Assignment 4 Solutions

CS44800 Staff

December 12, 2021

1 Query Transformation

1. Efficient relational algebra

   $$\pi_{T.branch\ name}((\pi_{branch\ name,assets}(\rho_T(branch))) \bowtie_{T.assets<S.assets} (\pi_{assets}(\sigma_{(\text{branch city='Brooklyn'})}(\rho_S(branch)))))$$

   This query is efficient, because *branch city='Brooklyn'* is the part of the query that will eliminate the most entries. By leaving this condition as the innermost one, the join will happen on the smallest set of entries possible.

2. We can use pipelining by pushing the results of

   $$\sigma'_{branch\ city='Brooklyn}$$

   into the assets projection right before it. Then we can pipeline those results into the join before it. We can pipeline again the left side projection of

   $$\pi_{branch\ name,assets}$$

   into the join.

   Using a block nested-join is the best option since the S relation can be completely fit inside the T relation. The other join options are not preferred because we don't index on assets for an index join; the relations are not sorted to use a merge join; we don't have enough information to justify a hash join.

3. The maximum number of I/Os we will require is equal to the size of our initial selects on T and S. The innermost select on S.branch city uses a very small subset of S, and the project on T uses all of T. Therefore the most I/Os we will require is equal to the size of T plus some small fraction of the size of S. Given that we will utilize pipelining in the processing of the expression, the amount of storage necessary is equal to the number of blocks necessary to store one tuple from T + one tuple from S.

4. The worst case occurs if all cities in S are "Brooklyn" because then M and N would both hold the same number of tuples, so there would be no benefit in having the inner and outer relations. However, this is likely to never happen and restricting the right side of the join to "Brooklyn" decreases the join input size as much as possible, therefore, making this the most efficient expression.

2 Cost Estimation

We compute the size as the following. We want to find min($\frac{1000*1500}{max(1100,900)}$, $\frac{1500*750}{max(50,100)}$) which we get $\frac{15000}{11}$. Then we will do $\frac{15000}{11} * \frac{750}{max(100,50)}$ which will get us $\frac{112500}{11}$ which is around 10227. This is the estimated

size of the join result.
As indicated by the min and max selection of values, it turns out joining r_1 and r_2 and then join the result with r_3 is the most efficient one because joining r_1 and r_2 first will create an intermediate relation closer to their original size and it is much smaller than joining r_2 and r_3 first.

3 Query Transformation

By applying the equivalence rule-1 (conjunctive selection operations can be deconstructed into a sequence of individual selections) on the left hand side, we can get the following:

$\sigma_{\theta_1}\sigma_{\theta_2}(E_1 \bowtie_{\theta_3} E_2)$

Now, by applying the equivalence rule-7 on the above expression, we can get the following:

$\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2} E2))$

Improving query performance: by pushing the select operator on E_2 before performing the join will reduce the number of tuples to be joined. As a result, we will be able to perform the join operation faster and hence the query performance should be improved.

4 Cost Estimation Techniques

1. We are going to assume that we have an equi-width histogram. The histogram represents the number of tuples in a certain range of the attribute A. We will start from the beginning of the histogram to the estimate A<v if the v is not lower than the lowest value of the histogram. If v is lower than the lowest value, then we just return 0. Else, we will add up the number of tuples for each range in the histogram until we reach the range where the value v is contained. Once we reach that range, we can estimate A = v by calculating the frequency count for that range divided by the number of distinct values that occur in that range. The sum of tuples calculated from A<v and A=v would be the estimated number of tuples.

2. Histogram approach can give better estimate than min max when a query is the part of the stored procedure, the value v may not be available when the query is optimized. In these cases, using histogram method is better.
   OR
   It is better to use a histogram when there are large outlier values for the minimum or maximum value since the data will be skewed. This will lead to an inaccurate cost estimation using the min max method.

5 Serializable Schedules

Assume we have two conflicting transactions that go through the entire dataset, e.g., SELECT sum(balance) FROM account) and UPDATE account SET balance=balance*1.01. Assuming the dataset does not fit in memory, when run serially each transaction will require reading in the entire account relation, for 2*blocks(account) I/Os. But if we were to, for example, let the sum(balance) transaction process the first block, then do the update on the first block, then let sum(balance) process the second block, then update that block, etc., it requires only one pass, for 1*blocks(account) I/Os. But this would still serialize as sum transaction followed by update transaction.

6 Two-Phase Locking

1. T1 reads A before W2 writes A, so T1 must be before T2. T1 doesn't do anything with A after T2 writes it, and T1 doesn't write A, so that is the only conflict on A. Only T1 operates on B, so we don't

need to worry about B. T1 reads C before W2 writes C, and doesn't operate on C after that, so again T1 must come before T2. Since these are the only conflicts, it serializes as T1 then T2.

2. Both T1 and T2 have to acquire locks on A before reading, so with a single lock type, T2 will have to wait until T1 has acquired all its locks. Assume T1 acquires all locks at the beginning, it can then release the lock on A after the read. If T2 acquires locks only before reading, it is able to do the first read, but blocks on the R2(C), so we can't get the order given (since T1 has to hold the lock on C until after the following operation.)

3. Switching the order of R2(C) and R1(C) would allow this to complete.

4. Since with strict 2-phase locking, we can't release locks until commit, no it does not. The R2(A) would have to wait until T1 is done.

5. With Shared/Exclusive locks, we can start with shared locks, so both can proceed until W2 needs an exclusive lock on C. But at this point, R1 is done with C, and since it acquired the lock on B at the beginning, it can release the lock on C (since it is in the second phase.)

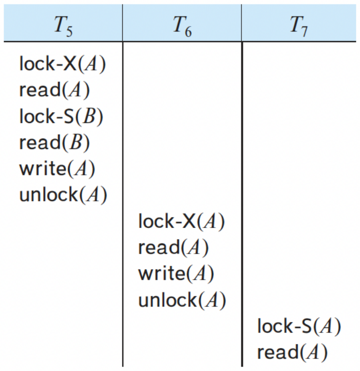
6. No need to rearrange.

7. No, because with Strict 2-PL, T1 will not be able to release any locks (including the one on A) until after its final write on B, so T2 will not be able to start.

7 Regular vs. Strict Two-Phase Locking

1. The regular two phase locking may result in cascading aborts as a transaction may observe uncommitted writes. The strict two-phase locking prevents this from happening. It also allows better recoverability.

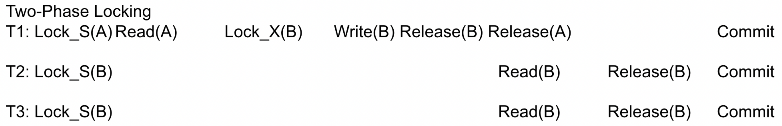
2. Consider the following example adopted from the textbook.



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

If there is a failure of T_5 right after T_7 reads A, both T_7 and T_6 need to rollback under regular 2pl.

3. Consider the following scenario.



Two-Phase Locking

T1: Lock_S(A)	Read(A)	Lock_X(B)	Write(B)	Release(B)	Release(A)		Commit
T2: Lock_S(B)					Read(B)	Release(B)	Commit
T3: Lock_S(B)					Read(B)	Release(B)	Commit

Strict Two-Phase Locking

T1:Lock_S(A)	Read(A)	Lock_X(B)	Write(B)	Release(B)	Release(A)	Commit		
T2:Lock_S(B)					Read(B)		Release(B)	Commit
T3:Lock_S(B)					Read(B)		Release(B)	Commit

Clearly, T_1 can release locks earlier under 2PL which reduces the wait time of T_2 and T_3.

8 Deadlock Avoidance

8.1) Let T_i and T_j are two transactions and T_i is older than T_j. Now, let us consider the simplest form of a deadlock situation where T_i is waiting for a lock that is held by T_j and T_j is waiting for a lock that is held by T_i (i.e., a graph with a cycle containing two vertices). Notice that if we follow a wound-wait protocol, the above situation will never occur. If the older transaction T_i request for a lock held by the younger transaction T_j, it will be granted immediately and T_j will be restarted. According to the wound-wait protocol, transactions only wait on older transactions so no cycle is created. As a result, a deadlock cannot occur using wound-wait protocol.

8.2) Let us consider the following example schedule in the following order:

T_1S(A)R(A); T_2X(B)W(B); T_1S(B); T_3S(C)R(C); T_2X(C); T_3X(A)

In the above example, a deadlock will be created where T_1 will wait for T_2, T_2 will wait for T_3 and T_3 will wait for T_1. However, if we follow the wait-die protocol, the deadlock can be avoided. According to the wait-die protocol, the the older transaction T_1 will be allowed to wait on T_2. Similarly, the older transaction T_2 will be allowed to wait for T_3. However, the younger transaction T_3 will not be allowed to wait for the older transaction T_1 and hence T_3 will be rolled back. As a result, T_3 will need to be restarted but no deadlock will occur.

8.3) Although there might be multiple ways to prevent the unnecessary roll back, one of the approaches is discussed below:

Suppose, a transaction T_i tries to lock an item X but unable to do so because X is locked by another transaction T_j. The rules for the proposed Cautious Waiting mechanism are as follows:

If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that Cautious Waiting is deadlock-free. Let us consider the b(T) at which each blocked transaction T was blocked. If two transactions T_i and T_j above both become blocked and T_i is waiting on T_j, then b(T_i) < b(T_j), since T_i can only wait on T_j at a time when T_j is not blocked. Hence, the blocking times form a total ordering on all blocked transactions, so NO cycle that causes deadlock can occur.

A second approach is to combine timeout and wound-wait or wait-die. Taking wait-die, for example, when a transaction is unable to acquire a lock that is held by an older transaction, instead of immediately aborting, it waits for a period of time. If there is no deadlock, this gives the older transaction time to release the lock and the waiting transaction can continue. If there is a deadlock, the waiting transaction will die at the end of the timeout, allowing the older transaction to continue. This is a tradeoff between how long we may delay when a deadlock occurs vs. not having to redo work.