A. Serializability I (8)

Consider the following schedule \( S \), consisting of transactions \( T_1, T_2 \) and \( T_3 \):

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( T_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w(A) )</td>
<td></td>
<td>( r(A) )</td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( w(B) )</td>
<td>( r(A) )</td>
<td></td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( w(A) )</td>
<td></td>
<td>( w(B) )</td>
<td>( r(B) )</td>
</tr>
<tr>
<td>( r(B) )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Give the precedence graph for \( S \)
- Is \( S \) conflict serializable? Justify your answer.
- Is \( S \) view serializable? Justify your answer.

Solution

\[ T_1 \rightarrow T_3 \rightarrow T_2 \]

Precedence Graph

- \( S \) is not conflict serializable because there is a circle \( T_1 - T_3 - T_1 \)
- \( S \) is view serializable because there is the following serial schedule \( S' \) where all reads of \( S \) and \( S' \) read the same value and all writes write the same value.

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( T_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w(A) )</td>
<td></td>
<td>( r(A) )</td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( w(B) )</td>
<td>( r(A) )</td>
<td>( w(A) )</td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( w(A) )</td>
<td>( r(B) )</td>
<td></td>
<td>( w(B) )</td>
</tr>
<tr>
<td>( r(B) )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B. **Serializability II (15)**

Two transactions are *not interleaved* in a schedule $S$ if every operation of one transaction precedes every operation of the other. (Note, the schedule may not be serial.) Give an example of a *serializable* schedule $S$ that has all of the following properties:

- transactions $T_1$ and $T_2$ are not interleaved in $S$
- $T_1$ starts before $T_2$ in $S$
- in any serial schedule equivalent to $S$, $T_2$ precedes $T_1$

Hint: The schedule may include more than two transactions.

**Solution**

Consider the following schedule $S$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r(A)$</td>
<td></td>
<td></td>
<td>$w(A)$</td>
</tr>
<tr>
<td>$r(B)$</td>
<td></td>
<td>$w(B)$</td>
<td></td>
</tr>
</tbody>
</table>

$S$ is serializable (acyclic precedence graph), $T_1$ and $T_2$ are not interleaved in $S$ and $T_1$ starts before $T_2$. In every serial schedule $S'$ equivalent to $S$, $T_3$ precedes $T_1$ because it is impossible to bring $r(A)$ ahead of $w(A)$. Similarly, $T_2$ precedes $T_3$ because it is impossible to bring $w(B)$ ahead of $r(B)$. Hence, in every serial schedule $T_2$ precedes $T_1$. 
C. Serializability III (15)

Consider two non-identical schedules $S$ and $S'$ consisting of transactions $T_1, ..., T_n$ where $n > 1$. For each of the following set of conditions decide whether $S$ and $S'$

- have to be **equivalent**, 
- have to be **nonequivalent**, 
- is **impossible to decide** whether they are equivalent or not using the given conditions 
- the described conditions **can not hold**

The conditions are

1. $S$ only reads, $S$ and $S'$ are serial and $n = 2$
2. All transactions $T_1, ..., T_n$ write a specific item $A$ exactly once and $S$ and $S'$ are serial. 
3. The precedence graphs of $S$ and $S'$ are identical and acyclic. 
4. $S$ is serializable but $S'$ is not serializable. 
5. No two serial schedules (of $T_1, ..., T_n$) are equivalent and $S$ and $S'$ are serializable.

<table>
<thead>
<tr>
<th></th>
<th>equivalent</th>
<th>nonequivalent</th>
<th>impossible to decide</th>
<th>can not hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

.Solution
# Two Phase Lock (20)

For each one of the following schedules decide whether
- they are serializable
- they can be produced by a Two Phase Lock (2PL) scheduler
- they can be produced by a strict 2PL scheduler

and check in the table below all entries that apply. Justify your answers in a concrete way. For example, if a schedule is 2PL show a series of lock/unlock operations that are compatible with the 2PL rules (you can use the empty lines of the schedules to place lock/unlock operations.)

Answers of the form “it is serializable because I can see that I can swap the operations” will not get full credit. Assume that shared locks can be used if in this way you can make a schedule 2PL or strict 2PL.

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(A)</td>
<td></td>
<td>r(D)</td>
</tr>
<tr>
<td></td>
<td>r(B)</td>
<td></td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
<td>w(D)</td>
</tr>
<tr>
<td></td>
<td>r(D)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r(B)</td>
<td>w(D)</td>
</tr>
<tr>
<td>w(B)</td>
<td></td>
<td>r(D)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(D)</td>
</tr>
</tbody>
</table>
It is serializable because it has an acyclic graph and 2PL because locks can be assigned as follows (many similar solutions are possible):
It can not be strict 2PL because $T_2$ will have to $unlock(B)$ at the very end and hence it will be impossible for $T_1$ to $w(B)$.

- It is not serializable because of a cycle in the precedence graph

![Precedence Graph](image1.png)

Every non-serializable schedule can not be 2PL or strict 2PL.

- It is serializable because it has an acyclic graph

![Precedence Graph](image2.png)

and 2PL because locks can be assigned as follows (many similar solutions are possible)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ls(A)$, $w(A)$</td>
<td>$ls(B)$; $ls(D)$, $R(B)$, $u(B)$</td>
<td>$ls(D)$, $r(D)$</td>
</tr>
<tr>
<td>$ls(B)$, $w(B)$, $u(A)$; $u(B)$</td>
<td>$R(D)$, $u(D)$</td>
<td>$lx(D)$, $w(D)$, $u(D)$</td>
</tr>
</tbody>
</table>

It can not be strict 2PL for the same reasons with the first schedule.
F. Recovery I (10)

1. **Undo logging** requires that before an item $X$ is modified on disk the log records pertaining to $X$ are in the disk. (This is the WAL rule.) Show using an example that an inconsistent database may result if log records for $X$ are not output to the disk before $X$.

2. **Redo logging**: Show using an example that an inconsistent database may result if some items are written on the disk before the commit is written on the log, even if WAL holds.

3. **Undo logging**: Show using an example that an inconsistent database may result if some items are not written in the disk by the time the commit is written. Assume that WAL holds.

Your examples should involve a crash and should clearly show (I) the write actions of the transaction, (II) the state of the log and the database at the time of the crash, and (III) why successful recovery can not be accomplished.

.Solution

1. Consider a transaction $\{w(A), w(B)\}$ that crashes after $w(A)$. At the time of the crash the new value of $A$ is on disk but the log record for $A$ is not on disk. After the crash we can not recover $A$ to the value it had before the crash.

2. Consider again the transaction $\{w(A), w(B)\}$ that crashes after $w(A)$. At the time of the crash both the new value of $A$ and the log record for $A$ are on disk. However, **Redo** does not undo the effect of any transaction. Hence the new value of $A$ stays on disk. (Indeed no recovery procedure can save us because the log record has the new value of $A$. The old value is lost.)

3. Consider a transaction $\{w(A), w(B)\}$ that completes but the computer crashes when $A$ is on disk, $B$ is not, and the commit record is. Upon recovery **Undo** will not deal with this transaction because it has will find the commit.
G. Recovery II (20)

We know that non strict 2PL may cause cascading aborts. Even worse, it could be the case that a crash occurs while cascading aborts take place. The goal of this exercise is to develop an Undo logging system that can handle cascading aborts, i.e., upon restart it can abort the transactions that have to be aborted and so on.

1. What additions are needed in the Undo logging data for handling cascading aborts
2. Write the rules of your enhanced “cascading abort Undo logging” (in the spirit of the notes). Try to pose rules that are not very restrictive.
3. Write an algorithm for recovery (restart) using notation similar with the notes.

Clarification on notes: In the slide of “Undo Logging Recovery” the condition “NO <T, Commit> (or <T, Abort>) in log” is parenthesized as “(NO <T, Commit>) OR <T, Abort> in log”.

Solution

1. We have to keep track of the reads performed by transactions so that we can tell which transaction has used values that have been written by aborted transactions.
2. In addition to the three rules of the notes the following rules should also be observed
   - For every read generate read log record containing the name of the read item.
   - Before commit is written on disk all read log entries should be permanently stored.
   - If write(A) happened before read(A) then the write log record appears before the read log record.

Let \( S = \) set of transactions \( T \) with \(<T,start>\) in log and no \(<T,commit>\) or an \(<T,abort>\) in log.

Loop until no more transaction \( T' \) can be added in \( S \)

If \( T' \) is committed and
   - there is a read log record of \( T' \) for item \( X \) and
   - there is a write log record of \( T \), where \( T \) is in \( S \), for item \( X \).

then put \( T' \) in \( S \).

Proceed by Undoing all transactions of \( S \) as done in notes.
Consider a transaction that performs the following actions in the given order
1. Write object A
2. Write object B
3. Commit

Decide whether each one of the following snapshots of the database and the log are possible or impossible at any point during or after the end of the transaction. The snapshots refer to data that are actually in the disk. Assume that if we do not mention explicitly that something is stored on the disk then it is not stored on the disk. For each item give an answer for both Undo and Redo logging.

1. Log has entry for write(A) and the database has the new value of A
2. The database has the new value for A
3. The log has entries for write(A) and write(B) and also has the commit entry
4. The log has entries for write(A) and write(B). The database has the new values for A and B
5. The log has entries for write(A), write(B), and the commit entry. The database has the new value for A
6. Log has entry for write(A) and commit and the database has the new value of A

<table>
<thead>
<tr>
<th></th>
<th>UNDO LOGGING</th>
<th>REDO LOGGING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>possible</td>
<td>impossible</td>
</tr>
<tr>
<td>1</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
I. Query Processing I (20)

Can it be worthwhile to temporarily create an index (B-tree index, hash table index, ...) to speed up the cost of executing a query? That is, does it make sense during the query execution to create an index that is discarded at the end of the query? If your answer is “no” discuss why other alternatives (nested loops, merge join, or hash join) will always be better. If your answer is "yes", give an example, including statistics and rough cost estimates, that shows that the index join (with index creation) is cheaper than nested loops or hash join. You may assume that merge join is too expensive if the relations are not sorted.

Solution

Consider the following join between $R$ and $S$

SELECT * FROM $R$, $S$ WHERE $R.A = S.A$

It is worthwhile to create an index on $R$ if (I) both $R$ and $S$ are too large, with respect to the main memory, for nested loops to be beneficial, and (II) $S$ is much larger than $R$ and only a few tuples of $S$ join with $R$, so that hash join will be too expensive. Here are some statistics that prove the point:

- $t(R) = 10^7$
- $t(S) = 10^{11}$
- $V(A,R) = 10^7$ (not really important)
- $s(R) = s(S) = 400$
- attribute A’s size is 100
- block size = 4000
- main memory = $4 \times 10^6$, i.e., $10^4$ tuples fit in main memory (any small number will do the trick)
- join selectivity = $10^{-11}$ (i.e., on the average every tuple of $R$ joins with one tuple of $S$)
- no sorting
- record pointer is 4 bytes long

The cost of

- Assume a hash index on $A$ with function $h$ is already in place. Then the cost of nested loops is $\sim 10^{11}$ because for every tuple of $S$ the join algorithm accesses the index to find joining tuples of $R$ and most probably it doesn’t find $h$. So, $k$ must be a small number such as 3 or 4. Creating the index on $A$ is in the order of $10^7$, i.e., it is insignificant. Notice that we get the above cost even without using the main memory.
- Nested loops cost is larger than $(10^7/10^6) 10^{11}/10 = 10^{13}$ block accesses.
- Let us assume that the same function $h$ is used. Then we end up having to hash into buckets the whole $S$ though only very tuples will actually join. Creating buckets for $S$ will occupy more than $10^{11}100/4000$ pages that will have to be written and reread.
J. Query Processing II (22)

Consider the following query:

```sql
SELECT *
FROM R, S, T
WHERE R.A = S.A and R.B = T.B AND T.C = S.C
     AND R.D = 16 AND S.F = 17
```

1. Give six algebraic expressions that (I) contain no cartesian product, (II) have pushed selections down, (III) each join is associated with an atomic condition (not a conjunctive condition).

2. Find the optimum plan, i.e., an algebraic expression where selections and projections are annotated with execution primitives. The execution primitives are `SCAN` and `INDEX` for the selection operator and `LOOPS`, `INDEX`, `MERGE`, and `HASH` for join. Compute an estimation of the optimal plan’s cost.

Assume the following schemas and statistics for the relations.

- **R(A, B, D, E)** has 1 million tuples, an index on D, \(V(D,R)=1000\), \(V(A,R)=1000000\), \(V(B,R)=1000000\).
- **S(A, C, F, G)** has 1 million tuples, an index on F, \(V(F,S)=1000\), \(V(A,S)=1000000\), and \(V(C,S)=1000000\).
- **T(B, C, H, I)** has 1 million tuples, an index on B, with \(V(B,T)=1000000\), \(V(C,T)=1000000\).

Assume the block size is 4096 Bytes, the block header size is 96 bytes, each indexed attribute occupies 5 bytes and the tuple size of each relation is 80 bytes.

To estimate the size of the joins use the following information:

- For every tuple of R there is exactly one tuple of S with the same A value and one tuple of T with the same B value. Equivalently, the chances that a given S tuple joins a given R tuple are \(1/1000000\).
- For every tuple of S there is exactly one tuple of R with the same A value and one tuple of T with the same C value.
- For every tuple of T there is exactly one tuple of R with the same B value and one tuple of S with the same B value.

Assume there is one megabyte of main memory available. Hint: You should use the main memory to avoid copying intermediate results to the disk, if there is enough space in main memory.
The optimal plan is derived from (1) by using INDEX for the selections \( \sigma_{D=16} R \) and \( \sigma_{F=17} S \). The resulting relations are so small that their join can be computed using nested loops in main memory and the join is also very small (indeed only one tuple) so that we can directly use INDEX for the top join.

An estimation of the cost:
- Retrieving the \( \frac{1000000}{1000} = 1000 \) tuples of \( \sigma_{D=16}^{INDEX} R \) will require \( \sim 1000 \) block accesses because the relation is not sorted. (There are a few extra accesses for the index.)
• Retrieving the \( \frac{1000000}{1000}=1000 \) tuples of \( \sigma_{F=17}^\text{INDEX} \) will require \( \sim 1000 \) block accesses because the relation is not sorted.

• We don’t store \( \sigma_{D=16}^\text{INDEX} \) and \( \sigma_{F=17}^\text{INDEX} \) in the disk because they are only \( \sim 80 \) Kbytes (each one.) We directly perform the nested loops join on \( A \) that costs nothing in terms of block accesses. The expected size of the result is 1 tuple because the probability of one tuple of \( R \) joining with a specific tuple of \( S \) is \( 1/1000000 \) and we have 1000 tuples of \( R \) and 1000 tuples of \( S \).

• We keep the 1 tuple in main memory and use the \( B \) index to find a joining \( T \) tuple. The cost is unimportant.

Overall the cost is \( \sim 2000 \) block accesses.