if you were told that R.a is a foreign key that refers to S.b?

Ans: Let $M = 1000$ be the number of pages in R, $N = 200$ be the number of pages in S, and $B = 52$ be the number of buffer pages available.

1. Basic idea is to read each page of the outer relation, and for each page scan the inner relation for matching tuples. Total cost would be

$$\#pagesinouter + (\#pagesinouter * \#pagesininner)$$

which is minimized by having the smaller relation be the outer relation.

$$TotalCost = N + (N * M) = 200200$$

The minimum number of buffer pages for this cost is 3.

2. This time read the outer relation in blocks, and for each block scan the inner relation for matching tuples. So the outer relation is still read once, but the inner relation is scanned only once for each outer block, of which there are *ceiling (#pages in outer / (B −2)) = ceiling (200/50) = 4*.

$$TotalCost = N + M * ceiling(N / (B-2)) = 4200$$

If the number of buffer pages is less than *52*, the number of scans of the inner would be more than 4 since *ceiling(200/49) = 5*. The minimum number of buffer pages for this cost is therefore 52.

3. Since $B > \sqrt{M} > \sqrt{N}$ we can use the refinement to Sort-Merge :

$$TotalCost = 3*(M + N) = 3600$$

NOTE: if *S.b* were not a key, then the merging phase could require more than one pass over one of the relations, making the cost of merging *M*N* I/Os in the worst case.

The minimum number of buffer pages required is *25*. With *25* buffer pages, the initial sorting pass will split R into *20* runs of size *50* and split *S* into *4* runs of size *50* (approximately). These *24* runs can then be merged in one pass, with one page left over to be used as an output buffer. With fewer than *25* buffer pages the number of runs produced by the first pass over both relations would exceed the number of available pages, making a one-pass merge impossible.

4. The cost of Hash Join is *3*(M+N)* if $B > \sqrt{(f * N)}$ where f is a fudge factor used to capture the small increase in size involved in building a hash table, and N is the number of pages in the smaller relation, *S* (see page 258). Since $\sqrt{N} \approx 14$, we can assume that this condition is met. We will also assume uniform partitioning from our hash function.

$$TotalCost = 3*(M + N) = 3600$$

Without knowing *f* we can only approximate the minimum number of buffer pages required, and a good guess is that we need $B > \sqrt{(f * N)}$.

5. The optimal cost would be achieved if each relation was only read once. We could do such a join by storing the entire smaller relation in memory, reading in the larger relation page-by-page, and for each tuple in the larger relation we search the smaller relation (which exists entirely in memory) for matching tuples. The buffer pool would have to hold the entire smaller relation, one page for reading in the larger relation, and one page to serve as an output buffer.

$$TotalCost = M + N = 1,\ 200$$

The minimum number of buffer pages for this cost is $N + 1 + 1 = 202$.

6. Any tuple in *R* can match at most one tuple in *S* because *S.b* is a primary key (which means the *S.b* field contains no duplicates). So the maximum number of tuples in the result is equal to the number of tuples in *R*, which is *10,000*.

The size of a tuple in the result could be as large as the size of an *R* tuple plus the size of an *S* tuple (minus the size of the shared attribute). This may allow only *5* tuples to be stored on a page. Storing *10,000* tuples at *5* per page would require *2000* pages in the result.

7. The foreign key constraint tells us that for every R tuple there is exactly one matching *S* tuple (because *S.b* is a key). The Sort-Merge and Hash Joins would not be affected, but we could reduce the cost of the two Nested Loops joins. If we make R the outer relation then for each tuple of *R* we only have to scan *S* until a match is found. This will require scanning only *50%* of *S* on average.

For *Page-Oriented Nested Loops*, the new cost would be

$$TotalCost = M + (M * (N/2)) = 101\ 000$$

and 3 buffer pages are still required.

For *Block Nested Loops*, the new cost would be

$$TotalCost = M + (N/2) * ceiling(\ M / (B-2)\ ) = 3000$$

and again this cost can only be achieved with 52 available buffer pages.

III. Consider a database with objects X and Y and assume that there are two transactions T1 and T2. Transaction T 1 reads objects X and Y and then writes object X. Transaction T2 reads objects X and Y and then writes objects X and Y.

1. Give an example schedule with actions of transactions T1 and T 2 on objects X and Y that results in a write-read conflict.
2. Give an example schedule with actions of transactions T1 and T 2 on objects X and Y that results in a read-write conflict.
3. Give an example schedule with actions of transactions T1 and T 2 on objects X and Y that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Ans:

1. The following schedule results in a write-read conflict:
   T2:R(X), T2:R(Y), T2:W(X), T1:R(X) ...
   T1:R(X) is a dirty read here.

2. The following schedule results in a read-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X) ...
   Now, T2 will get an unrepeatable read on X.

3. The following schedule results in a write-write conflict:
   T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X), T2:W(X) ...
   Now, T2 has overwritten uncommitted data.

4. Strict 2PL resolves these conflicts as follows:
   (a) In S2PL, T1 could not get a shared lock on X because T2 would be Holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.
   (b) Here T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.
   (c) Same as above.

IV. Consider the following classes of schedules: *serializable*, *conflict-serializable*, *view-serializable*, recoverable. For each of the following schedules, state which of the preceding classes it belongs to. If you cannot decide whether a schedule belongs in a certain class based on the listed actions, explain briefly.

The actions are listed in the order they are scheduled and prefixed with the transaction name. If a commit or abort is not shown, the schedule is incomplete; assume that abort or commit must follow all the listed actions.

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)
2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)
3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)
4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)
5. T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
7. T1:W(X), T2:R(X), T1:W(X), T2:Abort, T1:Commit
8. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. T2: R(X), T3:W(X), T3:Commit, T1:W(Y), T1:Commit, T2:R(Y), T2:W(Z), T2:Commit
11. T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit

Ans:

1. Not *serializable*, not *conflict-serializable*, not *view-serializable*; It is *recoverable*.
2. It is *serializable*, *conflict-serializable*, and *view-serializable*;
   We can not decide whether it's recoverable or not, since the abort/commit sequence of these two transactions are not specified.
3. It is the same with number 2 above.
4. It is NOT *serializable*, NOT *conflict-serializable*, NOT *view-serializable*;
   We can not decide whether it's recoverable or not, since the abort/commit sequence of these transactions are not specified.
5. It is *serializable*, *conflict-serializable*, and *view-serializable*;
   It is *recoverabl*.
6. It is *serializable* and *view-serializable*, not *conflict-serializable*;
   It is *recoverable*
7. It is not *serializable*, not *view-serializable*, not *conflict-serializable*;
   It is not *recoverable*.
8. It is not *serializable*, not *view-serializable*, not *conflict-serializable*;
   It is not *recoverable*.
9. It is *serializable*, *view-serializable*, and *conflict-serializable*;
   It is not *recoverable*.
10. It belongs to all above classes.
11. (assume the 2nd T2:Commit is instead T1:Commit).
    It is *serializable* and *view-serializable*, not *conflict-serializable*;
    It is *recoverable*.
12. It is *serializable* and *view-serializable*, not *conflict-serializable*;
    It is *recoverable*.