

## Assignment #3

Due: 2:00pm CST on October 6, 2006

Out: September 20, 2006

**NOTE:** Please submit a hard copy of your homework. Bring it to the lecture table at the beginning of the lecture on 6th October, 2006.

The hard copy should be as clearly readable as possible. You may be subtracted points for unreadability and ugly presentation.

**I2CS Students:** You should email your solutions to the TA(ylee11@uiuc.edu) in the **pdf** or the **word** format. Please send the file as attachment with your email by 2PM UIUC time (CST). I2CS students in other time zones should note that the deadline is according to CST.

## Problem 1 (30 pts)

Suppose you have 2 relations,  $R(A, B, C)$  and  $S(B, C, D, E)$ . You have a clustered unique (no duplicate keys) B+-tree index on attribute A for relation R. Assume this index is kept entirely in memory (i.e., you do not need to read it from disk).

For relation S, you have two indexes: (i) a non-clustered non-unique B+-tree index for attribute B, and (ii) a clustered non-unique B+-tree index for attribute C. Assume that these two indexes are also kept in memory. Also, assume that all of the tuples of S that agree on attribute C are stored in sequentially adjacent blocks on disk (that is, if more than one block is needed to store all of the tuples with some value of C, then these blocks will be sequentially located on the disk).

Other relevant data:

- 2000 tuples of R are stored per block on disk.
- $N(R) = 2,000,000$  (number of tuples of R)
- 200 tuples of S are stored per block on disk.
- $N(S) = 200,000$  (number of tuples of S)
- $V(B,S)=40,000$  (image size of attribute B in S)
- $V(C,S)=800$  (image size of attribute C in S)

You want to execute the following query:

2

```
SELECT *
FROM R, S
WHERE (R.B=S.B) AND (R.C=S.C)
```

We present you with two query plans:

**Plan 1:**

```
For every block BL of R, retrieved using the clustered index on A for R
  For every tuple r of BL
    Use the index on B for S to retrieve all
      of the tuples s of S such that s.B=r.B
    For each of these tuples s, if s.C=r.C, output r.A, r.B, r.C,
                                          s.B, s.C, s.D, s.E
```

**Plan 2:**

```
For every block BL of R, retrieved using the clustered index on A for R
  For every tuple r of BL
    Use the index on C for S to retrieve all
      of the tuples s of S such that s.C=r.C
    For each of these tuples s, if s.B=r.B, output r.A, r.B, r.C,
                                          s.B, s.C, s.D, s.E
```

- a) Analyze the above two plans carefully in terms of their behavior regarding accesses to disk, and explain which of the plans is therefore better. Be sure to include in your analysis which accesses to disk are sequential accesses and which ones are random accesses.
- b) Can you think of an alternative plan that is better than the above two in terms of disk access? You do not need to submit detail computations for this question.

## Solution

- a) For Plan 1, we need 1000 sequential block access for reading the tuples of relation R. For each tuple of relation R, we need to access relation S by using the B attribute. However, we only have a non-clustered non-unique B+-tree index for attribute B.

Therefore, each time we need to access  $200,000/40,000=5$  blocks randomly. Therefore<sup>3</sup>, plan 1 needs  $1000+2,000,000*5 = 10,001,000$  block accesses.

For Plan 2, we need 1000 sequential block access for reading the tuples of relation R. For each tuple of relation R, we need to access relation S by using the C attribute. We have a clustered non-unique B+-tree index for attribute C. Relation S has 1000 blocks. Therefore, each time we need to access  $1000/800=2$  blocks sequentially. Therefore, plan 2 needs  $1000+2,000,000*2 = 4,001,000$  block accesses.

We can tell that plan 2 is better than plan 1.

- b) There could be multiple ways for reducing the disk access. Here are some possible plans.
- As both relations have 1000 blocks, we can just read in the two relations and do join in memory. That means the disk access is just 2000 blocks. However, this scheme requires much more memory space.
  - We can modify plan 2 so that it would check to see if r.B exists in the index of S on attribute B before fetching all the tuples s of S such that  $s.C=r.C$ . Thus, if r.B does not exist, then we would avoid reading the tuples s such that  $s.C=r.C$ .

## Problem 2 (10 pts)

Consider the join  $R \bowtie_{R.a=S.b} S$ , given the following information about the relations to be joined. The cost metric is the number of page I/Os, 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 2,000 tuples and also has 10 tuples per page.
- Attribute  $b$  of relation  $S$  is the primary key for  $S$ .
- Neither relation has any indexes built on it.
- 52 buffer pages are available.

Can we use **hybrid hash join**? If yes, what is the cost? If no, why?

## Solution

4

Yes, we can pick  $k = 5$ , thus each bucket of the smaller relation  $S$  has about  $200/5$  or 40 pages of tuples. We still have  $52 - 40 = 12$  pages left, thus we can accommodate  $k - 1 = 4$  pages for each of the other buckets. The cost is  $(3 - 2 * B/N)(M + N) = 2976$ , where  $B=52$ ,  $M=10000/10=1000$ ,  $N=2000/10=200$ .

## Problem 3 (20 pts)

Query optimization has been considered as a key technique for the realization of the relational model.

- a) Why does the relational DBMS (in particular) need query optimization? Why this was not an issue for earlier DBMS (e.g., of the network model).
- b) System R has established the *Selinger-style* query optimization. What are the main techniques in this framework?
- c) Suppose we wish to compute the following:

$$\tau_b(R(a, b) \bowtie S(b, c) \bowtie T(c, d))$$

That is, we join the three relations and produce the result sorted on attribute  $b$ . Assume that we do **NOT** join  $R$  and  $T$  first, as that is a Cartesian product.

What are all the subexpressions and *interesting orders* that System R optimizer would consider?

## Solution

- a) Relational query languages provide a high-level “declarative” interface to access data stored in relational database. The query does not give the access paths; there can be many possible access paths. Thus, we need to choose the efficient access path, which is the task of the query optimizer. However, earlier DBMSs use the navigational model. The query itself explicitly states the access path it would take. Thus, the query optimization was not an issue.
- b) These are some techniques in this framework:
  - Space of query plans: To reduce the space, it pushes down projections and selections. Then, it only considers left-deep joins

- Cost estimation: Assuming that attributes and predicates are independent, the cost is the weighted sum of the I/O cost and CPU cost. Cost of an operator depends on input data size, data distribution, and physical layout. Thus, the optimizer uses statistics about the relations to estimate the cost. Also, it uses statistics on base relations and intermediate results.
  - Search algorithm: It uses the (bottom-up) dynamic programming.
- c) First, we can only join  $R \bowtie S$  or  $S \bowtie T$  first, since  $R \bowtie T$ , has been ruled out. System R optimizer considers only “left-deep” joins. Thus, there are *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  $R \bowtie S$  were much bigger than  $S$ , and therefore sorting  $S$  before the join would be an efficient way to perform a sort-join of  $R \bowtie S$  with  $T$ .
- If we compute  $S \bowtie T$  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.

## Problem 4 (15 pts)

Consider a database  $DB$ .  $DB$  has two relations  $R1$  and  $R2$ . The relation  $R1$  contains tuples  $t1$  and  $t2$ , while  $R2$  contains tuples  $t3$ ,  $t4$ , and  $t5$ . Assume that the database  $DB$ , relations, and tuples form a hierarchy of lockable database elements.

Tell the sequence of lock requests and the response of the locking scheduler, which is described in the paper “Granularity of Locks and Degrees of Consistency in a Shared Data Base”, to the following sequence of request. You may assume all requests occur just before they are needed, and all unlocks occur at the end of the transaction.

$r1(t1)$ ;  $w2(t2)$ ;  $r2(t3)$ ;  $w1(t4)$

[ $w2(t2)$  represents the creation of  $t2$  by transaction  $T2$ .]

## Solution

At the first step,  $T1$  puts a IS lock on the  $DB$  and on  $R1$ , and an S lock on  $t1$ .

At step 2,  $T_2$  puts an IX lock on the  $DB$  and on  $R_1$ , both of which are compatible with the IS locks already there.  $T_2$  also puts an X lock on  $t_2$ .

At step 3,  $T_2$  puts an IS lock on the  $DB$  and on  $R_2$  and an S lock on  $t_3$ , then releases its locks.

At step 4,  $T_1$  puts an IX lock on the  $DB$  and on  $R_2$  and an X lock on  $t_4$ , then releases its locks.

## Problem 5 (25 pts)

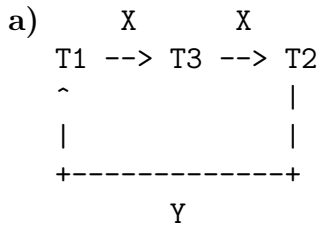
This problem studies serializability and degree of consistency. Examine the schedule below. There are three transactions  $T_1$ ,  $T_2$ , and  $T_3$ .

	T1	T2	T3
0	start		
1	read X		
2		start	
3		read Y	
4	write X		
5			start
6			read X
7			write X
8			commit
9		read X	
10		write Y	
11		write X	
12		commit	
13	read Y		
14	write Y		
15	commit		

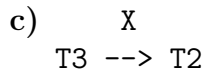
- Draw the precedence graph for this schedule.
- What is the equivalent serialization order for this schedule? If no order is possible, state NONE.
- Assume that transaction  $T_1$  did not run at all. Would the precedence graph be different in this case? Explain.
- What is the equivalent serialization order for this schedule? If no order is possible, state NONE.

- e) When all transactions run in the above schedule, identify the transactions with degree 3 consistency (using Definition 1 in the paper, “Granularity of Locks and Degrees of Consistency in a Shared Data Base”). Answer the same question when transaction T1 did not run at all. How is the degree of consistency relevant to serializability?

### Solution



- b) None. The graph has a cycle.



- d) The given schedule above is equivalent to a serial order of running T3 then T2.

- e) None is degree 3 consistency. T2 and T3 are not as they dirty T1's data, while T1 is not as its data is dirtied by other transactions.

However, if T1 does not run, both T2 and T3 are degree 3 consistency.

Degree 3 consistency holding long write and read locks is essentially strict 2PL. Consequently, when all transactions are degree 3 consistency, the schedule is guaranteed to be both serializable and recoverable. However, serializability alone cannot (necessarily) guarantee degree 3 consistency.