CONCURRENCY CONTROL INTRODUCTION

• Motivation: A dbms is multiprogrammed to increase the utilization of resources. While the CPU is processing one transaction, the disk can perform a write operation on behalf of another transaction. However, the interaction between multiple transactions must be controlled to ensure atomicity and consistency of the database.

• As an example, consider the following two transactions. $T_0$ transfers 50 from account A to B. $T_1$ transfers 10% of the balance from A to B.

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A=A-50</td>
<td>temp = A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B=B+50</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A=A-50</td>
<td>temp = A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(B)</td>
</tr>
<tr>
<td>B=B+50</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

• Assume the balance of A is 1000 and B is 2000. The bank's total balance is 3000. If the system executes $T_0$ before $T_1$, then A's balance will be 855 while B's balance will be 2145. On the other hand, if $T_1$ executes before $T_0$, then A's balance will be 850 and B's balance will be 2150. However, note that in both cases, the total balance of these two accounts is 3000.

• In this example both schedules are consistent.

• A schedule is a chronological execution order of multiple transactions by a system.

• A serial schedule is a sequence of processing multiple transactions which ensures atomicity of each transaction. Given $n$ transactions, there are $n!$ valid serial schedules.

• Concurrent execution of multiple transactions, where the instructions of different transactions are interleaved, may result in a non-serial schedule.

• To illustrate, assuming that a transaction makes a copy of each data item in order to manipulate it, consider the following execution:

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A=950</td>
<td>temp = A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - temp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(B)</td>
</tr>
<tr>
<td>B=2000</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

• In general, given two transactions $T_i$ and $T_j$ with instructions $I_i$ and $I_j$:

  1. If $I_i$ and $I_j$ refer to different data items then their execution order does not matter.
  2. If they refer to the same data item $Q$ then the order might be important:
     a. $I_i = \text{read}(Q)$
        $I_j = \text{read}(Q)$
        order does not matter
     b. $I_i = \text{read}(Q)$
        $I_j = \text{write}(Q)$
        order is important because:
        1. If the order is $I_i$, $I_j$, then $T_j$ does not observe $T_i$'s update
        2. If the order is $I_j$, $I_i$, then $T_i$ observes the update of $T_j$
     c. $I_i = \text{write}(Q)$
        $I_j = \text{read}(Q)$
        order is important because:
        1. If the order is $I_j$, $I_i$, then $T_i$ does not observe $T_j$'s update
        2. If the order is $I_i$, $I_j$, then $T_j$ observes the update of $T_i$
     d. $I_i = \text{write}(Q)$
        $I_j = \text{write}(Q)$
        order is important because the value obtained by the next read(Q) instruction observes either the value produced by $I_i$ or $I_j$ depending on whichever executed last.
LOCK-BASED PROTOCOLS

- There are alternative approaches to ensure serializability among multiple transactions.

Lock-based Protocols

- To ensure serializability, require access to data items to be performed in a mutually exclusive manner.
- This approach requires a transaction to lock a data item before accessing it. The simple protocol consists of two lock modes: Shared and Exclusive. A transaction locks a data item Q in S mode if it plans to read Q's value and X mode if it plans to write Q. The compatibility between these two lock modes is as follows:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>X</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

LOCK-BASED PROTOCOLS (Cont…)

- There can be multiple transactions with S locks on a particular data item. However, only one X lock is allowed on a data item. When a transaction requests a lock, if a compatible lock mode exists, it proceeds to lock the data item. Otherwise, it waits. Waiting might result in deadlocks.

LOCK-BASED PROTOCOLS (Cont…)

- As a solution to deadlocks, most systems construct a Transaction Wait for Graph (TWG). A transaction is represented as a node in TWG. When a transaction \( T_i \) waits for \( T_j \), an arc is attached from \( T_i \) to \( T_j \). Next, the system detects cycles in the TWG. If a cycle is detected then there exists a deadlock. The system breaks deadlocks by aborting one of the transactions (e.g., using log-based recovery protocol).

- With a two-phase locking protocol, each transaction is required to release its locks at the end of its execution. Thus, a transaction has two phases:
  1. growing phase: the transaction acquires locks
  2. shrinking phase: the transaction releases locks and acquires no additional locks

LOCK-BASED PROTOCOLS (Cont…)

Without a two-phase locking protocol, the schedule provided by an execution of transactions might no longer be serializable. This is specially true in the presence of aborts. Several possible situations might arise:

1. dirty reads: A transaction \( T_0 \) reads the value of a record Q at two different points in time (\( t_i \) and \( t_j \)) and observes a different value for this record. This is because an updating transaction \( T_1 \) produced the value of Q when \( T_0 \) read this value at time \( t_i \). However, \( T_1 \) aborted sometimes later (prior to \( t_j \)) and when \( T_0 \) tried to read the value of Q at \( t_j \), it observes the value of Q prior to execution of \( T_1 \).

2. un-repeatable reads: A transaction \( T_0 \) reads the value of a record Q at two different points in time (\( t_i \) and \( t_j \)) and observes a different value for this record. After \( T_0 \) reads the value of Q at time \( t_i \), an updating transaction \( T_1 \) updates the value of Q and commits prior to \( t_j \). When \( T_0 \) read this value of Q at time \( t_j \), it observes a different value for Q.
LOCK-BASED PROTOCOLS (Cont…)

Example of dirty reads:

\[
\begin{align*}
T_0 & \quad \text{lockX}(Q) \quad \text{read}(Q) \\
Q & = Q + 50 \quad \text{unlock}(Q) \\
T_1 & \quad \text{write}(Q) \quad \text{lockX}(Q) \\
T_0 & \quad \text{unlock}(Q) \quad \text{read}(Q) \\
T_1 & \quad \text{write}(Q) \quad \text{unlock}(Q) \\
T_0 & \quad \text{abort}
\end{align*}
\]

Example of unrepeatable reads:

\[
\begin{align*}
T_0 & \quad \text{lockS}(Q) \\
Q & = Q + 50 \quad \text{unlock}(Q) \\
T_1 & \quad \text{read}(Q) \quad \text{read}(Q) \\
T_0 & \quad \text{unlock}(Q) \\
T_1 & \quad \text{read}(Q) \quad \text{commit}
\end{align*}
\]

3. dirty writes (lost updates): T₀ and T₁ read the value of Q at two different points in time and produce a new value for this data item. Subsequently, they overwrite each other when updating Q. The execution paradigm that motivated the use of locking (earlier in the lecture notes) is an example of dirty writes.

- Most systems support four levels of lock granularities:
  - Level 3: locks held to end of a transaction (two phase locking that results in serializable schedules)
  - Level 2: write locks held to end of a transaction (un-repeatable reads)
  - Level 1: no read locks at all (dirty reads and un-repeatable reads)
  - Level 0: no locks (dirty writes, dirty reads and un-repeatable reads)

MULTI-GRANULARITY LOCKING

During the previous lecture we saw locking: S, X, IS, IX, SIX

This lecture will cover:
- Time-stamp based protocols
- Optimistic concurrency control

Time-stamp based protocol
- provide a mechanism to enforce order. How?
- Associated with each data item Q are two time stamp values:
  - W-TimeStamp(Q): Largest time stamp of the transaction that has written Q to date
  - R-TimeStamp(Q): Largest time stamp of the transaction that has read Q to date

TIME STAMP BASED PROTOCOL (Cont…)

- Consider the following schedule

\[
\begin{align*}
T_1 & \quad \text{Write}(Q) \\
T_2 & \quad \text{Read}(Q) \\
T_3 & \quad \text{Write}(Q)
\end{align*}
\]

- The rollback of T₂ is unnecessary because T₁ has already produced the final value. The right thing to do is to ignore the write operation performed by T₂.

THOMAS’S WRITE RULE

- Suppose transaction T₂ issues write(Q):
  - If TST(T₂) > W-TimeStamp(Q) then T₂ needs to read the value of Q which was already overwritten. Hence the read request is rejected and T₂ is rolled back.
  - If TST(T₂) = W-TimeStamp(Q) then the read operation is executed and the W-TimeStamp(Q) is set to the maximum of R-TimeStamp(Q) and TST(T₂).
- Suppose transaction T₃ issues write(Q):
  - If TST(T₃) < R-TimeStamp(Q) then this implies that some transaction has already consumed the value of Q and T₃ should have produced a value before that transaction read it. Hence the write request is rejected and T₃ is rolled back.
  - If TST(T₃) = W-TimeStamp(Q) then T₃ is trying to write an obsolete value of Q. Hence reject T₃’s request and roll it back.
  - Otherwise, execute the write(Q) operation and update W-TimeStamp(Q) to TST(T₃).
- When a transaction is rolled back, the system may assign a new timestamp to the transaction and restart its execution (as if it was just submitted).
- This approach is free from deadlocks.
OPTIMISTIC CC (VALIDATION TECHNIQUE)

- Argues that the overhead of locking is too high and not worthwhile for applications whose workload consists of read-only transactions.
- Each transaction $T_i$ has three phases:
  - Read phase: reads the value of data items and copies its contents to variables local to $T_i$. All write operations are performed on the temporary local variables.
  - Validation phase: $T_i$ determines whether the local variables whose values have been overwritten can be copied to the database. If not then abort. Otherwise, proceed to Write phase.
  - Write phase: The values stored in local variables overwrite the value of the data items in the database.
- A transaction has three time stamps:
  - Start($T_i$): When $T_i$ started its execution.
  - Validation($T_i$): When $T_i$ finished its read phase and started its validation.
  - Finish($T_i$): done with the write phase.
- $TS(T_i) = Validation(T_i)$ instead of Start($T_i$) because it produces a better response time if the conflict rate between transactions is low.

OPTIMISTIC CC (VALIDATION TECHNIQUE) (Cont...)

- When validating transaction $T_j$, for all transactions $T_i$ with $TS(T_i) < TS(T_j)$, one of the following must hold:
  - Finish($T_i$) < Start($T_j$), OR
  - Set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$. $T_i$ completes its write phase before $T_j$ starts its validation phase.
- Rational: Serializability is maintained because the write of $T_i$ cannot affect the read of $T_j$ and since $T_j$ cannot affect the read of $T_i$ because $Start(T_j) < Finish(T_i) < Validation(T_j)$

MULTIVERSION CHANGES

- The system keeps track of the old version of a data item $Q$.
- When a Write($Q$) operation is issued, the system creates a new version of $Q$.
- When a Read($Q$) operation is issued, the system selects the right version of $Q$ to read.
- As before, each transaction has a unique time stamp.
- A data item $Q$ has a sequence of versions associated with it: $Q_1, Q_2, ..., Q_n$.
- Each $Q_k$ has three data fields:
  - Content: its value.
  - W-TimeStamp: Time stamp of transaction that created version $k$.
  - R-TimeStamp: Time stamp of the largest transaction that successfully read version $k$.
- $T_i$ creates a new version $Q_k$ of data item when it issues Write($Q$). Its content holds the new value. The W-TimeStamp and R-TimeStamp are initialized to $TS(T_i)$ and $R-TimeStamp(Q_k) = TS(T_i)$.

MULTIVERSION CHANGES (Cont...)

- Assume that transaction $T_i$ issues either a read($Q$) or a write($Q$) operation.
- Let $Q_k$ denote the version of $Q$ whose write time stamp is the largest write time stamp less than $TS(T_i)$, i.e., $W-TimeStamp(Q_k) = TS(T_i)$.
- If $T_i$ issues a Read($Q$) then return the value of $Q_k$.
- If $T_i$ issues a Write($Q$), and $TS(T_i) < R-TimeStamp(Q_k)$ then $T_i$ is rolled back.
- Otherwise a new version of $Q_k$ is created.