The lost update problem

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>balance = b.getBalance();</code></td>
<td><code>balance = b.getBalance();</code></td>
</tr>
<tr>
<td><code>b.setBalance(balance*1.1);</code></td>
<td><code>b.setBalance(balance*1.1);</code></td>
</tr>
<tr>
<td><code>a.withdraw(balance/10)</code></td>
<td><code>c.withdraw(balance/10)</code></td>
</tr>
<tr>
<td><code>balance = b.getBalance();</code></td>
<td><code>balance = b.getBalance();</code></td>
</tr>
<tr>
<td><code>balance = $200</code></td>
<td><code>balance = $200</code></td>
</tr>
<tr>
<td><code>b.setBalance(balance*1.1);</code></td>
<td><code>b.setBalance(balance*1.1);</code></td>
</tr>
<tr>
<td><code>balance = $220</code></td>
<td><code>balance = $220</code></td>
</tr>
<tr>
<td><code>a.withdraw(balance/10)</code></td>
<td><code>a.withdraw(balance/10)</code></td>
</tr>
<tr>
<td><code>balance = $220</code></td>
<td><code>balance = $220</code></td>
</tr>
<tr>
<td><code>c.withdraw(balance/10)</code></td>
<td><code>c.withdraw(balance/10)</code></td>
</tr>
<tr>
<td><code>balance = $80</code></td>
<td><code>balance = $80</code></td>
</tr>
<tr>
<td><code>balance = $280</code></td>
<td><code>balance = $280</code></td>
</tr>
</tbody>
</table>

- the initial balances of accounts A, B, C are $100, $200, $300
- both transfer transactions increase B’s balance by 10%

The net effect should be to increase B by 10% twice - 200, 220, 242. but it only gets to 220. U’s update is lost.
The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction $V$:</th>
<th>Transaction $W$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.withdraw(100)</td>
<td>$a$.Branch.branchTotal()</td>
</tr>
<tr>
<td>$b$.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>$a$.withdraw(100);</td>
<td>total = $a$.getBalance() $100$</td>
</tr>
<tr>
<td></td>
<td>total = total + $b$.getBalance() $300$</td>
</tr>
<tr>
<td>Figure 13.6</td>
<td>total = total + $c$.getBalance()</td>
</tr>
<tr>
<td>$b$.deposit(100)</td>
<td>$300$</td>
</tr>
</tbody>
</table>

- the initial balances of accounts A and B are both $200$
- $V$ transfers $100$ from A to B while $W$ calculates branch total (which should be $400$)
A serially equivalent interleaving of $T$ and $U$
(lost updates cured)

If one of $T$ and $U$ runs before the other, they can’t get a lost update,
the same is true if they are run in a serially equivalent ordering.

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$balance = b.getBalance()$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$b.setBalance(balance*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$c.withdraw(balance/10)$</td>
</tr>
</tbody>
</table>

| balance = b.getBalance() | $200$ |
| $b.setBalance(balance*1.1)$ | $220$ |
| $a.withdraw(balance/10)$ | $80$ |
| | $220$ |
| | $242$ |
| | $80$ |
| | $278$ |

Figure 13.7

Their access to $B$ is serial, the other part can overlap.
A serially equivalent interleaving of $V$ and $W$

(inconsistent retrievals cured)

We could overlap the 1st line of $W$ with the 2nd line of $V$

<table>
<thead>
<tr>
<th>Transaction $V$:</th>
<th>Transaction $W$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
</tbody>
</table>

| a.withdraw(100); | $100 |
| b.deposit(100)   | $300 |

$\text{total} = a.\text{getBalance()}$  $\quad$ $\$100$

$\text{total} = \text{total} + b.\text{getBalance()}$  $\quad$ $\$400$

$\text{total} = \text{total} + c.\text{getBalance()}$

$\ldots$

- if $W$ is run before or after $V$, the problem will not occur
- therefore it will not occur in a serially equivalent ordering of $V$ and $W$
- the illustration is serial, but it need not be
**Read and write operation conflict rules**

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of <em>read</em> operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a <em>read</em> and a <em>write</em> operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of <em>write</em> operations depends on the order of their execution</td>
</tr>
</tbody>
</table>

- **Conflicting operations**
- a pair of operations conflicts if their combined effect depends on the order in which they were executed
  - e.g. *read* and *write* (whose effects are the result returned by *read* and the value set by *write*)
Serial equivalence defined in terms of conflicting operations

• For two transactions to be **serially equivalent**, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

• Consider
  - T: $x = \text{read}(i); \, \text{write}(i, 10); \, \text{write}(j, 20)$;
  - U: $y = \text{read}(j); \, \text{write}(j, 30); \, z = \text{read}(i)$;
  - serial equivalence requires that either
    - T accesses i before U and T accesses j before U. or
    - U accesses i before T and U accesses j before T.

• Serial equivalence is used as a criterion for designing concurrency control schemes.
A non-serially equivalent interleaving of operations of transactions $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = read(i)$</td>
<td>$y = read(j)$</td>
</tr>
<tr>
<td>write($i, 10$)</td>
<td>write($j, 30$)</td>
</tr>
<tr>
<td>write($j, 20$)</td>
<td>z = read ($i$)</td>
</tr>
</tbody>
</table>

- Each transaction’s access to $i$ and $j$ is serialised w.r.t one another, but
- $T$ makes all accesses to $i$ before $U$ does
- $U$ makes all accesses to $j$ before $T$ does
- therefore this interleaving is not serially equivalent
A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.getBalance()$</td>
<td>$a.getBalance()$</td>
</tr>
<tr>
<td>$a.setBalance(balance + 10)$</td>
<td>$a.setBalance(balance + 20)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$balance = a.getBalance()$</th>
<th>$balance = a.getBalance()$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100$</td>
<td>$110$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a.setBalance(balance + 10)$</th>
<th>$a.setBalance(balance + 20)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$110$</td>
<td>$130$</td>
</tr>
</tbody>
</table>

- $U$ reads $A$'s balance (which was set by $T$) and then commits

$U$ has performed a *dirty read*

$T$ subsequently aborts.

$U$ has committed, so it cannot be undone

Fix: $U$ waits until $T$ commits or aborts

These executions are serially equivalent

Figure 13.11
Premature writes - overwriting uncommitted values

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>before $T$ and $U$ the balance of $A$ was $100$</td>
<td>serially equivalent executions of $T$ and $U$</td>
</tr>
<tr>
<td>$a.setBalance(105)$</td>
<td>$a.setBalance(110)$</td>
</tr>
<tr>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td>$105$</td>
<td>$105$</td>
</tr>
</tbody>
</table>

Figure 13.12

- some database systems keep ‘before images’ and restore them after aborts.
  - e.g. $100$ is before image of $T$’s write, $105$ is before image of $U$’s write
  - if $U$ aborts we get the correct balance of $105$,
  - But if $U$ commits and then $T$ aborts, we get $100$ instead of $110$
  - Similarly, if $T$ aborts and then $U$ aborts, we get $105$ instead of $100$

- Fix: write operations must be delayed until earlier transactions that updated the same objects have committed or aborted
Locks

- Transactions must be scheduled so that their effect on shared objects is serially equivalent
- A server can achieve serial equivalence by serialising access to objects, e.g. by the use of locks
- for serial equivalence,
  - (a) all access by a transaction to a particular object must be serialized with respect to another transaction’s access.
  - (b) all pairs of conflicting operations of two transactions should be executed in the same order.
- to ensure (b), a transaction is not allowed any new locks after it has released a lock
  - **Two-phase locking** - has a ‘growing’ and a ‘shrinking’ phase
Transactions T and U with exclusive locks

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance()</td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td>b.setBalance(bal*1.1)</td>
<td>b.setBalance(bal*1.1)</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>c.withdraw(bal/10)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td></td>
<td>openTransaction</td>
<td></td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td>lock B</td>
<td>bal = b.getBalance()</td>
<td></td>
</tr>
<tr>
<td>b.setBalance(bal*1.1)</td>
<td></td>
<td>b.setBalance(bal*1.1)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>lock A</td>
<td>a.withdraw(bal/10)</td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlock A, B</td>
<td>closeTransaction</td>
<td></td>
</tr>
</tbody>
</table>

- Initially the balances of A, B and C unlocked
- The use of the lock on B effectively serializes access to B

When T is about to use B, it is locked for T, and U waits when U is about to use B, it is still locked for T. When T commits, it unlocks B. U can now continue.
Strict two-phase locking

- strict executions prevent dirty reads and premature writes (if transactions abort).
  - a transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted.
  - to enforce this, any locks applied during the progress of a transaction are held until the transaction commits or aborts.
  - this is called **strict two-phase locking**
  - For recovery purposes, locks are held until updated objects have been written to permanent storage
  - More than necessary: exclusive locks reduce concurrency
    - No conflict between pairs of read operations on an object
- The ‘many reader/ single writer’ scheme allows several transactions to read an object or a single transaction to write it (but not both)
- It uses read locks and write locks
  - read locks are sometimes called shared locks
Lock compatibility

- The operation conflict rules tell us that:
  1. If a transaction $T$ has already performed a *read* operation on a particular object, then a concurrent transaction $U$ must not *write* that object until $T$ commits or aborts.
  2. If a transaction $T$ has already performed a *write* operation on a particular object, then a concurrent transaction $U$ must not *read* or *write* that object until $T$ commits or aborts.

To enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction.

To enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction.

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock already set</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td>Lock already set</td>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>wait</td>
</tr>
<tr>
<td>write</td>
<td>wait</td>
<td>wait</td>
</tr>
</tbody>
</table>

Figure 13.15
Inconsistent retrieval and Lock updates

- Inconsistent retrievals are prevented by this locking scheme
- Lost updates – two transactions read an object and then use it to calculate a new value.
  - prevented by making later transactions delay their reads until the earlier ones have completed.
  - each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
  - when another transaction requires a read lock it will be delayed until any current transaction has completed
- Lock promotion: the conversion of a lock to a stronger lock – that is, a lock that is more exclusive.
  - demotion of locks (making them weaker) is not allowed
  - It may allow executions by other transactions that are inconsistent with serial equivalence
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

- The server applies locks when the read/write operations are about to be executed
- the server releases a transaction’s locks when it commits or aborts
- The granting of locks will be implemented by a separate object in the server that we call the lock manager.
Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>$a.deposit(100)$;</td>
<td>write lock $A$</td>
<td>$b.deposit(200)$</td>
<td>write lock $B$</td>
</tr>
<tr>
<td>$b.withdraw(100)$</td>
<td>waits for $U$'s</td>
<td>lock on $B$</td>
<td>$a.withdraw(200)$;</td>
<td>waits for $T$'s</td>
</tr>
<tr>
<td></td>
<td>lock on $B$</td>
<td></td>
<td></td>
<td>lock on $A$</td>
</tr>
</tbody>
</table>

T accesses A $\rightarrow$ B
U accesses B $\rightarrow$ A

- The `deposit` and `withdraw` methods are atomic. Although they read as well as write, they acquire write locks.
- The lock manager must be designed to deal with deadlocks.
The wait-for graph for the previous figure

- Definition of deadlock
  - deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  - a *wait-for graph* can be used to represent the waiting relationships between current transactions

- In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions

If one transaction is aborted, then its locks are released and that cycle is broken
Another wait-for graph

- $T$, $U$ and $V$ share a read lock on $C$ and
- $W$ holds write lock on $B$ (which $V$ is waiting for)
- $T$ and $W$ then request write locks on $C$ and deadlock occurs
e.g. $V$ is in two cycles - look on the left

Figure 13.22
Deadlock prevention is unrealistic

- One simple way: lock all of the objects used by a transaction when it starts
  - unnecessarily restricts access to shared resources.
  - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- There are some other similar techniques for deadlock prevention.
- One common shortcoming is a reduction in concurrency.
Deadlock detection

- by finding cycles in the wait-for graph
  - the software for deadlock detection can be part of the lock manager
  - it holds a representation of the wait-for graph so that it can check it for cycles from time to time
  - edges are added to the graph and removed from the graph by the lock manager’s `setLock` and `unLock` operations
  - when a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
  - it is hard to choose a victim - e.g. choose the oldest or the one in the most cycles
Timeouts on locks

• Lock timeouts can be used to resolve deadlocks
  – each lock is given a limited period. With that time, acts as usual
  – after this time,
    ♦ if no other transaction is competing for the locked object, the lock is allowed to remain;
    ♦ but if any other transaction is waiting to access the object protected by this lock, the lock is broken.
  – The transaction whose lock has been broken is normally aborted

• problems with lock timeouts
  • locks may be broken when there is no deadlock
  • if the system is overloaded, lock timeouts will happen more often and long-time transactions are more likely to be aborted
  • it is hard to select a suitable length for a timeout
Optimistic concurrency control

- the scheme is called optimistic because the likelihood of two transactions conflicting is low
- To avoid drawbacks in locking scheme
  - In lock scheme, even read-only transactions use locking in order to guarantee that the data is not modified by other transactions at the same time. But locking is necessary only in the worst case.
  - The use of locks can result in deadlocks.
  - To avoid cascading aborts, locks cannot be released until the end of the transaction. (reduce concurrency)
- a transaction proceeds without restriction until the close Transaction (no waiting, therefore no deadlock)
- it is then checked to see whether it has come into conflict with other transactions
- when a conflict arises, a transaction is aborted
- each transaction has three phases
three phases

Working phase
- the transaction uses a tentative version of the objects it accesses (dirty reads can’t occur as we read from a committed version or a copy of it)
- the coordinator records the readset and writeset of each transaction

Validation phase
- at closeTransaction the coordinator validates the transaction (looks for conflicts)
- if the validation is successful the transaction can commit.
- if it fails, either the current transaction, or one it conflicts with is aborted

Update phase
- If validated, the changes in its tentative versions are made permanent.
- read-only transactions can commit immediately after passing validation.
Timestamp ordering concurrency control

- each operation in a transaction is validated when it is carried out
  - if an operation cannot be validated, the transaction is aborted
  - each transaction is given a unique timestamp when it starts.
    - The timestamp defines its position in the time sequence of transactions.
    - from the server’s clock, or a counter similar to the counter used in logical clock.
  - requests from transactions can be totally ordered by their timestamps.
- basic timestamp ordering rule (based on operation conflicts)
  - A request to write an object is valid only if that object was last read and written by earlier transactions.
  - A request to read an object is valid only if that object was last written by an earlier transaction
- No deadlock since transactions only wait for earlier ones.
Timestamp-based concurrency control in the SDD-1 system

- Each object has a write timestamp and a set of tentative version (each version has a write timestamp); and a set of read timestamps.
- The write timestamp of the object is earlier than the write timestamp of any of its tentative version.
- Operation conflicts for timestamp ordering

<table>
<thead>
<tr>
<th>Rule</th>
<th>Operation</th>
<th>$T_c$ must not write an object that has been read by any $T_i$ where $T_i &gt; T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write read</td>
<td>$T_c &gt;$ write timestamp of the committed object.</td>
</tr>
<tr>
<td>2.</td>
<td>write write</td>
<td>$T_i &gt; T_c$</td>
</tr>
<tr>
<td>3.</td>
<td>read write</td>
<td>$T_i &gt; T_c$</td>
</tr>
</tbody>
</table>
Timestamp ordering write rule: combining rules 1 and 2

if ($T_c \geq$ maximum read timestamp on $D$ && $T_c >$ write timestamp on committed version of $D$)
perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
Abort transaction $T_c$

(a) $T_3$ write

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_2$</td>
<td>$T_2$ $T_3$</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ $T_2$</td>
<td>$T_1$ $T_2$ $T_3$</td>
</tr>
</tbody>
</table>

Key:

- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$)

$T_1 < T_2 < T_3 < T_4$
Timestamp ordering read rule: rule 3

if \( T_c > \) write timestamp on committed version of \( D \) \{ 
  let \( D_{\text{selected}} \) be the version of \( D \) with the maximum write timestamp \( \leq T_c \)
  if \( D_{\text{selected}} \) is committed
    perform read operation on the version \( D_{\text{selected}} \)
  else
    Wait until the transaction that made version \( D_{\text{selected}} \) commits or aborts
    then reapply the read rule
\} else
  Abort transaction \( T_c \)
Timestamp ordering read rule: rule 3

(a) T₃ read

(b) T₃ read

(c) T₃ read

(d) T₃ read

Key:

- Committed
- Tentative

object produced by transaction Tᵢ (with write timestamp Tᵢ)

Tᵢ < T₂ < T₃ < T₄
Transaction commits with timestamp ordering

- when a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - committed versions of an object must be created in timestamp order
  - the server may sometimes need to wait, but the client need not wait
  - to ensure recoverability, the server will save the ‘waiting to be committed versions’ in permanent storage

- the timestamp ordering algorithm is strict because
  - the read rule delays each read operation until previous transactions that had written the object had committed or aborted
  - writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted
Comparison of methods for concurrency control

- Pessimistic approach (detect conflicts as they arise)
  - Timestamp ordering: serialisation order decided statically
  - Locking: serialisation order decided dynamically
  - Timestamp ordering is better for transactions where reads >> writes,
  - Locking is better for transactions where writes >> reads
  - Strategy for aborts
    - Timestamp ordering – immediate
    - Locking – waits but can get deadlock

- Optimistic methods
  - All transactions proceed, but may need to abort at the end
  - Efficient operations when there are few conflicts, but aborts lead to repeating work

- The above methods are not always enough e.g.
  - In cooperative work there is a need for user notification
  - Applications such as cooperative CAD need user involvement in conflict resolution